

Redis源代码分析

文档审核人	胡戊(huwu)
文档拟制人	邹雨晗(yuhanzou)
文档提交时间	Jun 17, 2011

文档修改记录

文档更新时间	变更内容	变更提出部门	变更理由
Jun 17, 2011	初稿		

目 录

1. Redis介绍	4
2. 基本功能	4
2.1. 链表(<i>adlist.h/adlist.c</i>)	4
2.2. 字符串(<i>sds.h/sds.c</i>)	5
2.3. 哈希表(<i>dict.h/dict.c</i>)	6
2.4. 内存(<i>zmalloc.h/zmalloc.h</i>)	9
3. 服务器模型	11
3.1. 事件处理(<i>ae.h/ae.c</i>)	11
3.2. 套接字操作(<i>anet.h/anet.c</i>)	15
3.3. 客户端连接(<i>networking.h/networking.c, redis.c/redis.h</i>)	15
3.4. 命令处理	17
4. 虚拟内存	19
4.1. 数据读取过程	20
4.2. 数据交换策略	23
5. 备份机制	24
5.1. <i>Snapshot</i>	24
5.2. <i>AOF</i>	26
6. 主从同步	27
6.1. 建立连接	27
6.2. 指令同步	30
6.3. 主从转换	31

1. Redis介绍

Redis(<http://redis.io>)是一个开源的Key-Value存储引擎。它支持string、hash、list、set和sorted set等多种值类型。

2. 基本功能

2.1. 链表(adlist.h/adlist.c)

链表(list)是Redis中最基本的数据结构, 由adlist.h和adlist.c定义。

基本数据结构

listNode是最基本的结构, 标示链表中的一个结点。结点有向前(next)和向后(prev)连个指针, 链表是双向链表。保存的值是void*类型。

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

链表通过list定义, 提供头(head)、尾(tail)两个指针, 分别指向头部的结点和尾部的结点; 提供三个函数指针, 供用户传入自定义函数, 用于复制(dup)、释放(free)和匹配(match)链表中的结点的值(value); 通过无符号整数len, 标示链表的长度。

```
typedef struct list {
    listNode *head;
    listNode *tail;
    void *(*dup)(void *ptr);
    void (*free)(void *ptr);
    int (*match)(void *ptr, void *key);
    unsigned int len;
} list;
```

listIter是访问链表的迭代器, 指针(next)指向链表的某个结点, direction标示迭代访问的方向(宏AL_START_HEAD表示向前, AL_START_TAIL表示向后)。

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

使用方法

Redis定义了一系列的宏, 用于访问list及其内部结点。

链表创建时(listCreate), 通过Redis自己实现的zmalloc()分配堆空间。链表释放(listRelease)或删除结点(listDelNode)时, 如果定义了链表(list)的指针函数free, Redis会使用它释放链表的每一个结点的值(value), 否则需要用户手动释放。结点的内存使用Redis自己实现的zfree()释放。

对于迭代器, 通过方法listGetIterator()、listNext()、listReleaseIterator()、listRewind()和listRewindTail()使用, 例如对于链表list, 要从头向尾遍历, 可通过如下代码:

```
iter = listGetIterator(list, AL_START_HEAD); // 获取迭代器
```

```
while((node = listNext(iter)) != NULL) {
    DoItWithValue(listNodeValue(node)); // 用户实现DoItWithValue
}
listReleaseIterator(iter);
```

`listDup()`用于复制链表，如果用户实现了`dup`函数，则会使用它复制链表结点的`value`。`listSearchKey()`通过循环的方式在 $O(N)$ 的时间复杂度下查找值，若用户实现了`match`函数，则用它进行匹配，否则使用按引用匹配。

2.2. 字符串(sds.h/sds.c)

字符串是Redis中最基本的数据。Redis使用`key`作为存取`value`的唯一标示符，而`key`的通俗理解就是字符串。Redis中的字符串分为两类：二进制安全(Binary Safe)和非二进制安全。二进制安全的字符串，是指字符串中所有字符均可用256个字符(8bit)编码[2]。Redis中的`value`都是通过二进制安全的字符串存储的，而`key`使用的是非二进制安全的。

基本数据结构

Redis内部实现了字符串类型，由`sds.h`和`sds.c`定义。

`sds`本质是`char*`：

```
typedef char *sds;
```

通过`sdsnewlen()`创建时，Redis内部会创建`sdshdr`结构，这个结构的定义如下，其中`len`表示字符串的实际长度，`free`表示可以额外使用的字节数。`sdshdr`分配的内存空间，为`sizeof(int)+sizeof(int)+sizeof(char*)+len+free`。

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

在`sdsnewlen(const void *init, size_t initlen)`中，会分配`sizeof(struct sdshdr)+initlen+1`的空间，并将`*init`指向的字符串拷贝到`buf`中，在`buf`末尾补上`'\0'`。但是`sdsnewlen()`中返回给外部的，只有`sdshdr->buf`。

假如用户调用了`sdsnewlen("qqmail", 6)`，则64位系统中，Redis会分配 $24 + 6 + 1$ 字节的空间，在内存中的结构如下：

	6	0	qqmail	
--	---	---	--------	--

返回给用户的`sdshdr->buf`，指向内存中“qqmail”的起点，想通过`buf`获得指向`sdshdr`的指针是很容易的，例如方法`sdslen()`中：

```
size_t sdslen(const sds s) {
    struct sdshdr *sh = (void*) (s-(sizeof(struct sdshdr)));
    return sh->len;
}
```

通过`(s-(sizeof(struct sdshdr)))`即可得到`sdshdr`的地址。

使用方法

了解了`sdshdr`的结构之后，理解相关的方法就很容易了。而Redis这样仅暴露`buf`的做法，使得应用程序可以将`sds`简单地当成`char*`使用。

创建sds的方法有：

```
sds sdsnewlen(const void *init, size_t initlen);
sds sdsnew(const char *init);
sds sdsempty();
```

三个方法均用`zmalloc()`分配内存空间，`sdsempty()`创建空字符串。`void sdsfree(sds s)`用于释放字符串。`size_t sdslen(const sds s)`会返回字符串的总长度。而`size_t sdsavail(sds s)`返回字符串中可用的字节数。以下几个函数会涉及sds内存空间的分配：

```
sds sdsgrowzero(sds s, size_t len);
sds sdscatlen(sds s, void *t, size_t len);
sds sdscat(sds s, char *t);
sds sdscpylen(sds s, char *t, size_t len);
sds sdscpy(sds s, char *t);
```

其中`sdsgrowzero()`会增加sds的内存空间，并用`'\0'`填充，用`free`记录空余空间。`sdscat()`将字符串`t`连接在`s`的尾部，`sdscpy()`将字符串`t`复制到`s`中。若`s`中空间不够时，会调用`sdsgrowzero()`分配更多内存。

此外，sds还提供了一些字符串相关的操作函数，在此不详细分析。

2.3. 哈希表(dict.h/dict.c)

Redis的哈希表最大的特色就是自动扩容。当它的哈希表容量不够时，可以0/1切换，然后自动扩容。下面具体分析哈希表的实现。

整个哈希系统由结构体`dict`定义，其中`type`包含一系列哈希表需要用的函数，`dictht`类型的数组`ht[2]`表示两个哈希表实例，由`rehashidx`指明下一个需要扩容的哈希实例的编号，`iterators`记录外部使用哈希表的迭代器的数目。

```
typedef struct dict {
    dictType *type;
    void *privdata;
    dictht ht[2];
    int rehashidx; /* rehashing not in progress if rehashidx == -1 */
    int iterators; /* number of iterators currently running */
} dict;
```

`dictht`为哈希表具体实现的结构体，`table`指向哈希中的记录，用数组+开链的形式保存记录；`size`表示哈希表桶的大小，为2的指数；`sizemark=size-1`，方便哈希值根据`size`取模；`used`记录了哈希表中的记录数目。

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

哈希表使用开链的方式处理冲突，每条记录都是链表中的一个结点。`dictType`在哈希系统中包含了一系列可由应用程序定义的函数指针，包括Hash函数、Key复制、Value复制、Key比较、Key析构、Value析构，以增加哈希系统的灵活性。其中系统定义了三种默认的`type`，表示最常用的三种哈希表。

```
typedef struct dictType {
    unsigned int (*hashFunction)(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);
} dictType;
extern dictType dictTypeHeapStringCopyKey;
extern dictType dictTypeHeapStrings;
extern dictType dictTypeHeapStringCopyKeyValue;

```

Redis定义了一系列的宏用于操作哈希表，例如设置记录的value。对外提供的API，除了常规的创建哈希表，增、删、改记录之外，有两类是比较特别的：自动扩容和迭代器。

自动扩容

Redis用变量dict_can_resize记录哈希是否可以自动扩容，由两个方法dictEnableResize()和dictDisableResize()设置该变量。

应用程序可以使用dictResize()扩容，它首先判断是否允许扩容，及是否正在扩容。若可以扩容，则调用dictExpand()扩容。然后应用程序需要调用dictRehashMilliseconds()启动扩容过程，并指定扩容过程中记录拷贝速度。

