redis Documentation

Release 1.0

jacksu

November 09, 2015

Contents

Warning: 本文档主要是redis源码的解析， 在阅读本文档前最好先阅读 redis的设计与实现 。 源码的阅读主要是学习作者的代码风格、 编程技巧、 以及系统原理。 本文档主要就是介绍上面相关的三点， 源码是基于redis3.0。

Note: 源码阅读方法：

1. 自底向上：从耦合关系最小的模块开始读，然后逐渐过度到关系紧密的模块。就好像写程序的测试一样，先从单元测试开始，然后才到功能测试。我在刚开始读 Redis 源码的时候，使用的就是这种方法：先从单独的数据结构模块开始，然后再过渡到高层的功能模块。
2. 从功能入手：通过文件名（模块名）和函数名，快速定位到一个功能的具体实现，然后追踪整个实现的运作流程，从而了解该功能的实现方式。我在读阻塞列表、数据库这种流程和功能都比较复杂，和其他文件耦合也比较多的模块时，使用的就是这样的方法。
3. 自顶向下：从程序的 main() 函数，或者某个特别大的调用者函数为入口，以深度优先或者广度优先的方式阅读它的源码。我在阅读 redis.c/serverCron() 、 redis.c/main() 和 ae.c/aeMain() 这种有明显调用者性质的函数时，使用的就是这样的方法。

CHAPTER 1

第一部分 内部数据结构

作者为了系统的性能， 实现了很多数据结构， 如sds、adlist、dict和skiplist。

1.1 字符串（sds.h/sds.c）

为了方便计算字符串的长度、 以及提高字符串的拼接效率， 作者实现了自己的字符串结构sdshdr， 是二进制安全的， 并在后面自动添加0。

1.1.1 数据结构

```
typedef char *sds;

struct sdshdr {
        // 记录 buf 数组中已使用字节的数量
        // 等于 SDS 所保存字符串的长度
        int len;
        // 记录 buf 数组中未使用字节的数量
        int free;
        // 字节数组，用于保存字符串
        char buf[];
};
```

那么一个sds字符串实际申请的内存为： sizeof(sdshdr)+len+free+1, free新申请空间的时候为0， 拼接字符串的时候free就不为0。

1.1.2 技巧

1. 在函数sdsnewlen中，根据是否需要初始化使用zmalloc和zcalloc两个不同函数。
2. 计算字符串长度的时候，直接使用函数sdslen，不需要调用strlen。

3. 需要扩展free的空间时， 需要调用函数sdsMakeRoomFor， 该函数空间分配策略比较有意思， 如果free>=addlen,直接返回。 否则判断free＋addlen是否小于SDS_MAX_PREALLOC这个宏， 如果小于， 那么这次就分配2*（free＋addlen）的空间， 这样每次多分配一陪的空间； 否则就分配free+addlen+SDS_MAX_PREALLOC的空间。 这样可以控制最大多分配多少的空间， 以至于不要浪费太多空间。例如： `sds old＝sdsnew("test one"); sds new=sdscat(old,"test");` 此时有12的空余空间， 如果再次调用‘‘sdscat(new,”test”)‘‘, 那么就不需要分配空间。

4. 在函数sdscatvprintf中， 空间申请是以16，32，64..这样增长的， 无处不透露提高性能。

5. 在函数sdscmp中， 调用memcmp， 性能要比strcmp好， 而且还是二进制安全的。

6. 在函数sdssplitlen中， 默认分配的数组为5， 然后按照2的倍数进行增长， 这样做法，有点浪费空间，但是加快速度，不要每分割出来一个字符串就要申请空间。 比较的时候把seplen为1分出来， 也是加快字符串比较速度的考虑， 大部分时候应该是seplen为1。

1.2 双向链表(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。

1.2.1 数据结构

```
typedef struct listNode {

        //指向前一个节点
    struct listNode *prev;

    //指向后一个节点
    struct listNode *next;

    //值
    void *value;
} listNode;
```

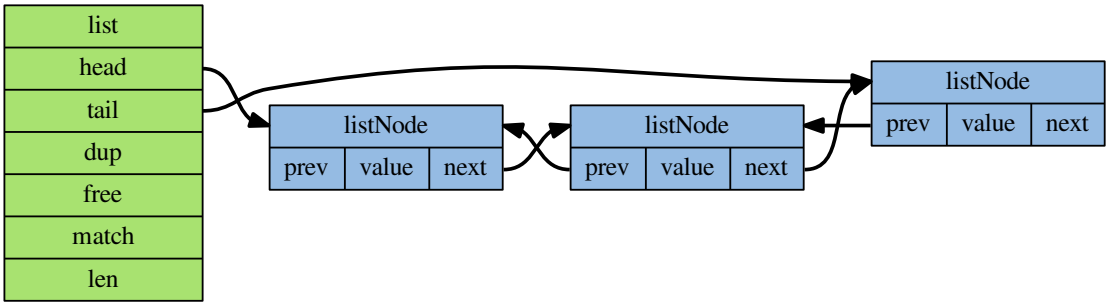
listNode是最基本的结构, 表示链表中的一个结点。 结点有向前(next)和向后 (prev)两个指针, 链表是双向链表。 保存的值是void*类型。

```
typedef struct list {

    listNode *head;

    listNode *tail;

    void *(*dup)(void *ptr);

    void (*free)(void *ptr);

    int (*match)(void *ptr, void *key);

    unsigned long len;
} list;
```

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); 通过无符号长整数len, 标示链表的长度。

```
typedef struct listIter {

    listNode *next;

    int direction;
} listIter;
```

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



1.2.2 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)
{
        dosomething;
}
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。

