Redis 源码学习

以下学习基于 redis 2.4.7 的 stable 版本，具体 git 提交版本不清楚。本学习文档使用的源码要的可以发邮件向我索要。

wolf711988@163.com

框架

概述

Redis 通过定义一个 struct redisServer 类型的全局变量 server 来保存服务器的相关信息（比如：配置信息，统计信息，服务器状态等等）。启动时通过读取配置文件里边的信息对 server 进行初始化（如果没有指定配置文件，将使用默认值对 sever 进行初始化），初始化的内容有：起监听端口，绑定有新连接时的回调函数，绑定服务器的定时函数，虚拟内存初始化，log 初始化等等。

启动

初始化服务器配置

先来看看 redis 的 main 函数的入口

Redis.c：1694

```
int main(int argc, char **argv) {
   time_t start;

   initServerConfig();
   if (argc == 2) {
      if (strcmp(argv[1], "-v") == 0 ||
          strcmp(argv[1], "--version") == 0) version();
      if (strcmp(argv[1], "--help") == 0) usage();
      resetServerSaveParams();
      loadServerConfig(argv[1]);
   } else if ((argc > 2)) {
      usage();
   } else {
      ...
   }
   if (server.daemonize) daemonize();
   initServer();
```

```
    ...
```

[3]initServerConfig 初始化全局变量 server 的属性为默认值。

[4,9]如果命令行指定了配置文件， resetServerSaveParams 重置对落地备份的配置（即重置为默认值）并读取配置文件的内容对全局变量 server 再进行初始化 ，没有在配置文件中配置的将使用默认值。

[15]如果服务器配置成后台执行，则对服务器进行 daemonize。

[16]initServer 初始化服务器，主要是设置信号处理函数，初始化事件轮询，起监听端口，绑定有新连接时的回调函数，绑定服务器的定时函数，初始化虚拟内存和 log 等等。

创建服务器监听端口

Redis.c：923

```
    if (server.port != 0) {
        server.ipfd= anetTcpServer(server.neterr,server.port,server.bindaddr);
        if (server.ipfd == ANET_ERR) {
            redisLog(REDIS_WARNING, "Opening port %d: %s",
                server.port, server.neterr);
            exit(1);
        }
    }
```

[2]anetTcpServer 创建一个 socket 并进行监听，然后把返回的 socket fd 赋值给 server.ipfd

事件轮询结构体定义

先看看事件轮询的结构体定义

Ae.h：88

```
2.  /* State of an event based program */
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;
```

[3]maxfd 是最大的文件描述符，主要用来判断是否有文件事件需要处理(ae.c:293)和当使用 select 来处理网络 IO 时作为 select 的参数(ae_select.c:50)。

[4]timeEventNextId 是下一个定时事件的 ID

[5]events[AE_SETSIZE]用于保存通过 aeCreateFileEvent 函数创建的文件事件，在 sendReplyToClient 函数和 freeClient 函数中通过调用

aeDeleteFileEvent 函数删除已经处理完的事件

[6]fired[AE_SETSIZE]用于保存已经触发的文件事件，在对应的网络 I/O 函数中进行赋值（epoll,select,kqueue），不会对 fired 进行删除操作，只会一直覆盖原来的值。然后在 aeProcessEvents 函数中对已经触发的事件进行处理。

[7]timeEventHead 是定时事件链表的头，定时事件的存储用链表实现。

[8]Stop 用于停止事件轮询处理

[9]apidata 用于保存轮询 api 需要的数据，即 aeApiState 结构体，对于 epoll 来说，aeApiState 结构体的定义如下：

```
typedef struct aeApiState {
    int epfd;
    struct epoll_event events[AE_SETSIZE];
} aeApiState;
```

[10]beforesleep 是每次进入处理事件时执行的函数

创建事件轮询

Redis.c：920

```
server.el = aeCreateEventLoop();
```

Ae.c：55

```
aeEventLoop *aeCreateEventLoop(void) {
    aeEventLoop *eventLoop;
    int i;

    eventLoop = zmalloc(sizeof(*eventLoop));
    if (!eventLoop) return NULL;
    eventLoop->timeEventHead = NULL;
    eventLoop->timeEventNextId = 0;
    eventLoop->stop = 0;
    eventLoop->maxfd = -1;
    eventLoop->beforesleep = NULL;
    if (aeApiCreate(eventLoop) == -1) {
        zfree(eventLoop);
        return NULL;
    }
    /* Events with mask == AE_NONE are not set. So let's initialize
     * the vector with it. */
    for (i = 0; i < AE_SETSIZE; i++)
        eventLoop->events[i].mask = AE_NONE;
    return eventLoop;
}
```

绑定定时函数和有新连接时的回调函数

redis.c：973

```
aeCreateTimeEvent(server.el, 1, serverCron, NULL, NULL);
if (server.ipfd > 0 &&
    aeCreateFileEvent(server.el,server.ipfd,AE_READABLE,
    acceptTcpHandler,NULL) == AE_ERR) oom("creating file event");
```

aeCreateTimeEvent 创建定时事件并绑定回调函数 serverCron，这个定时事件第一次是超过 1 毫秒就有权限执行，如果其他事件的处理时间比较长，可能会出现超过一定时间都没执行情况。这里的 1 毫秒只是超过后有可执行的权限，并不是一定会执行。第一次执行后，如果还要执行，是由定时函数的返回值确定的，在 processTimeEvents（ae.c：219）中，当调用定时回调函数后，获取定时回调函数的返回值，如果返回值不等于-1，则设置定时回调函数的下一次触发时间为当前时间加上定时回调函数的返回值，即调用间隔时间。serverCron 的返回值是 100ms，表明从二次开始，每超过 100ms 就有权限执行。（定时回调函数 serverCron 用于更新 lru 时钟，更新服务器的状态，打印一些服务器信息，符合条件的情况下对 hash 表进行重哈希，启动后端写 AOF 或者检查后端写 AOF 或者备份是否完成，检查过期的 KEY 等等）

aeCreateFileEvent 创建监听端口的 socket fd 的文件读事件（即注册网络 io 事件）并绑定回调函数 acceptTcpHandler

进入事件轮询

初始化后将进入事件轮询

Redis.c：1733

```
    aeSetBeforeSleepProc(server.el,beforeSleep);
    aeMain(server.el);
    aeDeleteEventLoop(server.el);
```

设置每次进入事件处理前会执行的函数 beforeSleep

进入事件轮询 aeMain

退出事件轮询后删除事件轮询，释放事件轮询占用内存 aeDeleteEventLoop（不过没在代码中发现有执行到这一步的可能，服务器接到 shutdown 命令时通过一些处理后直接就通过 exit 退出了，可能是我看错了，待验证）

事件轮询函数 aeMain

看看 aeMain 的内容

Ae.c：382

```
void aeMain(aeEventLoop *eventLoop) {
    eventLoop->stop = 0;
    while (!eventLoop->stop) {
```

```
        if (eventLoop->beforesleep != NULL)
            eventLoop->beforesleep(eventLoop);
        aeProcessEvents(eventLoop, AE_ALL_EVENTS);
    }
}
```

每次进入事件处理前，都会调用设置的 beforesleep，beforeSleep 函数主要是处理被阻塞的命令和根据配置写 AOF。

aeProcessEvents 处理定时事件和网络 io 事件

启动完毕，等待客户端请求

到进入事件轮询函数后，redis 的启动工作就做完了，接下来就是等待客户端的请求了。

接收请求

新连接到来时的回调函数

在绑定定时函数和有新连接时的回调函数中说到了绑定有新连接来时的回调函数 acceptTcpHandler，现在来看看这个函数的具体内容

Networking.c：427

```
void acceptTcpHandler(aeEventLoop *el, int fd, void *privdata, int mask) {
    int cport, cfd;
    char cip[128];
    REDIS_NOTUSED(el);
    REDIS_NOTUSED(mask);
    REDIS_NOTUSED(privdata);

    cfd = anetTcpAccept(server.neterr, fd, cip, &cport);
    if (cfd == AE_ERR) {
        redisLog(REDIS_WARNING,"Accepting client connection: %s",
server.neterr);
        return;
    }
    redisLog(REDIS_VERBOSE,"Accepted %s:%d", cip, cport);
    acceptCommonHandler(cfd);
}
```

anetTcpAccept 函数 accept 新连接，返回的 cfd 是新连接的 socket fd.

acceptCommonHandler 函数是对新建立的连接进行处理，这个函数在使用 unix socket 时也会被用到。

接收客户端的新连接

接下来看看 anetTcpAccept 函数的具体内容

Anet.c：330

```
int anetTcpAccept(char *err, int s, char *ip, int *port) {
    int fd;
    struct sockaddr_in sa;
    socklen_t salen = sizeof(sa);
    if ((fd = anetGenericAccept(err,s,(struct sockaddr*)&sa,&salen)) == ANET_ERR)
        return ANET_ERR;

    if (ip) strcpy(ip,inet_ntoa(sa.sin_addr));
    if (port) *port = ntohs(sa.sin_port);
    return fd;
}
```

再进去 anetGenericAccept 看看

Anet.c：313

```
static int anetGenericAccept(char *err, int s, struct sockaddr *sa, socklen_t
*len) {
    int fd;
    while(1) {
        fd = accept(s,sa,len);
        if (fd == -1) {
            if (errno == EINTR)
                continue;
            else {
                anetSetError(err, "accept: %s", strerror(errno));
                return ANET_ERR;
            }
        }
        break;
    }
    return fd;
}
```

anetTcpAccept 函数中调用 anetGenericAccept 函数进行接收新连接，anetGenericAccept 函数在 unix socket 的新连接处理中也会用到

anetTcpAccept 函数接收新连接后，获取客户端得 ip，port 并返回

创建 redisClient 进行接收处理

anetTcpAccept 运行完后，返回新连接的 socket fd，然后返回到调用函数 acceptTcpHandler 中，继续执行 acceptCommonHandler 函数

Networking.c: 403

```
static void acceptCommonHandler(int fd) {
   redisClient *c;
   if ((c = createClient(fd)) == NULL) {
      redisLog(REDIS_WARNING,"Error allocating resoures for the client");
      close(fd); /* May be already closed, just ingore errors */
      return;
   }
   /* If maxclient directive is set and this is one client more... close the
    * connection. Note that we create the client instead to check before
    * for this condition, since now the socket is already set in nonblocking
    * mode and we can send an error for free using the Kernel I/O */
   if (server.maxclients && listLength(server.clients) > server.maxclients) {
      char *err = "-ERR max number of clients reached\r\n";

      /* That's a best effort error message, don't check write errors */
      if (write(c->fd,err,strlen(err)) == -1) {
         /* Nothing to do, Just to avoid the warning... */
      }
      freeClient(c);
      return;
   }
   server.stat_numconnections++;
}
```

创建一个 redisClient 来处理新连接，每个连接都会创建一个 redisClient 来处理。
如果配置了最大并发客户端，则对现有的连接数进行检查和处理
最后统计连接数

绑定有数据可读时的回调函数

Networking.c: 15

```
redisClient *createClient(int fd) {
   redisClient *c = zmalloc(sizeof(redisClient));
   c->bufpos = 0;

   anetNonBlock(NULL,fd);
   anetTcpNoDelay(NULL,fd);
   if (aeCreateFileEvent(server.el,fd,AE_READABLE,
      readQueryFromClient, c) == AE_ERR)
   {
      close(fd);
      zfree(c);
      return NULL;
   }
```

```
    selectDb(c,0);
    c->fd = fd;
    c->querybuf = sdsempty();
     c->reqtype = 0;
     ...
}
```

创建新连接的 socket fd 对应的文件读事件，绑定回调函数 readQueryFromClient

如果创建成功，则对 redisClient 进行一系列的初始化，因为 redisClient 是通用的，即不管是什么命令的请求，都是通过创建一个 redisClient 来处理的，所以会有比较多的字段需要初始化。

createClient 函数执行完后返回到调用处 acceptCommonHandler 函数，然后从 acceptCommonHandler 函数再返回到 acceptTcpHandler 函数。

接收请求完毕，准备接收客户端得数据

到此为止，新连接到来时的回调函数 acceptTcpHandler 执行完毕，在这个回调函数中创建了一个 redisClient 来处理这个客户端接下来的请求，并绑定了接收的新连接的读文件事件。当有数据可读时，网络 i/o 轮询（比如 epoll）会有事件触发，此时绑定的回调函数 readQueryFromClient 将会调用来处理客户端发送过来的数据。

读取客户端请求的数据

在绑定有数据可读时的回调函数中的 createClient 函数中绑定了一个有数据可读时的回调函数 readQueryFromClient 函数，现在看看这个函数的具体内容

Networking.c:874

```
void readQueryFromClient(aeEventLoop *el, int fd, void *privdata, int mask) {
    redisClient *c = (redisClient*) privdata;
    char buf[REDIS_IOBUF_LEN];
    int nread;
    REDIS_NOTUSED(el);
    REDIS_NOTUSED(mask);

    server.current_client = c;
    nread = read(fd, buf, REDIS_IOBUF_LEN);
    if (nread == -1) {
        if (errno == EAGAIN) {
            nread = 0;
        } else {
            redisLog(REDIS_VERBOSE, "Reading from client: %s",strerror(errno));
```

```
            freeClient(c);
            return;
        }
    } else if (nread == 0) {
        redisLog(REDIS_VERBOSE, "Client closed connection");
        freeClient(c);
        return;
    }
    if (nread) {
        c->querybuf = sdscatlen(c->querybuf,buf,nread);
        c->lastinteraction = time(NULL);
    } else {
        server.current_client = NULL;
        return;
    }
    if (sdslen(c->querybuf) > server.client_max_querybuf_len) {
        sds ci = getClientInfoString(c), bytes = sdsempty();

        bytes = sdscatrepr(bytes,c->querybuf,64);
        redisLog(REDIS_WARNING,"Closing client that reached max query buffer
length: %s (qbuf initial bytes: %s)", ci, bytes);
        sdsfree(ci);
        sdsfree(bytes);
        freeClient(c);
        return;
    }
    processInputBuffer(c);
    server.current_client = NULL;
}
```

调用系统函数 read 来读取客户端传送过来的数据，调用 read 后对读取过程中被系统中断的情况（nread == -1 && errno == EAGAIN），客户端关闭的情况（nread == 0）进行了判断处理。

如果读取的数据超过限制（1GB）则报错。

读取完后进入 processInputBuffer 进行协议解析

请求协议

从 readQueryFromClient 函数读取客户端传过来的数据，进入 processInputBuffer 函数进行协议解析，可以把 processInputBuffer 函数看作是输入数据的协议解析器

Networking.c：835

```
void processInputBuffer(redisClient *c) {
    /* Keep processing while there is something in the input buffer */
    while(sdslen(c->querybuf)) {
```

```
        /* Immediately abort if the client is in the middle of something. */
        if (c->flags & REDIS_BLOCKED || c->flags & REDIS_IO_WAIT) return;

        /* REDIS_CLOSE_AFTER_REPLY closes the connection once the reply is
         * written to the client. Make sure to not let the reply grow after
         * this flag has been set (i.e. don't process more commands). */
        if (c->flags & REDIS_CLOSE_AFTER_REPLY) return;

        /* Determine request type when unknown. */
        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");
        }

        /* Multibulk processing could see a <= 0 length. */
        if (c->argc == 0) {
            resetClient(c);
        } else {
            /* Only reset the client when the command was executed. */
            if (processCommand(c) == REDIS_OK)
                resetClient(c);
        }
    }
}
```

Redis 支持两种协议，一种是 inline，一种是 multibulk。inline 协议是老协议，现在一般只在命令行下的 redis 客户端使用，其他情况一般是使用 multibulk 协议。

如果客户端传送的数据的第一个字符时‘*’，那么传送数据将被当做 multibulk 协议处理，否则将被当做 inline 协议处理。Inline 协议的具体解析函数是 `processInlineBuffer`，multibulk 协议的具体解析函数是 `processMultibulkBuffer`。

当协议解析完毕，即客户端传送的数据已经解析出命令字段和参数字段，接下来进行命令处理，命令处理函数是 `processCommand`。

Inline 请求协议

Networking.c：679

```
int processInlineBuffer(redisClient *c) {
    ...
}
```

根据空格分割客户端传送过来的数据，把传送过来的命令和参数保存在 argv 数组中，把参数个数保存在 argc 中，argc 的值包括了命令参数本身。即 set key value 命令，argc 的值为 3。

详细解析见协议详解

Multibulk 请求协议

Multibulk 协议比 inline 协议复杂，它是二进制安全的，即传送数据可以包含不安全字符。Inline 协议不是二进制安全的，比如，如果 set key value 命令中的 key 或 value 包含空白字符，那么 inline 协议解析时将会失败，因为解析出来的参数个数与命令需要的的参数个数会不一致。

协议格式

*<number of arguments> CR LF

$<number of bytes of argument 1> CR LF

<argument data> CR LF

...

$<number of bytes of argument N> CR LF

<argument data> CR LF

协议举例

*3

$3

SET

$5

mykey

$7

myvalue

具体解析代码位于

Networking.c：731

```
int processMultibulkBuffer(redisClient *c) {
    ...
}
```

详细解析见协议详解

处理命令

当协议解析完毕，则表示客户端的命令输入已经全部读取并已经解析成功，接下来就是执行客户端命令前的准备和执行客户端传送过来的命令

Redis.c：1062

```
/* If this function gets called we already read a whole
 * command, argments are in the client argv/argc fields.
 * processCommand() execute the command or prepare the
 * server for a bulk read from the client.
 *
 * If 1 is returned the client is still alive and valid and
 * and other operations can be performed by the caller. Otherwise
 * if 0 is returned the client was destroied (i.e. after QUIT). */
int processCommand(redisClient *c) {
    ...
 /* Now lookup the command and check ASAP about trivial error conditions
  * such as wrong arity, bad command name and so forth. */
    c->cmd = c->lastcmd = lookupCommand(c->argv[0]->ptr);
    ...
    call(c);
    ...
}
```

lookupCommand 先根据客户端传送过来的数据查找该命令并找到命令的对应处理函数

Call 函数调用该命令函数来处理命令，命令与对应处理函数的绑定位于

Redi.c：72

```
struct redisCommand *commandTable;
struct redisCommand readonlyCommandTable[] = {
{"get",getCommand,2,0,NULL,1,1,1},
    ...
}
```

回复请求

回复请求位于对应的命令中，以 get 命令为例

T_string.c：67

```
void getCommand(redisClient *c) {
   getGenericCommand(c);
}
```

T_string.c：52

```
int getGenericCommand(redisClient *c) {
   robj *o;

   if ((o = lookupKeyReadOrReply(c,c->argv[1],shared.nullbulk)) == NULL)
```

```
        return REDIS_OK;

    if (o->type != REDIS_STRING) {
        addReply(c,shared.wrongtypeerr);
        return REDIS_ERR;
    } else {
        addReplyBulk(c,o);
        return REDIS_OK;
    }
}
```

getGenericCommand 在 getset 命令中也会用到

lookupKeyReadOrReply 是以读数据为目的查询 key 函数，并且如果该 key 不存在，则在该函数中做不存在的回包处理。

如果该 key 存在，则返回该 key 对应的数据，addReply 函数以及以 addReply 函数开头的都是回包函数。

绑定写数据的回调函数

接下来看看 addReply 函数里的内容

Networking.c:190

```
void addReply(redisClient *c, robj *obj) {
    if (_installWriteEvent(c) != REDIS_OK) return;
    ...
}
```

Networking.c:64

```
int _installWriteEvent(redisClient *c) {
    if (c->fd <= 0) return REDIS_ERR;
    if (c->bufpos == 0 && listLength(c->reply) == 0 &&
        (c->replstate == REDIS_REPL_NONE ||
         c->replstate == REDIS_REPL_ONLINE) &&
        aeCreateFileEvent(server.el, c->fd, AE_WRITABLE,
        sendReplyToClient, c) == AE_ERR) return REDIS_ERR;
    return REDIS_OK;
}
```

addReply 函数一进来就先调用绑定写数据的回调函数 installWriteEvent

installWriteEvent 函数中创建了一个文件写事件和绑定写事件的回调函数为 sendReplyToClient。

准备写的数据内容

addReply 函数一进来后就绑定写数据的回调函数，接下来就是准备写的数据内容

Networking.c: 190

```
void addReply(redisClient *c, robj *obj) {
```

```
    if (_installWriteEvent(c) != REDIS_OK) return;
    redisAssert(!server.vm_enabled || obj->storage == REDIS_VM_MEMORY);

    /* This is an important place where we can avoid copy-on-write
     * when there is a saving child running, avoiding touching the
     * refcount field of the object if it's not needed.
     *
     * If the encoding is RAW and there is room in the static buffer
     * we'll be able to send the object to the client without
     * messing with its page. */
    if (obj->encoding == REDIS_ENCODING_RAW) {
        if (_addReplyToBuffer(c,obj->ptr,sdslen(obj->ptr)) != REDIS_OK)
            _addReplyObjectToList(c,obj);
    } else {
        /* FIXME: convert the long into string and use _addReplyToBuffer()
         * instead of calling getDecodedObject. As this place in the
         * code is too performance critical. */
        obj = getDecodedObject(obj);
        if (_addReplyToBuffer(c,obj->ptr,sdslen(obj->ptr)) != REDIS_OK)
            _addReplyObjectToList(c,obj);
        decrRefCount(obj);
    }
}
```

先尝试把要返回的内容添加到发送数据缓冲区中（redisClient->buf），如果该缓冲区的大小已经放不下这次想放进去的数据，或者已经有数据在排队（redisClient->reply 链表不为空），则把数据添加到发送链表的尾部。

给客户端答复数据

在绑定写数据的回调函数中看到绑定了回调函数 sendReplyToClient，现在来看看这个函数的主要内容

Networking.c：566

```
void sendReplyToClient(aeEventLoop *el, int fd, ...) {
    ...
while(c->bufpos > 0 || listLength(c->reply)) {
    ...
    if(c->bufpos > 0){
        ...
         nwritten=write(fd,...,c->bufpos-c->sentlen);
            ...
        } else {
            o = listNodeValue(listFirst(c->reply));
            ...
            nwritten=write(fd,...,objlen-c->sentlen);
```

```
            ...
        }
    }
}
```