除了应用程序指定的扩容外，在调用dictAdd()往哈希中添加记录时，系统也会通过调用_dictExpandIfNeeded()判断是否需要扩容。_dictExpandIfNeeded()中，如果正在扩容，则不会重复进行扩容；如果哈希表size=0，即桶数目为0，则扩容到初始大小；否则如果used>=size，并且can_resize==1或used/size超过阈值（默认为5）时，以max(used, size)的两倍为基数，调用dictExpand()扩容。

```

if (dictIsRehashing(d)) return DICT_OK;
if (d->ht[0].size == 0) return dictExpand(d, DICT_HT_INITIAL_SIZE);

if (d->ht[0].used >= d->ht[0].size &&
    (dict_can_resize ||
     d->ht[0].used/d->ht[0].size > dict_force_resize_ratio))
{
    return dictExpand(d, ((d->ht[0].size > d->ht[0].used) ?
                          d->ht[0].size : d->ht[0].used)*2);
}
return DICT_OK;

```

dictExpand()进行扩容时，会先选择一个满足size需求的2的指数，然后分配内存空间，创建新的哈希表。如果此时ht[0]为空，则直接将哈希表赋值给ht[0]；否则，赋值给ht[1]，并启动拷贝过程，将ht[0]的记录逐个桶地拷贝到ht[1]中。置rehashidx=0，表明正在扩容，且待拷贝的桶为ht[0]->table[rehashidx]。

```

dictht n; /* the new hashtable */
unsigned long realsize = _dictNextPower(size);

if (dictIsRehashing(d) || d->ht[0].used > size)
    return DICT_ERR;

n.size = realsize;
n.sizemask = realsize-1;
n.table = zcalloc(realsize*sizeof(dictEntry*));
n.used = 0;

if (d->ht[0].table == NULL) {
    d->ht[0] = n;

```

```

        return DICT_OK;
    }

    d->ht[1] = n;
    d->rehashidx = 0;
    return DICT_OK;

```

`dictIsRehashing()`通过`rehashidx`来判断是否正在扩容。这个方法在多处被调用，当`dictAdd()`往哈希表中添加记录时，也会通过该方法判断是否正在扩容。若正在扩容，则调用`_dictRehashStep()`，该函数判断，若此时`iterators==0`，即没有迭代器时，就从`ht[0]`中拷贝一部分记录到`ht[1]`。

拷贝过程在`dictRehash()`中完成，该函数返回0时，表示扩容结束，`ht[0]`中所有记录都已拷贝到`ht[1]`，且`rehashidx`被置为-1；否则返回1，表示扩容未结束。拷贝过程中，将`ht[0]->table[rehashidx]`拷贝到`ht[1]`后，`rehashidx++`，直到`used==0`，即所有记录拷贝完成。拷贝一个桶时，需要对桶中所有元素重新求哈希值，然后一个个放入`ht[1]`中。`dictRehash()`通过参数，控制每次拷贝的桶的数目。该过程由下面代码描述：

```

int dictRehash(dict *d, int n) {
    if (!dictIsRehashing(d)) return 0;

    while(n--) {
        dictEntry *de, *nextde;

        if (d->ht[0].used == 0) {
            zfree(d->ht[0].table);
            d->ht[0] = d->ht[1];
            _dictReset(&d->ht[1]);
            d->rehashidx = -1;
            return 0;
        }

        while(d->ht[0].table[d->rehashidx] == NULL) d->rehashidx++;
        de = d->ht[0].table[d->rehashidx];

        while(de) {
            unsigned int h;

            nextde = de->next;
            /* Get the index in the new hash table */
            h = dictHashKey(d, de->key) & d->ht[1].sizemask;
            de->next = d->ht[1].table[h];
            d->ht[1].table[h] = de;
            d->ht[0].used--;
            d->ht[1].used++;
            de = nextde;
        }
        d->ht[0].table[d->rehashidx] = NULL;
        d->rehashidx++;
    }
    return 1;
}

```


迭代器

迭代器提供了遍历哈希表中所有元素的方法，通过`dictGetIterator()`获得迭代器后，使用`dictNext(dictIterator *)`方法获得下一个元素。当外部持有的迭代器数目不为0时，哈希表会暂停扩容操作。

迭代器遍历的过程，从`ht[0]`开始，依次从第一个桶`table[0]`开始遍历桶中的元素，然后`table[1]`，`table[2]`，...，`table[size]`，若正在扩容，则会继续遍历`ht[1]`中的桶。遍历桶中元素时，依次访问链表中的每个元素。

2.4. 内存(zmalloc.h/zmalloc.h)

前文已提到，Redis通过自己的方法管理内存，主要方法有`zmalloc()`，`zrealloc()`，`zcalloc()`和`zfree()`，分别对应C中的`malloc()`，`realloc()`，`calloc()`和`free()`。相关代码在`zmalloc.h`和`zmalloc.c`中。

Redis自己管理内存的好处主要有两个：可以利用内存池等手段提高内存分配的性能；可以掌握更多的内存信息，以便于Redis虚拟内存(VM)等功能中，决定何时将数据swap到磁盘。

先回忆各个系统中常见的内存分配函数：

malloc()分配一块指定大小的内存区域，并返回指向区域开头的指针，若分配失败，则返回NULL。

calloc()与`malloc()`一样，分配一块指定大小的内存区域，成功时返回区域头指针，失败返回NULL。区别在于，`calloc()`的输入参数为`count`和`size`，即分配的项的数目，及每一项的大小。`calloc()`在成功分配内存空间后，会将空间内所有值置0。

realloc()修改已分配的内存块的大小。若已分配的内存块后没有足够的空间用于扩展内存块，则重新申请一块满足需要的内存块，并将旧的数据拷贝到新位置，释放旧的内存块，返回指向新的内存块的指针；否则直接扩展原有的内存块。若分配失败，返回NULL。

free()释放已分配的内存块。

内存分配

在Redis中，如果系统中包含TCMALLOC，则会使用`tc_malloc()`等TCMALLOC中的方法代替`malloc()`等原有的分配内存方法。TCmalloc是google perftools中的一个组件。

```
#if defined(USE_TCMALLOC)
#define malloc(size) tc_malloc(size)
```

首先看`zmalloc()`和`zfree()`两个最常用的方法。Redis在申请内存时，除了申请需要的`size`外，还会多申请一块定长(`PREFIX_SIZE`)的区域用于记录所申请的内存块的长度。如果申请成功，Redis会使用宏函数（Redis中为性能考虑，大量使用宏函数）`update_zmalloc_stat_alloc(size+PREFIX_SIZE, size)`记录申请的内存块的相关信息，以便监控内存使用状况；当内存块被`zfree()`释放时，根据头部的信息可以快速获知被释放的内存区域的长度，然后通过宏函数`update_zmalloc_stat_free()`标记释放。源代码中，若系统支持`malloc_size()`方法，则会使用它返回指针所指向的内存块的大小（Mac OS X 10.4以上支持该方法[3]）。有疑惑的是，在支持`malloc_size()`的系统中，为何还要多申请`PREFIX_SIZE`的内存？

```

void *zmalloc(size_t size) {
    void *ptr = malloc(size+PREFIX_SIZE);

    if (!ptr) zmalloc_oom(size);
#ifdef HAVE_MALLOC_SIZE
    update_zmalloc_stat_alloc(redis_malloc_size(ptr),size);
    return ptr;
#else
    *((size_t*)ptr) = size; // 在头部记录内存块的长度
    update_zmalloc_stat_alloc(size+PREFIX_SIZE,size);
    return (char*)ptr+PREFIX_SIZE;
#endif
}

```

宏`update_zmalloc_stat_alloc()`中，首先将要分配的空间与内存对齐，然后会根据宏`zmalloc_thread_safe`判断是否需要对内存信息记录表的相关操作加锁。虽然Redis在大部分场景中是单线程读写的，即`thread_safe`的，但启用虚拟内存(VM)，或持久化dump到磁盘等操作时会启动多线程，因此在多线程模式中，需要对部分操作加锁。

内存监控

`used_memory`记录了Redis使用的内存总数。而多线程下`malloc()`是线程安全的。`zmalloc_allocations[]`记录了各个size分配的内存块的数目，大于256个字节的按256算。应用程序可以通过`zmalloc_allocations_for_size(size)`获得对应size的内存块的分配数目；也可以通过`zmalloc_used_memory()`获得Redis占用的总内存。这些监控类的方法在Redis的日志系统中被用到。

```

#define update_zmalloc_stat_alloc(__n,__size) do { \
    size_t _n = (__n); \
    size_t _stat_slot = (__size < ZMALLOC_MAX_ALLOC_STAT) ? __size : \
    ZMALLOC_MAX_ALLOC_STAT; \
    if (_n&(sizeof(long)-1)) _n += sizeof(long)-(_n&(sizeof(long)-1)); \
    if (zmalloc_thread_safe) { \
        pthread_mutex_lock(&used_memory_mutex); \
        used_memory += _n; \
        zmalloc_allocations[_stat_slot]++; \
        pthread_mutex_unlock(&used_memory_mutex); \
    } else { \
        used_memory += _n; \
        zmalloc_allocations[_stat_slot]++; \
    } \
} while(0)

```

外部应用程序通过`zmalloc_enable_thread_safeness()`方法开启Redis内存模块的线程安全模式，后文会分析哪些功能需要开启线程安全。

`zcalloc(size)`、`zrealloc()`与`zmalloc()`的处理策略类似，不再详述。

在部分操作系统中，Redis可以通过`zmalloc_get_rss()`方法获得自己的进程占用的内存信息。该信息通过操作系统提供，往往比Redis自己记录的`used_memory`更准确，但其获取速度也较慢。这些信息也是用于虚拟内存功能。