1.2.3 应用

除了实现列表类型以外， 双端链表还被很多 Redis 内部模块所应用：

- 事务模块使用双端链表依序保存输入的命令；
- 服务器模块使用双端链表来保存多个客户端；
- 订阅/发送模块使用双端链表来保存订阅模式的多个客户端；
- 事件模块使用双端链表来保存时间事件（time event）；

1.3 字典(dict.h/dict.c)

Redis字典具有以下特点：

- Redis字典的底层实现为哈希表，

- 每个字典使用两个哈希表， 一般情况下只使用 0 号哈希表， 只有在 rehash 进行时， 才会同时使用 0 号和 1 号哈希表。
- 哈希表使用链地址法来解决键冲突的问题。
- 自动 Rehash 扩展或收缩哈希表。
- 对哈希表的 rehash 是分多次、渐进式地进行的。

1.3.1 数据结构

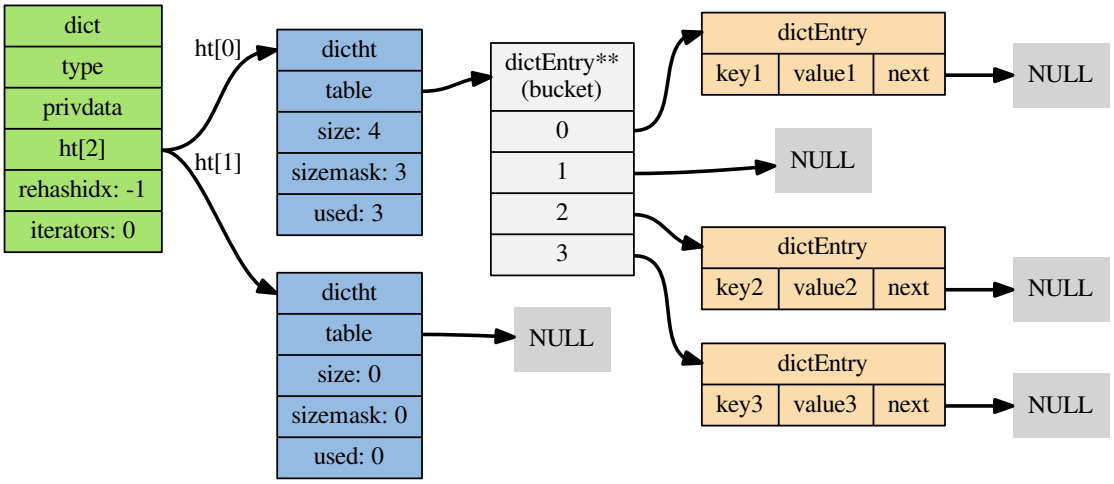
```
/*
 * hash节点
 */

typedef struct dictEntry {

    //键
    void *key;

    //值
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
    } v;

    //指向下一个节点
    struct dictEntry *next;
} dictEntry;
```

```
typedef struct dictht {

    //桶
    dictEntry **table;

    //指针数组大小
    unsigned long size;

    //指针数组掩码，用于计算索引值
    unsigned long sizemask;

    //hash表现有节点数量
    unsigned long used;
} dictht;
```

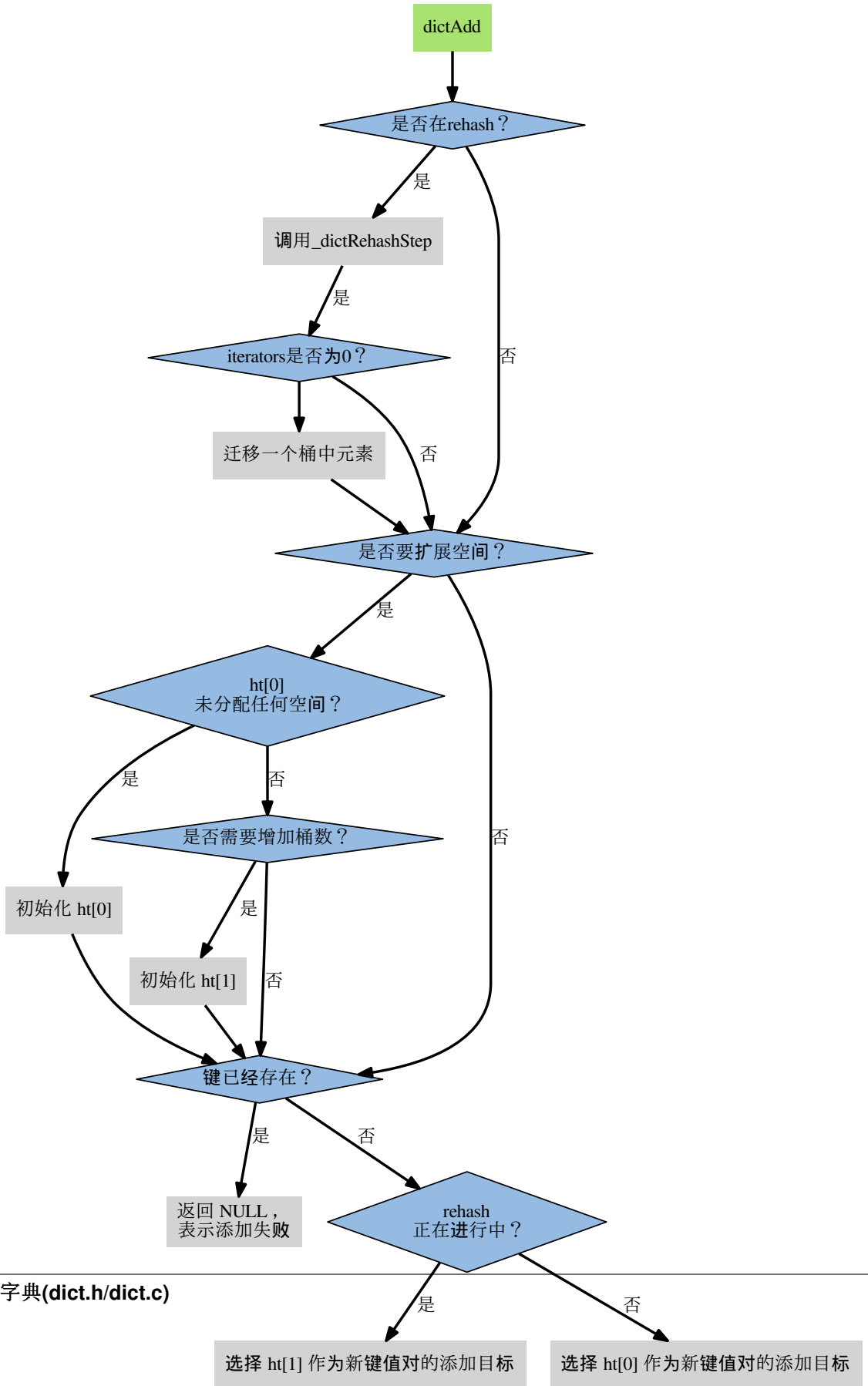
```
typedef struct dict {

    //类型处理函数
    dictType *type;

    //类型处理函数私有值
    void *privdata;

    //两个hash表
    dictht ht[2];

    //rehash标示，为－1表示不在rehash，不为0表示正在rehash的桶
```

```
    int rehashidx; /* rehashing not in progress if rehashidx == -1 */

    //当前正在运行的安全迭代器数量
    int iterators; /* number of iterators currently running */
} dict;
```