通过调用系统函数 write 给客户端发送数据，如果缓冲区有数据就把缓冲区的数据发送给客户端，缓冲区的数据发送完了，如果有排队数据，则继续发送。

退出

Redis 服务器的退出是通过 shutdown 命令来退出的，退出前会做一系列的清理工作

Db.c：347

```
void shutdownCommand(redisClient *c) {
    if (prepareForShutdown() == REDIS_OK)
        exit(0);
    addReplyError(c,"Errors trying to SHUTDOWN. Check logs.");
}
```

总结

框架从启动，接收请求，读取客户端数据，请求协议解析，处理命令，回复请求，退出对 redis 运行的整个流程做了一个梳理。对整个 redis 的运作和框架有了一个初步的了解。

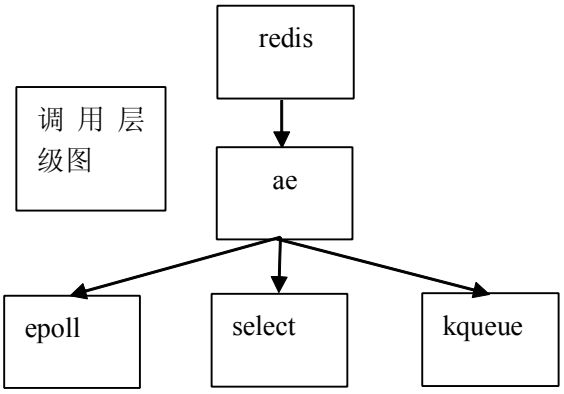
使用 GDB 跟踪 redis 的执行

编译源码时请使用 make noopt 命令，否则跟踪调试时很多变量和执行会被优化，无法和源码一一对应

Redis 中的库

网络 I/O 轮询

网络 i/o 轮询是通过封装系统支持的网络 i/o 处理库（epoll,select,kqueue），使其具有相同接口，调用层也封装了一个库 ae，这两个库实现 redis 的网络 i/o 轮询的功能。



Ae 库中网络 i/o 轮询相关结构体

aeFileEvent

Ae.c：63

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

aeFileProc 的定义如下

Ae.c：58

```
typedef void aeFileProc(struct aeEventLoop *eventLoop, int fd, void *clientData,
int mask);
```

Mask 表明这个事件关注的是什么类型的事件，可以是读事件 AE_READABLE，也可以是写事件 AE_WRITABLE，或者是读写事件（AE_READABLE|AE_WRITABLE)

rfileProc 是读事件发生时的处理函数

wfileProc 是写事件发生时的处理函数

clientData 是 redisClient 的引用，在调用 rfileProc 和 wfileProc 时需要作为参数 clientData 传入

aeEventLoop 中网络 i/o 轮询相关字段

Ae.h：88

```
/* State of an event based program */
typedef struct aeEventLoop {
   int maxfd;
   aeFileEvent events[AE_SETSIZE]; /* Registered events */
   aeFiredEvent fired[AE_SETSIZE]; /* Fired events */
   void *apidata; /* This is used for polling API specific data */
   ...
} aeEventLoop;
```

maxfd 是最大的文件描述符，主要用来判断是否有文件事件需要处理(ae.c:293)和当使用 select 来处理网络 IO 时使用时作为 select 的参数(ae_select.c:50)。

events[AE_SETSIZE]用于保存通过 aeCreateFileEvent 创建的文件事件，在 sendReplyToClient 和 freeClient 函数中通过调用 aeDeleteFileEvent 删除已经处理完的事件

fired[AE_SETSIZE]用于保存已经触发的文件事件，在对应的网络 I/O 函数中进行赋值（epoll,select,kqueue），不会对 fired 进行删除操作，只会一直覆盖原来的值。然后在 aeProcessEvents 函数中对已经触发的事件进行处理。

apidata 用于保存轮询 api 需要的数据，即 aeApiState 结构体，这个字段屏蔽了不同系统轮询 api 的差异性，对于 epoll 来说，aeApiState 结构体的定义如下：

```
typedef struct aeApiState {
    int epfd;
    struct epoll_event events[AE_SETSIZE];
} aeApiState;
```

Ae 库中网络 i/o 主要相关接口

Ae 库中的主要相关接口有以下：

```
aeEventLoop *aeCreateEventLoop(void);
```

创建一个事件轮询，分配内存并对事件轮询的字段进行初始化

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

创建一个文件事件（网络i/o事件）保存于eventLoop的事件数组events[fd]中，fd是文件fd，mask标识该事件时读事件还是写事件，proc是事件发生时的回调函数，clientData是一个redisClient指针，事件回调函数调用时作为参数。

```
int aeProcessEvents(aeEventLoop *eventLoop, int flags);
```

调用事件的绑定的回调函数来处理已经发生的事件。Flag标识需要处理的事件类型，是所有事件AE_ALL_EVENTS还是文件读写事件AE_FILE_EVENTS，还是定时事件AE_TIME_EVENTS。

```
void aeDeleteFileEvent(aeEventLoop *eventLoop, int fd, int mask);
```
删除文件事件，当一个文件事件处理完了要进行删除，fd是事件绑定的文件描述符，mask标识删除读事件还是写事件，还是都删除。

```
void aeDeleteEventLoop(aeEventLoop *eventLoop);
```
删除事件轮询，回收内存和做一些清理工作。当服务器退出时调用。

网络 api 库中的选取

网络 api 库的实现有 3 个（epoll,select,kqueue），它们都提供统一的接口。
根据平台选取其中一个最优的来使用。
选取相关代码如下：
Ae.c：43

```
/* Include the best multiplexing layer supported by this system.
 * The following should be ordered by performances, descending. */
#ifdef HAVE_EPOLL
#include "ae_epoll.c"
#else
    #ifdef HAVE_KQUEUE
    #include "ae_kqueue.c"
    #else
    #include "ae_select.c"
    #endif
#endif
```
尽量选取效率高的底层库，选取顺序是 epoll --> kqueue --> select

网络 api 库中的统一接口

不管是 epoll，还是 kqueue，还是 select，他们都实现了相同的接口，在 ae 库层，这些接口的实现对它来说是透明的，它只知道该接口可以实现什么功能，而不关心实现。比如，aeApiPoll，可能是 select 实现，也可能是 epoll 实现。

Api 库的接口主要有以下：

```
static int aeApiCreate(aeEventLoop *eventLoop);
```
创建并初始化 api 库，创建一个 aeApiState 结构体变量。以 epoll 的实现为例，这个函数会调用 epoll_create，初始化 aeApiState 结构体变量并保存于 eventLoop 的 apidata 字段。

```
static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int mask);
```
添加该 fd 感兴趣的事件。以 epoll 实现为例，这个函数里边会调用 epoll_ctl()，操作为 EPOLL_CTL_ADD 或者 EPOLL_CTL_MOD。

```
static int aeApiPoll(aeEventLoop *eventLoop, struct timeval *tvp);
```

事件轮询。以 epoll 实现为例，这个函数里边会调用 epoll_wait 函数，并把触发事件保存于 eventLoop 的 fired 字段。

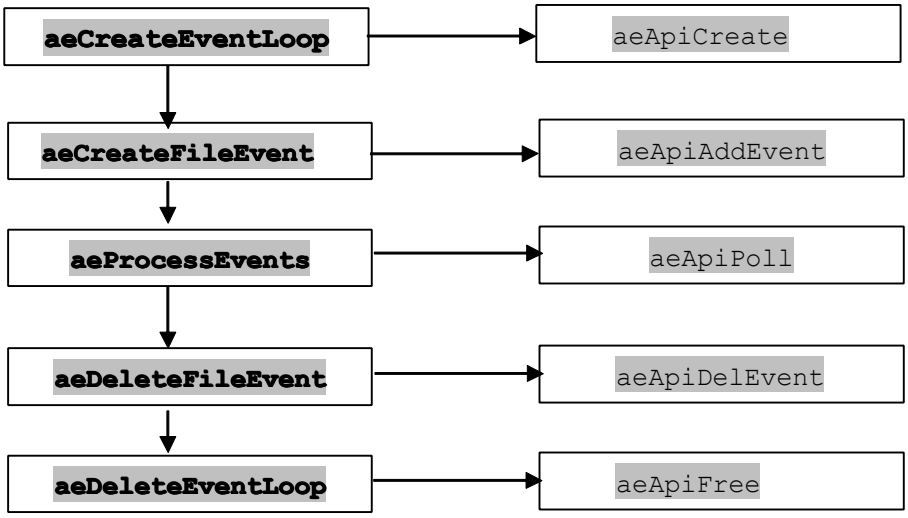
```
static void aeApiDelEvent(aeEventLoop *eventLoop, int fd, int delmask);
```

删除该 fd 不再感兴趣的事件。以 epoll 实现为例，这个函数里边会调用 epoll_ctl()，操作为 EPOLL_CTL_DEL。

```
static void aeApiFree(aeEventLoop *eventLoop);
```

释放事件轮询。

Ae 库与网络 api 库的调用顺序和关系



zmalloc（内存分配库）

Zmalloc 是在底层内存分配库上再封装的一个内存分配库，主要为了跟踪总共使用的内存，方便掌握内存使用情况。在计算使用内存时充分考虑了内存分配时的对齐（使用 sizeof(long)长度对齐）和底层分配库在不同平台的格外内存使用。

底层分配库

底层分配库的选取

Zmallo.c：48

```
/* Explicitly override malloc/free etc when using tcmalloc. */
#if defined(USE_TCMALLOC)
#define malloc(size) tc_malloc(size)
#define calloc(count,size) tc_calloc(count,size)
#define realloc(ptr,size) tc_realloc(ptr,size)
#define free(ptr) tc_free(ptr)
#elif defined(USE_JEMALLOC)
#define malloc(size) je_malloc(size)
#define calloc(count,size) je_calloc(count,size)
#define realloc(ptr,size) je_realloc(ptr,size)
#define free(ptr) je_free(ptr)
#endif
```

Redis 使用的底层分配库有默认的 malloc（glibc2.3 使用的是 ptmalloc2）,tc_malloc，je_malloc 三个库，使用的优先关系 tc_malloc --> je_malloc --> malloc(ptmalloc2)

tc_malloc

TCMalloc（Thread-Caching Malloc）是 google-perftools 工具中的一个，与标准的 glibc 库的 malloc 相比，TCMalloc 在内存的分配上效率和速度要高得多。
详细内容可以参考：
http://code.google.com/p/gperftools/?redir=1
http://shiningray.cn/tcmalloc-thread-caching-malloc.html
http://goog-perftools.sourceforge.net/doc/tcmalloc.html
http://hi.baidu.com/rodimus/blog/item/65cbbd17f080661ac83d6d7b.html

je_malloc

jemalloc 起源于 Jason Evans 2006 年在 BSDcan conference 发表的论文：A Scalable Concurrent malloc Implementation for FreeBSD

jason 认为 phkmalloc（FreeBSD's previous malloc implementation by Kamp (1998)）没有考虑多处理器的情况，因此在多线程并发下性能低下(事实如此)，而 jemalloc 适合多线程下内存分配管理。
详细内容可以参考：
http://blog.sina.com.cn/s/blog_836e02450100tjfj.html
http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

内存分配库中的宏

HAVE_MALLOC_SIZE

Zmalloc.h：38

```
#if defined(USE_TCMALLOC)
#define     ZMALLOC_LIB     ("tcmalloc-"     __xstr(TC_VERSION_MAJOR)     "."
__xstr(TC_VERSION_MINOR))
#include <google/tcmalloc.h>
#if TC_VERSION_MAJOR >= 1 && TC_VERSION_MINOR >= 6
#define HAVE_MALLOC_SIZE 1
#define zmalloc_size(p) tc_malloc_size(p)
#else
#error "Newer version of tcmalloc required"
#endif

#elif defined(USE_JEMALLOC)
#define    ZMALLOC_LIB    ("jemalloc-"    __xstr(JEMALLOC_VERSION_MAJOR)    "."
__xstr(JEMALLOC_VERSION_MINOR) "." __xstr(JEMALLOC_VERSION_BUGFIX))
#define JEMALLOC_MANGLE
#include <jemalloc/jemalloc.h>
#if JEMALLOC_VERSION_MAJOR >= 2 && JEMALLOC_VERSION_MINOR >= 1
#define HAVE_MALLOC_SIZE 1
#define zmalloc_size(p) JEMALLOC_P(malloc_usable_size)(p)
#else
#error "Newer version of jemalloc required"
#endif

#elif defined(__APPLE__)
#include <malloc/malloc.h>
#define HAVE_MALLOC_SIZE 1
#define zmalloc_size(p) malloc_size(p)
#endif
```

如果提供了计算一个指针对应内存大小功能，则定义宏 HAVE_MALLOC_SIZE 为 1

- Tcmalloc 的 1.6 版本及以上版本提供了计算一个指针对应内存大小功能。

Jemallc 的 2.1 版本及以上版本提供了计算一个指针对应内存大小功能。
苹果平台也有自己的计算一个指针对应内存大小的功能。

PREFIX_SIZE

Zmalloc.c：38

```
#ifdef HAVE_MALLOC_SIZE
#define PREFIX_SIZE (0)
#else
#if defined(__sun) || defined(__sparc) || defined(__sparc__)
#define PREFIX_SIZE (sizeof(long long))
#else
#define PREFIX_SIZE (sizeof(size_t))
#endif
#endif
```

PREFIX_SIZE 是指在用底层分配库分配内存时用来记录所分配内存大小的空间大小，位于所分配内存的最前面。是为了在释放所用内存时能知道被释放的内存有多大而可以更新记录的使用总内存大小。

有些平台上的底层库提供了计算一个指针对应内存大小的功能（比如：tc_malloc_size()函数和 malloc_size()函数），有些没有。没有计算一个指针对应内存大小的功能的情况下，就需要一个空间来保存分配的大小，而 PREFIX_SIZE 则是保存这个大小所需的空间。

如果定义了宏 HAVE_MALLOC_SIZE(即有计算一个指针对应内存大小功能)，则无需格外空间，此时定义宏 PREFIX_SIZE 为 0。

宏__sun 表运行平台是 Solaris 操作系统，宏__sparc 或者宏__sparc__表示运行平台的处理器是 SPARC 系列的处理器。SPARC，全称为“可扩充处理器架构”（Scalable Processor ARChitecture），是 RISC 微处理器架构之一。

如果是 sun 平台，则使用 long long 类型大小的空间存储所分配内存大小的空间大小
其他平台则使用 size_t 类型大小的空间存储所分配内存大小的空间大小（如 linux）

update_zmalloc_stat_alloc & update_zmalloc_stat_free

分配内存时，更新使用总内存，使用总内存增加。

Zmalloc.c：61

```
#define update_zmalloc_stat_alloc(__n,__size) do { \
   size_t _n = (__n); \
   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; \
      pthread_mutex_unlock(&used_memory_mutex); \
   } else { \
```

```
        used_memory += _n; \
    } \
} while(0)

#define update_zmalloc_stat_free(__n) do { \
    size_t _n = (__n); \
    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; \
        pthread_mutex_unlock(&used_memory_mutex); \
    } else { \
        used_memory -= _n; \
    } \
} while(0)
```

需要的内存大小是__n，包含了 PREFIX_SIZE。如果不是 sizeof(long) 的整数倍，则向上对齐为 sizeof(long) 的整数倍，因为实际分配内存时，真正分配的内存大小是对齐的，即 sizeof(long) 的整数倍。

__size 是数据需要的内存大小，即不包含 PREFIX_SIZE，在这个宏中暂时没用到，可能是历史遗留，也可能是留作以后使用。

如果配置了虚拟内存的多线程内存与磁盘同步数据，在更新总内存时就要加线程锁，保证总内存计数的准确性。

全局变量

Zmalloc.c：85

```
static size_t used_memory = 0;
```

使用的总内存

Zmalloc.c：86

```
static int zmalloc_thread_safe = 0;
```

是否需要检查线程安全，当使用多线程同步内存与数据的时需要

Zmalloc.c：87

```
pthread_mutex_t used_memory_mutex = PTHREAD_MUTEX_INITIALIZER;
```

线程互斥变量，用于多线程时解决竞争问题

内存分配库中的接口

内存分配的接口大部分与我们底层分配库的接口类似，只不过在分配时会更新使用的总内存，释放时也会更新使用的总内存。

主要接口如下：

```
void *zmalloc(size_t size);
```

内存分配接口，底层接口使用malloc

```
void *zcalloc(size_t size);
```

内存分配接口，底层接口使用calloc，但屏蔽了底层api calloc函数的第一个参数，其作用与zmalloc的不同之处是：calloc在动态分配完内存后，自动初始化该内存空间为零，而malloc不初始化，里边数据是随机的垃圾数据 。

```
void *zrealloc(void *ptr, size_t size);
```

对一个指针重新分配内存接口，底层接口使用realloc。

```
void zfree(void *ptr);
```

内存释放接口，底层接口使用free。

```
char *zstrdup(const char *s);
```

字符串复制接口，把s指向的字符串复制一份并返回

```
size_t zmalloc_used_memory(void);
```

返回统计出来的使用的总内存大小。

```
void zmalloc_enable_thread_safeness(void);
```

启动线程安全，更新总内存大小时需要加线程锁。函数只是简单对全局变量zmalloc_thread_safe赋值为1.

```
size_t zmalloc_get_rss(void);
```

取 RSS 大小。 RSS，即 resident set size，是进程实际驻存在物理内存的部分的大小。因为一个进程执行不需要把整个进程都全部驻存到物理内存。相对应的有 VSS（virtual set size），一个进程的总的大小。只有当进程执行时，整个进程都驻存到物理内存时才 RSS=VSS。

取 RSS 的实现：

1：如果定义了宏 HAVE_PROCFS，那么将从"/proc/[[pid]/stat"中读取；宏 HAVE_PROCFS 是在 config.h:19 定义的，只有在 linux 系统下才会定义宏 HAVE_PROCFS。

2：如果定义了宏 HAVE_TASKINFO，将从该进程 id 对应的 task_info_t 结构中读取 resident_size 变量；宏 HAVE_TASKINFO 是在 config.h：24 行定义的，只有在 apple 平台才会定义宏 HAVE_TASKINFO。

3：:如果宏 HAVE_PROCFS 和宏 HAVE_TASKINFO 都没有定义，那么简单地返回统计的总内存大小。

```
float zmalloc_get_fragmentation_ratio(void);
```

取存储的碎片率。碎片率=(float)zmalloc_get_rss()/zmalloc_used_memory()

sds(simple dynamic string，字符串处理库)

Redis 的字符串库。redis 中有很多字符串操作，redis 实现自己的字符串库，提高分配效率，提高计算长度效率等等。

sds 库中的类型

```
typedef char *sds;

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

- sds 类型其实就是 char * 类型，存储的就是以 '\0' 结尾的字符串

结构体类型 struct sdshdr 是 sds 的 header，len 字段是 sds 的长度，free 字段是总共为这个 buf 分配的内存减去 len 的值

sds 库中的主要接口

sds 库中的接口与我们常见的接口都很类似，就是每次操作时需要额外更新两个字段 len 和 free，或者提供处理这两个字段的接口

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

生成一个 sds，长度为 initlen，初始化为 init 的前 initlen 个字符并返回

```
sds sdsnew(const char *init);
```

生成一个 sds，初始化为 init 并返回

```
sds sdsempty();
```

生成一个空 sds 并返回

```
size_t sdslen(const sds s);
```

求 sds 长度

```
sds sdsdup(const sds s);
```

复制 sds 并返回

```
void sdsfree(sds s);
```

释放 sds

```
size_t sdsavail(sds s);
```

取这个 sds 还可以用的存储大小，即 free 字段值

```
sds sdsgrowzero(sds s, size_t len);
```

把 sds 增长到 len，增大的空间由 '\0' 填充并返回

```
sds sdscatlen(sds s, void *t, size_t len);
```
把字符串 t 的前 len 个字符拼接到 sds 后边并返回

```
sds sdscat(sds s, char *t);
```
把字符串拼接到 sds 后边并返回

```
sds sdscatsds(sds s, sds t);
```
把 sds 拼接到 sds 后边并返回

```
sds sdscpylen(sds s, char *t, size_t len);
```
把字符串的前 len 个字符复制到 sds 中。并返回

```
sds sdscpy(sds s, char *t);
```
复制字符串到 sds 中并返回

```
sds sdscatvprintf(sds s, const char *fmt, va_list ap);
```
封装 vsnprintf，把格式化输出串接到 sds 后边并返回

```
sds sdscatprintf(sds s, const char *fmt, ...);
```
实现类 printf 方法，把格式化输出串接到 sds 后边并返回

```
sds sdstrim(sds s, const char *cset);
```
去掉 sds 前后在 cset 中包含的字符并返回

```
sds sdsrange(sds s, int start, int end);
```
重置 sds 为位于 start 和 end 之间的字符串，start 和 end 可以小于 0，小于 0 时则从后往回算，最好一个字符对应的字符位置为-1， 比如：sdsrange(s, -1, -1)截取的字符串为最后一个字符。并返回

```
void sdsupdatelen(sds s);
```
重新计算 sds 的真正长度并更新 free 和 len 字段

```
void sdsclear(sds s);
```
将 sds 清空为空字符串

```
int sdscmp(sds s1, sds s2);
```
比较两个 sds，当 s1<s2 时，返回值<0，当 s1=s2 时，返回值=0；当 s1>s2 时，返回值>0

```
sds *sdssplitlen(char *s, int len, char *sep, int seplen, int *count);
```
把 s 根据 sep 分割，返回分割后的 sds 数组，这个函数二进制安全，len 是 s 的长度，seplen 是 sep 的长度。

```
void sdsfreesplitres(sds *tokens, int count);
```
释放一个 sds 数组

```
void sdstolower(sds s);
```
把 sds 转换成小写

```
void sdstoupper(sds s);
```
把 sds 转换成大写

```
sds sdsfromlonglong(long long value);
```
把长整型转换成 sds 并返回

```
sds sdscatrepr(sds s, char *p, size_t len);
```
把不可打印字符转成可见，并返回转换后的字符串

```
sds *sdssplitargs(char *line, int *argc);
```
根据输入参数分割成 sds 数组，参数个数存储于 argc，返回参数的 sds 数组

dict（dictionary）

Dict 是 redis 中非常重要和基础的数据结构，redis 中所有用户数据的 key 都是存储于 dict 中。Redis 很多辅助数据，比如 redis 的命令，也是用 dict 存储的，还有比如 hash 数据类型也可以通过 dict 来存储。Dict 其实就是一个哈希表的实现，采用链表解决冲突，很多细节先放开，先来看看 dict 的结构体。

Dict 中的结构体

存储的元素--dictEntry

Dict.h：45