除了内存相关的操作外，Redis在此还提供了一个复制字符串的方法`zstrdup(char*)`，该方法将申请一块与源字符串长度相同的内存区域，并用`memcpy()`拷贝字符串的内容。

3. 服务器模型

3.1. 事件处理(ae.h/ae.c)

Redis是单线程模型（虚拟内存等功能会启动其它线程(进程)），通过事件机制异步地处理所有请求。

Redis的事件模型在不同的操作系统中提供了不同的实现，ae_epoll.h/ae_epoll.c为epoll的实现，ae_select.h/ae_select.c是select的实现，ae_kqueue.h/ae_kqueue.c是bsd中kqueue的实现。

基本事件

Redis提供两种基本事件：FileEvent和TimeEvent。前者是基于操作系统的异步机制(epoll/kqueue)实现的文件事件，后者是Redis自己实现的定时器。

aeEventLoop是事件模型中最基本的结构体：

```
typedef struct aeEventLoop {
    int maxfd;
    long long timeEventNextId;
    aeFileEvent events[AE_SETSIZE]; /* Registered events */
    aeFiredEvent fired[AE_SETSIZE]; /* Fired events */
    aeTimeEvent *timeEventHead;
    int stop;
    void *apidata; /* This is used for polling API specific data */
    aeBeforeSleepProc *beforesleep;
} aeEventLoop;
```

maxfd标示当前最大的事件描述符。events和fired分别保存了已注册和已注销的FileEvent，AE_SETSIZE是Redis中可以注册的事件的上限，默认为1024*10。timeEventHead指向一个TimeEvent的链表，timeEventNextId标示下一个定时器。beforesleep是每次事件轮询前都会执行的函数，相当于hook。stop用于停止事件轮询。

```
typedef struct aeFileEvent {
    int mask; /* one of AE_(READABLE|WRITABLE) */
    aeFileProc *rfileProc;
    aeFileProc *wfileProc;
    void *clientData;
} aeFileEvent;
```

aeFileEvent可以用于socket事件的监听。mask表示要监听的事件类型，rfileProc和wfileProc分别为读事件和写事件的响应函数。clientData。

```
typedef struct aeTimeEvent {
    long long id; /* time event identifier. */
    long when_sec; /* seconds */
    long when_ms; /* milliseconds */
    aeTimeProc *timeProc;
    aeEventFinalizerProc *finalizerProc;
    void *clientData;
    struct aeTimeEvent *next;
} aeTimeEvent;
```

aeTimeEvent用于定时器(timer)，其实现是一个链表，其中每一个结点是一个timer，并有独立id。when_sec和when_ms指定了定时器的触发时间，timeProc为响应函数，finalizerProc为删除定时器时的“析构函数”。

`aeFiredEvent`表示触发的`FileEvent`，其中`fd`为文件描述符，`mask`为事件类型。

相关操作

Redis提供了一系列的API用于操作事件。`aeCreateEventLoop()`和`aeDeleteEventLoop(aeEventLoop*)`分别用于初始化和删除事件循环。初始化`eventLoop`时，会初始化对应操作系统的实现，例如Linux中会使用`ae_epoll.c`中的`aeApiCreate()`初始化事件使用的文件描述符和事件列表。Redis中使用了三个宏`AE_NONE`、`AE_READABLE`和`AE_WRITABLE`表示监听的`FileEvent`的类型，初始时`eventLoop`中所有`events`的监听类型为`AE_NONE`。`aeStop(aeEventLoop*)`用户暂停事件轮询。

`aeCreateFileEvent()`和`aeDeleteFileEvent()`用于创建和删除`FileEvent`，在Linux系统中，会使用`epoll`监听文件描述符，并用`EPOLLIN`和`EPOLLOUT`监听文件的Read和Write事件，并在`eventLoop->apiData`中记录相关信息。

`aeCreateTimeEvent()`和`aeDeleteTimeEvent()`用于创建和删除`TimeEvent`。

`aeMain(aeEventLoop*)`是启动事件轮询的入口，内部实现是一个死循环，每一次循环中先执行`eventLoop->beforesleep()`方法，然后使用`aeProcessEvents()`轮询事件。`aeProcessEvents()`中，首先找到最近的`timer`（最先超时的），记下距该`timer`触发的时间，并作为检查`FileEvent`的超时时间。Redis通过不同操作系统实现(Linux:`ae_epoll.c`)中的`aeApiPoll()`方法，检查是否触发了`FileEvent`，若触发则根据`mask`决定响应函数(`rfileProc/wfileProc`)。`FileEvent`的优先级高于`TimeEvent`，因此可以将`TimerEvent`的触发时间用作`FileEvent`轮询的超时时间。

检查完`FileEvent`后，通过调用`processTimeEvents()`检查`TimeEvent`，触发所有已超时的`timer`的`timeProc`，根据`timeProc`的返回值，决定是删除`timer`还是继续增加`timer`的触发时间。例如，`timeProc`返回100时，则将`timer`的触发时间增加100ms，即100ms后再次触发该`timer`。若`timeProc`一直返回100，则该`timer`会每隔100ms触发一次。其中重要代码如下：

```
...
retval = te->timeProc(eventLoop, id, te->clientData);
processed++;
if (retval != AE_NOMORE) {
    aeAddMillisecondsToNow(retval, &te->when_sec, &te->when_ms);
} else {
    aeDeleteTimeEvent(eventLoop, id);
}
...
```

除了上述异步的事件机制外，Redis还提供了一个同步处理事件的API：`aeWait(int fd, int mask, long long milliseconds)`，`fd`指明了需要监听的文件描述符，`mask`说明监听的事件类型，`milliseconds`指明超时事件。该方法内部通过`select`实现了阻塞型等待。

Redis提供`aeSetBeforeSleepProc()`用于设置事件轮询中每次循环前执行的方法。

初始化事件轮询

Redis启动时便会初始化事件轮询并启动，`int main()`中，调用`initServer()`初始化服务端。在`initServer()`中，初始化事件轮询，通过`TimeEvent`定制一个定时器

serverCron, 每100ms执行一次服务器相关的操作。在Redis所有初始化工作完成后, 执行aeMain() 启动事件轮询。

```
int initServer() {
    // ...
    server.el = aeCreateEventLoop();
    // ...
}
int main(int argc, char **argv) {
    // ...
    initServer();
    // ....
    aeSetBeforeSleepProc(server.el,beforeSleep);
    aeMain(server.el);
    aeDeleteEventLoop(server.el);
}
```

事件系统在处理TimeEvent和FileEvent之前, 会调用BeforeSleepProc, Redis中事件循环前会调用beforeSleep()。该函数负责许多功能, 例如处理已经ready的客户端连接, 处理虚拟内存的换入换出机制, 处理aof等等。

serverCron()是Redis中非常重要的函数, 负责协调Redis中许多功能。首先更新全局的时间戳server.unixtime, Redis中各个模块中对时间要求不严格的地方, 均通过这个时间戳获取系统时间, 以避免调用time(NULL)的时间消耗。然后更新虚拟内存换入换出策略相关的时间戳server.lruclock。

```
/* We take a cached value of the unix time in the global state because
 * with virtual memory and aging there is to store the current time
 * in objects at every object access, and accuracy is not needed.
 * To access a global var is faster than calling time(NULL) */
server.unixtime = time(NULL);
/* We have just 22 bits per object for LRU information.
 * So we use an (eventually wrapping) LRU clock with 10 seconds resolution.
 * 2^22 bits with 10 seconds resolution is more or less 1.5 years.
 *
 * Note that even if this will wrap after 1.5 years it's not a problem,
 * everything will still work but just some object will appear younger
 * to Redis. But for this to happen a given object should never be touched
 * for 1.5 years.
 *
 * Note that you can change the resolution altering the
 * REDIS_LRU_CLOCK_RESOLUTION define.
 */
updateLRUClock();
```

然后, 如果收到SIGTERM信息, 会停止服务器。然后打印一些数据库相关的运行日志。当没有后台的虚拟内存控制进程和备份进程时, 会尝试对哈希表进行扩容。

```
/* We don't want to resize the hash tables while a background saving
 * is in progress: the saving child is created using fork() that is
 * implemented with a copy-on-write semantic in most modern systems, so
 * if we resize the HT while there is the saving child at work actually
 * a lot of memory movements in the parent will cause a lot of pages
 * copied. */
if (server.bgsavechildpid == -1 && server.bgrewritechildpid == -1) {
    if (!(loops % 10)) tryResizeHashTables();
    if (server.activerehashing) incrementallyRehash();
}
```

然后输出客户端连接的相关数据日志，并关闭超时连接。再次检查，如果有后台进程在进行备份，则等待它们完成，并进行一些清理，然后开始执行哈希表扩容。如果没有上述后台进程，则判断是否需要进行备份。

```
/* Check if a background saving or AOF rewrite in progress terminated */
if (server.bgsavechildpid != -1 || server.bgrewritechildpid != -1) {
    int statloc;
    pid_t pid;

    if ((pid = wait3(&statloc, WNOHANG, NULL)) != 0) {
        if (pid == server.bgsavechildpid) {
            backgroundSaveDoneHandler(statloc);
        } else {
            backgroundRewriteDoneHandler(statloc);
        }
        updateDictResizePolicy();
    }
} else {
    /* If there is not a background saving in progress check if
     * we have to save now */
    time_t now = time(NULL);
    for (j = 0; j < server.saveparamslen; j++) {
        struct saveparam *sp = server.saveparams+j;

        if (server.dirty >= sp->changes &&
            now-server.lastsave > sp->seconds) {
            redisLog(REDIS_NOTICE, "%d changes in %d seconds. Saving...",
                sp->changes, sp->seconds);
            rdbSaveBackground(server.dbfilename);
            break;
        }
    }
}
```