dict的结构如下：



1.3.2 字典中添加元素的过程



创建一个比 ht[0]->table 更大的 ht[1]->table ， size为大于used*2的2的指数, 开始值为4;

将 ht[0]->table 中的所有键值对迁移到 ht[1]->table ;

将原有 ht[0] 的数据清空，并将 ht[1] 替换为新的 ht[0] ;

进行rehash的条件:

自然 rehash： ratio >= 1 ，且变量 dict_can_resize 为真。

强制 rehash： ratio 大于变量 dict_force_resize_ratio （目前版本中， dict_force_resize_ratio 的值为 5 ）。

阶进rehash:

主动方式：databaseCron中调用dictRehashMilliseconds执行一毫秒。

被动方式：调用dictAdd，dicFind，dictDelete，dictGetRandomKey时，调用_dictRehashStep，迁移一个非空桶。

CHAPTER 2

第二部分 内存压缩结构

redis 还使用了一些特殊的存储结构， 在条件容许的情况下， 会使用压缩数据结构替代内部数据结构。 创建它们所消耗的内存通常比作用类似的内部数据结构要少得多， 如果使用得当， 压缩数据结构可以为用户节省大量的内存。 压缩数据结构的编码和操作方式要比内部数据结构要复杂得多， 所以所占用的 CPU 时间会比作用类似的内部数据结构要多。

2.1 整数集(intset.h/intset.c)

intset是集合键的底层实现之一， 保存的元素是有序的。

可作为集合键底层实现， 如果一个集合满足以下两个条件：

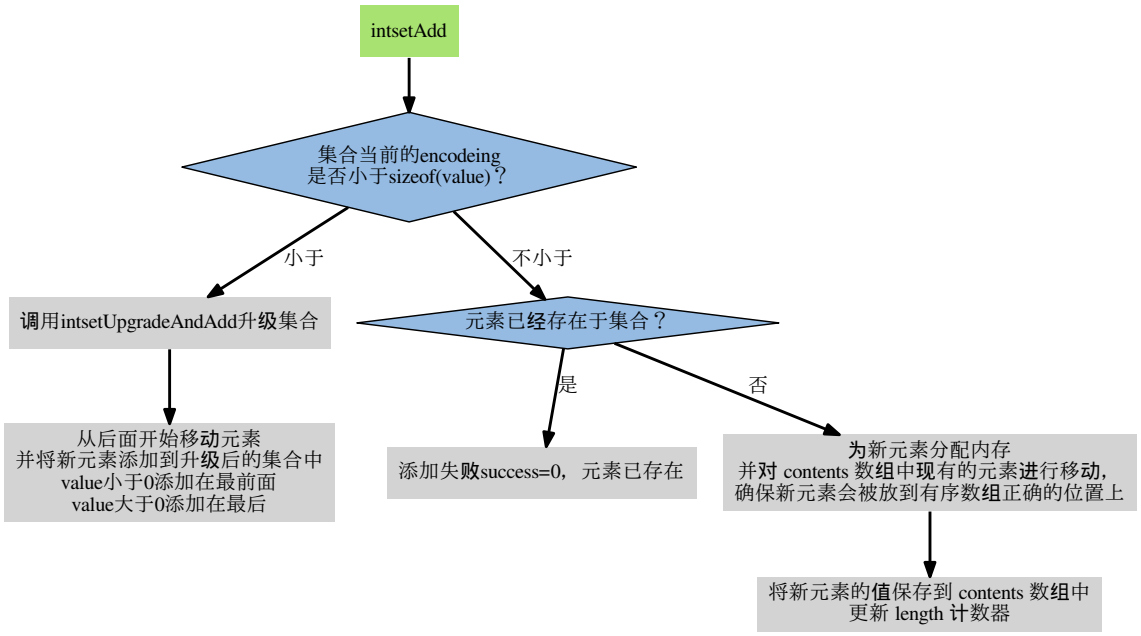
1 保存可转化为long long类型的元素

2 元素数量不多

2.1.1 数据结构

```
typedef struct intset {
    //保存元素所使用类型的长度
    uint32_t encoding;
    //保存元素的个数
    uint32_t length;
    //保存元素的数组
    int8_t contents[];
} intset;
```