```
typedef struct dictEntry {
    void *key;
    void *val;
    struct dictEntry *next;
} dictEntry;
```
Key：元素的 key
Val：key 对应的值
Next：dict 采用链表解决冲突，next 是指向下一个元素的指针

Dict 类型--dictType

这个结构体里的值决定了 dict 实例间的区别

Dict.h：51

```
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;
```

hashFunction：key 到索引的映射函数指针

- keyDup：key 的深复制函数指针

valDup：val 的深复制函数指针

keyCompare：key 的比较函数指针，相同返回 true，不同返回 false

keyDestructor：key 的析构函数指针

valDestructor：val 的析构函数指针

哈希表--dictht

Dict.h：62

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

Table：存储元素的哈希表，是存储元素结构体 dictEntry 指针的一维数组

Size：一维指针数组 table 的大小

Sizemask：该值比 size 小 1，用于与哈希函数算出来的值按位与，保证要操作的索引不会超出 table 的索引范围。

Used：哈希表中已经存储了的元素个数

字典--Dict

Dict.h：69

```
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;
```

- Type：dict 类型结构体指针

Privdata：dict 实例的关联数据

Ht：哈希表，每个 dict 有两个哈希表，用于增量哈希

Rehashidx：ht[0]中下一个需要增量重哈希的索引，当 rehashidx=-1 时表增量哈希当前未进行

Iterators：当前正在使用的迭代器数，当迭代一个 dict 时，如果当前 hash 表是 ht[0]且迭代器的 safe 字段为 1 且开始遍历 ht[0]（即迭代器的 index 字段为-1），则对 iterators 进行加 1，因此也说明 iterators 这个字段其实只有两种可能的值，0 和 1。当 iterators 字段不为 0 时（安全迭代器，即迭代器的 safe 字段为 1，只有在写 rdb 文件或者重写 aof 文件时才会使用安全迭代器），不会进行 lazy rehashing（当有后端写 rdb 文件进程或者重写 aof 文件进程时，也不会进行 active rehash），因为当迭代器的 safe 字段为 1 时，在迭代的过程中是可以对 dict 进行增删查改操作，如果进行 lazy 或者 active 重 hash，可能导致数据重复或丢失（重复例子：为了保存 rdb 文件，当前正在遍历 ht[0]，索引 0~4 都已经遍历完，如果此时又在 lazy rehashing，且 rehashing 到了 ht[0]的索引 2，就会把 ht[0]的索引 2 的内容挪到 ht[1]，这样当保存 rdb 文件时的指针遍历到 ht[1]，又会把从 ht[0]的索引 2 的挪过来的内容再写一份到 rdb 文件中。）。迭代完后，调用 dictReleaseIterator 函数进行释放，并对 dict 的 iterators 字段进行减 1 操作（减 1 操作的判断条件与加 1 操作的判断条件相同，如果进行过加 1 操作，则要进行减 1 操作）。

迭代器--dictIterator

Dict.h：77

```
/* If safe is set to 1 this is a safe iteartor, that means, you can call
 * dictAdd, dictFind, and other functions against the dictionary even while
 * iterating. Otherwise it is a non safe iterator, and only dictNext()
 * should be called while iterating. */
typedef struct dictIterator {
    dict *d;
    int table, index, safe;
    dictEntry *entry, *nextEntry;
} dictIterator;
```

d：这个迭代器所在的 dict 指针

table：迭代器当前所在的表的索引（table=0 或 table=1）

index：迭代器当前所在的索引

safe：如果 safe 等于 1，表示即使在迭代中，也可以对这个 dict 做增删查改操作，此时每开始迭代一个 dict，会对 dict 的 iteators 进行加 1。

entry：迭代器当前所在的元素的指针

nextEntry：迭代器下一个元素的指针

dict 的建立

Dict.c：119

```
/* Create a new hash table */
dict *dictCreate(dictType *type,
```

```
       void *privDataPtr)
{
    dict *d = zmalloc(sizeof(*d));

    _dictInit(d,type,privDataPtr);
    return d;
}

/* Initialize the hash table */
int _dictInit(dict *d, dictType *type,
       void *privDataPtr)
{
    _dictReset(&d->ht[0]);
    _dictReset(&d->ht[1]);
    d->type = type;
    d->privdata = privDataPtr;
    d->rehashidx = -1;
    d->iterators = 0;
    return DICT_OK;
}
```

Dict 的建立很简单，就是给 dict 结构体分配内存，然后初始化相关字段
内存分配时并没有给两个哈希表分配内存，在_dictReset 中把 table 字段初始化为 null

dict 的增加操作

Dict.c：258

```
/* Add an element to the target hash table */
int dictAdd(dict *d, void *key, void *val)
{
    int index;
    dictEntry *entry;
    dictht *ht;

    if (dictIsRehashing(d)) _dictRehashStep(d);

    /* Get the index of the new element, or -1 if
     * the element already exists. */
    if ((index = _dictKeyIndex(d, key)) == -1)
        return DICT_ERR;

    /* Allocates the memory and stores key */
    ht = dictIsRehashing(d) ? &d->ht[1] : &d->ht[0];
    entry = zmalloc(sizeof(*entry));
    entry->next = ht->table[index];
```

```
    ht->table[index] = entry;
    ht->used++;

    /* Set the hash entry fields. */
    dictSetHashKey(d, entry, key);
    dictSetHashVal(d, entry, val);
    return DICT_OK;
}
```

_dictKeyIndex 先求 key 对应的 index，如果该 key 已经存在，则返回的 index 为-1，则 dictAdd 返回错误，因为增加操作只能增加未存在的 key。另外_dictKeyIndex 函数里还调用了给 hash 表分配内存或者增加内存的函数_dictExpandIfNeeded，具体见 dict 的动态调整如果该 key 还未存在，则给需要添加的元素分配内存并插入到哈希表对应索引的链表的头部

设置新添加的元素的 key 和 value 字段

dict 的删除操作

Dict.c：349

```
int dictDelete(dict *ht, const void *key) {
    return dictGenericDelete(ht,key,0);
}
```

Dict.c：311

```
/* Search and remove an element */
static int dictGenericDelete(dict *d, const void *key, int nofree)
{
    unsigned int h, idx;
    dictEntry *he, *prevHe;
    int table;

    if (d->ht[0].size == 0) return DICT_ERR; /* d->ht[0].table is NULL */
    if (dictIsRehashing(d)) _dictRehashStep(d);
    h = dictHashKey(d, key);

    for (table = 0; table <= 1; table++) {
        idx = h & d->ht[table].sizemask;
        he = d->ht[table].table[idx];
        prevHe = NULL;
        while(he) {
            if (dictCompareHashKeys(d, key, he->key)) {
                /* Unlink the element from the list */
                if (prevHe)
                    prevHe->next = he->next;
                else
                    d->ht[table].table[idx] = he->next;
```

```
            if (!nofree) {
                dictFreeEntryKey(d, he);
                dictFreeEntryVal(d, he);
            }
            zfree(he);
            d->ht[table].used--;
            return DICT_OK;
        }
        prevHe = he;
        he = he->next;
    }
    if (!dictIsRehashing(d)) break;
  }
  return DICT_ERR; /* not found */
}
```

根据 key 求出对应的对应的索引
根据索引找到对应链表，遍历索引对应的链表，如果该 key 不存在，则返回错误
若找到该 key，则删除该 key，并把删除元素前后的链表结点串起来
若 nofree 参数为 0，则对 key 和 value 的内存进行释放

dict 的查询操作

Dict.c：412

```
void *dictFetchValue(dict *d, const void *key) {
    dictEntry *he;

    he = dictFind(d,key);
    return he ? dictGetEntryVal(he) : NULL;
}
```

Dict.c：391

```
dictEntry *dictFind(dict *d, const void *key)
{
   dictEntry *he;
   unsigned int h, idx, table;

   if (d->ht[0].size == 0) return NULL; /* We don't have a table at all */
   if (dictIsRehashing(d)) _dictRehashStep(d);
   h = dictHashKey(d, key);
   for (table = 0; table <= 1; table++) {
      idx = h & d->ht[table].sizemask;
      he = d->ht[table].table[idx];
      while(he) {
         if (dictCompareHashKeys(d, key, he->key))
            return he;
```

```
            he = he->next;
        }
        if (!dictIsRehashing(d)) return NULL;
    }
    return NULL;
}
```

根据 key 求出对应的对应的索引

- 根据索引找到对应链表，遍历索引对应的链表，如果该 key 不存在，则返回空指针

寻到对应结点后，则返回该结点的 value 指针

dict 的改变操作

Dict.c：285

```
/* Add an element, discarding the old if the key already exists.
 * Return 1 if the key was added from scratch, 0 if there was already an
 * element with such key and dictReplace() just performed a value update
 * operation. */
int dictReplace(dict *d, void *key, void *val)
{
    dictEntry *entry, auxentry;

    /* Try to add the element. If the key
     * does not exists dictAdd will suceed. */
    if (dictAdd(d, key, val) == DICT_OK)
        return 1;
    /* It already exists, get the entry */
    entry = dictFind(d, key);
    /* Free the old value and set the new one */
    /* Set the new value and free the old one. Note that it is important
     * to do that in this order, as the value may just be exactly the same
     * as the previous one. In this context, think to reference counting,
     * you want to increment (set), and then decrement (free), and not the
     * reverse. */
    auxentry = *entry;
    dictSetHashVal(d, entry, val);
    dictFreeEntryVal(d, &auxentry);
    return 0;
}
```

先尝试添加该 key，如果该 key 不存在，则添加成功并返回 1

如果添加失败，则查找该 key 对应的存储元素，然后修改该元素的 value 字段

Dict 的动态调整

Dict.c：518

```
/* Expand the hash table if needed */
static int _dictExpandIfNeeded(dict *d)
{
    /* Incremental rehashing already in progress. Return. */
    if (dictIsRehashing(d)) return DICT_OK;

    /* If the hash table is empty expand it to the intial size. */
    if (d->ht[0].size == 0) return dictExpand(d, DICT_HT_INITIAL_SIZE);

    /* If we reached the 1:1 ratio, and we are allowed to resize the hash
     * table (global setting) or we should avoid it but the ratio between
     * elements/buckets is over the "safe" threshold, we resize doubling
     * the number of buckets. */
    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;
}
```

Dict.c:155

```
/* Expand or create the hashtable */
int dictExpand(dict *d, unsigned long size)
{
    dictht n; /* the new hashtable */
    unsigned long realsize = _dictNextPower(size);

    /* the size is invalid if it is smaller than the number of
     * elements already inside the hashtable */
    if (dictIsRehashing(d) || d->ht[0].used > size)
        return DICT_ERR;

    /* Allocate the new hashtable and initialize all pointers to NULL */
    n.size = realsize;
    n.sizemask = realsize-1;
    n.table = zcalloc(realsize*sizeof(dictEntry*));
    n.used = 0;
```

```
    /* Is this the first initialization? If so it's not really a rehashing
     * we just set the first hash table so that it can accept keys. */
    if (d->ht[0].table == NULL) {
        d->ht[0] = n;
        return DICT_OK;
    }

    /* Prepare a second hash table for incremental rehashing */
    d->ht[1] = n;
    d->rehashidx = 0;
    return DICT_OK;
}
```

Dict 中的哈希表只要符合以下扩展条件之一则会进行扩展：

1：哈希表还未分配过内存，则给 ht[0]分配初始的内存

2：哈希表中的元素数量（used）大于等于哈希表中的索引数量（size）且 dict_can_resize 为 1。

3：哈希表中的元素数量（used）大于等于哈希表中的索引数量（size）且 used/size 的比例大于 dict_force_resize_ratio。

Dict 中的哈希表的扩展的大小都是 2 的 n 次方。

除了初始化时，扩展的都是 ht[1]，具体原因见 dict 的增量重哈希

Dict 的增量重哈希

重哈希的实现

Dict.c：185

```
/* Performs N steps of incremental rehashing. Returns 1 if there are still
 * keys to move from the old to the new hash table, otherwise 0 is returned.
 * Note that a rehashing step consists in moving a bucket (that may have more
 * thank one key as we use chaining) from the old to the new hash table. */
int dictRehash(dict *d, int n) {
    if (!dictIsRehashing(d)) return 0;

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

        /* Check if we already rehashed the whole table... */
        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;
```

```
        }

        /* Note that rehashidx can't overflow as we are sure there are more
         * elements because ht[0].used != 0 */
        while(d->ht[0].table[d->rehashidx] == NULL) d->rehashidx++;
        de = d->ht[0].table[d->rehashidx];
        /* Move all the keys in this bucket from the old to the new hash HT */
        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;
}
```

每次重哈希时可以指定想重哈希的步骤数 n

如果重哈希完成，则释放 ht[0]的指针数组 table，并把 ht[1]赋值给 ht[0]，然后重置 ht[1]，并重置 dict 的 rehashidx 为-1，表此次重哈希已经全部完成

如果重哈希未完成，则找到下一个需要重哈希的索引，遍历这个索引对应的链表，求出链表中元素的 key 在 ht[1]的索引并把它串接 ht[1]索引对应的链表中

Active rehashing

Active rehashing：在定时函数 serverCron 中调用增量重哈希函数，每次至少重哈希 100 个索引，每次重哈希时间一旦超过 1ms 则结束此次主动重哈希。

Redis.c：447

```
/* Our hash table implementation performs rehashing incrementally while
 * we write/read from the hash table. Still if the server is idle, the hash
 * table will use two tables for a long time. So we try to use 1 millisecond
 * of CPU time at every serverCron() loop in order to rehash some key. */
void incrementallyRehash(void) {
    int j;

    for (j = 0; j < server.dbnum; j++) {
        if (dictIsRehashing(server.db[j].dict)) {
```

```
            dictRehashMilliseconds(server.db[j].dict,1);
            break; /* already used our millisecond for this loop... */
        }
    }
}
```

Dict.c：234

```
/* Rehash for an amount of time between ms milliseconds and ms+1 milliseconds
*/
int dictRehashMilliseconds(dict *d, int ms) {
    long long start = timeInMilliseconds();
    int rehashes = 0;

    while(dictRehash(d,100)) {
        rehashes += 100;
        if (timeInMilliseconds()-start > ms) break;
    }
    return rehashes;
}
```

每次每个 db 的 dict 增量重哈希一旦超过 1ms，这个 db 的 dict 的此次重哈希就完成了。每次重哈希以 100 步为单位，一步一个索引。

Lazy rehashing

Lazy rehashing：每次对 dict 进行操作时重哈希一个索引。

Lazy rehashing 函数 _dictRehashStep，在 dictAdd， dictGenericDelete， dictFind，dictGetRandomKey 函数中都会进行调用，这几个函数覆盖了 dict 的增删查改操作。

Dict.c：246

```
/* This function performs just a step of rehashing, and only if there are
 * no safe iterators bound to our hash table. When we have iterators in the
 * middle of a rehashing we can't mess with the two hash tables otherwise
 * some element can be missed or duplicated.
 *
 * This function is called by common lookup or update operations in the
 * dictionary so that the hash table automatically migrates from H1 to H2
 * while it is actively used. */
static void _dictRehashStep(dict *d) {
    if (d->iterators == 0) dictRehash(d,1);
}
```

每次重哈希一个索引

Dict 的迭代器初始化

不安全的迭代器

不安全的迭代器表示在使用该迭代器时，不能对 dict 进行增删查改
Dict.c：419

```
dictIterator *dictGetIterator(dict *d)
{
    dictIterator *iter = zmalloc(sizeof(*iter));

    iter->d = d;
    iter->table = 0;
    iter->index = -1;
    iter->safe = 0;
    iter->entry = NULL;
    iter->nextEntry = NULL;
    return iter;
}
```

分配内存
对相关字段进行初始化

安全的迭代器

安全的迭代器表示在使用该迭代器时，可以对 dict 进行增删查改
Dict.c：432

```
dictIterator *dictGetSafeIterator(dict *d) {
    dictIterator *i = dictGetIterator(d);

    i->safe = 1;
    return i;
}
```

与不安全的迭代器的唯一区别是，safe 字段初始化为 1
只有在写 rdb 文件或 aof 文件时会使用安全迭代器

dict 的下一个元素

Dict.c：439

```
dictEntry *dictNext(dictIterator *iter)
{
    while (1) {
        if (iter->entry == NULL) {
```

```
            dictht *ht = &iter->d->ht[iter->table];
            if (iter->safe && iter->index == -1 && iter->table == 0)
                iter->d->iterators++;
            iter->index++;
            if (iter->index >= (signed) ht->size) {
                if (dictIsRehashing(iter->d) && iter->table == 0) {
                    iter->table++;
                    iter->index = 0;
                    ht = &iter->d->ht[1];
                } else {
                    break;
                }
            }
            iter->entry = ht->table[iter->index];
        } else {
            iter->entry = iter->nextEntry;
        }
        if (iter->entry) {
            /* We need to save the 'next' here, the iterator user
             * may delete the entry we are returning. */
            iter->nextEntry = iter->entry->next;
            return iter->entry;
        }
    }
    return NULL;
}
```

iter->entry == NULL 表明要从一个新的索引开始遍历，否则继续遍历该冲突链表

确定要遍历的 hash 表，dict 有两个 hash 表用于增量重 hash

如果该迭代器是安全迭代器且是刚开始遍历第一个 hash 表，则 dict 的 iterators 进行加 1

如果第一个 hash 表遍历完了且正在重 hash，即有一部分数据是在 ht[1]，则对 ht[1] 进行遍历

Dict 的迭代器的释放

Dict.c：470

```
void dictReleaseIterator(dictIterator *iter)
{
    if (iter->safe && !(iter->index == -1 && iter->table == 0))
        iter->d->iterators--;
    zfree(iter);
}
```

如果迭代器的 safe 字段是 1 开始过对 ht[0]的遍且历(即对 dict 的 iterators 字段进

行过加 1,）则对 dict 的 iterators 字段进行减 1
释放迭代器占用的内存

Dict 的增删查改使用实例

增加操作实例：set 命令

setCommand（t_string.c：37）

删除操作实例：del 命令

delCommand（db.c：224）

改变操作实例：set 命令

setCommand（t_string.c：37）

查询操作实例：get 命令

getCommand（t_string.c：67）

adlist

Adlist 是 redis 中的双向链表的实现。
Redis 的结构体都封装的很好，看起来很舒服，迭代器，增删查改很明了
代码很简单，看起来很舒服，简直就是一种享受呀
具体代码位于 adlist.h 和 adlist.c

adlist 中的结构体

listNode

Adlist.h：36

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

双向链表结点结构体
Prev 是指向上一个结点的指针
Next 是指向下一个结点的指针

listIter

Adlist.h：42

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

双向链表的迭代器结构体
Next 是迭代到下一个的指针
Direction 是迭代的方向，有两种方向：AL_START_HEAD 是从头往尾开始迭代，
AL_START_TAIL 是从尾往头开始迭代。

list 结构体

Adlist.h：47

```
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;
```

双向链表的结构体
Head 是链表头指针
Tail 是链表尾指针
Dup 是链表结点结构体中 value 的复制函数
Free 是链表结点结构体中 value 的释放函数
Match 是链表结点结构体中 value 的匹配函数
Len 是链表的长度

Adlist 提供的接口

接口很明了，封装的很好，见名知意，就不再解释了。

```
list *listCreate(void);
void listRelease(list *list);
list *listAddNodeHead(list *list, void *value);
```

```
list *listAddNodeTail(list *list, void *value);
list *listInsertNode(list *list, listNode *old_node, void *value, int after);
void listDelNode(list *list, listNode *node);
listIter *listGetIterator(list *list, int direction);
listNode *listNext(listIter *iter);
void listReleaseIterator(listIter *iter);
list *listDup(list *orig);
listNode *listSearchKey(list *list, void *key);
listNode *listIndex(list *list, int index);
void listRewind(list *list, listIter *li);
void listRewindTail(list *list, listIter *li);
```

Intset

Intset 是 redis 实现的整数集合。

实现要点

Intset 使用一个数组来从小到大存储所有集合中的整数。

为了节省空间，intset 使用了编码类型。每个整数占用的字节以集合中占用最大的为准。如果一开始集合中的整数都可以用 2 个字节表示，突然插入一个要用 4 个字节才能存储的整数，则整个集合都会升级到每个整数使用 4 个字节存储。

因为 intset 是排序存储，增删查改的时间复杂度都是 O(log(n))

Intset 主要接口

接口很明了，封装的很好，见名知意，就不再解释了。

```
intset *intsetNew(void);
intset *intsetAdd(intset *is, int64_t value, uint8_t *success);
intset *intsetRemove(intset *is, int64_t value, int *success);
uint8_t intsetFind(intset *is, int64_t value);
int64_t intsetRandom(intset *is);
uint8_t intsetGet(intset *is, uint32_t pos, int64_t *value);
uint32_t intsetLen(intset *is);
size_t intsetBlobLen(intset *is);
```

ziplist

ziplist 是用一个字符串来实现的双向链表结构，顾名思义，使用 ziplist 可以减少双向链表的存储空间，主要是节省了链表指针的存储，如果存储指向上一个链表结点和指向下一个链表

结点的指针需要 8 个字节，而转化成存储上一个结点长度和当前结点长度在大多数情况下可以节省很多空间（最好的情况下只需 2 个字节）。但是每次向链表增加元素都需要重新分配内存。

ziplist 中的结构体

```
typedef struct zlentry {
    unsigned int prevrawlensize, prevrawlen;
    unsigned int lensize, len;
    unsigned int headersize;
    unsigned char encoding;
    unsigned char *p;
} zlentry;
```

Prevrawlen：上个链表结点占用的长度
Prevrawlensize：上个链表结点长度的存储占用的字节数
Len：当前链表结点占用的长度
Lensize：当前链表结点长度的存储占用的字节数
Headersize：当前链表结点的头部大小， headersize = prevrawlensize + lensize
Encoding：当前链表结点长度（即字段 len）使用的编码类型
P：指向当前结点起始位置的指针

Ziplist 的存储结构

链表存储结构

<zlbytes><zltail><zllen><entry><entry><zlend>
Zlbytes：一个 4 字节的无符号整型，存储的是整个 ziplist 占用的字节数，用于重分配内存时使用。
Zltail：一个 4 字节的无符号整型，存储的是链表最后一个结点的偏移值，即链表开头地址+zltail 即为最后一个结点的起始地址
Zllen：一个 2 字节的无符号整型，存储的是链表中存储的结点数，当这个值存储的是 2 字节无符号整型的最大值时，需要遍历链表获取链表的结点数
Entry：链表结点，链表结点的存储格式见结点存储结构
Zlend：占用 1 字节的链表的结尾符，值为 255

相关的宏定义
Ziplist.c: 89