若服务器为master，则释放过期的key；若为slave，则等待同步指令DEL。

```
/* Expire a few keys per cycle, only if this is a master.
 * On slaves we wait for DEL operations synthesized by the master
 * in order to guarantee a strict consistency. */
if (server.masterhost == NULL) activeExpireCycle();
```

然后检查物理内存使用率，若超过阈值，则将部分不常用的Entity换出到磁盘。Redis的虚拟内存的换入换出机制分两种，根据server.vm_max_threads设置分别使用不同的机制。

```
/* Swap a few keys on disk if we are over the memory limit and VM
 * is enabled. Try to free objects from the free list first. */
if (vmCanSwapOut()) {
    while (server.vm_enabled && zmalloc_used_memory() >
           server.vm_max_memory)
    {
        int retval = (server.vm_max_threads == 0) ?
            vmSwapOneObjectBlocking() :
            vmSwapOneObjectThreaded();
        if (retval == REDIS_ERR && !(loops % 300) &&
            zmalloc_used_memory() >
            (server.vm_max_memory+server.vm_max_memory/10))
        {
            redisLog(REDIS_WARNING, "WARNING: vm-max-memory limit exceeded by
more than 10%% but unable to swap more objects out!");
        }
    }
}
```



```

        /* Note that when using threaded I/O we free just one object,
        * because anyway when the I/O thread in charge to swap this
        * object out will finish, the handler of completed jobs
        * will try to swap more objects if we are still out of memory. */
        if (retval == REDIS_ERR || server.vm_max_threads > 0) break;
    }
}

```

最后，若服务器为slave，则每10次serverCron（即1秒）检查一次与master的连接式否正常。

```

/* Replication cron function -- used to reconnect to master and
 * to detect transfer failures. */
if (!(loops % 10)) replicationCron();

// Todo slave不正常

```

3.2. 套接字操作(anet.h/anet.c)

Redis在`anet.h/c`中封装了所有底层套接字操作：

```

int anetTcpConnect(char *err, char *addr, int port);
int anetTcpNonBlockConnect(char *err, char *addr, int port);
int anetUnixConnect(char *err, char *path);
int anetUnixNonBlockConnect(char *err, char *path);
int anetRead(int fd, char *buf, int count);
int anetResolve(char *err, char *host, char *ipbuf);
int anetTcpServer(char *err, int port, char *bindaddr);
int anetUnixServer(char *err, char *path);
int anetTcpAccept(char *err, int serversock, char *ip, int *port);
int anetUnixAccept(char *err, int serversock);
int anetWrite(int fd, char *buf, int count);
int anetNonBlock(char *err, int fd);
int anetTcpNoDelay(char *err, int fd);
int anetTcpKeepAlive(char *err, int fd);

```

`anetTcpServer()`是建立网络套接字服务器的方法，即对`socket()`、`bind()`、`listen()`等一系列操作的封装，并返回`socket`的`fd`。`anetUnixServer()`建立本地套接字服务器。

`anetTcpConnect()`、`anetTcpNonBlockConnect()`用于建立阻塞型和非阻塞型网络套接字连接。`anetUnixConnect()`、`anetUnixNonBlockConnect()`用于建立阻塞型和非阻塞型的本地套接字连接。

`anetNonBlock()`将一个`fd`设置为非阻塞型。`anetTcpNoDelay()`将TCP连接设为非延迟的，即屏蔽Nagle算法。`anetTcpKeepAlive()`开启连接检测，避免对方宕机、网络中断时`fd`永远阻塞。

`anetRead()`和`anetWrite()`用于套接字上的读写功能，`anetTcpAccept()`和`anetUnixAccept()`分别在网络套接字和本地套接字上新增连接。

Redis通过这些API封装了所有套接字操作，但这些操作并没有什么特别之处，不再详细分析其实现。

3.3. 客户端连接(networking.h/networking.c, redis.c/redis.h)

本节关注Redis的并发框架，及客户端连接的建立和请求处理。上述事件模型，及网络操作，拼凑出Redis的服务器并发框架。

初始化服务器

`main()` 中先调用 `initServerConfig()` 初始化服务器配置。包括端口、物理文件等基本信息，虚拟内存的各种设置，及 `command table(server.commands)` 等。前文已说过，Redis的dict支持自定义各种行为（例如Hash函数，比较函数等），初始化 `command table` 时，首先用 `commandTableDictType` 指定将 `command dict` 的各种行为，然后 `populateCommand()` 初始化各种命令。Redis中命令由结构体 `redisCommand` 定义，后文分析。这里，Redis会将一系列的命令载入 `command table` 中。

然后 `main()` 调用 `initServer()` 初始化事件轮询，紧接着会调用 `anetTcpServer()` 初始化服务器。

```
initServer() {
    // ....
    server.ipfd = anetTcpServer(server.neterr,server.port,server.bindaddr);
    // ...
    if (server.ipfd > 0 && aeCreateFileEvent(server.el,server.ipfd,AE_READABLE,
acceptTcpHandler,NULL) == AE_ERR) oom("creating file event");
    // ...
}
```

完成服务器的初始化后，Redis已开始监听端口，通过 `aeCreateFileEvent()` 监听端口的 `READABLE` 事件，若该事件触发，则表示有新连接建立。`acceptTcpHandler()` 响应 `READABLE` 事件，调用 `anetTcpAccept()` 建立连接，获得客户端连接的 `fd`。获得 `fd` 后，通过 `createClient()` 方法，创建连接相关信息，保存在结构体 `redisClient` 中。该结构体保存了连接的 `fd`，输入 `buf`，输出 `buf` 等一系列的信息。**Redis在处理请求时完全**是用单线程异步模式，通过 `epoll` 监听所有连接的读写事件，并通过相应的响应函数处理。**Redis使用这样的设计来满足需求，很大程度上因为它服务的是高并发的短连接，且请求处理事件非常短暂。这个模型下，一旦有请求阻塞了服务，整个Redis的服务将受影响。Redis采用单线程模型可以避免系统上下文切换的开销。**

Redis的连接由结构体 `redisClient(redis.h)` 定义，默认使用非阻塞、非延迟型的连接。连接由 `createClient()` 方法创建，由 `freeClient()` 方法释放。`createClient()` 的过程中，在连接的 `fd` 上监听 `READABLE` 事件，当该事件触发时说明接收到客户端请求，调用 `readQueryFromClient()` 方法处理请求。然后初始化连接的输入缓冲区，参数列表等等。最后将连接加入全局的连接列表 `server.clients` 中。

```
/* With multiplexing we need to take per-clinet state.
 * Clients are taken in a liked list. */
typedef struct redisClient {
    int fd;
    redisDb *db;
    int dictid;
    sds querybuf;
    int argc;
    robj **argv;
    int reqtype;
    int multibulklen; /* number of multi bulk arguments left to read */
    long bulklen; /* length of bulk argument in multi bulk request */
    list *reply;
    int sentlen;
    time_t lastinteraction; /* time of the last interaction, used for timeout 最后一次交互的时间戳 */
    int flags; /* REDIS_SLAVE | REDIS_MONITOR | REDIS_MULTI ... 连接的状态标记，初始时为0 */
}
```



```

int slaveseldb;          /* slave selected db, if this client is a slave */
int authenticated;      /* when requirepass is non-NULL */
int replstate;          /* replication state if this is a slave */
int repldbfd;           /* replication DB file descriptor */
long repldloff;         /* replication DB file offset */
off_t repldbsize;       /* replication DB file size */
multiState mstate;      /* MULTI/EXEC state */
blockingState bpop;     /* blocking state */
list *io_keys;          /* Keys this client is waiting to be loaded from the
                        * swap file in order to continue. */

list *watched_keys;     /* Keys WATCHED for MULTI/EXEC CAS */
dict *pubsub_channels;  /* channels a client is interested in (SUBSCRIBE) */
list *pubsub_patterns;  /* patterns a client is interested in (SUBSCRIBE) */

/* Response buffer */
int bufpos;
char buf[REDIS_REPLY_CHUNK_BYTES];
} redisClient;

```

以下几种情况会使用`freeClient()`释放连接：从客户端读数据错误、已到达服务器最大连接数(`server.maxclients`)、

客户端的状态标记中，有三个状态会让Redis不再处理客户端命令：

`REDIS_BLOCKED`（处理阻塞操作）和`REDIS_IO_WAIT`（等待虚拟内存）会让客户端读入数据后阻塞，不处理命令。`REDIS_CLOSE_AFTER_REPLY`标记会让客户端输出数据后立即关闭，此时也不再处理输入的命令。

3.4. 命令处理

Redis的命令由结构体`redisCommand`定义，包含了指向命令处理函数的指针。

```

typedef void redisCommandProc(redisClient *c);
typedef void redisVmPreloadProc(redisClient *c, struct redisCommand *cmd, int
argc, robj **argv);
struct redisCommand {
    char *name;
    redisCommandProc *proc;
    int arity;
    int flags;

    redisVmPreloadProc *vm_preload_proc;

    int vm_firstkey;
    int vm_lastkey;
    int vm_keystep;
};

```

Redis在`readQueryFromClient()`中处理客户端的Input，每次尝试读取长度为`REDIS_IOBUF_LEN`（默认为1024）的数据，输入的数据保存在`redisClient`结构体的`querybuf`中`sdscatlen(c->querybuf,buf,nread)`，然后调用`processInputBuffer()`处理数据。