2.1.2 添加元素

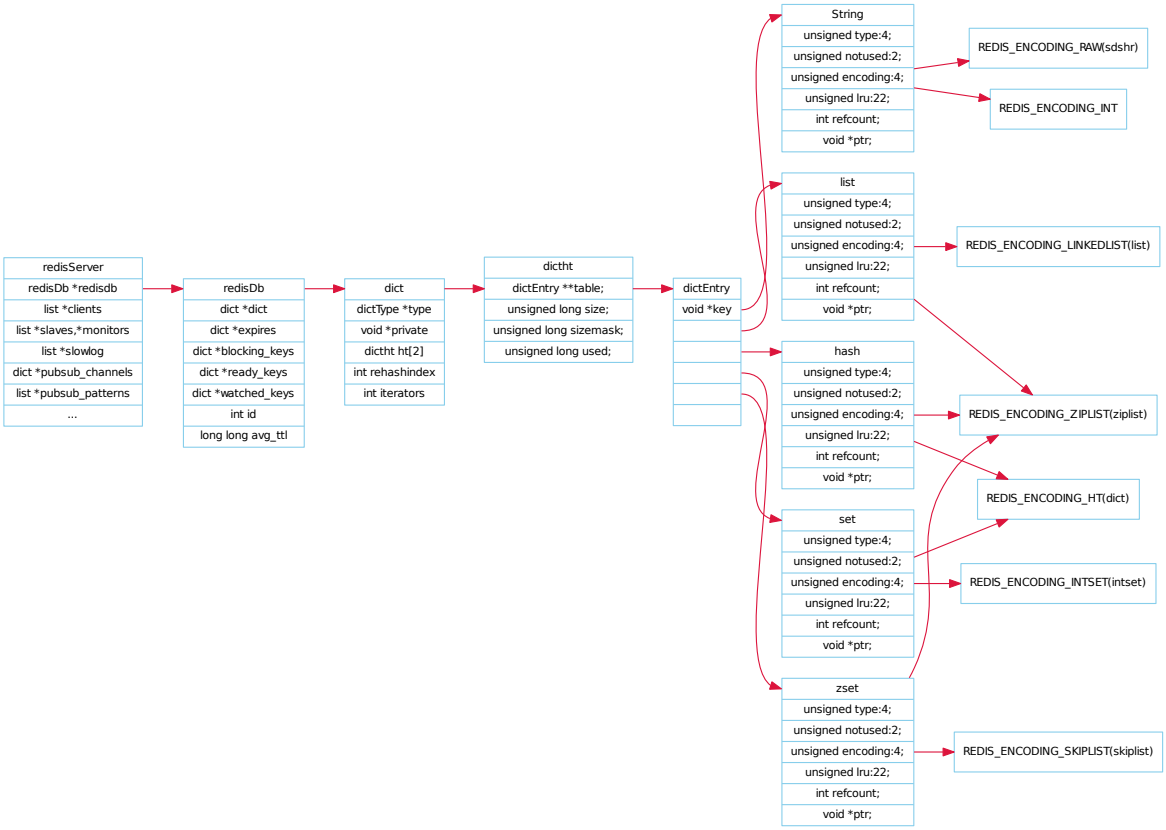


Note: intset初始化的时候是int16_t类型；只支持升级，不支持降级；查找使用二分查找法。

CHAPTER 3

第三部分 内存存储结构

Redis 是支持多key-value数据库(表)的,并用 RedisDb 来表示一个key-value数据库(表). redisServer 中有一个 redisDb *db成员变量, RedisServer 在初始化时,会根据配置文件的 db 数量来创建一个 redisDb 数组. 客户端在连接后,通过 SELECT 指令来选择一个 reidsDb,如果不指定,则缺省是redisDb数组的第1个(即下标是 0) redisDb. 一个客户端在选择 redisDb 后,其后续操作都是在此 redisDb 上进行的. 下面会详细介绍一下 redisDb 的内存结构.



CHAPTER 4

第四部分 初始化

主要介绍redis.h／redis.c/redis-cli.c的内容，主要介绍介绍一些初始化过程，详细的应该在服务器调优的时候才可以体会。

4.1 初始化服务器

从启动 Redis 服务器， 到服务器可以接受外来客户端的网络连接这段时间， Redis 需要执行一系列初始化操作。

整个初始化过程可以分为以下六个步骤:

1. 初始化服务器全局状态。
2. 载入配置文件。
3. 创建 daemon 进程。
4. 初始化服务器功能模块。
5. 载入数据。
6. 开始事件循环。

以下各个小节将介绍 Redis 服务器初始化的各个步骤。

4.1.1 初始化服务器全局状态

`redis.h/redisServer` 结构记录了和服务器相关的所有数据， 这个结构主要包含以下信息:

- 服务器中的所有数据库。
- 命令表：在执行命令时，根据字符来查找相应命令的实现函数。
- 事件状态。
- 服务器的网络连接信息：套接字地址、端口，以及套接字描述符。
- 所有已连接客户端的信息。
- 日志（log）和慢查询日志（slowlog）的选项和相关信息。
- 服务器配置选项：比如要创建多少个数据库，是否将服务器进程作为 daemon 进程来运行，最大连接多少个客户端，压缩结构（zip structure）的实体数量，等等。
- 统计信息：比如键有多少次命令、不命中，服务器的运行时间，内存占用，等等。