```
/* Utility macros */
#define ZIPLIST_BYTES(zl)       (*((uint32_t*)(zl)))
#define ZIPLIST_TAIL_OFFSET(zl) (*((uint32_t*)((zl)+sizeof(uint32_t))))
#define ZIPLIST_LENGTH(zl)      (*((uint16_t*)((zl)+sizeof(uint32_t)*2)))
```

```
#define ZIPLIST_HEADER_SIZE     (sizeof(uint32_t)*2+sizeof(uint16_t))
#define ZIPLIST_ENTRY_HEAD(zl)  ((zl)+ZIPLIST_HEADER_SIZE)
#define ZIPLIST_ENTRY_TAIL(zl)  ((zl)+ZIPLIST_TAIL_OFFSET(zl))
#define ZIPLIST_ENTRY_END(zl)   ((zl)+ZIPLIST_BYTES(zl)-1)
```

结点存储结构

<上一个链表结点占用的长度><当前链表结点占用的长度><当前结点数据>

上一个链表结点占用的长度

上一个链表结点占用的长度占用的字节数根据编码类型而定
当长度数据小于 254 使用一个字节存储，该字节存储的数值就是该长度，
当长度数据大于等于 254 时，使用 5 个字节存储，第一个字节的数值为 254，表示接下来的 4 个字节才真正表示长度

当前链表结点用的长度存储和数据存储

第一个字节的前两位用于区分长度存储编码类型和数据编码类型，具体如下
字符串类型编码
|00pppppp|
长度小于等于 63（2^6-1）字节的字符串，后 6 位用于存储字符串长度，长度与类型总共占用了 1 个字节

|01pppppp|qqqqqqqq|
长度小于等于 16383（2^14-1）字节的字符串，后 14 位用于存储字符串长度，长度与类型总共占用了 2 个字节

|10_____|qqqqqqqq|rrrrrrrr|ssssssss|tttttttt|
长度大于等于 16384 字节的字符串，后 4 个字节用于存储字符串长度，长度与类型总共占用了 5 个字节

整型编码
|1100____|
整型类型，后 2 个字节存储的值就是该整数
|1101____|
整型类型，后 4 个字节存储的值就是该整数
|1110____|
整型类型，后 8 个字节存储的值就是该整数

相关的宏定义
Ziplist.c：77

```
/* Different encoding/length possibilities */
#define ZIP_STR_06B (0 << 6)
```

```
#define ZIP_STR_14B (1 << 6)
#define ZIP_STR_32B (2 << 6)
#define ZIP_INT_16B (0xc0 | 0<<4)
#define ZIP_INT_32B (0xc0 | 1<<4)
#define ZIP_INT_64B (0xc0 | 2<<4)

/* Macro's to determine type */
#define ZIP_IS_STR(enc) (((enc) & 0xc0) < 0xc0)
#define ZIP_IS_INT(enc) (!ZIP_IS_STR(enc) && ((enc) & 0x30) < 0x30)
```

Ziplist 提供的接口

```
unsigned char *ziplistNew(void);
```

创建一个 ziplist
返回创建的 ziplist 的指针

```
unsigned char *ziplistPush(unsigned char *zl, unsigned char *s, unsigned int slen,
int where);
```

在 ziplist 的尾端或头部添加一个结点
zl 是 ziplist 的指针
s 是待添加结点的值
slen 是待添加结点的值长度
返回最新的 ziplist 的指针

```
unsigned char *ziplistIndex(unsigned char *zl, int index);
```

根据索引获取 ziplist 的结点，封装类似数组接口
zl 是 ziplist 的指针
index 是索引，从 0 开始，0 即取链表的第一个结点，index 可以是负数，负数表从后往前算，
-1 就是取链表的最后一个元素
如果 index 处有结点，则返回指向改结点的指针，否则返回 NULL

```
unsigned char *ziplistNext(unsigned char *zl, unsigned char *p);
```

获取 ziplist 的下一个结点
zl 是无用参数
p 是当前结点指针
如果还有下一个结点，则返回下一个结点的指针，否则返回 NULL

```
unsigned char *ziplistPrev(unsigned char *zl, unsigned char *p);
```

获取 ziplist 的上一个结点
zl 是 ziplist 的指针
p 是当前结点指针
如果还有上一个结点，则返回上一个结点的指针，否则返回 NULL

```
unsigned int ziplistGet(unsigned char *p, unsigned char **sval, unsigned int *slen,
long long *lval);
```

获取 p 指向的当前结点的值

p 是指向当前结点的指针

sval 保存获取到的当前结点的值的指针

- slen 是获取到的当前结点的值的长度

lval 是当值是整型时保存返回的数值

如果 p 指向的结点是合法结点返回 1，否则返回 0

```
unsigned char *ziplistInsert(unsigned char *zl, unsigned char *p, unsigned char
*s, unsigned int slen);
```

在指针 p 指向的位置插入一个结点

zl 是 ziplist 的指针

p 是待插入结点的位置

s 是待插入结点的值

slen 是待插入结点的值的长度

返回最新的 ziplist 的指针

```
unsigned char *ziplistDelete(unsigned char *zl, unsigned char **p);
```

删掉*p 指向的结点

zl 是 ziplist 的指针

p 是一个 value-result 参数，传入需删除的结点，返回被删除结点下一个结点的指针

返回最新的 ziplist 的指针

```
unsigned char *ziplistDeleteRange(unsigned char *zl, unsigned int index, unsigned
int num);
```

删除连续的一批结点

- zl 是 ziplist 的指针

index 是开始删除的索引

num 是删除的个数

返回最新的 ziplist 的指针

```
unsigned int ziplistCompare(unsigned char *p, unsigned char *s, unsigned int
slen);
```

p 指向的结点的值和 s 对应的值做比较

p 是 ziplist 结点的指针

s 是呆比较的值

slen 是 s 的长度

相等返回 1，否则返回 0

```
unsigned int ziplistLen(unsigned char *zl);
```

取 ziplist 链表中元素的个数

zl 是 ziplist 的指针

返回 ziplist 链表中元素的个数

```
size_t ziplistBlobLen(unsigned char *zl);
```

取 ziplist 链表占用的字节数
zl 是 ziplist 的指针
返回 ziplist 链表占用的字节数

zipmap

zipmap 与 ziplist 类似，也是作者为了节省空间实现的 map。顾名思义，zip 是压缩的意思，即这种 map 实现比起一般的 map 实现可以节省空间。

Zipmap 用于 redis 中的 hash 数据类型，当 hash 类型里边的 key 的个数小于等于配置的值 hash-max-zipmap-entries 的时候，hash 数据类型用 zipmap 存储。当 hash 类型里边的 key 的个数大于配置的值 hash-max-zipmap-entries 的时候，hash 数据类型用 dict 存储。

Zipmap 与 dict 的转换函数是 convertToRealHash（t_hash.c：248）

Zipmap 的存储结构

<zmlen><keylen>key<vallen><free>val<keylen>key<vallen><free>val <zmend>

Zmlen：存储当前 zipmap 中的 key 的个数。占用一个字节，如果个数等于或大于 254，这个字节存储的值将没有意义了，需要遍历整个 zipmap 来获取 key 的个数。

keyen：存储 key 的长度。当 keylen 小于 254 时，keylen 占用一个字节，这个字节里存储的值就是 key 的长度；如果 keylen 大于等于 254，keylen 占用 5 个字节，这个字节存储的值就是 254，接下来的 4 个字节里存储的是 keylen 的长度。

key：存储 zipmap 中的 key 的值。

- vallen：存储 val 的长度，vallen 的存储方法与 keylen 相同。

Val：存储 zipmap 中 val 的值。

Free：存储 val 值后边的没有使用的字节数。作者为了避免分配和移动内存太频繁做的一种优化。如果一个 key 一开始的值的长度是 10，后来被更新成长度为 5 的值，此时不需要把后边的数据往前移动，只需设置 free 为 5 即可。如果后来再被更新成长度为 8 的值，也不需要重新分配内存，只需设置 free 为 2 即可。

Zmend：zipmap 的结尾标识符，占用一个字节，值为 255。

Zipmap 提供的接口

```
unsigned char *zipmapNew(void);
```

创建一个 zipmap
返回创建的 zipmap 指针

```
unsigned char *zipmapSet(unsigned char *zm, unsigned char *key, unsigned int klen,
```

```
unsigned char *val, unsigned int vlen, int *update);
```

在 zipmap 中添加一个 key—value 对

- Zm 是 zipmap 指针

Key 是待添加 key 的内容
Klen 是 key 的长度
Val 是待添加的 value 的内容
Vlen 是待添加的 value 的字符串的长度
Update 是一个返回值参数，如果该 key 已经存在，*update 返回 1，否则返回 0

```
unsigned char *zipmapDel(unsigned char *zm, unsigned char *key, unsigned int klen,
int *deleted);
```

在 zipmap 中删除一个 key—value 对

- Zm 是 zipmap 指针

Key 是待删除 key 的内容
Klen 是 key 的长度
Deleted 是一个返回值参数，如果该 key 删除成功，*Deleted 返回 1，如果该 key 不存在则返回 0

```
unsigned char *zipmapRewind(unsigned char *zm);
```

返回指向 zipmap 第一个结点的指针
Zm 是 zipmap 指针

```
unsigned char *zipmapNext(unsigned char *zm, unsigned char **key, unsigned int
*klen, unsigned char **value, unsigned int *vlen);
```

获取 zm 指向的结点的 key，klen，value 和 vlen
Zm 是 zipmap 指针
Key 是返回值参数，返回当前结点的 key 的内容
Klen 是返回值参数，返回当前结点的 key 的长度
Value 是返回值参数，返回当前结点的 value 的内容
Vlen 是返回值参数，返回当前结点的 value 的长度
返回指向返回结点内容的下一个结点的指针

```
int zipmapGet(unsigned char *zm, unsigned char *key, unsigned int klen, unsigned
char **value, unsigned int *vlen);
```

取某个 key 对应的 value
Zm 是 zipmap 指针
Key 是待查找的 key 的内容
Klen 是待查找的 key 的长度
Value 是返回值参数，返回查找到的 value 的内容
Vlen 是返回值参数，返回查找到的 value 的长度
返回 1 表获取成功，返回 0 该 key 不存在，获取失败

```
int zipmapExists(unsigned char *zm, unsigned char *key, unsigned int klen);
```

判断一个 key 是否存在于 zipmap 中

Zm 是 zipmap 指针
Key 是待查找的 key 的内容
Klen 是待查找的 key 的长度
返回 1 表查找成功，返回 0 该 key 不存在，查找失败

```
unsigned int zipmapLen(unsigned char *zm);
```

获取 zipmap 中结点的数量
Zm 是 zipmap 指针
返回 zipmap 中结点的数量

```
size_t zipmapBlobLen(unsigned char *zm);
```

获取 zipmap 占用的字节数
Zm 是 zipmap 指针

Redis 的模块

协议详解

在框架鸟瞰中简单的提到了 redis 的 inline 协议和 multibulk 协议，这里进行更细致的分析

请求协议

Inline 协议

协议格式：argv[0] argv[1] argv[2] ...
举例：SET mykey 6
Networking.c：679

```
int processInlineBuffer(redisClient *c) {
    char *newline = strstr(c->querybuf,"\r\n");
    int argc, j;
    sds *argv;
    size_t querylen;

    /* Nothing to do without a \r\n */
    if (newline == NULL) {
        if (sdslen(c->querybuf) > REDIS_INLINE_MAX_SIZE) {
            addReplyError(c,"Protocol error: too big inline request");
            setProtocolError(c,0);
        }
        return REDIS_ERR;
    }
```

```
    /* Split the input buffer up to the \r\n */
    querylen = newline-(c->querybuf);
    argv = sdssplitlen(c->querybuf,querylen," ",1,&argc);

    /* Leave data after the first line of the query in the buffer */
    c->querybuf = sdsrange(c->querybuf,querylen+2,-1);

    /* Setup argv array on client structure */
    if (c->argv) zfree(c->argv);
    c->argv = zmalloc(sizeof(robj*)*argc);

    /* Create redis objects for all arguments. */
    for (c->argc = 0, j = 0; j < argc; j++) {
        if (sdslen(argv[j])) {
            c->argv[c->argc] = createObject(REDIS_STRING,argv[j]);
            c->argc++;
        } else {
            sdsfree(argv[j]);
        }
    }
    zfree(argv);
    return REDIS_OK;
}
```

查找 '\r\n'，一个命令以 '\r\n' 结尾，如果没有 '\r\n' ，表明这个请求字符串还没有接收完或者发送的命令错误。

把 '\r\n' 前的字符串根据空格分割成多个参数存储于 argv 数组内，参数个数存储于 argc 内，argc 的值包括了命令字段。即 set key value 命令，argc 的值为 3。

把缓冲区内已经取出来的字符串（ '\r\n' 前的字符串，包括 '\r\n' 在内）删除，让剩余数据继续保持于这个缓冲区内

初始化 redisClient 中的 argc 字段和 argv 字段，redisClient 中的 argc 依然是参数个数，redisClient 中的 argv 是 robj 指针的一维数组。

Multibulk 协议

协议格式

*<number of arguments> CR LF

$<number of bytes of argument 1> CR LF

<argument data> CR LF

...

$<number of bytes of argument N> CR LF

<argument data> CR LF

协议举例
*3
$3
SET
$5
mykey
$7
myvalue

Network：731

```
int processMultibulkBuffer(redisClient *c) {
   char *newline = NULL;
   int pos = 0, ok;
   long long ll;

   if (c->multibulklen == 0) {
      /* The client should have been reset */
      redisAssert(c->argc == 0);

      /* Multi bulk length cannot be read without a \r\n */
      newline = strchr(c->querybuf,'\r');
      if (newline == NULL) {
         if (sdslen(c->querybuf) > REDIS_INLINE_MAX_SIZE) {
            addReplyError(c,"Protocol error: too big mbulk count string");
            setProtocolError(c,0);
         }
         return REDIS_ERR;
      }

      /* Buffer should also contain \n */
      if (newline-(c->querybuf) > ((signed)sdslen(c->querybuf)-2))
         return REDIS_ERR;

      /* We know for sure there is a whole line since newline != NULL,
       * so go ahead and find out the multi bulk length. */
      redisAssert(c->querybuf[0] == '*');
      ok = string2ll(c->querybuf+1,newline-(c->querybuf+1),&ll);
      if (!ok || ll > 1024*1024) {
         addReplyError(c,"Protocol error: invalid multibulk length");
         setProtocolError(c,pos);
         return REDIS_ERR;
      }

      pos = (newline-c->querybuf)+2;
```

```
    if (ll <= 0) {
        c->querybuf = sdsrange(c->querybuf,pos,-1);
        return REDIS_OK;
    }

    c->multibulklen = ll;

    /* Setup argv array on client structure */
    if (c->argv) zfree(c->argv);
    c->argv = zmalloc(sizeof(robj*)*c->multibulklen);
}

redisAssert(c->multibulklen > 0);
while(c->multibulklen) {
    /* Read bulk length if unknown */
    if (c->bulklen == -1) {
        newline = strchr(c->querybuf+pos,'\r');
        if (newline == NULL) {
            if (sdslen(c->querybuf) > REDIS_INLINE_MAX_SIZE) {
                addReplyError(c,"Protocol error: too big bulk count string");
                setProtocolError(c,0);
            }
            break;
        }

        /* Buffer should also contain \n */
        if (newline-(c->querybuf) > ((signed)sdslen(c->querybuf)-2))
            break;

        if (c->querybuf[pos] != '$') {
            addReplyErrorFormat(c,
                "Protocol error: expected '$', got '%c'",
                c->querybuf[pos]);
            setProtocolError(c,pos);
            return REDIS_ERR;
        }

        ok = string2ll(c->querybuf+pos+1,newline-(c->querybuf+pos+1),&ll);
        if (!ok || ll < 0 || ll > 512*1024*1024) {
            addReplyError(c,"Protocol error: invalid bulk length");
            setProtocolError(c,pos);
            return REDIS_ERR;
        }
```

```
            pos += newline-(c->querybuf+pos)+2;
            c->bulklen = ll;
        }

        /* Read bulk argument */
        if (sdslen(c->querybuf)-pos < (unsigned)(c->bulklen+2)) {
            /* Not enough data (+2 == trailing \r\n) */
            break;
        } else {
            c->argv[c->argc++] = createStringObject(c->querybuf+pos,c->bulklen);
            pos += c->bulklen+2;
            c->bulklen = -1;
            c->multibulklen--;
        }
    }

    /* Trim to pos */
    c->querybuf = sdsrange(c->querybuf,pos,-1);

    /* We're done when c->multibulk == 0 */
    if (c->multibulklen == 0) return REDIS_OK;

    /* Still not read to process the command */
    return REDIS_ERR;
}
```

判断有没有 '\r\n' 字符串，如果没有，返回错误
判断输入是不是以 * 开头，如果不是，记 log 并退出
读取传送过来的第一个值， number of arguments
根据 number of arguments 遍历参数，根据参数的长度读取参数字符串，最后存储于 redisClient 的 argv 字段

回复协议

redis 的回包数据散落在具体的命令中。

共用的协议

对应一些错误或简单的成功返回，inline 和 multibulk 是一样的
比如：
+OK\r\n
或者
-ERR\r\n
等等

这些字符串会在一启动时初始化，初始化的代码位于
Redis.c：746

```
void createSharedObjects(void) {
    int j;

    shared.crlf = createObject(REDIS_STRING,sdsnew("\r\n"));
    shared.ok = createObject(REDIS_STRING,sdsnew("+OK\r\n"));
    shared.err = createObject(REDIS_STRING,sdsnew("-ERR\r\n"));
    shared.emptybulk = createObject(REDIS_STRING,sdsnew("$0\r\n\r\n"));
    shared.czero = createObject(REDIS_STRING,sdsnew(":0\r\n"));
    shared.cone = createObject(REDIS_STRING,sdsnew(":1\r\n"));
    shared.cnegone = createObject(REDIS_STRING,sdsnew(":-1\r\n"));
    shared.nullbulk = createObject(REDIS_STRING,sdsnew("$-1\r\n"));
    shared.nullmultibulk = createObject(REDIS_STRING,sdsnew("*-1\r\n"));
    shared.emptymultibulk = createObject(REDIS_STRING,sdsnew("*0\r\n"));
    shared.pong = createObject(REDIS_STRING,sdsnew("+PONG\r\n"));
    shared.queued = createObject(REDIS_STRING,sdsnew("+QUEUED\r\n"));
    shared.wrongtypeerr = createObject(REDIS_STRING,sdsnew(
        "-ERR Operation against a key holding the wrong kind of value\r\n"));
    shared.nokeyerr = createObject(REDIS_STRING,sdsnew(
        "-ERR no such key\r\n"));
    shared.syntaxerr = createObject(REDIS_STRING,sdsnew(
        "-ERR syntax error\r\n"));
    shared.sameobjecterr = createObject(REDIS_STRING,sdsnew(
        "-ERR source and destination objects are the same\r\n"));
    shared.outofrangeerr = createObject(REDIS_STRING,sdsnew(
        "-ERR index out of range\r\n"));
    shared.loadingerr = createObject(REDIS_STRING,sdsnew(
        "-LOADING Redis is loading the dataset in memory\r\n"));
    shared.space = createObject(REDIS_STRING,sdsnew(" "));
    shared.colon = createObject(REDIS_STRING,sdsnew(":"));
    shared.plus = createObject(REDIS_STRING,sdsnew("+"));
    ...
}
```

Inline 协议

与请求协议类似，具体可参考：
http://redis.io/topics/protocol

Multibulk 协议

与请求协议类似，具体可参考：

http://redis.io/topics/protocol

VM（虚拟内存）

Reids 官方已经在 http://redis.io/topics/virtual-memory 说明，reids 在 2.4 以后的版本将不再支持 virtual memory，并且说即使是支持 virtual memory 的版本也不建议使用 virtual memory。Redis 以后将致力于做一个最好的纯内存数据库（当然，一如既往地支持落地存储功能），以后将主要在支持脚步，集群和更好的落地存储功能努力。

不管怎样，来学习一下 redis 的 VM 实现。

实现原因

根据作者 antirez 的这篇文章 http://antirez.com/post/redis-virtual-memory-story.html ，Redis 的虚拟内存的是为了解决冷数据的问题。基本上所有互联网业务都有这个问题，即每天的活跃用户只占总用户数的一小部分。而 redis 是一个内存数据库，如果把所有数据放在内存中（不管这个用户是不是活跃或者这个用户永远都不会再回来），这样对很多业务来说是不现实的，并且对宝贵的内存空间也是一种很多的浪费。于是作者想通过虚拟内存来解决这个问题。

Redis 的虚拟内存是在应用层自己实现的，并没有使用操作系统的虚拟内存，主要原因作者也在 http://antirez.com/post/redis-virtual-memory-story.html 这篇文章说到了：

操作系统交换很少使用的页，但操作系统每页的大小是 4K。Redis 使用了各种各样的数据结构，有的结构占用空间很大，比如 hash, set, list 等等，它们可能几十 K，所以它们可能连续占用了好几个页。有的结构占用空间很小，这些小的结构却也可能分散到很多个操作系统的页中。所以，即使只有一小部分热点数据，这些数据也有可能分布在操作系统的很多页中。由于操作系统的一个页中只有 1byte 的热点数据也会让操作系统把这个页保留在内存中。反过来说，如果内存不够时，某个比较少使用的页被换出了，但如果上边有 1byte 数据是 redis 想用的，又将导致操作系统将这个页换入，而内存不够时，这个页将再次被换出，严重影响了 redis 的性能。

在 redis 中，不管是简单的对象还是复杂的对象，它们在内存中时占用的空间大小远远大于它们被系列化到磁盘后的大小。因为系列化到磁盘存储时，它们既没有指针，也没有元数据。一般来说，redis 中的对象在内存中时占用的空间大概是它们被系列化到磁盘后的 10 倍，因为 redis 可以很好地对存储在磁盘上的对象进行编码。这也就是说，对于同一份大小的数据，redis 在应用层实现的 VM 比与操作系统的 VM 可以减少 10 倍左右的磁盘 I/O 操作。

实现要点

虚拟内存其实就是一个磁盘上的文件 swap file，当内存不够用时，把数据写到这个文件中。

Redis 只会把 value 换出到虚拟内存，不会把 key 换出到虚拟内存，redis 通过这样来保证使用虚拟内存后还是可以保证很高的查询效率。Redis 这样的设计目的是使得当启用 VM

且热点数据保存在内存中时可以与没有启动 VM 的情况拥有差不多的查询性能。

Value 存储在 swap file 中的格式与 rdb 的格式一样，所以当实现换出换入数据时可以直接调 rdb 的接口来实现，大大减少了 vm 模块的复杂度。

Swap file 的格式与操作系统的类似，也是按 page 来分，通过配置可以设置 page 大小和 page 数量，虚拟内存的大小等于(page 大小 * page 数量）

Redis 在内存中保存一个位图来标识某个 page 是否被使用

计算可交换性公式：swappability = idle-time * log(estimated size)，可交换性越大，越有可能被换出

Redis 会尝试随机取 100 次来尽量取出 5 个在内存中存储的对象，根据可交换性公式计算出可交换性，最后把交换性最大的换出内存。取 100 次是想保证即使可交换的对象的比例只有 1%或者 2%也能尽量取到一个可交换的对象

Redis 把 val 对象换出后，会把原来 val 对象的指针类型变成 redisVmPointer，该指针指向的对象存储了把 val 对象换入内存时需要的信息。

阻塞式 VM

阻塞式 VM 在换入或者换出时会阻塞当前的所有任务，使得业务请求的响应时间变慢。

当配置的 vm-max-threads 配置为 0 时使用阻塞式 VM

阻塞式换出

阻塞式换出是在定时函数中调用的，定时函数每 100 毫秒就有权限执行一次，也就是说，每次运行到定时函数内后如果条件满足都会进行阻塞式换出，一直换出到不能再换出为止，具体换出的条件见下边。

Redis.c：523