若客户端处于`ready`状态（即`flags`中无`REDIS_BLOCKED`、`REDIS_IO_WAIT`和`REDIS_CLOSE_AFTER_REPLY`）状态，则尝试处理输入数据。若客户端没有通过`reqtype`指定使用的协议，则根据数据的第一个字节判断，若`querybuf[0]=='*'`，则使用`MULTIBULK`协议，否则为`INLINE`协议，分别调用对应的协议解包函数并处理命令，完成后重置客户端(`resetClient()`)。

```

void processInputBuffer(redisClient *c) {
    while(sdslen(c->querybuf)) {
        if (c->flags & REDIS_BLOCKED || c->flags & REDIS_IO_WAIT) return;
        if (c->flags & REDIS_CLOSE_AFTER_REPLY) return;

        if (!c->reqtype) {
            if (c->querybuf[0] == '*') {
                c->reqtype = REDIS_REQ_MULTIBULK;
            } else {
                c->reqtype = REDIS_REQ_INLINE;
            }
        }

        if (c->reqtype == REDIS_REQ_INLINE) {
            if (processInlineBuffer(c) != REDIS_OK) break;
        } else if (c->reqtype == REDIS_REQ_MULTIBULK) {
            if (processMultibulkBuffer(c) != REDIS_OK) break;
        } else {
            redisPanic("Unknown request type");
        }

        if (c->argc == 0) {
            resetClient(c);
        } else {
            if (processCommand(c) == REDIS_OK)
                resetClient(c);
        }
    }
}

```

INLINE协议非常简单，按换行符（'\r\n'）分隔，一行是一条命令。命令中参数用空格分隔，通过processInlineBuffer()解析后，存放在argc/argv中，其中argv是redisObject对象的数组。

MULTIBULK协议 //TODO

processCommand()中处理所有客户端的命令。首先是“quit”，Redis接收到这个命令后将客户端连接断开；否则，使用lookupCommand()从command table中查找命令，然后通过命令的处理函数(cmd->proc())处理命令。

以‘get’命令为例，它在readonlyCommandTable中定义，在initServerConfig()初始化过程中载入command table，若服务器接收到get命令，则通过lookupCommand()找到对应的redisCommand结构体的实例，通过处理函数getCommand()处理命令。

```

struct redisCommand readonlyCommandTable[] = {
    {"get",getCommand,2,0,NULL,1,1,1},
    // ....
}

```

redis将命令处理函数的实现写在t_***.c中，例如t_string.c实现了字符串操作相关的命令，包括getCommand()。

4. 虚拟内存

虚拟内存是Redis中最复杂的模块，它几乎影响了Redis所有模块的功能。Redis作为一个内存数据库，数据量受内存制约，为了存储更多的数据，Redis在2.0加入了虚拟内存功能，允许将不常访问的数据的value放入磁盘中。

虚拟内存部分的分析分为两部分：获取在vm中的数据的过程，数据在vm中换入换出策略。分析前者前，先看Redis为支持虚拟内存的数据结构。

Redis底层用结构体redisObject表示保存的数据项。

```
/* The actual Redis Object */
#define REDIS_LRU_CLOCK_MAX ((1<<21)-1) /* Max value of obj->lru */
#define REDIS_LRU_CLOCK_RESOLUTION 10 /* LRU clock resolution in seconds */
typedef struct redisObject {
    unsigned type:4;
    unsigned storage:2; /* REDIS_VM_MEMORY or REDIS_VM_SWAPPING */
    unsigned encoding:4;
    unsigned lru:22; /* lru time (relative to server.lruclock) */
    int refcount;
    void *ptr;
    /* VM fields are only allocated if VM is active, otherwise the
     * object allocation function will just allocate
     * sizeof(redisObjt) minus sizeof(redisObjectVM), so using
     * Redis without VM active will not have any overhead. */
} robj;
```

其中type表示value的数据类型：string、list、set；storage表示存储的位置，Redis共定义了四种存储状态，分别为内存，磁盘（虚拟内存），正在从内存换出到磁盘和正在从磁盘换入内存；encoding；lru与server.lruclock相关；refcount为引用计数。

```
#define REDIS_VM_MEMORY 0 /* The object is on memory */
#define REDIS_VM_SWAPPED 1 /* The object is on disk */
#define REDIS_VM_SWAPPING 2 /* Redis is swapping this object on disk */
#define REDIS_VM_LOADING 3 /* Redis is loading this object from disk */
```

与此同时，Redis定义结构体vmPointer，表示被从内存换出到磁盘上的数据项。

```
/* The VM pointer structure - identifies an object in the swap file.
 *
 * This object is stored in place of the value
 * object in the main key->value hash table representing a database.
 * Note that the first fields (type, storage) are the same as the redisObject
 * structure so that vmPointer strucuters can be accessed even when casted
 * as redisObject structures.
 *
 * This is useful as we don't know if a value object is or not on disk, but we
 * are always able to read obj->storage to check this. For vmPointer
 * structures "type" is set to REDIS_VMPINTER (even if without this field
 * is still possible to check the kind of object from the value of 'storage').*/
typedef struct vmPointer {
    unsigned type:4;
    unsigned storage:2; /* REDIS_VM_SWAPPED or REDIS_VM_LOADING */
    unsigned notused:26;
    unsigned int vtype; /* type of the object stored in the swap file */
    off_t page; /* the page at witch the object is stored on disk */
    off_t usedpages; /* number of pages used on disk */
} vmpointer;
```

这个设计非常巧妙，`vmpointer`的前32个字节与`redisObject`是可以互相转换的。
`vmPointer`的具体定义待后文再行分析。

```
off_t vm_page_size;
off_t vm_pages;
off_t vm_next_page; /* Next probably empty page */
off_t vm_near_pages; /* Number of pages allocated sequentially */
unsigned char *vm_bitmap; /* Bitmap of free/used pages */
```

Redis将内存按页(Page)管理，页的大小与数目在服务启动时指定。Redis通过位图索引`server.vm_bitmap`管理页，其中0表示未用，1表示已用。Redis中，一个页最多只存放一条数据，而一条数据可以拆分存放在多个页中。

```
/* An I/O thread process an element taken from the io_jobs queue and
 * put the result of the operation in the io_done list. While the
 * job is being processed, it's put on io_processing queue. */
list *io_newjobs; /* List of VM I/O jobs yet to be processed */
list *io_processing; /* List of VM I/O jobs being processed */
list *io_processed; /* List of VM I/O jobs already processed */
```

Redis中，将VM相关的读写都用任务(job)来描述及执行，`io_newjobs`表示新增的job，`processing`和`processed`分别表示正在执行和完成的job。Job总共有3种：LOAD、PREPARE_SWAP和DO_SWAP，分别为从磁盘读入内存、计算换出数据需要交换的页及换出到磁盘。

```
/* VM threaded I/O request message */
#define REDIS_IOJOB_LOAD 0 /* Load from disk to memory */
#define REDIS_IOJOB_PREPARE_SWAP 1 /* Compute needed pages */
#define REDIS_IOJOB_DO_SWAP 2 /* Swap from memory to disk */
typedef struct iobjob {
    int type; /* Request type, REDIS_IOJOB_* */
    redisDb *db; /* Redis database */
    robj *key; /* This I/O request is about swapping this key */
    robj *id; /* Unique identifier of this job:
               this is the object to swap for REDIS_IOREQ_*_SWAP, or the
               vmpointer object for REDIS_IOREQ_LOAD. */
    robj *val; /* the value to swap for REDIS_IOREQ_*_SWAP, otherwise this
               * field is populated by the I/O thread for REDIS_IOREQ_LOAD. */
    off_t page; /* Swap page where to read/write the object */
    off_t pages; /* Swap pages needed to save object. PREPARE_SWAP return val */
    int canceled; /* True if this command was canceled by blocking side of VM */
    pthread_t thread; /* ID of the thread processing this entry */
} iobjob;
```

job的结构体种，记录了相关的信息。任何Job，都通过`queueIOJob()`加入`server.io_newjobs`，当Redis发现能启动新的VM线程，或者有空闲的VM线程时，会执行job，线程的入口为`IOThreadEntryPoint`，该函数负责执行job，并将job从`server.io_newjobs`移入`server.io_processing`。然后Redis通过管道在线程间交流，VM线程将完成job的信号通过管道发送到主线程，主线程进行善后。Job具体的执行过程，在后文的数据读取过程中有更详细的分析。

4.1. 数据读取过程

对于`getCommand()`，从Redis中查询数据，Redis先用`lookupKey()`从哈希表中查找对应的`dictEntity`，校验其合法性后，如果服务器没有后台备份进程在工作，就更新它的时间戳。但如果启动了虚拟内存功能，就必须判断`dictEntity`的存储状态。如果该`dictEntity`正在内存中(`REDIS_VM_MEMORY`)，则不做操作；如果它正在被换出到磁盘

(REDIS_VM_SWAPPING), 则停止换出操作; 如果它在磁盘, 则调用`vmLoadObject()`将`dictEntity`从磁盘读出; 如果它正在从磁盘换入到内存, 则先调用`vmLoadObject()`将它立即从磁盘读出。具体来看这个过程, 在命令处理的入口`processCommand()`中,

```
if (server.vm_enabled && server.vm_max_threads > 0 &&
    blockClientOnSwappedKeys(c,cmd)) return REDIS_ERR;
call(c,cmd);
```