- 数据持久化（AOF 和 RDB）的配置和状态。

— slave信息

— master信息

- 实现订阅与发布（pub/sub）功能所需的数据结构。

— 是否运行集群及相关选项。

- Lua 脚本的运行环境及相关选项。

— 调试信息选项

```
/*server对象*/

struct redisServer {
        /* General */

        //配置文件路径
        char *configfile;           /* Absolute config file path, or NULL */

        //serverCron()调用频率
        int hz;                     /* serverCron() calls frequency in hertz */

        //数据库对象
        redisDb *db;

        //支持的命令列表
        dict *commands;             /* Command table */

        //没有转化的命令
        dict *orig_commands;        /* Command table before command renaming. */

        //事件
        aeEventLoop *el;

        //每分钟增加一次
        unsigned lruclock:22;       /* Clock incrementing every minute, for LRU */

        unsigned lruclock_padding:10;

        int shutdown_asap;          /* SHUTDOWN needed ASAP */

        int activerehashing;        /* Incremental rehash in serverCron() */

        //验证密码
        char *requirepass;          /* Pass for AUTH command, or NULL */

        char *pidfile;              /* PID file path */

        int arch_bits;              /* 32 or 64 depending on sizeof(long) */

        int cronloops;              /* Number of times the cron function run */

        char runid[REDIS_RUN_ID_SIZE+1];  /* ID always different at every exec. */

        int sentinel_mode;          /* True if this instance is a Sentinel. */


        /* Networking */
```

```
    int port;                   /* TCP listening port */
    int tcp_backlog;            /* TCP listen() backlog */
    char *bindaddr[REDIS_BINDADDR_MAX]; /* Addresses we should bind to */
    int bindaddr_count;         /* Number of addresses in server.bindaddr[] */
    char *unixsocket;           /* UNIX socket path */
    mode_t unixsocketperm;      /* UNIX socket permission */
    int ipfd[REDIS_BINDADDR_MAX]; /* TCP socket file descriptors */
    int ipfd_count;             /* Used slots in ipfd[] */
    int sofd;                   /* Unix socket file descriptor */
    int cfd[REDIS_BINDADDR_MAX];/* Cluster bus listening socket */
    int cfd_count;              /* Used slots in cfd[] */
    //连接客户端
    list *clients;              /* List of active clients */
    list *clients_to_close;     /* Clients to close asynchronously */
    list *slaves, *monitors;    /* List of slaves and MONITORs */
    redisClient *current_client; /* Current client, only used on crash report */
    int clients_paused;         /* True if clients are currently paused */
    mstime_t clients_pause_end_time; /* Time when we undo clients_paused */
    char neterr[ANET_ERR_LEN];   /* Error buffer for anet.c */
    dict *migrate_cached_sockets;/* MIGRATE cached sockets */


    /* RDB / AOF loading information */
    int loading;                /* We are loading data from disk if true */
    off_t loading_total_bytes;
    off_t loading_loaded_bytes;
    time_t loading_start_time;
    off_t loading_process_events_interval_bytes;
    /* Fast pointers to often looked up command */
    struct redisCommand *delCommand, *multiCommand, *lpushCommand, *lpopCommand,
    *rpopCommand;


    /* Fields used only for stats */
    time_t stat_starttime;          /* Server start time */
    long long stat_numcommands;     /* Number of processed commands */
    long long stat_numconnections;  /* Number of connections received */
    long long stat_expiredkeys;     /* Number of expired keys */
    long long stat_evictedkeys;     /* Number of evicted keys (maxmemory) */
    long long stat_keyspace_hits;   /* Number of successful lookups of keys */
    long long stat_keyspace_misses; /* Number of failed lookups of keys */
    size_t stat_peak_memory;        /* Max used memory record */
    long long stat_fork_time;       /* Time needed to perform latest fork() */
    long long stat_rejected_conn;   /* Clients rejected because of maxclients */
    long long stat_sync_full;       /* Number of full resyncs with slaves. */
    long long stat_sync_partial_ok; /* Number of accepted PSYNC requests. */
    long long stat_sync_partial_err;/* Number of unaccepted PSYNC requests. */

    //保存慢日志命令
    list *slowlog;                  /* SLOWLOG list of commands */
    long long slowlog_entry_id;     /* SLOWLOG current entry ID */
    long long slowlog_log_slower_than; /* SLOWLOG time limit (to get logged) */
    unsigned long slowlog_max_len;     /* SLOWLOG max number of items logged */
    /* The following two are used to track instantaneous "load" in terms
    * of operations per second. */
    long long ops_sec_last_sample_time; /* Timestamp of last sample (in ms) */
    long long ops_sec_last_sample_ops;  /* numcommands in last sample */
    long long ops_sec_samples[REDIS_OPS_SEC_SAMPLES];
```

```
    int ops_sec_idx;


    /* Configuration */
    int verbosity;                  /* Loglevel in redis.conf */
    int maxidletime;                /* Client timeout in seconds */
    int tcpkeepalive;               /* Set SO_KEEPALIVE if non-zero. */
    int active_expire_enabled;      /* Can be disabled for testing purposes. */
    size_t client_max_querybuf_len; /* Limit for client query buffer length */
    int dbnum;                      /* Total number of configured DBs */
    int daemonize;                  /* True if running as a daemon */
    clientBufferLimitsConfig client_obuf_limits[REDIS_CLIENT_LIMIT_NUM_CLASSES];


    /* AOF persistence */
    int aof_state;                  /* REDIS_AOF_(ON|OFF|WAIT_REWRITE) */
    int aof_fsync;                  /* Kind of fsync() policy */
    char *aof_filename;             /* Name of the AOF file */
    int aof_no_fsync_on_rewrite;    /* Don't fsync if a rewrite is in prog. */
    int aof_rewrite_perc;           /* Rewrite AOF if % growth is > M and... */
    off_t aof_rewrite_min_size;     /* the AOF file is at least N bytes. */
    off_t aof_rewrite_base_size;    /* AOF size on latest startup or rewrite. */
    off_t aof_current_size;         /* AOF current size. */
    int aof_rewrite_scheduled;      /* Rewrite once BGSAVE terminates. */
    pid_t aof_child_pid;            /* PID if rewriting process */
    list *aof_rewrite_buf_blocks;   /* Hold changes during an AOF rewrite. */
    sds aof_buf;      /* AOF buffer, written before entering the event loop */
    int aof_fd;       /* File descriptor of currently selected AOF file */
    int aof_selected_db; /* Currently selected DB in AOF */
    time_t aof_flush_postponed_start; /* UNIX time of postponed AOF flush */
    time_t aof_last_fsync;            /* UNIX time of last fsync() */
    time_t aof_rewrite_time_last;   /* Time used by last AOF rewrite run. */
    time_t aof_rewrite_time_start;  /* Current AOF rewrite start time. */
    int aof_lastbgrewrite_status;   /* REDIS_OK or REDIS_ERR */
    unsigned long aof_delayed_fsync;  /* delayed AOF fsync() counter */
    int aof_rewrite_incremental_fsync;/* fsync incrementally while rewriting? */
    int aof_last_write_status;      /* REDIS_OK or REDIS_ERR */
    int aof_last_write_errno;       /* Valid if aof_last_write_status is ERR */


    /* RDB persistence */
    long long dirty;                /* Changes to DB from the last save */
    long long dirty_before_bgsave;  /* Used to restore dirty on failed BGSAVE */
    pid_t rdb_child_pid;            /* PID of RDB saving child */
    struct saveparam *saveparams;   /* Save points array for RDB */