```
int serverCron(struct aeEventLoop *eventLoop, long long id, void *clientData)
{
    ...
    /* Swap a few keys on disk if we are over the memory limit and VM
     * is enbled. 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 threade 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;
        }
    }
    ...
    return 100;
}
```

Vm.c：539

```
/* Return true if it's safe to swap out objects in a given moment.
 * Basically we don't want to swap objects out while there is a BGSAVE
 * or a BGAEOREWRITE running in backgroud. */
int vmCanSwapOut(void) {
    return (server.bgsavechildpid == -1 && server.bgrewritechildpid == -1);
}
```

当没有写 rdb 文件和 aof 文件的子进程时将可以进行换出。

定时函数执行时会检查是否满足了同时换出的下边四个条件，如果都同时满足将一直进行换出，直到其中有一个条件不满足

配置启用了 VM，即 vm-enabled yes

占用内存超过了配置 vm-max-memory

swap file 中还有合适的 page 可用

能选取出可交换到虚拟内存中的对象

如果配置 vm-max-threads 为 0，则进行阻塞式换出

我们再进去看看执行阻塞式换出操作的函数 vmSwapOneObjectBlocking

Vm.c：531

```
int vmSwapOneObjectBlocking() {
    return vmSwapOneObject(0);
}
```

vmSwapOneObjectBlocking 函数是 vmSwapOneObject 函数的简单封装

接着进 vmSwapOneObject 函数看看

Vm.c：459

```
/* Try to swap an object that's a good candidate for swapping.
 * Returns REDIS_OK if the object was swapped, REDIS_ERR if it's not possible
 * to swap any object at all.
 *
 * If 'usethreaded' is true, Redis will try to swap the object in background
 * using I/O threads. */
int vmSwapOneObject(int usethreads) {
    int j, i;
```

```
struct dictEntry *best = NULL;
double best_swappability = 0;
redisDb *best_db = NULL;
robj *val;
sds key;

for (j = 0; j < server.dbnum; j++) {
   redisDb *db = server.db+j;
   /* Why maxtries is set to 100?
    * Because this way (usually) we'll find 1 object even if just 1% - 2%
    * are swappable objects */
   int maxtries = 100;

   if (dictSize(db->dict) == 0) continue;
   for (i = 0; i < 5; i++) {
      dictEntry *de;
      double swappability;

      if (maxtries) maxtries--;
      de = dictGetRandomKey(db->dict);
      val = dictGetEntryVal(de);
      /* Only swap objects that are currently in memory.
       *
       * Also don't swap shared objects: not a good idea in general and
       * we need to ensure that the main thread does not touch the
       * object while the I/O thread is using it, but we can't
       * control other keys without adding additional mutex. */
      if (val->storage != REDIS_VM_MEMORY || val->refcount != 1) {
         if (maxtries) i--; /* don't count this try */
         continue;
      }
      swappability = computeObjectSwappability(val);
      if (!best || swappability > best_swappability) {
         best = de;
         best_swappability = swappability;
         best_db = db;
      }
   }
}
if (best == NULL) return REDIS_ERR;
key = dictGetEntryKey(best);
val = dictGetEntryVal(best);

redisLog(REDIS_DEBUG,"Key with best swappability: %s, %f",
```

```
        key, best_swappability);

    /* Swap it */
    if (usethreads) {
        robj *keyobj = createStringObject(key,sdslen(key));
        vmSwapObjectThreaded(keyobj,val,best_db);
        decrRefCount(keyobj);
        return REDIS_OK;
    } else {
        vmpointer *vp;

        if ((vp = vmSwapObjectBlocking(val)) != NULL) {
            dictGetEntryVal(best) = vp;
            return REDIS_OK;
        } else {
            return REDIS_ERR;
        }
    }
}
```

寻找最合适换出的 key，寻找的算法是遍历所有 db，在每个 db 中找出 5 个在内存中且引用计数为 1 的 key，计算可交换性。每个 db 中尝试的次数为 100 次，尝试 100 次是为了保证即使有 1%~2%左右可交换的对象也尽可能找到可交换的对象。

计算可交换性的算法是 swappability = idle-time * log(estimated size)，idle-time 是通过对象的 lru 字段减去 server.lruclock 字段得到的值。每次查找对象时都会更新这个对象的 lru 字段（见 db.c:18）。estimated size 是对象的占用空间的大小，具体根据对象类型不同，计算方法不同。比如：字符串类型占用空间大小计算方法为：对象指针大小+字符串长度+字符串类型中 len 和 free 占用的空间；hash 类型的长度的计算方法为：（dictEntry 结构体大小+随机取 hash 中一个值的大小）*hash 表长度。

找到最适合交换的 key 后，调用 vmSwapObjectBlocking 函数进行交换，返回虚拟函数指针，这里记录了到时我们把对象 load 回内存时需要的信息。

我们再跟着进去看看 vmSwapObjectBlocking 函数

Vm.c：259

```
/* Transfers the 'val' object to disk. Store all the information
 * a 'vmpointer' object containing all the information needed to load the
 * object back later is returned.
 *
 * If we can't find enough contiguous empty pages to swap the object on disk
 * NULL is returned. */
vmpointer *vmSwapObjectBlocking(robj *val) {
    off_t pages = rdbSavedObjectPages(val);
    off_t page;
    vmpointer *vp;
```

```
    redisAssert(val->storage == REDIS_VM_MEMORY);
    redisAssert(val->refcount == 1);
    if (vmFindContiguousPages(&page,pages) == REDIS_ERR) return NULL;
    if (vmWriteObjectOnSwap(val,page) == REDIS_ERR) return NULL;

    vp = createVmPointer(val);
    vp->page = page;
    vp->usedpages = pages;
    decrRefCount(val); /* Deallocate the object from memory. */
    vmMarkPagesUsed(page,pages);
    redisLog(REDIS_DEBUG,"VM: object %p swapped out at %lld (%lld pages)",
        (void*) val,
        (unsigned long long) page, (unsigned long long) pages);
    server.vm_stats_swapped_objects++;
    server.vm_stats_swapouts++;
    return vp;
}
```

计算存储到虚拟内存的需要的页数

查找可以存储下这个对象大小的连续页的位置，如果找不到则返回 NULL

写对象到虚拟内存中，虚拟内存的存储格式是与 rdb 文件一样的，所以写虚拟内存的接口使用的是 rdb 那套。

创建一个虚拟内存对象指针 redisVmPointer，记下对象在虚拟内存中占用的页数和起始页，这些信息在把这个对象 load 回内存时需要用到。

标示该对象所占页为已用

然后统计交换数，交换对象数

返回虚拟内存指针 redisVmPointer

阻塞式换入

阻塞式换入是在需要操作这个对象时，发现这个对象不在内存中，然后就把这个对象换入内存。阻塞式换入是在操作对象的统一入口处 lookupKey 函数统一处理的，我们来看看 lookupKey 函数的实现。

db.c：9

```
robj *lookupKey(redisDb *db, robj *key) {
    dictEntry *de = dictFind(db->dict,key->ptr);
    if (de) {
        robj *val = dictGetEntryVal(de);

        /* Update the access time for the aging algorithm.
         * Don't do it if we have a saving child, as this will trigger
         * a copy on write madness. */
        if (server.bgsavechildpid == -1 && server.bgrewritechildpid == -1)
            val->lru = server.lruclock;
```

```
        if (server.vm_enabled) {
            if (val->storage == REDIS_VM_MEMORY ||
                val->storage == REDIS_VM_SWAPPING)
            {
                /* If we were swapping the object out, cancel the operation */
                if (val->storage == REDIS_VM_SWAPPING)
                    vmCancelThreadedIOJob(val);
            } else {
                int notify = (val->storage == REDIS_VM_LOADING);

                /* Our value was swapped on disk. Bring it at home. */
                redisAssert(val->type == REDIS_VMPOINTER);
                val = vmLoadObject(val);
                dictGetEntryVal(de) = val;

                /* Clients blocked by the VM subsystem may be waiting for
                 * this key... */
                if (notify) handleClientsBlockedOnSwappedKey(db,key);
            }
        }
        return val;
    } else {
        return NULL;
    }
}
```

根据 key 查找要操作的对象，如果对象在内存，返回的 val 就是该对象的指针，如果对象被换出了内存中，返回的 val 其实是一个 redisVmPointer。

如果开启了虚拟内存且对象在内存中，直接取出 val 返回。

如果开启了虚拟内存且对象正处在换出内存的过程中，则取消换出，这种情况只在多线程 VM 中可能发生。

如果开启了虚拟内存且对象在磁盘上，则调用函数 vmLoadObject 将对象换入。

如果开启了虚拟内存且对象正处在从磁盘换入到内存的过程中，则将这个请求对应的 redisClient 对象挂到等待链表的尾端。这种情况只在多线程 VM 中可能发生。所有的等待链表组成了一个以 key 为键的哈希表（哈希表的具体见库--dict），当这个对象换入内存后会通知这个等待这个 key 的 redisClient 对象。

多线程 VM

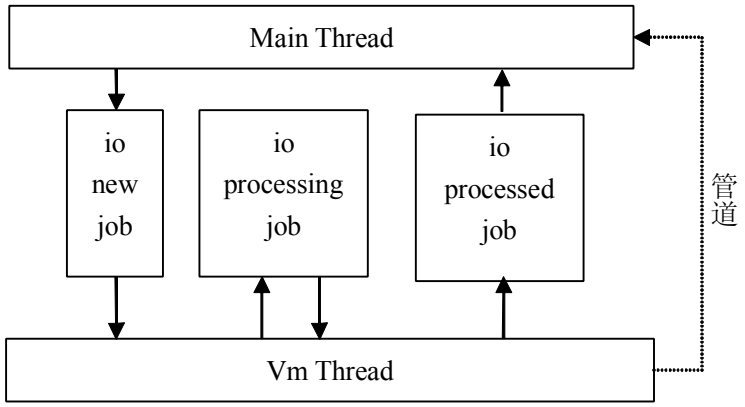
Redis 是一个单线程的 server，并且经常被用在实时响应性要求很高的场景。对于阻塞式 VM 有点让人接受不了。因为如果一个 client 请求需要的数据需要从虚拟内存 load 进来的话，会阻塞所有 client 的请求。

作者 antirez 最终采用了多线程来处理内存与虚拟内存之间的换入换出工作。这样当一个

client 请求需要的数据需要从虚拟内存 load 进来时，redis 把这个把数据 load 进内存的任务交给 vm 的线程处理，然后把这个 client 挂起，等待通知数据换入到内存。然后 redis 继续处理其他 client 的请求。这样，当一个 client 请求需要的数据需要从虚拟内存 load 进来时只会阻塞一个 client，而不会影响到其他 client。

vm 的多线程只局限于 vm 这个模块，没有影响到其他模块，多线程编程的复杂性也局限于 vm 模块。

任务对象的足迹图（从左到右）



中间的三大块分别是 server.io_newjobs 队列，server.io_processing 队列和 server.io_processed 队列。

主线程把换入换出的任务放到 server.io_newjobs 队列中

Vm 线程一直循环从 server.io_newjobs 队列中读取任务，当开始处理时，把任务放到 server.io_processing 队列中。

Vm 线程处理完任务后，把任务放到 server.io_processed 队列中，然后通过在管道中写入来通知主线程有任务完成，可以去 server.io_processing 队列读取已经完成的任务。

多线程换出

多线程换出与阻塞式换出很多大体相同，唯一的区别是，阻塞式换出是主线程直接从内存把最适合换出的对象写到虚拟内存中，而多线程式换出是把这些换出包装成任务，丢给 vm 线程去做换出真正的换入换出工作。这样，主线程就可以避开磁盘 I/O 操作的瓶颈了。

多线程换出把换出分成了两步，第一步是在虚拟内存中准备好存储换出对象的页，第二步是真正换出。每一步都是一个任务，即多线程换出一个可以需要主线程发送两个任务给 vm 线程。

Vm.c：465

```
int vmSwapOneObject(int usethreads) {
    ...
```

```
    /* Swap it */
    if (usethreads) {
        robj *keyobj = createStringObject(key,sdslen(key));
        vmSwapObjectThreaded(keyobj,val,best_db);
        decrRefCount(keyobj);
        return REDIS_OK;
    } else {
        vmpointer *vp;

        if ((vp = vmSwapObjectBlocking(val)) != NULL) {
            dictGetEntryVal(best) = vp;
            return REDIS_OK;
        } else {
            return REDIS_ERR;
        }
    }
}
```

多线程换出与阻塞式换出的区别，多线程 vm 调用 vmSwapObjectThreaded 函数把换出这个任务放到 vm 线程的新任务队列 server.io_newjobs 中。

接下来看看多线程换出的第一步，把 REDIS_IOJOB_PREPARE_SWAP 任务放到 vm 线程的新任务队列中。

Vm.c：924

```
int vmSwapObjectThreaded(robj *key, robj *val, redisDb *db) {
    iojob *j;

    j = zmalloc(sizeof(*j));
    j->type = REDIS_IOJOB_PREPARE_SWAP;
    j->db = db;
    j->key = key;
    incrRefCount(key);
    j->id = j->val = val;
    incrRefCount(val);
    j->canceled = 0;
    j->thread = (pthread_t) -1;
    val->storage = REDIS_VM_SWAPPING;

    lockThreadedIO();
    queueIOJob(j);
    unlockThreadedIO();
    return REDIS_OK;
}
```

构造任务对象

把将被换出的 val 对象的 storage 字段置为 REDIS_VM_SWAPPING

Vm.c：915

```
/* This function must be called while with threaded IO locked */
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();
}
```

把换出任务放到 vm 线程的新任务队列 server.io_newjobs 中的细节。

Vm 线程收到 REDIS_IOJOB_PREPARE_SWAP 任务的处理如下：

Vm.c：829

```
        /* Process the Job */
        if (j->type == REDIS_IOJOB_LOAD) {
            vmpointer *vp = (vmpointer*)j->id;
            j->val = vmReadObjectFromSwap(j->page,vp->vtype);
        } else if (j->type == REDIS_IOJOB_PREPARE_SWAP) {
            j->pages = rdbSavedObjectPages(j->val);
        } else if (j->type == REDIS_IOJOB_DO_SWAP) {
            if (vmWriteObjectOnSwap(j->val,j->page) == REDIS_ERR)
                j->canceled = 1;
        }
```

Vm 线程对 REDIS_IOJOB_PREPARE_SWAP 任务的处理只是调用函数 rdbSavedObjectPages 计算把这个对象换出到虚拟内存需要的 page 数，然后通知主线程该任务的完成。

我们继续来看看主线程收到 REDIS_IOJOB_PREPARE_SWAP 任务完成的处理

Vm.c：632

```
        } else if (j->type == REDIS_IOJOB_PREPARE_SWAP) {
            /* Now we know the amount of pages required to swap this object.
             * Let's find some space for it, and queue this task again
             * rebranded as REDIS_IOJOB_DO_SWAP. */
            if (!vmCanSwapOut() ||
                vmFindContiguousPages(&j->page,j->pages) == REDIS_ERR)
            {
                /* Ooops... no space or we can't swap as there is
                 * a fork()ed Redis trying to save stuff on disk. */
                j->val->storage = REDIS_VM_MEMORY; /* undo operation */
                freeIOJob(j);
            } else {
                /* Note that we need to mark this pages as used now,
                 * if the job will be canceled, we'll mark them as freed
```

```
             * again. */
            vmMarkPagesUsed(j->page,j->pages);
            j->type = REDIS_IOJOB_DO_SWAP;
            lockThreadedIO();
            queueIOJob(j);
            unlockThreadedIO();
        }
    }
```

如果当前不能进行换出操作或者虚拟内存已经没有连续的可以放下这个对象的空间，则这个换出任务取消。同时把准备换出的 val 对象的 storage 字段重置为 REDIS_VM_MEMORY，并释放任务对象。

如果当前可以进行换出操作且虚拟内存有连续的可以放下这个对象的空间，则把任务对象的类型改为 REDIS_IOJOB_DO_SWAP 并把这个任务再次放入到 vm 线程的新任务队列 server.io_newjobs 中。

接下来看看 vm 线程收到 REDIS_IOJOB_DO_SWAP 任务时的处理

Vm.c：829

```
    /* Process the Job */
    if (j->type == REDIS_IOJOB_LOAD) {
        vmpointer *vp = (vmpointer*)j->id;
        j->val = vmReadObjectFromSwap(j->page,vp->vtype);
    } else if (j->type == REDIS_IOJOB_PREPARE_SWAP) {
        j->pages = rdbSavedObjectPages(j->val);
    } else if (j->type == REDIS_IOJOB_DO_SWAP) {
        if (vmWriteObjectOnSwap(j->val,j->page) == REDIS_ERR)
            j->canceled = 1;
    }
```

Vm 线程对 REDIS_IOJOB_DO_SWAP 任务的处理就是把这个对象写到虚拟内存以 page 为开始的空间中，然后通知主线程该任务的完成。

最后来看看主线程收到 REDIS_IOJOB_DO_SWAP 任务完成的处理

Vm.c：653

```
    } else if (j->type == REDIS_IOJOB_DO_SWAP) {
        vmpointer *vp;

        /* Key swapped. We can finally free some memory. */
        if (j->val->storage != REDIS_VM_SWAPPING) {
            vmpointer *vp = (vmpointer*) j->id;
            printf("storage: %d\n",vp->storage);
            printf("key->name: %s\n",(char*)j->key->ptr);
            printf("val: %p\n",(void*)j->val);
            printf("val->type: %d\n",j->val->type);
            printf("val->ptr: %s\n",(char*)j->val->ptr);
        }
```