在虚拟内存开启, 并以多线程模式执行时, 会调用`blockClientOnSwappedKeys()`, 将可能已经换出到磁盘的数据加载到内存, 并在加载未完成前阻塞客户端连接(将客户端状态置为`REDIS_IO_WAIT`)。在`blockClientOnSwappedKeys()`中, 如果命令设置了预加载函数(`vm_preload_proc`), 则调用预加载函数处理, 否则调用`waitForMultipleSwappedKeys()`, 将需要的key载入内存。

```
int blockClientOnSwappedKeys(redisClient *c, struct redisCommand *cmd) {
    if (cmd->vm_preload_proc != NULL) {
        cmd->vm_preload_proc(c,cmd,c->argc,c->argv);
    } else {
        waitForMultipleSwappedKeys(c,cmd,c->argc,c->argv);
    }

    /* If the client was blocked for at least one key, mark it as blocked. */
    if (listLength(c->io_keys)) {
        c->flags |= REDIS_IO_WAIT;
        aeDeleteFileEvent(server.el,c->fd,AE_READABLE);
        server.vm_blocked_clients++;
        return 1;
    } else {
        return 0;
    }
}
```

`waitForMultipleSwappedKeys`和命令设置的预加载函数的载入过程均由`waitForSwappedKey()`开始, 该函数首先从数据库中找到`cmd`需要的数据项的`dictEntity`对象, 根据它存储的位置, 如果为`REDIS_VM_MEMORY`, 则直接返回, 无需阻塞客户端; 如果为`REDIS_VM_SWAPPING`, 则用`vmCancelThreadedIOJob(o)`取消正在进行的换出操作, 并直接返回, 无需阻塞客户端。其余情况, 均阻塞客户端。

每个客户端的结构体均有链表`c->io_keys`, 保存所有等待swapping的对象的key。如果`cmd`需要的数据项在磁盘上, 则将它的关键字加入`c->io_keys`。然后将`{c, key}`加入`c->db->io_keys`, 这个db其实是全局的`server->db`, 它为每个key, 保存了一个阻塞在它上面的客户端的队列。

接下来, 如果该key已经换出到磁盘, 并且没有job将它从磁盘读出, 则创建一个job, 将它从磁盘换出, job中用`vmpointer`对象标记需要换出的`dictEntity`。

```
int waitForSwappedKey(redisClient *c, robj *key) {
    struct dictEntry *de;
    robj *o;
    list *l;

    /* If the key does not exist or is already in RAM we don't need to
     * block the client at all. */
    de = dictFind(c->db->dict,key->ptr);
    if (de == NULL) return 0;
    o = dictGetEntryVal(de);
    if (o->storage == REDIS_VM_MEMORY) {
```

```

    return 0;
} else if (o->storage == REDIS_VM_SWAPPING) {
    /* We were swapping the key, undo it! */
    vmCancelThreadedIOJob(o);
    return 0;
}

/* OK: the key is either swapped, or being loaded just now. */

/* Add the key to the list of keys this client is waiting for.
 * This maps clients to keys they are waiting for. */
listAddNodeTail(c->io_keys,key);
incrRefCount(key);

/* Add the client to the swapped keys => clients waiting map. */
de = dictFind(c->db->io_keys,key);
if (de == NULL) {
    int retval;

    /* For every key we take a list of clients blocked for it */
    l = listCreate();
    retval = dictAdd(c->db->io_keys,key,l);
    incrRefCount(key);
    redisAssert(retval == DICT_OK);
} else {
    l = dictGetEntryVal(de);
}
listAddNodeTail(l,c);

/* Are we already loading the key from disk? If not create a job */
if (o->storage == REDIS_VM_SWAPPED) {
    iojob *j;
    vmpointer *vp = (vmpointer*)o;

    o->storage = REDIS_VM_LOADING;
    j = zmalloc(sizeof(*j));
    j->type = REDIS_IOJOB_LOAD;
    j->db = c->db;
    j->id = (robj*)vp;
    j->key = key;
    incrRefCount(key);
    j->page = vp->page;
    j->val = NULL;
    j->canceled = 0;
    j->thread = (pthread_t) -1;
    lockThreadedIO();
    queueIOJob(j);
    unlockThreadedIO();
}
return 1;
}

```

由于vm可能有多线程操作，所以需要将job队列操作放入临界区。queueIOJob的过程非常简单，将job加入server->io_newjobs队列，若当前的IO线程数少于上限，则创建一个IO线程处理该Job。

```

void queueIOJob(iojob *j) {
    redisLog(REDIS_DEBUG,"Queued IO Job %p type %d about key '%s'\n",
        (void*)j, j->type, (char*)j->key->ptr);
    listAddNodeTail(server.io_newjobs,j);
}

```

```

if (server.io_active_threads < server.vm_max_threads)
    spawnIOThread();
}

```

IO线程的入口为IOThreadEntryPoint(), 该函数循环处理server->io_newjobs中的job, 正在被处理的任务, 从server->io_newjobs中移入server->processing, 根据job类型选择不同的处理方法, 如REDIS_IOJOB_LOAD的job, 由vmReadObjectFromSwap()处理, 将磁盘的数据读入内存。完成的job被移入server->processed。最后, 通过server->io_ready_pipe_write通知其它线程, 处理server->processed中的job, 每当完成一个job时, 向io_ready_pipe写入一个字符。

server->io_ready_pipe_read和server->io_ready_pipe_write向线程间提供通信服务, 在虚拟内存初始化过程vmInit()中, 监听io_ready_pipe_read上的读事件, 事件响应函数vmThreadedIOCompletedJob处理完成的job。

vmThreadedIOCompletedJob根据job的类型处理其“善后”工作, 例如REDIS_IOJOB_LOAD, 此时key已被从vm中换入内存, 调用handleClientsBlockedOnSwappedKey唤醒被该key阻塞的客户端。如前文所述, c->db->io_keys (即server->db->io_keys)中保存了{key, c}, 即阻塞在该key上的所有客户端。对于db->io_keys中的所有c, 调用dontWaitForSwappedKey()将他们从db->io_keys中删除; 同时, c->io_keys里保存了该客户端需要阻塞读取的key, 也将被换出的key从c->io_keys中删除。最后, 若c已经不再被任何key阻塞, 将它加入server.io_ready_clients, 准备进行“正常”的读写操作。

最后一个问题, 阻塞状态的客户端, 加入server.io_ready_clients回到正常状态后, 在哪里重新监听READABLE事件? 前文分析Redis的事件机制时, 曾提到aeMain()在每次epoll循环前, 会执行beforeSleep()方法, Redis在该方法中检查server.io_ready_clients, 若有ready的客户端, 则重新在它上面监听READABLE事件, 并处理它的命令。

4.2. 数据交换策略

Redis的虚拟内存系统有两种可用的换入换出策略: 阻塞式(Blocking Virtual Memory)和多线程非阻塞式(Threaded Virtual Memory IO), 相关代码都在vm.c中。

Redis在第一次建库时(无论是AOF备份还是Snapshot备份), 如果需要将数据换出到vm, 均使用阻塞式交换; 之后Redis会在serverCron()中与vm交换数据。

```

/* Swap a few keys on disk if we are over the memory limit and VM
 * is enabled. Try to free objects from the free list first. */
if (vmCanSwapOut()) {
    while (server.vm_enabled && zmalloc_used_memory() >
           server.vm_max_memory)
    {
        int retval = (server.vm_max_threads == 0) ?
                     vmSwapOneObjectBlocking() :
                     vmSwapOneObjectThreaded();

        // ...
        if (retval == REDIS_ERR || server.vm_max_threads > 0) break;
    }
}

```


`vmSwapOneObjectBlocking()`是阻塞式交换，`vmSwapOneObjectThreaded()`是多线程交换，最终交换的代码都在`vmSwapOneObject()`中。Redis每次从每个库中随机选择5个数据，用`computeObjectSwappability()`测试它们换出到磁盘的可行性，最后选择1个最需要换出到磁盘的，用阻塞或非阻塞的方式将它换出到磁盘。

多线程模式下，换出的方式和读入类似，也是创建一个类型为`REDIS_IOJOB_PREPARE_SWAP`的job，启动该线程后，按Snapshot备份的编码方式，将需要换出的数据编码，计算其长度，进而得到需要换出的页的数目。然后寻找一块连续的满足要换出的页的数目的区域，创建类型为`REDIS_IOJOB_DO_SWAP`的job，进行换出，并标记该数据的状态。Redis在vm中寻找连续空闲区域的算法比较简单，直接进行简单的枚举：如果当前区域空闲，则判断是否有满足需求连续的空闲区域；如果当前区域有数据，则标记遇到有数据的区域的次数，如果连续`REDIS_VM_MAX_RANDOM_JUMP/4`次遇到非空闲区域，则将指针往前跳跃一个不超过`REDIS_VM_MAX_RANDOM_JUMP`的随机数。

阻塞式换出就比较简单了，只是将多线程模式下的几个步骤阻塞式执行。

5. 备份机制

Redis是内存数据库，支持两种方式——snapshot和aof，将数据从内存dump到磁盘。snapshot是快照式备份，每次将Redis全库dump到磁盘，aof是流水式备份，每次将Redis数据库的修改日志dump到磁盘，并定期整理日志。