    int saveparamslen;              /* Number of saving points */
    char *rdb_filename;             /* Name of RDB file */
    int rdb_compression;            /* Use compression in RDB? */
    int rdb_checksum;               /* Use RDB checksum? */
    time_t lastsave;                /* Unix time of last successful save */
    time_t lastbgsave_try;          /* Unix time of last attempted bgsave */
    time_t rdb_save_time_last;      /* Time used by last RDB save run. */
    time_t rdb_save_time_start;     /* Current RDB save start time. */
    int lastbgsave_status;          /* REDIS_OK or REDIS_ERR */
    int stop_writes_on_bgsave_err;  /* Don't allow writes if can't BGSAVE */
    /* Propagation of commands in AOF / replication */
    redisOpArray also_propagate;    /* Additional command to propagate. */
```

```
    /* Logging */
    char *logfile;                  /* Path of log file */
    int syslog_enabled;             /* Is syslog enabled? */
    char *syslog_ident;             /* Syslog ident */
    int syslog_facility;            /* Syslog facility */


    /* Replication (master) */
    int slaveseldb;                 /* Last SELECTed DB in replication output */
    long long master_repl_offset;   /* Global replication offset */
    int repl_ping_slave_period;     /* Master pings the slave every N seconds */
    char *repl_backlog;             /* Replication backlog for partial syncs */
    long long repl_backlog_size;    /* Backlog circular buffer size */
    long long repl_backlog_histlen; /* Backlog actual data length */
    long long repl_backlog_idx;     /* Backlog circular buffer current offset */
    long long repl_backlog_off;     /* Replication offset of first byte in the
    backlog buffer. */
    time_t repl_backlog_time_limit; /* Time without slaves after the backlog
    gets released. */
    time_t repl_no_slaves_since;    /* We have no slaves since that time.
    Only valid if server.slaves len is 0. */
    int repl_min_slaves_to_write;   /* Min number of slaves to write. */
    int repl_min_slaves_max_lag;    /* Max lag of <count> slaves to write. */
    int repl_good_slaves_count;     /* Number of slaves with lag <= max_lag. */


    /* Replication (slave) */
    char *masterauth;               /* AUTH with this password with master */
    char *masterhost;               /* Hostname of master */
    int masterport;                 /* Port of master */
    int repl_timeout;               /* Timeout after N seconds of master idle */
    redisClient *master;     /* Client that is master for this slave */
    redisClient *cached_master; /* Cached master to be reused for PSYNC. */
    int repl_syncio_timeout; /* Timeout for synchronous I/O calls */
    int repl_state;          /* Replication status if the instance is a slave */
    off_t repl_transfer_size; /* Size of RDB to read from master during sync. */
    off_t repl_transfer_read; /* Amount of RDB read from master during sync. */
    off_t repl_transfer_last_fsync_off; /* Offset when we fsync-ed last time. */
    int repl_transfer_s;     /* Slave -> Master SYNC socket */
    int repl_transfer_fd;    /* Slave -> Master SYNC temp file descriptor */
    char *repl_transfer_tmpfile; /* Slave-> master SYNC temp file name */
    time_t repl_transfer_lastio; /* Unix time of the latest read, for timeout */
    int repl_serve_stale_data; /* Serve stale data when link is down? */
    int repl_slave_ro;          /* Slave is read only? */
    time_t repl_down_since; /* Unix time at which link with master went down */
    int repl_disable_tcp_nodelay;   /* Disable TCP_NODELAY after SYNC? */
    int slave_priority;             /* Reported in INFO and used by Sentinel. */
    char repl_master_runid[REDIS_RUN_ID_SIZE+1];  /* Master run id for PSYNC. */
    long long repl_master_initial_offset;         /* Master PSYNC offset. */


    /* Replication script cache. */
    dict *repl_scriptcache_dict;        /* SHA1 all slaves are aware of. */
    list *repl_scriptcache_fifo;        /* First in, first out LRU eviction. */
    int repl_scriptcache_size;          /* Max number of elements. */
    /* Synchronous replication. */
    list *clients_waiting_acks;         /* Clients waiting in WAIT command. */
    int get_ack_from_slaves;            /* If true we send REPLCONF GETACK. */
```

```
    /* Limits */
    unsigned int maxclients;        /* Max number of simultaneous clients */
    unsigned long long maxmemory;   /* Max number of memory bytes to use */
    int maxmemory_policy;           /* Policy for key eviction */
    int maxmemory_samples;          /* Pricision of random sampling */


    /* Blocked clients */
    unsigned int bpop_blocked_clients; /* Number of clients blocked by lists */
    list *unblocked_clients; /* list of clients to unblock before next loop */
    list *ready_keys;        /* List of readyList structures for BLPOP & co */
    /* Sort parameters - qsort_r() is only available under BSD so we
    * have to take this state global, in order to pass it to sortCompare() */
    int sort_desc;
    int sort_alpha;
    int sort_bypattern;
    int sort_store;


    /* Zip structure config, see redis.conf for more information  */
    size_t hash_max_ziplist_entries;
    size_t hash_max_ziplist_value;
    size_t list_max_ziplist_entries;
    size_t list_max_ziplist_value;
    size_t set_max_intset_entries;
    size_t zset_max_ziplist_entries;
    size_t zset_max_ziplist_value;
    time_t unixtime;        /* Unix time sampled every cron cycle. */
    long long mstime;       /* Like 'unixtime' but with milliseconds resolution. */


    /* Pubsub */
    dict *pubsub_channels;  /* Map channels to list of subscribed clients */
    list *pubsub_patterns;  /* A list of pubsub_patterns */
    int notify_keyspace_events; /* Events to propagate via Pub/Sub. This is an
    xor of REDIS_NOTIFY... flags. */


    /* Cluster */
    int cluster_enabled;      /* Is cluster enabled? */
    mstime_t cluster_node_timeout; /* Cluster node timeout. */
    char *cluster_configfile; /* Cluster auto-generated config file name. */
    struct clusterState *cluster;  /* State of the cluster */
    int cluster_migration_barrier; /* Cluster replicas migration barrier. */


    /* Scripting */
    lua_State *lua; /* The Lua interpreter. We use just one for all clients */
    redisClient *lua_client;   /* The "fake client" to query Redis from Lua */
    redisClient *lua_caller;   /* The client running EVAL right now, or NULL */
    dict *lua_scripts;         /* A dictionary of SHA1 -> Lua scripts */
    mstime_t lua_time_limit;  /* Script timeout in milliseconds */
    mstime_t lua_time_start;  /* Start time of script, milliseconds time */
    int lua_write_dirty;  /* True if a write command was called during the
    execution of the current script. */
    int lua_random_dirty; /* True if a random command was called during the
    execution of the current script. */
```

```
        int lua_timedout;     /* True if we reached the time limit for script
        execution. */
        int lua_kill;         /* Kill the script if true. */


        /* Assert & bug reporting */
        char *assert_failed;
        char *assert_file;
        int assert_line;
        int bug_report_start; /* True if bug report header was already logged. */
        int watchdog_period;  /* Software watchdog period in ms. 0 = off */
};
```