```
            redisAssert(j->val->storage == REDIS_VM_SWAPPING);
            vp = createVmPointer(j->val);
            vp->page = j->page;
            vp->usedpages = j->pages;
            dictGetEntryVal(de) = vp;
            /* Fix the storage otherwise decrRefCount will attempt to
             * remove the associated I/O job */
            j->val->storage = REDIS_VM_MEMORY;
            decrRefCount(j->val);
            redisLog(REDIS_DEBUG,
                "VM: object %s swapped out at %lld (%lld pages) (threaded)",
                (unsigned char*) j->key->ptr,
                (unsigned long long) j->page, (unsigned long long) j->pages);
            server.vm_stats_swapped_objects++;
            server.vm_stats_swapouts++;
            freeIOJob(j);
            /* Put a few more swap requests in queue if we are still
             * out of memory */
            if (trytoswap && vmCanSwapOut() &&
                zmalloc_used_memory() > server.vm_max_memory)
            {
                int more = 1;
                while(more) {
                    lockThreadedIO();
                    more = listLength(server.io_newjobs) <
                           (unsigned) server.vm_max_threads;
                    unlockThreadedIO();
                    /* Don't waste CPU time if swappable objects are rare. */
                    if (vmSwapOneObjectThreaded() == REDIS_ERR) {
                        trytoswap = 0;
                        break;
                    }
                }
            }
        }
```

创建 vmpointer 指针，分配内存并初始化 vmpointer 的 page 和 usedpages 字段。

把 vmpointer 指针赋值给 val 的指针变量

因为 val 的指针变量不再指向 val 对象了，所以调用 decrRefCount 函数减少 val 对象的引用次数，当引用次数减为 0 时则释放 val 对象占用的内存，这里不会执行到释放对象的对方，因为 iojob 对象中有对 val 的引用

释放 iojob 对象，里边会继续调用 decrRefCount，此次调用会释放 val 对象占用的内存。

最后，如果发现使用内存还是超过配置的内存且满足换出条件，则继续进行往 vm 的任务队列中添加换出任务

总结一下：
多线程换出经历了两个阶段，准备换出和进行换出。主线程和 vm 线程间也交互了两次。

多线程换入

主线程接收到一个请求，如果请求中涉及的数据在 vm 上，主线程就把需要的且在 vm 上的数据包装成一个任务放进 vm 线程的处理队列中，vm 线程处理完后会通知主线程，当这个请求涉及到的所有数据都 load 回内存后，该被阻塞的 client 将会被恢复。

vm 的多线程换入其实是阻塞式换入的补充，因为当多线程 vm 在某些情况下不能把数据换入时，多线程换入就会转变为阻塞式换入。比如说， mget 命令请求 5 个 key 的数据，有两个 key 是在虚拟内存中，此时，这个 client 将会被阻塞，等待这两个 key 换入。当这两个 key 换入后，vm 线程会通知主线程可以继续这个被阻塞的 client 了。但是，如果此时另外三个 key 又被换出时，此时就会使用阻塞式换入把这三个 key 换入，然后继续处理这个 client 的请求。

在 processCommand 函数中会调用函数 blockClientOnSwappedKeys（调用位置 redis.c：1144）判断该命令请求的数据是不是所有都在内存中，如果都在，则继续执行。如果有一个 key 是在虚拟内存中，则把换入的任务放到 vm 线程的任务队列 server.io_newjobs 中，同时把要等待换入的 key 加入到该 client 的 io_keys 链表中，并且把该 client 放到 db 的 io_keys 哈希表下（该哈希表以等待换入的 key 为键，解决冲突方式是链表）。该 client 将被阻塞，等待 vm 线程完成换入任务。如果有多个 key 需要换入，对每个 key 进行循环处理。

vm 线程读取 server.io_newjobs 的任务，然后把这个任务从 server.io_newjobs 队列中移出并放入到 server.io_processing 队列中。然后根据任务的类型处理。因为此时任务类型为换入，vm 线程将读取虚拟内存数据，把读出来的数据保存于任务的 val 字段。然后把任务从 server.io_processing 队列中移出并放入到 server.io_processed 队列中。此时 vm 线程将会发送一个字节到管道 server.io_ready_pipe_write 中。管道 server.io_ready_pipe_read 在初始化虚拟内存时就绑定了可读事件的回调函数 vmThreadedIOCompletedJob。

管道 server.io_ready_pipe_read 收到一个字节后，函数 vmThreadedIOCompletedJob 被触发，然后读取 server.io_processed 队列中的任务并将该任务从 server.io_processed 队列中删除。因为此时任务类型为换入，所以接下来的就是释放虚拟内存占用的空间，把从虚拟内存读取的数据放回内存中。然后从 db 的 io_keys 哈希表找到该 key 对应的链表（该链表下的元素是所有等待该 key 的 client），然后遍历所有 client，把该 key 从 client 的 io_keys 链表中删除。如果某个 client 的 io_keys 链表在删除该 key 后是空的，则表明该 client 要等待的所有 key 都已经换入内存中。于是把这个 client 加入到 server.io_ready_clients 链表中。

在进入事件处理前都会执行 beforeSleep 函数并对 server.io_ready_clients 链表进行处理，如果开启了虚拟内存且 server.io_ready_clients 链表不为空，则会继续处理该 client 的请求，此时该 client 的请求被恢复。

接下来看看实现的代码细节
Redis.c：1062

```
int processCommand(redisClient *c) {
    ...
```

```
    /* Exec the command */
    if (c->flags & REDIS_MULTI &&
        c->cmd->proc != execCommand && c->cmd->proc != discardCommand &&
        c->cmd->proc != multiCommand && c->cmd->proc != watchCommand)
    {
        queueMultiCommand(c);
        addReply(c,shared.queued);
    } else {
        if (server.vm_enabled && server.vm_max_threads > 0 &&
            blockClientOnSwappedKeys(c)) return REDIS_ERR;
        call(c);
    }
    return REDIS_OK;
}
```

在所有请求都需要执行的函数 processCommand 中，如果启动了多线程 vm，则进入函数 blockClientOnSwappedKeys，如果 blockClientOnSwappedKeys 函数返回 1 表示请求涉及的某些 key 需从虚拟内存换入，该 client 将会被阻塞。返回 0 表所有 key 都在内存中，可以继续执行。

我们进 blockClientOnSwappedKeys 函数探个究竟

Vm.c：1067

```
/* Is this client attempting to run a command against swapped keys?
 * If so, block it ASAP, load the keys in background, then resume it.
 *
 * The important idea about this function is that it can fail! If keys will
 * still be swapped when the client is resumed, this key lookups will
 * just block loading keys from disk. In practical terms this should only
 * happen with SORT BY command or if there is a bug in this function.
 *
 * Return 1 if the client is marked as blocked, 0 if the client can
 * continue as the keys it is going to access appear to be in memory. */
int blockClientOnSwappedKeys(redisClient *c) {
    if (c->cmd->vm_preload_proc != NULL) {
        c->cmd->vm_preload_proc(c,c->cmd,c->argc,c->argv);
    } else {
        waitForMultipleSwappedKeys(c,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 函数将需要换入到内存的 key 的任务放进 vm 线程的新任务队列 server.io_newjobs 中。同时记下任务完成时需要的一些信息，比如把需要换入的 key 放到 client 的 io_keys 链表中。

如果 client 的 io_keys 链表不为空，则表示有需要交换的 key，标识这个 client 为阻塞态，并把这个 client 的读事件从事件轮询中删除，最后返回 1，表示这个 client 将要被阻塞。

如果 client 的 io_keys 链表为空，则表示没有需要交换的 key，返回 0，表示这个 client 的请求可以继续处理。

继续接着进去 waitForMultipleSwappedKeys 函数

Vm.c：1014

```
/* Preload keys for any command with first, last and step values for
 * the command keys prototype, as defined in the command table. */
void waitForMultipleSwappedKeys(redisClient *c, struct redisCommand *cmd, int
argc, robj **argv) {
    int j, last;
    if (cmd->vm_firstkey == 0) return;
    last = cmd->vm_lastkey;
    if (last < 0) last = argc+last;
    for (j = cmd->vm_firstkey; j <= last; j += cmd->vm_keystep) {
        redisAssert(j < argc);
        waitForSwappedKey(c,argv[j]);
    }
}
```

遍历检查请求的命令对应的所有 key 是否需要换入，如果需要则做相应的处理

接着继续进入 waitForSwappedKey 函数

vm.c：946

```
/* This function makes the clinet 'c' waiting for the key 'key' to be loaded.
 * If there is not already a job loading the key, it is craeted.
 * The key is added to the io_keys list in the client structure, and also
 * in the hash table mapping swapped keys to waiting clients, that is,
 * server.io_waited_keys. */
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;
}
```

查找该 key 对应的 val，如果 val 对象是在内存中则立即返回，如果该 key 对应的 val 对象正在被换出到内存的过程中，则取消换出。

把等待换入的 key 加入到该 client 的 io_keys 链表中，用于判断该 client 是否可以恢复。当该 client 的 io_keys 链表的为空时则可以恢复。每换入一个 key 都会更新等待该 key 的 client 的 io_keys 链表。

把该 client 放到 db 的 io_keys 哈希表下以等待换入的 key 为键的链表中。当主线程收到有一个 key 被换入的通知时就可以通过这个哈希表找到等待该 key 的 client。然后就可以更新 client 的 io_keys 链表。

最后生成一个任务对象，把这个任务对象放到 vm 线程的新任务队列 server.io_newjobs。

我们接下来看看线程函数是怎样处理换入的

线程入口函数 IOThreadEntryPoint

Vm.c：801

```
void *IOThreadEntryPoint(void *arg) {
    iojob *j;
    listNode *ln;
    REDIS_NOTUSED(arg);

    pthread_detach(pthread_self());
    while(1) {
        /* Get a new job to process */
        lockThreadedIO();
        if (listLength(server.io_newjobs) == 0) {
            /* No new jobs in queue, exit. */
            redisLog(REDIS_DEBUG,"Thread %ld exiting, nothing to do",
                (long) pthread_self());
            server.io_active_threads--;
            unlockThreadedIO();
            return NULL;
        }
        ln = listFirst(server.io_newjobs);
        j = ln->value;
        listDelNode(server.io_newjobs,ln);
        /* Add the job in the processing queue */
```

```
        j->thread = pthread_self();
        listAddNodeTail(server.io_processing,j);
        ln = listLast(server.io_processing); /* We use ln later to remove it */
        unlockThreadedIO();
        redisLog(REDIS_DEBUG,"Thread %ld got a new job (type %d): %p about key
'%s'",
            (long) pthread_self(), j->type, (void*)j, (char*)j->key->ptr);

        /* Process the Job */
        if (j->type == REDIS_IOJOB_LOAD) {
            vmpointer *vp = (vmpointer*)j->id;
            j->val = vmReadObjectFromSwap(j->page,vp->vtype);
        } else if (j->type == REDIS_IOJOB_PREPARE_SWAP) {
            j->pages = rdbSavedObjectPages(j->val);
        } else if (j->type == REDIS_IOJOB_DO_SWAP) {
            if (vmWriteObjectOnSwap(j->val,j->page) == REDIS_ERR)
                j->canceled = 1;
        }

        /* Done: insert the job into the processed queue */
        redisLog(REDIS_DEBUG,"Thread %ld completed the job: %p (key %s)",
            (long) pthread_self(), (void*)j, (char*)j->key->ptr);
        lockThreadedIO();
        listDelNode(server.io_processing,ln);
        listAddNodeTail(server.io_processed,j);
        unlockThreadedIO();

        /* Signal the main thread there is new stuff to process */
        redisAssert(write(server.io_ready_pipe_write,"x",1) == 1);
    }
    return NULL; /* never reached */
}
```

把任务从 server.io_newjobs 队列中取出并放到 server.io_processing 队列中。

- 处理换入换出任务

把任务放到 server.io_processed 队列中。

往管道中写入一字节来通知主线程有任务完成，可以去 server.io_processing 队列读取已经完成的任务。

我们接着看看主线程时怎么处理管道信息和恢复 client 请求的

先看看管道可读事件的绑定

Redis.c：998

```
if (server.vm_enabled) vmInit();
```

Vm.c：42

```
void vmInit(void) {
```

```
    ...
    int pipefds[2];
    ...
    if (pipe(pipefds) == -1) {
        redisLog(REDIS_WARNING,"Unable to intialized VM: pipe(2): %s. Exiting."
            ,strerror(errno));
        exit(1);
    }
    server.io_ready_pipe_read = pipefds[0];
    server.io_ready_pipe_write = pipefds[1];
    redisAssert(anetNonBlock(NULL,server.io_ready_pipe_read) != ANET_ERR);
    ...
    if (aeCreateFileEvent(server.el, server.io_ready_pipe_read, AE_READABLE,
        vmThreadedIOCompletedJob, NULL) == AE_ERR)
        oom("creating file event");
}
```

创建管道并绑定管道的可读事件的回调函数为 vmThreadedIOCompletedJob

进去 vmThreadedIOCompletedJob 函数内部看看

Vm.c：574

```
void vmThreadedIOCompletedJob(aeEventLoop *el, int fd, void *privdata,
          int mask)
{
    ...
    while((retval = read(fd,buf,1)) == 1) {
        iojob *j;
        listNode *ln;
        struct dictEntry *de;

        redisLog(REDIS_DEBUG,"Processing I/O completed job");

        /* Get the processed element (the oldest one) */
        lockThreadedIO();
        redisAssert(listLength(server.io_processed) != 0);
        ...
        ln = listFirst(server.io_processed);
        j = ln->value;
        listDelNode(server.io_processed,ln);
        unlockThreadedIO();
        ...
        de = dictFind(j->db->dict,j->key->ptr);
        redisAssert(de != NULL);
        if (j->type == REDIS_IOJOB_LOAD) {
            redisDb *db;
```

```
            vmpointer *vp = dictGetEntryVal(de);

            /* Key loaded, bring it at home */
            vmMarkPagesFree(vp->page,vp->usedpages);
            redisLog(REDIS_DEBUG, "VM: object %s loaded from disk (threaded)",
                (unsigned char*) j->key->ptr);
            server.vm_stats_swapped_objects--;
            server.vm_stats_swapins++;
            dictGetEntryVal(de) = j->val;
            incrRefCount(j->val);
            db = j->db;
            /* Handle clients waiting for this key to be loaded. */
            handleClientsBlockedOnSwappedKey(db,j->key);
            freeIOJob(j);
            zfree(vp);
        } else if (j->type == REDIS_IOJOB_PREPARE_SWAP) {
            ...
        } else if (j->type == REDIS_IOJOB_DO_SWAP) {
            ...
        }
        processed++;
        if (processed == toprocess) return;
    }
    ...
}
```

读取管道中的字节，每一个字节代表着一个任务完成

如果任务类型是 REDIS_IOJOB_LOAD，则把从虚拟内存 load 进来的数据放回到内存。

调用 handleClientsBlockedOnSwappedKey 函数，查询判断这个 key 换入到内存中后有没有 client 是不是不再需要等待了，如果某个 client 所有等待的 key 都换入内存了，则把这个 client 放到 server.io_ready_clients 链表中。

每当一个 key 换入到内存 handleClientsBlockedOnSwappedKey 函数都会被调用（另一个主要调用点在 db.c:37）。

如果内存当时很吃紧，在判断某个对象是否在内存中时，该对象还是在内存中或者刚被换进来，但当需要读取该对象时，该对象又被换出去了或者正在被换出的过程中，redis 是怎么来处理这种情况的呢？我们来看看 reids 是怎么实现多线程换入是阻塞式换入的补充的。我们先来看看 lookupKey 函数

Db.c：9

```
robj *lookupKey(redisDb *db, robj *key) {
    dictEntry *de = dictFind(db->dict,key->ptr);
    if (de) {
        robj *val = dictGetEntryVal(de);

        /* Update the access time for the aging algorithm.
```

```
        * Don't do it if we have a saving child, as this will trigger
        * a copy on write madness. */
       if (server.bgsavechildpid == -1 && server.bgrewritechildpid == -1)
          val->lru = server.lruclock;

       if (server.vm_enabled) {
          if (val->storage == REDIS_VM_MEMORY ||
              val->storage == REDIS_VM_SWAPPING)
          {
              /* If we were swapping the object out, cancel the operation */
              if (val->storage == REDIS_VM_SWAPPING)
                 vmCancelThreadedIOJob(val);
          } else {
              int notify = (val->storage == REDIS_VM_LOADING);

              /* Our value was swapped on disk. Bring it at home. */
              redisAssert(val->type == REDIS_VMPOINTER);
              val = vmLoadObject(val);
              dictGetEntryVal(de) = val;

              /* Clients blocked by the VM subsystem may be waiting for
               * this key... */
              if (notify) handleClientsBlockedOnSwappedKey(db,key);
          }
       }
       return val;
    } else {
       return NULL;
    }
}
```

当执行到 lookupKey 函数时，表明要换入的对象都已经换入了。

如果启动了多线程 vm 并且换出这个 val 的任务（该 val 在判断是否在内存中时还在内存中或者刚被换进来）正在执行中，这个任务将会被取消掉。

如果发现查找 key 时发现该 key 对应的 val（该 val 在判断是否在内存中时还在内存中或者刚被换进来）在虚拟内存中时，此时 redis 将使用阻塞式 vm 直接去把相关对象换入

这就是上边说到的“vm 的多线程换入其实是阻塞式换入的补充，因为当多线程 vm 在某些情况下不能把数据换入时，多线程换入就会转变为阻塞式换入。”

我们接着继续往下跟，进入 handleClientsBlockedOnSwappedKey 函数

Vm.c：1134

```
/* Every time we now a key was loaded back in memory, we handle clients
 * waiting for this key if any. */
void handleClientsBlockedOnSwappedKey(redisDb *db, robj *key) {
    struct dictEntry *de;
```

```
    list *l;
    listNode *ln;
    int len;

    de = dictFind(db->io_keys,key);
    if (!de) return;

    l = dictGetEntryVal(de);
    len = listLength(l);
    /* Note: we can't use something like while(listLength(l)) as the list
     * can be freed by the calling function when we remove the last element. */
    while (len--) {
        ln = listFirst(l);
        redisClient *c = ln->value;

        if (dontWaitForSwappedKey(c,key)) {
            /* Put the client in the list of clients ready to go as we
             * loaded all the keys about it. */
            listAddNodeTail(server.io_ready_clients,c);
        }
    }
}
```

从 db->io_keys 哈希表中找出所有等待这个 key 的 client

对所有 client 调用 dontWaitForSwappedKey 函数，如果该 client 要等待的所有 key 都换入了则返回 1，否则返回 0。

如果该 client 等待的所有 key 都换入了，则把这个 client 加入到 server.io_ready_clients 链表中。

好吧，有点深了，坚持一下下，很快就到底了。

进入 dontWaitForSwappedKey 函数

Vm.c：1095

```
/* Remove the 'key' from the list of blocked keys for a given client.
 *
 * The function returns 1 when there are no longer blocking keys after
 * the current one was removed (and the client can be unblocked). */
int dontWaitForSwappedKey(redisClient *c, robj *key) {
    list *l;
    listNode *ln;
    listIter li;
    struct dictEntry *de;

    /* The key object might be destroyed when deleted from the c->io_keys
     * list (and the "key" argument is physically the same object as the
     * object inside the list), so we need to protect it. */
```

```
    incrRefCount(key);

    /* Remove the key from the list of keys this client is waiting for. */
    listRewind(c->io_keys,&li);
    while ((ln = listNext(&li)) != NULL) {
        if (equalStringObjects(ln->value,key)) {
            listDelNode(c->io_keys,ln);
            break;
        }
    }
    redisAssert(ln != NULL);

    /* Remove the client form the key => waiting clients map. */
    de = dictFind(c->db->io_keys,key);
    redisAssert(de != NULL);
    l = dictGetEntryVal(de);
    ln = listSearchKey(l,c);
    redisAssert(ln != NULL);
    listDelNode(l,ln);
    if (listLength(l) == 0)
        dictDelete(c->db->io_keys,key);

    decrRefCount(key);
    return listLength(c->io_keys) == 0;
}
```

从 client 的 io_keys 链表中删除这个刚换入的 key

从 db 的 io_keys 哈希表中以这个 key 为索引的链表摘除该 client。链表中的元素是等待这个 key 的所有 client。如果最后这个链表为空时，则把这个以 key 为索引的链表也删除。（注意：在删除过程中，vm 线程有可能往这里添加 client）

回到 dontWaitForSwappedKey 函数的上层调用处 handleClientsBlockedOnSwappedKey 中，我们知道如果该 client 等待的所有 key 都换入了，则把这个 client 加入到 server.io_ready_clients 链表中。然后 redis 又怎么继续处理这个 client 呢？

我们来看看 beforeSleep 函数，beforeSleep 函数是在进入事件处理前都会被调用的函数

Redis.c：691

```
/* This function gets called every time Redis is entering the
 * main loop of the event driven library, that is, before to sleep
 * for ready file descriptors. */
void beforeSleep(struct aeEventLoop *eventLoop) {
    REDIS_NOTUSED(eventLoop);
    listNode *ln;
    redisClient *c;

    /* Awake clients that got all the swapped keys they requested */
```

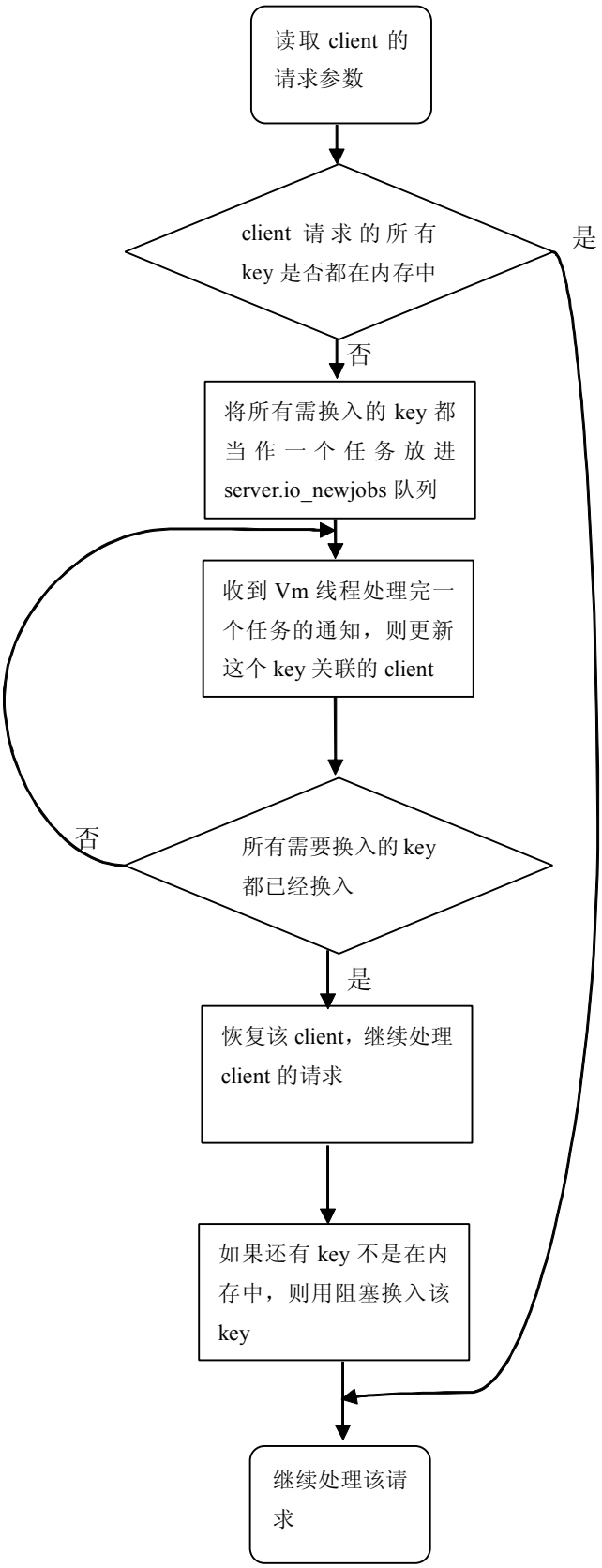
```
    if (server.vm_enabled && listLength(server.io_ready_clients)) {
        listIter li;

        listRewind(server.io_ready_clients,&li);
        while((ln = listNext(&li))) {
            c = ln->value;
            struct redisCommand *cmd;

            /* Resume the client. */
            listDelNode(server.io_ready_clients,ln);
            c->flags &= (~REDIS_IO_WAIT);
            server.vm_blocked_clients--;
            aeCreateFileEvent(server.el, c->fd, AE_READABLE,
                readQueryFromClient, c);
            cmd = lookupCommand(c->argv[0]->ptr);
            redisAssert(cmd != NULL);
            call(c);
            resetClient(c);
            /* There may be more data to process in the input buffer. */
            if (c->querybuf && sdslen(c->querybuf) > 0) {
                server.current_client = c;
                processInputBuffer(c);
                server.current_client = NULL;
            }
        }
    }
    ...
}
```