5.1. Snapshot

Redis支持Snapshot（快照）式备份。Redis可以自动检测备份时机，设置每N秒检查，若发生M次以上数据更新操作，则开始Snapshot备份，也可以由客户端发送save或bgsave命令手动启动备份。

save是阻塞式备份，备份过程中Redis会停止服务；bgsave是后台备份，Redis将新建一个备份进程负责将全库dump到磁盘。

`saveCommand()`中，首先检查是否有bgsave的后台备份进程，若没有，则执行`rdbSave()`进行全库备份。

`bgsaveCommand()`中，也会检查是否有bgsave的后台备份进程，若没有，则执行`rdbSaveBackground()`，启动备份子进程，在后台全库备份。备份子进程最终也是通过`rdbSave()`备份数据库。

`rdbSaveBackground()`中，首先通过前文所说的`waitEmptyIOJobsQueue()`检查是否有vm的IO线程在进行数据交换，如果有则阻塞，等待所有vm的IO线程完成工作。然后`fork()`备份子进程。在父进程中，`server.bgsavechildpid`记录了备份子进程的pid，并通过`updateDictResizePolicy()`禁止Hash表自动扩容。在子进程中，如果启用了vm，则打开vm swap文件，然后调用`rdbSave()`执行备份。因为之前已经检查了所有vm的IO线程是否完成，因此这里打开vm的swap文件并不会和vm的IO线程冲突。

```
int rdbSaveBackground(char *filename) {
    pid_t childpid;

    if (server.bgsavechildpid != -1) return REDIS_ERR;
    if (server.vm_enabled) waitEmptyIOJobsQueue();
    server.dirty_before_bgsave = server.dirty;
```



```

if ((childpid = fork()) == 0) {
    /* Child */
    if (server.vm_enabled) vmReopenSwapFile();
    if (server.ipfd > 0) close(server.ipfd);
    if (server.sofd > 0) close(server.sofd);
    if (rdbSave(filename) == REDIS_OK) {
        _exit(0);
    } else {
        _exit(1);
    }
} else {
    /* Parent */
    if (childpid == -1) {
        redisLog(REDIS_WARNING,"Can't save in background: fork: %s",
            strerror(errno));
        return REDIS_ERR;
    }
    redisLog(REDIS_NOTICE,"Background saving started by pid %d",childpid);
    server.bgsavechildpid = childpid;
    updateDictResizePolicy();
    return REDIS_OK;
}
return REDIS_OK; /* unreachable */
}

```

`rdbSave()`中首先通过`waitEmptyIOJobsQueue()`检查vm的IO线程，并阻塞直到所有vm的IO线程完成。因为备份过程中，需要备份vm的swap文件，如果有未完成的vm的IO线程，会导致数据不同步。

首先建立一个tmp文件，向它写入备份数据。依次循环每个db: `server.db`, `server.db+1`, ..., `server.db+server.dbnum`, 通过dict的迭代器`dictIterator`访问db中的所有数据，依次写入tmp文件。

Redis在写备份文件时做了许多优化，例如`rdbSaveLen()`用于写入长度信息，写入的数字不超过32bit整数。Redis将长度数字按二进制长度分三类：小于6bit，6bit至14bit，14bit至32bit，并用前缀`REDIS_RDB_6BITLEN(00)`、`REDIS_RDB_14BITLEN(01)`、`REDIS_RDB_32BITLEN(10)`标示数字长度。将前缀与数字一起写入磁盘。这个优化类似于Protobuf里压缩数字的方法。类似的压缩还有很多，再此不一一分析。

```

int rdbSaveLen(FILE *fp, uint32_t len) {
    unsigned char buf[2];
    int nwritten;

    if (len < (1<<6)) {
        /* Save a 6 bit len */
        buf[0] = (len&0xFF)|(REDIS_RDB_6BITLEN<<6);
        if (rdbWriteRaw(fp,buf,1) == -1) return -1;
        nwritten = 1;
    } else if (len < (1<<14)) {
        /* Save a 14 bit len */
        buf[0] = ((len>>8)&0xFF)|(REDIS_RDB_14BITLEN<<6);
        buf[1] = len&0xFF;
        if (rdbWriteRaw(fp,buf,2) == -1) return -1;
        nwritten = 2;
    } else {
        /* Save a 32 bit len */
        buf[0] = (REDIS_RDB_32BITLEN<<6);

```

```

        if (rdbWriteRaw(fp,buf,1) == -1) return -1;
        len = htonl(len);
        if (rdbWriteRaw(fp,&len,4) == -1) return -1;
        nwritten = 1+4;
    }
    return nwritten;
}

```

在备份文件中，Redis写入type/key/value数据，并放弃所有过期的数据。特别地，对于vm中的数据，会先读入内存，再将数据写入备份文件。

父进程在serverCron中检查备份进程，如果存在bgsave进程或者bgrewrite进程（AOF备份，下节分析），则wait3()等待进程完成，完成后调用backgroundSaveDoneHandler()或者backgroundRewriteDoneHandler()处理。

snapshot式备份中，Redis会停止vm的换入换出操作，停止更新LRU策略中的标记信息，但不会影响Redis对内存中的数据的读写。Redis通过调整snapshot备份的粒度，适应各种应用需求。

5.2. AOF

除了Snapshot备份模式外Redis还支持AOF(流水式)备份，它保存所有操作的commit log，并能自动地进行全库备份和删除无用的commit log。

Redis在写commit log时，Redis并不是每次处理请求时都将请求写入磁盘，它会用一段内存Buffer缓存commit log，并在一定时间将缓存中的内容统一写入磁盘。为了避免磁盘缓存的影响，可以设置三种策略进行fsync()：每次写操作后均调用fsync()，每秒调用一次fsync()，从不调用fsync()，三种不同的策略获得不同的性能和一致性。

Redis将所有操作保存在commit log中，commit log的文件是追加写。当commit log的大小无法承受时，可以手工通过bgrewriteaof命令产生一个快照（这个不同于Snapshot备份），具体构造方式后文分析。

AOF有四种操作：

- feedAppendOnlyFile() – 将命令写入AOF。该方法在call()中被调用，用于将所有Redis处理的命令写入AOF；该方法在Redis发现过期的Keys被调用，记录清除过期Key的操作

call()是Redis中所有命令处理的入口，当启用了AOF且数据有变化时(server.dirty有变化)，将命令用feedAppendOnlyFile()写入AOF，该函数中，将命令编码后写入server.aofbuf，当后台rewriteaof进程存在时，同时将编码后的命令写入server.bgrewritebuf，bgrewritebuf的作用后文解释。

```

/* Append to the AOF buffer. This will be flushed on disk just before
 * of re-entering the event loop, so before the client will get a
 * positive reply about the operation performed. */
server.aofbuf = sdscatlen(server.aofbuf,buf,sdslen(buf));

/* If a background append only file rewriting is in progress we want to
 * accumulate the differences between the child DB and the current one
 * in a buffer, so that when the child process will do its work we
 * can append the differences to the new append only file. */
if (server.bgrewritechildpid != -1)
    server.bgrewritebuf = sdscatlen(server.bgrewritebuf,buf,sdslen(buf));

```

- `rewriteAppendOnlyFile()` – 产生快照，并更新aof文件。

`rewriteAppendOnlyFile()`只在 `rewriteAppendOnlyFileBackground()`中被调用。Redis产生AOF快照主要分为以下几步：

- (1) `fork()`一个rewrite子进程，调用`rewriteAppendOnlyFile()`产生快照。
- (2) 当rewrite进程开始后，`rewriteAppendOnlyFile()`扫描数据库中**所有数据**，将他们编码后写入临时文件。AOF文件的编码形式为文本，即人肉可读的编码。
- (3) rewrite时父进程照常接收请求，并将流水日志写入 `server.bgrewriteaofbuf`中。
- (4) 子进程完成工作，父进程在`serverCron()`中通过`wait3()`获知其状态后，调用`backgroundRewriteDoneHandler()`将`server.bgrewriteaofbuf`中的内容作为新的commit log，覆盖原有`server.aofbuf`。
- (5) 父进程将rewrite子进程生成的临时文件改名，作为新的aof文件。用于未来恢复数据。

Redis的AOF快照机制的基础，是数据库的大小一定小于操作日志的大小，否则产生的快照文件可能远远超过commit log文件的大小。因此，此处有一点值得考虑，**rewrite过程是否可以只写入aofbuf中改变了的数据的快照？**

- `flushAppendOnlyFile()` – 将内存buffer中的commit log写到磁盘上。Redis在`beforeSleep()`中，及关闭AOF功能时，将内存buffer中的log写入磁盘。
- `loadAppendOnlyFile()` – 重放AOF文件。该方法在Redis启动时被调用，从AOF文件中重建数据库。

Redis通过AOF重建时，构造一个虚拟的client()，向自己发送重建的命令。而恢复数据只需要将AOF快照重新载入数据库，并回放commit log中的操作。

6. 主从同步

Redis支持主从同步，其官方文档 <http://www.redis.io/topics/replication> 简要介绍了它的主从同步机制。翻译官方文档的文章，Redis的主从同步有以下特点：

- 一个Master可以有多个Slave
- Slave可以接受其它Slave的连接，因此Masters和Slaves之间可以行程一个网络
- 同步时，作为Master的一端（可能其状态是Slave）的同步是非阻塞的，它可以继续处理命令；作为Slave一端的第一次同步是阻塞的。
- 主从机制可以用于提高扩展性，比如可以用多个Slaves处理复杂的读请求。
- 可以通过Slave备份数据，以降低Master的负载。