程序创建一个的 `redisServer` 结构的实例变量 `server`， 调用函数 `initServerConfig()`，将 `server` 的各个属性初始化为默认值。

当 `server` 变量的初始化完成之后， 程序进入服务器初始化的下一步： 读入配置文件。

4.1.2 读入配置文件

在初始化服务器的上一步中， 程序为 `server` 变量（也即是服务器状态）的各个属性设置了默认值， 但这些默认值有时候并不是最合适的：

- 用户可能想使用 AOF 持久化，而不是默认的 RDB 持久化。
- 用户可能想用其他端口来运行 Redis ，以避免端口冲突。
- 用户可能不想使用默认的 16 个数据库，而是分配更多或更少数量的数据库。
- 用户可能想对默认的内存限制措施和回收策略做调整。

等等。

为了让使用者按自己的要求配置服务器， Redis 允许用户在运行服务器时， 提供相应的配置文件（config file）或者显式的选项（options）， Redis 在初始化完 `server` 变量之后， 会读入配置文件和选项， 然后根据这些配置来对 `server` 变量的属性值做相应的修改：

1. 如果单纯执行 `redis-server` 命令，那么服务器以默认的配置来运行 Redis 。
2. 另一方面， 如果给 Redis 服务器送入一个配置文件， 那么 Redis 将按配置文件的设置来更新服务器的状态。

   比如说， 通过命令 `redis-server /etc/my-redis.conf` ， Redis 会根据 `my-redis.conf` 文件的内容来对服务器状态做相应的修改。
3. 除此之外， 还可以显式地给服务器传入选项， 直接修改服务器配置。

   举个例子， 通过命令 `redis-server --port 10086` ， 可以让 Redis 服务器端口变更为 `10086` 。
4. 当然， 同时使用配置文件和显式选项也是可以的， 如果文件和选项有冲突的地方， 那么优先使用选项所指定的配置值。

   举个例子， 如果运行命令 `redis-server /etc/my-redis.conf --port 10086` ， 并且 `my-redis.conf` 也指定了 `port` 选项， 那么服务器将优先使用 `--port 10086` （实际上是选项指定的值覆盖了配置文件中的值）。