遍历 server.io_ready_clients 链表，恢复所有被阻塞的 client，让它们继续执行。

到次为止，多线程换入的整个流程就完了，总结一下


读取 client 的请求参数
client 请求的所有 key 是否都在内存中
是
否
将所有需换入的 key 都当作一个任务放进 server.io_newjobs 队列
收到 Vm 线程处理完一个任务的通知，则更新这个 key 关联的 client
否
所有需要换入的 key 都已经换入
是
恢复该 client，继续处理 client 的请求
如果还有 key 不是在内存中，则用阻塞换入该 key
继续处理该请求

多线程取消换出换入

在某些情况下需要取消多线程的换入换出。比如，正在换出某个 key 时突然有一个 client 来请求这个 key（vm.c：965），此时则需要取消多线程的换出；或者需要尽快换入时发现还处于多线程的换入处理中（db.c：32→vm.c：338），此时则需要换出。

我们也来看下多线程取消换出换入时是怎样实现的

Vm.c：718

```
/* Remove the specified object from the threaded I/O queue if still not
 * processed, otherwise make sure to flag it as canceled. */
void vmCancelThreadedIOJob(robj *o) {
    list *lists[3] = {
        server.io_newjobs,      /* 0 */
        server.io_processing,   /* 1 */
        server.io_processed     /* 2 */
    };
    int i;

    redisAssert(o->storage    ==    REDIS_VM_LOADING    ||    o->storage    ==
REDIS_VM_SWAPPING);
again:
    lockThreadedIO();
    /* Search for a matching object in one of the queues */
    for (i = 0; i < 3; i++) {
        listNode *ln;
        listIter li;

        listRewind(lists[i],&li);
        while ((ln = listNext(&li)) != NULL) {
            iojob *job = ln->value;

            if (job->canceled) continue; /* Skip this, already canceled. */
            if (job->id == o) {
                redisLog(REDIS_DEBUG,"*** CANCELED %p (key %s) (type %d) (LIST ID
%d)\n",
                    (void*)job, (char*)job->key->ptr, job->type, i);
                /* Mark the pages as free since the swap didn't happened
                 * or happened but is now discarded. */
                if (i != 1 && job->type == REDIS_IOJOB_DO_SWAP)
                    vmMarkPagesFree(job->page,job->pages);
                /* Cancel the job. It depends on the list the job is
                 * living in. */
```

```
            switch(i) {
            case 0: /* io_newjobs */
                /* If the job was yet not processed the best thing to do
                 * is to remove it from the queue at all */
                freeIOJob(job);
                listDelNode(lists[i],ln);
                break;
            case 1: /* io_processing */
                /* Oh Shi- the thread is messing with the Job:
                 *
                 * Probably it's accessing the object if this is a
                 * PREPARE_SWAP or DO_SWAP job.
                 * If it's a LOAD job it may be reading from disk and
                 * if we don't wait for the job to terminate before to
                 * cancel it, maybe in a few microseconds data can be
                 * corrupted in this pages. So the short story is:
                 *
                 * Better to wait for the job to move into the
                 * next queue (processed)... */

                /* We try again and again until the job is completed. */
                unlockThreadedIO();
                /* But let's wait some time for the I/O thread
                 * to finish with this job. After all this condition
                 * should be very rare. */
                usleep(1);
                goto again;
            case 2: /* io_processed */
                /* The job was already processed, that's easy...
                 * just mark it as canceled so that we'll ignore it
                 * when processing completed jobs. */
                job->canceled = 1;
                break;
            }
            /* Finally we have to adjust the storage type of the object
             * in order to "UNDO" the operaiton. */
            if (o->storage == REDIS_VM_LOADING)
                o->storage = REDIS_VM_SWAPPED;
            else if (o->storage == REDIS_VM_SWAPPING)
                o->storage = REDIS_VM_MEMORY;
            unlockThreadedIO();
            redisLog(REDIS_DEBUG,"*** DONE");
            return;
        }
```

```
        }
    }
    unlockThreadedIO();
    printf("Not found: %p\n", (void*)o);
    redisAssert(1 != 1); /* We should never reach this */
}
```

遍历 server.io_newjobs，server.io_processing，server.io_processed 三个队列中的所有元素，查找出要取消换出的 key 是在哪个队列中。

如果当前任务不是在 server.io_processing 队列中且任务类型是 REDIS_IOJOB_DO_SWAP，则先把之前标识为使用状态的虚拟内存还原为空闲态。

如果是在 server.io_newjobs 中，直接把这个任务删除即可

如果是在 server.io_processing 中，则让主线程挂起，等待 vm 线程处理完这个任务，然后继续循环判断该任务是已经到了 server.io_processed 队列

如果是在 server.io_processed 中，则直接标识该任务的 canceled 为 1 即可，当主线程接收到任务完成的通知时，如果判断到该任务的 canceled 为 1 会直接释放这个任务对象。

接下来看看当主线程接收到任务完成的通知时且该任务的 canceled 为 1 的处理

Vm.c：605

```
        /* If this job is marked as canceled, just ignore it */
        if (j->canceled) {
            freeIOJob(j);
            continue;
        }
```

Vm.c：548

```
void freeIOJob(iojob *j) {
    if ((j->type == REDIS_IOJOB_PREPARE_SWAP ||
        j->type == REDIS_IOJOB_DO_SWAP ||
        j->type == REDIS_IOJOB_LOAD) && j->val != NULL)
    {
         /* we fix the storage type, otherwise decrRefCount() will try to
          * kill the I/O thread Job (that does no longer exists). */
        if (j->val->storage == REDIS_VM_SWAPPING)
            j->val->storage = REDIS_VM_MEMORY;
        decrRefCount(j->val);
    }
    decrRefCount(j->key);
    zfree(j);
}
```

我们看到如果该任务被标识为 `canceled` 后调用 `freeIOJob` 函数进行释放 `iojob` 对象。

rdb（redis database）

rdb 是 redis 保存内存数据到磁盘数据的其中一种方式（另一种是 AOF）。Rdb 的主要原理就是在某个时间点把内存中的所有数据的快照保存一份到磁盘上。在条件达到时通过 fork 一

个子进程把内存中的数据写到一个临时文件中来实现保存数据快照。在所有数据写完后再把这个临时文件用原子函数 rename(2)重命名为目标 rdb 文件。这种实现方式充分利用 fork 的 copy on write。
另外一种是通过 save 命令主动触发保存数据快照，这种是阻塞式的，即不会通过生成子进程来进行数据集快照的保存。

相关配置

save <seconds> <changes>
经过多少秒且多少个 key 有改变就进行，可以配置多个，只要有一个满足就进行保存数据快照到磁盘

rdbcompression yes
保存数据到 rdb 文件时是否进行压缩，如果不想可以配置成'no'，默认是'yes'，因为压缩可以减少 I/O，当然，压缩需要消耗一些 cpu 资源。

dbfilename dump.rdb
快照文件名

dir ./
快照文件所在的目录，同时也是 AOF 文件所在的目录

Rdb 文件格式

注：本节所说的类型，值在没有特别标明的情况下都是针对 rdb 文件来说的

Rdb 文件的整体格式

文件签名 | 版本号 | 类型 | 值 | 类型 | 值 | ... | 类型 | 值
注：竖线和空格是为了便于阅读而加入的，rdb 文件中是没有竖线和空格分隔的

文件签名是字符串：REDIS
版本号是字符串：0002
类型是指值的类型，redis 值的类型有很多种，下边一一介绍
值是对应的类型下的值，不同类型的值格式不一样。这里的值包含了 redis 中的 key 与 val。
而不是单指 redis 中 val。

REDIS_SELECTDB 类型与 REDIS_EOF 类型

REDIS_SELECTDB 类型：对应的值是 redis db 的编号，从 0 开始到比 db 数小 1 的数值。redis

中可以配置 db 数，每个 key 只属于一个 db。

存储 redis db 的编号时使用的是存储长度时使用的格式，为了尽量压缩 rdb 文件，存储长度使用的字节数是不一样的，具体见下边 rdb 中长度的存储

REDIS_EOF 类型：没有对应的值。rdb 文件的结束符。

把这 REDIS_SELECTDB 类型和 REDIS_EOF 类型代入到上边的 rdb 文件的格式中，那么 rdb 文件的整体格式变成为：

文件签名 | 版本号 | REDIS_SELECTDB 类型 | db 编号 | 类型 | 值 | ... | REDIS_SELECTD 类型 | db 编号 | 类型 | 值 | ... | REDIS_EOF 类型

每个 db 编号后边到下一个 REDIS_SELECTDB 类型出现之前的数据都是该 db 下边的 key 和 value 的数据

相关代码

Rdb.c：394

```
int rdbSave(char *filename) {
    ...
    fp = fopen(tmpfile,"w");
    if (!fp) {
        redisLog(REDIS_WARNING, "Failed saving the DB: %s", strerror(errno));
        return REDIS_ERR;
    }
    if (fwrite("REDIS0002",9,1,fp) == 0) goto werr;
    for (j = 0; j < server.dbnum; j++) {
        ...
        /* Write the SELECT DB opcode */
        if (rdbSaveType(fp,REDIS_SELECTDB) == -1) goto werr;
        if (rdbSaveLen(fp,j) == -1) goto werr;

        /* Iterate this DB writing every entry */
        while((de = dictNext(di)) != NULL) {
            ...
            initStaticStringObject(key,keystr);
            expiretime = getExpire(db,&key);

            /* Save the expire time */
            if (expiretime != -1) {
                /* If this key is already expired skip it */
                if (expiretime < now) continue;
                if (rdbSaveType(fp,REDIS_EXPIRETIME) == -1) goto werr;
                if (rdbSaveTime(fp,expiretime) == -1) goto werr;
            }
            /* Save the key and associated value. This requires special
             * handling if the value is swapped out. */
            if (!server.vm_enabled || o->storage == REDIS_VM_MEMORY ||
```

```
                                o->storage == REDIS_VM_SWAPPING) {
                int otype = getObjectSaveType(o);

                /* Save type, key, value */
                if (rdbSaveType(fp,otype) == -1) goto werr;
                if (rdbSaveStringObject(fp,&key) == -1) goto werr;
                if (rdbSaveObject(fp,o) == -1) goto werr;
            } else {
                /* REDIS_VM_SWAPPED or REDIS_VM_LOADING */
                robj *po;
                /* Get a preview of the object in memory */
                po = vmPreviewObject(o);
                /* Save type, key, value */
                if (rdbSaveType(fp,getObjectSaveType(po)) == -1)
                    goto werr;
                if (rdbSaveStringObject(fp,&key) == -1) goto werr;
                if (rdbSaveObject(fp,po) == -1) goto werr;
                /* Remove the loaded object from memory */
                decrRefCount(po);
            }
        }
        dictReleaseIterator(di);
    }
    /* EOF opcode */
    if (rdbSaveType(fp,REDIS_EOF) == -1) goto werr;
    ...
}
```

Rdb 中长度的存储

Redis 为了尽量压缩 rdb 文件真是费尽心思，先来看看 redis 为了压缩使用的长度存储。长度主要用在字符串长度，链表长度，hash 表的大小存储上。

Redis 把长度的存储分为四种，最左边字节的从左到右的前两位用于区分长度的存储类型。

类型位表示	类型整型表示	占用字节数	类型解析
00	0	1	当长度能用 6 位表示使用此类型
01	1	2	当长度不能用 6 位表示且能用 14 位表示使用此类型
10	2	5	当长度不能用 14 位表示且能用 32 位表示使用此类型

相关代码

Rdb.c：31

```
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;
}
```

也许你发现了，上边的表格中只有 3 种，还有一种哪去了呢？
把这种特别放开是因为这种比较特殊

类型位表示	类型整型表示	字节后 6 位含义	类型解析
11	3	编码类型	如果字符串是通过编码后存储的，则存储长度的类型的位表示为 11，然后根据后 6 位的编码类型来确定怎样读取和解析接下来的数据

是不是觉得这种长度类型很奇怪，为什么要这样做？
Redis 在两种情况下需要对存储的内容进行编码
1：把字符串转成整数存储。
比如：‘-100’需要 4 个字节存储，转换整数只需要一个字节
相关函数 rdbTryIntegerEncoding（rdb.c：88）
2：使用 lzf 算法压缩字符串。
相关函数 lzf_compress（lzf_c.c：99），lzf 的算法解释见 lzf 字符串压缩算法

当 redis 使用这两种编码对字符串进行编码时，在读取时需要区分改字符串有没有被编码过，对编码过的字符串需要特别处理，因为长度信息是存储在字符串的前面得，所以可以通过在存储长度的位置上加入编码类型的信息。

我们来看看相关代码
Rdb.c：557

```
uint32_t rdbLoadLen(FILE *fp, int *isencoded) {
    unsigned char buf[2];
    uint32_t len;
    int type;

    if (isencoded) *isencoded = 0;
    if (fread(buf,1,1,fp) == 0) return REDIS_RDB_LENERR;
    type = (buf[0]&0xC0)>>6;
    if (type == REDIS_RDB_6BITLEN) {
        /* Read a 6 bit len */
        return buf[0]&0x3F;
    } else if (type == REDIS_RDB_ENCVAL) {
        /* Read a 6 bit len encoding type */
        if (isencoded) *isencoded = 1;
        return buf[0]&0x3F;
    } else if (type == REDIS_RDB_14BITLEN) {
        /* Read a 14 bit len */
        if (fread(buf+1,1,1,fp) == 0) return REDIS_RDB_LENERR;
        return ((buf[0]&0x3F)<<8)|buf[1];
    } else {
        /* Read a 32 bit len */
        if (fread(&len,4,1,fp) == 0) return REDIS_RDB_LENERR;
        return ntohl(len);
    }
}
```

我们可以看到，在读取 rdb 文件时，当发现长度类型是 REDIS_RDB_ENCVAL，把编码类型返回。

我们来看看知道编码类型后的处理

Rdb.c：633

```
robj *rdbGenericLoadStringObject(FILE*fp, int encode) {
    int isencoded;
    uint32_t len;
    sds val;

    len = rdbLoadLen(fp,&isencoded);
    if (isencoded) {
        switch(len) {
        case REDIS_RDB_ENC_INT8:
        case REDIS_RDB_ENC_INT16:
        case REDIS_RDB_ENC_INT32:
            return rdbLoadIntegerObject(fp,len,encode);
        case REDIS_RDB_ENC_LZF:
            return rdbLoadLzfStringObject(fp);
```

```
        default:
            redisPanic("Unknown RDB encoding type");
        }
    }

    if (len == REDIS_RDB_LENERR) return NULL;
    val = sdsnewlen(NULL,len);
    if (len && fread(val,len,1,fp) == 0) {
        sdsfree(val);
        return NULL;
    }
    return createObject(REDIS_STRING,val);
}
```

读取长度
如果长度类型是有编码信息的，则根据编码类型进行读取
如果长度类型是有效长度，则根据长度信息读取字符串

REDIS_EXPIRETIME 类型

如果一个 key 被 expire 设置过，那么在该 key 与 value 的前面会有一个 REDIS_EXPIRETIME 类型与其对应的值。
REDIS_EXPIRETIME 类型对应的值是过期时间点的 timestamp
REDIS_EXPIRETIME 类型与其值是可选的，不是必须的，只有被 expire 设置过的 key 才有这个值

假设有一个 key 被 expire 命令设置过，把这 REDIS_EXPIRETIME 类型代入到上边的 rdb 文件的格式中，那么 rdb 文件的整体格式变成为：
文件签名 | 版本号 | REDIS_SELECTDB 类型 | db 编号 | REDIS_EXPIRETIME 类型 | timestamp | 类型 | 值 | ... | REDIS_SELECTD 类型 | db 编号 | 类型 | 值 | ... | REDIS_EOF 类型

数据类型

数据类型主要有以下类型：
REDIS_STRING 类型
REDIS_LIST 类型
REDIS_SET 类型
REDIS_ZSET 类型
REDIS_HASH 类型
REDIS_VMPOINTER 类型
REDIS_HASH_ZIPMAP 类型
REDIS_LIST_ZIPLIST 类型

REDIS_SET_INTSET 类型
REDIS_ZSET_ZIPLIST 类型

其中 REDIS_HASH_ZIPMAP ， REDIS_LIST_ZIPLIST ， REDIS_SET_INTSET 和 REDIS_ZSET_ZIPLIST 这四种数据类型都是只在 rdb 文件中才有的类型，其他的数据类型其实就是 val 对象中 type 字段存储的值。

下边以 REDIS_STRING 类型和 REDIS_LIST 类型为例进行详解，其他类型都类似

REDIS_STRING 类型

假设 rdb 文件中有一个值是 REDIS_STRING 类型，比如执行了一个 set mykey myval 命令，则在 rdb 文件表示为
REDIS_STRING 类型 | 值
其中值包含了 key 的长度，key 的值，val 的长度和 val 的值，把 REDIS_STRING 类型值的格式代入得
REDIS_STRING 类型 | keylen | mykey | vallen | myval
长度的存储格式见 rdb 中长度的存储

REDIS_LIST 类型

List
REDIS_LIST | listlen | len | value | len | value
Listlen 是链表长度
Len 是链表结点的值 value 的长度
Value 是链表结点的值
Ziplist
REDIS_ENCODING_ZIPLIST | ziplist
Ziplist 就是通过字符串来实现的，直接将其存储于 rdb 文件中即可

快照保存

我们接下来看看具体实现细节
不管是触发条件满足后通过 fork 子进程来保存快照还是通过 save 命令来触发，其实都是调用的同一个函数 rdbSave（rdb.c：394）。

先来看看触发条件满足后通过 fork 子进程的实现保存快照的的实现
在每 100ms 调用一次的 serverCron 函数中会对快照保存的条件进行检查，如果满足了则进行快照保存
Redis.c：604

```
/* 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);
         }
          ...
         updateDictResizePolicy();
      }
   } else {
       time_t now = time(NULL);

      /* If there is not a background saving in progress check if
       * we have to save now */
       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;
          }
       }
        ...
}
```

如果后端有写 rdb 的子进程或者写 aof 的子进程，则检查 rdb 子进程是否退出了，如果退出了则进行一些收尾处理，比如更新脏数据计数 server.dirty 和最近快照保存时间 server.lastsave。

如果后端没有写 rdb 的子进程且没有写 aof 的子进程，则判断下是否有触发写 rdb 的条件满足了，如果有条件满足，则通过调用 rdbSaveBackground 函数进行快照保存。

跟着进 rdbSaveBackground 函数里边看看

Rdb.c：499

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

   if (server.bgsavechildpid != -1) return REDIS_ERR;
   if (server.vm_enabled) waitEmptyIOJobsQueue();
   server.dirty_before_bgsave = server.dirty;
   start = ustime();
   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 */
        server.stat_fork_time = ustime()-start;
        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; /* unreached */
}
```

对是否已经有写 rdb 的子进程进行了判断，如果已经有保存快照的子进程，则返回错误。

如果启动了虚拟内存，则等待所有处理换出换入的任务线程退出，如果还有 vm 任务在处理就会一直循环等待。一直到所有换入换出任务都完成且所有 vm 线程退出。

保存当前的脏数据计数，当快照保存完后用于更新当前的脏数据计数（见函数 backgroundSaveDoneHandler，rdb.c：1062）

记下当前时间，用于统计 fork 一个进程需要的时间

Fork 一个字进程，子进程调用 rdbSave 进行快照保存

父进程统计 fork 一个子进程消耗的时间： server.stat_fork_time = ustime()-start，这个统计可以通过 info 命令获得。

保存子进程 ID 和更新增量重哈希的策略，即此时不应该再进行增量重哈希，不然大量 key 的改变可能导致 fork 的 copy-on-write 进行大量的写。

到了这里我们知道，rdb 的快照保存是通过函数 rdbSave 函数（rdb.c：394）来实现的。其实 save 命令也是通过调用这个函数来实现的。我们来简单看看

Db.c：323

```
void saveCommand(redisClient *c) {
    if (server.bgsavechildpid != -1) {
        addReplyError(c,"Background save already in progress");
        return;
    }
    if (rdbSave(server.dbfilename) == REDIS_OK) {
        addReply(c,shared.ok);
```

```
    } else {
        addReply(c,shared.err);
    }
```

最后我们进 rdbSave 函数看看

rdb.c：394

```
int rdbSave(char *filename) {
    ...
    /* Wait for I/O therads to terminate, just in case this is a
     * foreground-saving, to avoid seeking the swap file descriptor at the
     * same time. */
    if (server.vm_enabled)
        waitEmptyIOJobsQueue();

    snprintf(tmpfile,256,"temp-%d.rdb", (int) getpid());
    fp = fopen(tmpfile,"w");
    if (!fp) {
        redisLog(REDIS_WARNING, "Failed saving the DB: %s", strerror(errno));
        return REDIS_ERR;
    }
    if (fwrite("REDIS0002",9,1,fp) == 0) goto werr;
    for (j = 0; j < server.dbnum; j++) {
        redisDb *db = server.db+j;
        dict *d = db->dict;
        if (dictSize(d) == 0) continue;
        di = dictGetSafeIterator(d);
        if (!di) {
            fclose(fp);
            return REDIS_ERR;
        }

        /* Write the SELECT DB opcode */
        if (rdbSaveType(fp,REDIS_SELECTDB) == -1) goto werr;
        if (rdbSaveLen(fp,j) == -1) goto werr;

        /* Iterate this DB writing every entry */
        while((de = dictNext(di)) != NULL) {
            sds keystr = dictGetEntryKey(de);
            robj key, *o = dictGetEntryVal(de);
            time_t expiretime;

            initStaticStringObject(key,keystr);
            expiretime = getExpire(db,&key);
```

```
        /* Save the expire time */
        if (expiretime != -1) {
            /* If this key is already expired skip it */
            if (expiretime < now) continue;
            if (rdbSaveType(fp,REDIS_EXPIRETIME) == -1) goto werr;
            if (rdbSaveTime(fp,expiretime) == -1) goto werr;
        }
        /* Save the key and associated value. This requires special
         * handling if the value is swapped out. */
        if (!server.vm_enabled || o->storage == REDIS_VM_MEMORY ||
                                  o->storage == REDIS_VM_SWAPPING) {
            int otype = getObjectSaveType(o);