6.1. 建立连接

接下来我们看Redis中Slave和Master是如何建立连接的。Redis通过`server.replstate`表明Master和Slave的状态，初始时，状态均为

REDIS_REPL_NONE。server.masterhost和server.masterport为Master的地址，Master结点的这两个变量为NULL。

Slave的server.replstate状态除了REDIS_REPL_NONE外，还有

```
/* Slave replication state - slave side */
#define REDIS_REPL_NONE 0 /* No active replication */
#define REDIS_REPL_CONNECT 1 /* Must connect to master */
#define REDIS_REPL_TRANSFER 2 /* Receiving .rdb from master */
#define REDIS_REPL_CONNECTED 3 /* Connected to master */
```

Master对于每个Slave都维护一个状态，在结构体redisClient.replstate中，因为Slave对于Master，其本质也是一个客户端。Master对每个Slave标记状态，有四种：

```
/* Slave replication state - from the point of view of master
 * Note that in SEND_BULK and ONLINE state the slave receives new updates
 * in its output queue. In the WAIT_BGSAVE state instead the server is waiting
 * to start the next background saving in order to send updates to it. */
#define REDIS_REPL_WAIT_BGSAVE_START 3 /* master waits bgsave to start feeding
it */
#define REDIS_REPL_WAIT_BGSAVE_END 4 /* master waits bgsave to start bulk DB
transmission */
#define REDIS_REPL_SEND_BULK 5 /* master is sending the bulk DB */
#define REDIS_REPL_ONLINE 6 /* bulk DB already transmitted, receive updates */
```

REDIS_REPL_WAIT_BGSAVE_START表示Master准备dump数据到磁盘，任何新建的Slave连接均处于该状态；Master开始数据dump后，将Slave的状态修改为REDIS_REPL_WAIT_BGSAVE_END，当dump完成后，通知Slave，状态变为REDIS_REPL_SEND_BULK，准备向其发送数据；当发送完成，Slave就绪后，状态变为REDIS_REPL_ONLINE，至此Master可以向状态为REDIS_REPL_ONLINE的Slave发送更新命令。

对于Slave，如果server.replstate!=REDIS_REPL_CONNECTED，则无法处理命令（获取Slave状态的命令除外）；对于Master，如果c.replstate!=REDIS_REPL_ONLINE，也不会向它发送命令。

serverCron()中每隔10秒会调用replicationCron()，处理主从同步相关的功能。replicationCron()会检测同步数据传输超时，Slave与Master连接断开等问题。在分析可能的故障前，先分析建立连接的过程。

6.1.1. Master

Redis启动时可以通过配置文件指定为Master状态或Slave状态，Slave也可以通过Sync命令与另一个Master（可能是Master，也可能是与Master有连接的Slave）建立主从连接。下面分析syncCommand()的处理。

首先，如果发送sync命令的结点已经是Slave状态，则无需再次同步。如果接受sync指令的结点，本身也是Slave，且没有与任何Master建立连接，则无法响应sync命令。如果Slave与Master之间还有其它交互命令没有处理完，也无法继续响应sync命令。

然后Master开始检测bgsave状态，如果已经有bgsave进程，再判断是否正在响应其它Slave的sync命令，且正在dump数据（Slave的replstate为REDIS_REPL_BGSAVE_END），则直接将新Slave的replstate状态置为REDIS_REPL_BGSAVE_END，并准备将dump好的数据传给新Slave。如果没有dump好的

数据，则将新Slave的状态置为REDIS_REPL_BGSAVE_START，并等待bgsave进程完成，之后再启动bgsave，为Slave dump数据。

如果没有bgsave进程，则启动新的bgsave进程，并将Slave的状态置为REDIS_REPL_BGSAVE_END。

最后，将新Slave加入Master的server.slaves列表。

```
void syncCommand(redisClient *c) {
    /* ignore SYNC if already slave or in monitor mode */
    if (c->flags & REDIS_SLAVE) return;

    /* Refuse SYNC requests if we are a slave but the link with our master
     * is not ok... */
    if (server.masterhost && server.replstate != REDIS_REPL_CONNECTED) {
        addReplyError(c,"Can't SYNC while not connected with my master");
        return;
    }

    /* SYNC can't be issued when the server has pending data to send to
     * the client about already issued commands. We need a fresh reply
     * buffer registering the differences between the BGSAVE and the current
     * dataset, so that we can copy to other slaves if needed. */
    if (listLength(c->reply) != 0) {
        addReplyError(c,"SYNC is invalid with pending input");
        return;
    }

    redisLog(REDIS_NOTICE,"Slave ask for synchronization");
    /* Here we need to check if there is a background saving operation
     * in progress, or if it is required to start one */
    if (server.bgsavechildpid != -1) {
        /* Ok a background save is in progress. Let's check if it is a good
         * one for replication, i.e. if there is another slave that is
         * registering differences since the server forked to save */
        redisClient *slave;
        listNode *ln;
        listIter li;

        listRewind(server.slaves,&li);
        while((ln = listNext(&li))) {
            slave = ln->value;
            if (slave->replstate == REDIS_REPL_WAIT_BGSAVE_END) break;
        }
        if (ln) {
            /* Perfect, the server is already registering differences for
             * another slave. Set the right state, and copy the buffer. */
            listRelease(c->reply);
            c->reply = listDup(slave->reply);
            c->replstate = REDIS_REPL_WAIT_BGSAVE_END;
            redisLog(REDIS_NOTICE,"Waiting for end of BGSAVE for SYNC");
        } else {
            /* No way, we need to wait for the next BGSAVE in order to
             * register differences */
            c->replstate = REDIS_REPL_WAIT_BGSAVE_START;
            redisLog(REDIS_NOTICE,"Waiting for next BGSAVE for SYNC");
        }
    } else {
        /* Ok we don't have a BGSAVE in progress, let's start one */
        redisLog(REDIS_NOTICE,"Starting BGSAVE for SYNC");
    }
}
```

```

        if (rdbSaveBackground(server.dbfilename) != REDIS_OK) {
            redisLog(REDIS_NOTICE,"Replication failed, can't BGSAVE");
            addReplyError(c,"Unable to perform background save");
            return;
        }
        c->replstate = REDIS_REPL_WAIT_BGSAVE_END;
    }
    c->repldbfd = -1;
    c->flags |= REDIS_SLAVE;
    c->slaveseldb = 0;
    listAddNodeTail(server.slaves,c);
    return;
}

```

如前文Snapshot快照分析，当bgsave进程备份完成时，在serverCron()中会调用backgroundSaveDoneHandler()处理，该函数中调用updateSlavesWaitingBgsave()处理等待bgsave的Slave结点。

updateSlavesWaitingBgsave()检查server.slaves列表，当发现状态为REDIS_REPL_WAIT_BGSAVE_START的结点时，会再次启动bgsave进程进行dump数据。当发现状态为REDIS_REPL_WAIT_BGSAVE_START的结点时，会准备将数据发送给Slave，并将Slave的状态修改为REDIS_REPL_SEND_BULK。Master在Slave上监听WRITEABLE事件，当Slave可写时，调用sendBulkToSlave()将数据发送到Slave。

在Master中，sendBulkToSlave()将数据从磁盘读入，通过socket发送到Slave，完成后删除WRITEABLE事件，并将Slave状态改为REDIS_REPL_ONLINE。然后再次监听Slave的WRITEABLE事件，当Slave可写时，表明Slave已经准备好开始同步指令。

6.1.2. Slave

Redis可以在配置文件中指定为Slave模式，并指定Master的地址，这样的Slave结点启动时即为REDIS_REPL_ONLINE状态，然后在replicationCron()中调用syncWithMaster()与Master同步。

syncWithMaster()中，向Master发送sync命令，并准备接收数据的目录。然后通过readSyncBulkPayload()处理Master发送的dump的数据。Slave通过syncRead()和syncWrite()与Master传输数据，这两个方法底层是异步IO的，但封装成阻塞型，因此Slave第一次同步是阻塞的。

replicationCron()会定时Master的dump数据是否发送完成，如果长时间没有收到dump数据的数据包，Slave会通过replicationAbortSyncTransfer()取消数据同步。

6.2. 指令同步

在call()中，Master会将改变了数据的命令通过replicationFeedSlaves()同步到Slaves中。Slave以处理普通命令的流程处理这些Master发来的命令。

特别地在serverCron()中，Master会删除过期的数据，Slave则等待Master同步DEL指令将过期数据删除。

`replicationCron()`中, Master会定时向所有Slave发送心跳指令, 同时, Slave的`replicationCron()`会通过Ping指令检查Slave与Master的连接状态, 如果Slave长时间没有收到主机的Ping, 则会断开与主机的连接。

6.3. 主从转换

`slaveof`可以转换Redis中结点的状态。对于Slave结点, `slaveof no one`可以将Slave与Master断开, 并将Slave转为主机。对于Master, `slaveof ip port`可以将主机转为Slave。

Redis的主从转换非常简单, 如果同步未完成, 则取消同步; 否则, 直接改变`server.masterhost`等状态。

参考文献

- [1] Redis, <http://redis.io>
- [2] Binary-safe, http://en.wikipedia.org/wiki/Binary_safe
- [3] malloc_size(), Mac OS X Manual Page, http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man3/malloc_size.3.html
- [4] BeansDB, <http://code.google.com/p/beansdb/>
- [5] Redis进阶教程-aof(append only file)日志文件, <http://lgone.com/html/y2010/757.html>
- [6] Redis源代码分析, <http://www.petermao.com/category/redis>