其实在读入配置文件前， 还要判断是不是sentinel， 如果sentinel， 还需要通过initSentinelConfig()和initSentinel()初始化， 才通过resetServerSaveParams()重置param选项， 通过loadServerConfig(configfile,options)读入配置文件和显选项。

4.1.3 创建 daemon 进程

Redis 默认不以 daemon 进程的方式运行。

若服务器初始化进行到这一步时， 程序将创建 daemon 进程来运行 Redis ， 并创建相应的 pid 文件。

4.1.4 初始化服务器功能模块

在这一步， 初始化程序完成两件事：

- 为 `server` 变量的数据结构子属性分配内存。
- 初始化这些数据结构。

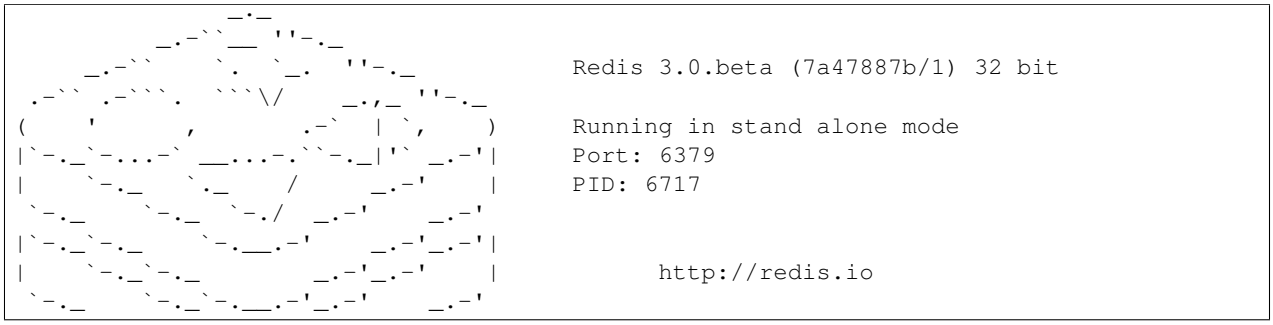
为数据结构分配内存， 并初始化这些数据结构， 等同于对相应的功能进行初始化。

比如说， 当为订阅与发布所需的链表分配内存之后， 订阅与发布功能就处于就绪状态， 随时可以为 Redis 所用了。

在这一步， initServer()完成的主要动作如下：

- 初始化 Redis 进程的信号功能。
- 初始化日志功能。
- 初始化客户端功能。
- 初始化共享对象。
- 初始化事件功能。
- 初始化网络连接。
- 初始化数据库。
- 初始化订阅与发布功能。
- 初始化各个统计变量。
- 关联服务器常规操作（cron job）到时间事件，关联客户端应答处理器到文件事件。
- 如果 AOF 功能已打开，那么打开或创建 AOF 文件。
- 设置内存限制。
- 初始化 Lua 脚本环境。
- 初始化慢查询功能。
- 初始化后台操作线程。

完成这一步之后， 服务器redisAsciiArt()打印出 Redis 的 ASCII LOGO 、服务器版本等信息， 表示所有功能模块已经就绪， 可以等待被使用了：



```
|`-._`-._     `-.__.-'     _.-'_.-'|
|    `-._`-._         _.-'_.-'     |
 `-._     `-._`-.__.-'_.-'     _.-'
     `-._     `-.__.-'     _.-'
         `-._         _.-'
             `-.__.-'
```

虽然所有功能已经就绪， 但这时服务器的数据库还是一片空白， 程序还需要将服务器上一次执行时记录的数据载入到当前服务器中， 服务器的初始化才算真正完成。

4.1.5 载入数据

在这一步， 如果不为sentinel， 程序需要将持久化在 RDB 或者 AOF 文件里的数据， 载入到服务器进程里面。

如果服务器有启用 AOF 功能的话， 那么使用 AOF 文件来还原数据； 否则， 程序使用 RDB 文件来还原数据。

当执行完这一步时， 服务器打印出一段载入完成信息：

```
[6717] 22 Feb 11:59:14.830 * DB loaded from disk: 0.068 seconds
```

Note: 如果是集群， 还要检查数据的一致性。

4.1.6 开始事件循环

到了这一步， 服务器的初始化已经完成， 程序打开事件循环， 开始接受客户端连接。

以下是服务器在这一步打印的信息：

```
[6717] 22 Feb 11:59:14.830 * The server is now ready to accept connections on port 6379
```

4.2 初始化客户端

客户端使用了linenoise库， linenoise比较简单， 不需要任何配置，支持单行、多行模式，history命令查询，自动补全等。 help.h是当前所有的命令文件汇总， 用于tab自动补全功能的源数据。

客户端初始化主要通过一下几步：

1.初始化客户端默认状态

2.查看是否终端输出

3.初始化help

4.根据参数初始化变量

5.判断以那种方式工作

4.2.1 初始化客户端默认状态

4.2.2 查看是否终端输出

!isatty(fileno(stdout)) && (getenv("FAKETTY") == NULL)判断是否终端输出， 实现了如下功能:

```
$ redis-cli exists akey
(integer) 0
$ echo $(redis-cli exists akey)
0
```

后面一个命令的输出中 (integer) 去哪里了?

看了看 redis-cli 帮助中有个 –raw 选项，可以控制输出格式:

```
$ redis-cli --raw exists akey
0
```

4.2.3 初始化help

调用cliInitHelp()， 初始化help命令， 包括group、command命令。

4.2.4 根据参数初始化变量

调用parseOptions(argc,argv)， 修改config的默认值。

4.2.5 判断以那种方式工作

client有九种运行模式： latency、slave、RDB、Pipe、find big keys、stat、scan、交互、eval模式， 根据不同设置， 初始化不同的运行模式， 默认是交互模式。