            /* Save type, key, value */
            if (rdbSaveType(fp,otype) == -1) goto werr;
            if (rdbSaveStringObject(fp,&key) == -1) goto werr;
            if (rdbSaveObject(fp,o) == -1) goto werr;
        } else {
            /* REDIS_VM_SWAPPED or REDIS_VM_LOADING */
            robj *po;
            /* Get a preview of the object in memory */
            po = vmPreviewObject(o);
            /* Save type, key, value */
            if (rdbSaveType(fp,getObjectSaveType(po)) == -1)
                goto werr;
            if (rdbSaveStringObject(fp,&key) == -1) goto werr;
            if (rdbSaveObject(fp,po) == -1) goto werr;
            /* Remove the loaded object from memory */
            decrRefCount(po);
        }
    }
    dictReleaseIterator(di);
}
/* EOF opcode */
if (rdbSaveType(fp,REDIS_EOF) == -1) goto werr;

/* Make sure data will not remain on the OS's output buffers */
fflush(fp);
fsync(fileno(fp));
fclose(fp);

/* Use RENAME to make sure the DB file is changed atomically only
 * if the generate DB file is ok. */
if (rename(tmpfile,filename) == -1) {
```

```
        redisLog(REDIS_WARNING,"Error moving temp DB file on the final destination:
%s", strerror(errno));
        unlink(tmpfile);
        return REDIS_ERR;
    }
    redisLog(REDIS_NOTICE,"DB saved on disk");
    server.dirty = 0;
    server.lastsave = time(NULL);
    return REDIS_OK;

werr:
    fclose(fp);
    unlink(tmpfile);
    redisLog(REDIS_WARNING,"Write error saving DB on disk: %s", strerror(errno));
    if (di) dictReleaseIterator(di);
    return REDIS_ERR;
}
```

对是否有 vm 线程进行再次判断，因为如果是通过 save 命令过来的是没有判断过 vm 线程的。

创建并打开临时文件

- 写入文件签名“REDIS”和版本号“0002”

遍历所有 db 中的所有 key

对每个 key，先判断是否设置了 expireTime，如果设置了，则保存 expireTime 到 rdb 文件中。然后判断该 key 对应的 value 是否则内存中，如果是在内存中，则取出来写入到 rdb 文件中保存，如果被换出到虚拟内存了，则从虚拟内存读取然后写入到 rdb 文件中。

不同类型有有不同的存储格式，详细见 rdb 文件格式

最后写入 rdb 文件的结束符

关闭文件并重命名临时文件名到正式文件名

更新脏数据计数 server.dirty 为 0 和最近写 rdb 文件的时间 server.lastsave 为当前时间，这个只是在通过 save 命令触发的情况下有用。因为如果是通过 fork 一个子进程来写 rdb 文件的，更新无效，因为更新的是子进程的数据。

如果是通过 fork 一个子进程来写 rdb 文件（即不是通过 save 命令触发的），在写 rdb 文件的过程中，可能又有一些数据被更改了，那此时的脏数据计数 server.dirty 怎么更新呢？ redis 是怎样处理的呢?

我们来看看写 rdb 的子进程推出时得处理

Redis.c：605

```
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();
    }
}
```

如果捕捉到写 rdb 文件的子进程退出，则调用 backgroundSaveDoneHandler 进行处理

接着看看 backgroundSaveDoneHandler 函数

Rdb.c：1062

```
void backgroundSaveDoneHandler(int statloc) {
    int exitcode = WEXITSTATUS(statloc);
    int bysignal = WIFSIGNALED(statloc);

    if (!bysignal && exitcode == 0) {
        redisLog(REDIS_NOTICE,
            "Background saving terminated with success");
        server.dirty = server.dirty - server.dirty_before_bgsave;
        server.lastsave = time(NULL);
    } else if (!bysignal && exitcode != 0) {
        redisLog(REDIS_WARNING, "Background saving error");
    } else {
        redisLog(REDIS_WARNING,
            "Background saving terminated by signal %d", WTERMSIG(statloc));
        rdbRemoveTempFile(server.bgsavechildpid);
    }
    server.bgsavechildpid = -1;
    /* Possibly there are slaves waiting for a BGSAVE in order to be served
     * (the first stage of SYNC is a bulk transfer of dump.rdb) */
    updateSlavesWaitingBgsave(exitcode == 0 ? REDIS_OK : REDIS_ERR);
}
```

更新脏数据计数 server.dirty 为 0 和最近写 rdb 文件的时间 server.lastsave 为当前时间

唤醒因为正在保存快照而等待的 slave，关于 slave 的具体内容，见 replication

快照导入

当 redis 因为停电或者某些原因挂掉了，此时重启 redis 时，我们就需要从 rdb 文件中读取快照文件，把保存到 rdb 文件中的数据重新导入到内存中。

先来看看启动时对快照导入的处理

Redis.c：1717

```
    if (server.appendonly) {
        if (loadAppendOnlyFile(server.appendfilename) == REDIS_OK)
            redisLog(REDIS_NOTICE,"DB  loaded  from  append  only  file:  %ld
seconds",time(NULL)-start);
```

```
    } else {
        if (rdbLoad(server.dbfilename) == REDIS_OK) {
            redisLog(REDIS_NOTICE,"DB loaded from disk: %ld seconds",
                time(NULL)-start);
        } else if (errno != ENOENT) {
            redisLog(REDIS_WARNING,"Fatal error loading the DB. Exiting.");
            exit(1);
        }
    }
```

如果保存了 AOF 文件，则使用 AOF 文件来恢复数据，AOF 的具体内容见 AOF
如果没有 AOF，则使用 rdb 文件恢复数据，调用 rdbLoad 函数

接着看看 rdbLoad 函数

Rdb.c：929

```
int rdbLoad(char *filename) {
    ...
    fp = fopen(filename,"r");
    if (!fp) {
        errno = ENOENT;
        return REDIS_ERR;
    }
    if (fread(buf,9,1,fp) == 0) goto eoferr;
    buf[9] = '\0';
    if (memcmp(buf,"REDIS",5) != 0) {
        fclose(fp);
        redisLog(REDIS_WARNING,"Wrong signature trying to load DB from file");
        errno = EINVAL;
        return REDIS_ERR;
    }
    rdbver = atoi(buf+5);
    if (rdbver < 1 || rdbver > 2) {
        fclose(fp);
        redisLog(REDIS_WARNING,"Can't handle RDB format version %d",rdbver);
        errno = EINVAL;
        return REDIS_ERR;
    }

    startLoading(fp);
    while(1) {
        robj *key, *val;
        int force_swapout;

        expiretime = -1;
```

```
        /* Serve the clients from time to time */
        if (!(loops++ % 1000)) {
            loadingProgress(ftello(fp));
            aeProcessEvents(server.el, AE_FILE_EVENTS|AE_DONT_WAIT);
        }

        /* Read type. */
        if ((type = rdbLoadType(fp)) == -1) goto eoferr;
        if (type == REDIS_EXPIRETIME) {
            if ((expiretime = rdbLoadTime(fp)) == -1) goto eoferr;
            /* We read the time so we need to read the object type again */
            if ((type = rdbLoadType(fp)) == -1) goto eoferr;
        }
        if (type == REDIS_EOF) break;
        /* Handle SELECT DB opcode as a special case */
        if (type == REDIS_SELECTDB) {
            if ((dbid = rdbLoadLen(fp,NULL)) == REDIS_RDB_LENERR)
                goto eoferr;
            if (dbid >= (unsigned)server.dbnum) {
                redisLog(REDIS_WARNING,"FATAL: Data file was created with a Redis
server configured to handle more than %d databases. Exiting\n", server.dbnum);
                exit(1);
            }
            db = server.db+dbid;
            continue;
        }
        /* Read key */
        if ((key = rdbLoadStringObject(fp)) == NULL) goto eoferr;
        /* Read value */
        if ((val = rdbLoadObject(type,fp)) == NULL) goto eoferr;
        /* Check if the key already expired. This function is used when loading
         * an RDB file from disk, either at startup, or when an RDB was
         * received from the master. In the latter case, the master is
         * responsible for key expiry. If we would expire keys here, the
         * snapshot taken by the master may not be reflected on the slave. */
        if (server.masterhost == NULL && expiretime != -1 && expiretime < now) {
            decrRefCount(key);
            decrRefCount(val);
            continue;
        }
        /* Add the new object in the hash table */
        dbAdd(db,key,val);

        /* Set the expire time if needed */
```

```
    if (expiretime != -1) setExpire(db,key,expiretime);

    /* Handle swapping while loading big datasets when VM is on */

    /* If we detecter we are hopeless about fitting something in memory
     * we just swap every new key on disk. Directly...
     * Note that's important to check for this condition before resorting
     * to random sampling, otherwise we may try to swap already
     * swapped keys. */
    if (swap_all_values) {
       dictEntry *de = dictFind(db->dict,key->ptr);

       /* de may be NULL since the key already expired */
       if (de) {
          vmpointer *vp;
          val = dictGetEntryVal(de);

          if (val->refcount == 1 &&
              (vp = vmSwapObjectBlocking(val)) != NULL)
             dictGetEntryVal(de) = vp;
       }
       decrRefCount(key);
       continue;
    }
    decrRefCount(key);

    /* Flush data on disk once 32 MB of additional RAM are used... */
    force_swapout = 0;
    if ((zmalloc_used_memory() - server.vm_max_memory) > 1024*1024*32)
       force_swapout = 1;

    /* If we have still some hope of having some value fitting memory
     * then we try random sampling. */
    if (!swap_all_values && server.vm_enabled && force_swapout) {
       while (zmalloc_used_memory() > server.vm_max_memory) {
          if (vmSwapOneObjectBlocking() == REDIS_ERR) break;
       }
       if (zmalloc_used_memory() > server.vm_max_memory)
          swap_all_values = 1; /* We are already using too much mem */
    }
}
fclose(fp);
stopLoading();
return REDIS_OK;
```

```
eoferr: /* unexpected end of file is handled here with a fatal exit */
    redisLog(REDIS_WARNING,"Short read or OOM loading DB. Unrecoverable error,
aborting now.");
    exit(1);
    return REDIS_ERR; /* Just to avoid warning */
}
```

打开 rdb 文件

读取 rdb 文件的签名和版本号

开始进入类型 | 值 | 类型 | 值的循环读取，可参考 rdb 文件格式

作者还做了导入的进度条，是有人反馈说 rdb 文件很大时导入时要很久，但又不知道进度，所以作者就加了导入的进度条，改善用户体验

读取类型

如果类型是过期时间类型 REDIS_EXPIRETIME，则读取过期时间

如果类型是文件结束类型 REDIS_EOF，则跳出类型 | 值 | 类型 | 值的循环读取

如果类型是选择 db 类型 REDIS_SELECTDB，则读取 db 索引并把当前 db 转成该 db，然后继续类型 | 值 | 类型 | 值的循环读取。

如果不是以上类型，则表明该类型是数据类型，读取作为 key 的字符串，即读取字符串类型的值，然后接着读取作为 value 的字符串。不同类型的编码不一样，根据写入时得规则解释读取到的值即可

读取到 key 和 value 后，判断下该 key 是否过期，如果过期则丢弃，不再导入，然后继续类型 | 值 | 类型 | 值的循环读取。

如果读取成功，则导入到内存，如果有过期时间则设置过期时间

如果配置了虚拟内存并且内存的使用比虚拟内存配置的大 32M 时，开始随机的取一些数据换出到虚拟内存中。

从上边我们也可以看到，如果没有配置虚拟内存，rdb 文件导入时会尽可能地占用操作系统的内存，甚至可能全部用完。

总结

落地存储是数据设计的一大重点也是难点。原理很简单，定义某种协议，然后按照某种协议写入读出。Redis 为了节省空间和读写时的 I/O 操作，做了很多很细致的工作来压缩数据。另外 redis 的丰富的数据类型也加大了落地的实现难度。作者也曾经在他的博客说过，redis 的丰富的数据类型导致了很多经典的优化办法无法在 redis 上实现。

AOF（append only file）

Aof 文件是 redis 保存内存数据到磁盘数据的其中一种方式（另外一种是 rdb），aof 落地原理很简单，客户端的每一条命令都会保存到磁盘，导入数据时只需把所有命令执行一遍就可以了。Aof 文件比 rdb 更精确和更安全，更能保存数据一致性。rdb 是定时保存内存中的数据镜像到磁盘，aof 是客户端每发一条命令就进行保存。

相关配置

appendonly no
是否启动 aof 保存数据到磁盘，可配置成 yes 或者 no
yes 表启动
no 表不启动

appendfilename appendonly.aof
配置 aof 文件名字

appendfsync no
写入磁盘的策略
调用 write 系统函数写数据，其实并不一定就写入到了磁盘，因为内核为了提高写磁盘的性能实现了一层缓存区，等数据积累到一定程度后在一起写入。如果只写入到了缓存区且掉电了，那么缓存区里边的数据将丢失。
可配置成 no 或者 always 或者 everysec
No 表使用系统的自动写入，即使用系统的写入策略
Always 表每次调用 write 系统函数后，直接再调用 fsync 系统函数写到磁盘
Everysec 表每秒中调用一次 fsync 系统函数写数据到磁盘

Aof 文件格式

Aof 记录的是一条条命令，记录命令的格式就是请求协议。具体见 multibulk 请求协议

Aof 存储时协议转换相关代码
Aof.c：158

```
sds catAppendOnlyGenericCommand(sds dst, int argc, robj **argv) {
    char buf[32];
    int len, j;
    robj *o;

    buf[0] = '*';
    len = 1+ll2string(buf+1,sizeof(buf)-1,argc);
    buf[len++] = '\r';
    buf[len++] = '\n';
    dst = sdscatlen(dst,buf,len);

    for (j = 0; j < argc; j++) {
        o = getDecodedObject(argv[j]);
        buf[0] = '$';
        len = 1+ll2string(buf+1,sizeof(buf)-1,sdslen(o->ptr));
        buf[len++] = '\r';
        buf[len++] = '\n';
```

```
        dst = sdscatlen(dst,buf,len);
        dst = sdscatlen(dst,o->ptr,sdslen(o->ptr));
        dst = sdscatlen(dst,"\r\n",2);
        decrRefCount(o);
    }
    return dst;
}
```

根据 multibulk 请求协议拼接字符，返回拼接好的字符串

写入 aof

Redis 在启动时就打开了 aof 文件，做好写入的准备

Redis.c：979

```
    if (server.appendonly) {
        server.appendfd                                                 =
open(server.appendfilename,O_WRONLY|O_APPEND|O_CREAT,0644);
        if (server.appendfd == -1) {
            redisLog(REDIS_WARNING, "Can't open the append-only file: %s",
                strerror(errno));
            exit(1);
        }
}
```

打开 aof 文件

然后在命令的统一调用入口（call 函数）调用添加命令到 aof 文件的函数

函数调用层级关系：processInputBuffer-->processCommand-->call

Redis.c：1035

```
void call(redisClient *c) {
    long long dirty, start = ustime(), duration;

    dirty = server.dirty;
    c->cmd->proc(c);
    dirty = server.dirty-dirty;
    duration = ustime()-start;
    slowlogPushEntryIfNeeded(c->argv,c->argc,duration);

    if (server.appendonly && dirty > 0)
        feedAppendOnlyFile(c->cmd,c->db->id,c->argv,c->argc);
    ...
}
```

计算出当前命令导致的脏数据数量，如果当前命令没有产生脏数据，则该命令不需要添加到 aof 文件中

如果启动了 aof 且数据被更新了，则添加到 aof 文件中，注意看 feedAppendOnlyFile 的参数，

从左到右分别是命令对象，key 所在的 db 的 id，命令的参数数组，命令参数的个数。

接着进入 feedAppendOnlyFile 函数看看

Aof.c：203

```
void feedAppendOnlyFile(struct redisCommand *cmd, int dictid, robj **argv, int
argc) {
    sds buf = sdsempty();
    robj *tmpargv[3];

    /* The DB this command was targetting is not the same as the last command
     * we appendend. To issue a SELECT command is needed. */
    if (dictid != server.appendseldb) {
        char seldb[64];

        snprintf(seldb,sizeof(seldb),"%d",dictid);
        buf = sdscatprintf(buf,"*2\r\n$6\r\nSELECT\r\n$%lu\r\n%s\r\n",
            (unsigned long)strlen(seldb),seldb);
        server.appendseldb = dictid;
    }

    if (cmd->proc == expireCommand) {
        /* Translate EXPIRE into EXPIREAT */
        buf = catAppendOnlyExpireAtCommand(buf,argv[1],argv[2]);
    } else if (cmd->proc == setexCommand) {
        /* Translate SETEX to SET and EXPIREAT */
        tmpargv[0] = createStringObject("SET",3);
        tmpargv[1] = argv[1];
        tmpargv[2] = argv[3];
        buf = catAppendOnlyGenericCommand(buf,3,tmpargv);
        decrRefCount(tmpargv[0]);
        buf = catAppendOnlyExpireAtCommand(buf,argv[1],argv[2]);
    } else {
        buf = catAppendOnlyGenericCommand(buf,argc,argv);
    }

    /* 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));

    sdsfree(buf);
}
```

如果当前要添加到 aof 文件的命令与上一个添加到 aof 文件的命令不同，则需要先添加一条选择 db 的命令

涉及到过期时间的命令需要特殊处理，因为过期的剩余时间需要转换成 unix 时间戳。

setex 命令分解成 set 命令和 expire 命令处理

命令的存储格式与请求协议相同，详细见 aof 文件格式

- 最后把要写入的内容添加到 server.aofbuf，根据配置的写入策略写入

如果正在重写 aof 文件，则需要把这些在重写过程中接收到的更新数据的命令也添加到新文件末尾去

导出 aof

Redis 在启动时，如果发现启动了 aof，则会去尝试读取 aof 文件内容，把数据导入到内存中

先看看启动时的相关代码

Redis.c：1717

```
if (server.appendonly) {
        if (loadAppendOnlyFile(server.appendfilename) == REDIS_OK)
            redisLog(REDIS_NOTICE,"DB loaded from append only file: %ld
seconds",time(NULL)-start);
}
```

如果配置了 aof，则尝试把数据导入到内存

接着进函数 loadAppendOnlyFile 看看

Redis.c：282

```
int loadAppendOnlyFile(char *filename) {
   ...
    FILE *fp = fopen(filename,"r");
    ...
    if (fp && redis_fstat(fileno(fp),&sb) != -1 && sb.st_size == 0) {
        server.appendonly_current_size = 0;
        fclose(fp);
        return REDIS_ERR;
    }

    if (fp == NULL) {
        redisLog(REDIS_WARNING,"Fatal error: can't open the append log file for
reading: %s",strerror(errno));
        exit(1);
```

```
    }

    /* Temporarily disable AOF, to prevent EXEC from feeding a MULTI
     * to the same file we're about to read. */
    server.appendonly = 0;

    fakeClient = createFakeClient();
    startLoading(fp);

    while(1) {
        ...
        /* Serve the clients from time to time */
        if (!(loops++ % 1000)) {
            loadingProgress(ftello(fp));
            aeProcessEvents(server.el, AE_FILE_EVENTS|AE_DONT_WAIT);
        }

        if (fgets(buf,sizeof(buf),fp) == NULL) {
            if (feof(fp))
                break;
            else
                goto readerr;
        }
        if (buf[0] != '*') goto fmterr;
        argc = atoi(buf+1);
        if (argc < 1) goto fmterr;

        argv = zmalloc(sizeof(robj*)*argc);
        for (j = 0; j < argc; j++) {
            if (fgets(buf,sizeof(buf),fp) == NULL) goto readerr;
            if (buf[0] != '$') goto fmterr;
            len = strtol(buf+1,NULL,10);
            argsds = sdsnewlen(NULL,len);
            if (len && fread(argsds,len,1,fp) == 0) goto fmterr;
            argv[j] = createObject(REDIS_STRING,argsds);
            if (fread(buf,2,1,fp) == 0) goto fmterr; /* discard CRLF */
        }

        /* Command lookup */
        cmd = lookupCommand(argv[0]->ptr);
        if (!cmd) {
            redisLog(REDIS_WARNING,"Unknown command '%s' reading the append only
file", argv[0]->ptr);
            exit(1);
```

```
        }
        /* Run the command in the context of a fake client */
        fakeClient->argc = argc;
        fakeClient->argv = argv;
        cmd->proc(fakeClient);

        /* The fake client should not have a reply */
        redisAssert(fakeClient->bufpos == 0 && listLength(fakeClient->reply) ==
0);
        /* The fake client should never get blocked */
        redisAssert((fakeClient->flags & REDIS_BLOCKED) == 0);

        /* Clean up. Command code may have changed argv/argc so we use the
         * argv/argc of the client instead of the local variables. */
        for (j = 0; j < fakeClient->argc; j++)
            decrRefCount(fakeClient->argv[j]);
        zfree(fakeClient->argv);

        /* Handle swapping while loading big datasets when VM is on */
        force_swapout = 0;
        if ((zmalloc_used_memory() - server.vm_max_memory) > 1024*1024*32)
            force_swapout = 1;

        if (server.vm_enabled && force_swapout) {
            while (zmalloc_used_memory() > server.vm_max_memory) {
                if (vmSwapOneObjectBlocking() == REDIS_ERR) break;
            }
        }
    }

    /* This point can only be reached when EOF is reached without errors.
     * If the client is in the middle of a MULTI/EXEC, log error and quit. */
    if (fakeClient->flags & REDIS_MULTI) goto readerr;

    fclose(fp);
    freeFakeClient(fakeClient);
    server.appendonly = appendonly;
    stopLoading();
    aofUpdateCurrentSize();
    server.auto_aofrewrite_base_size = server.appendonly_current_size;
    return REDIS_OK;

readerr:
    if (feof(fp)) {
```

```
      redisLog(REDIS_WARNING,"Unexpected end of file reading the append only
file");
   } else {
      redisLog(REDIS_WARNING,"Unrecoverable error reading the append only file:
%s", strerror(errno));
   }
   exit(1);
fmterr:
   redisLog(REDIS_WARNING,"Bad file format reading the append only file: make
a backup of your AOF file, then use ./redis-check-aof --fix <filename>");
   exit(1);
}
```

打开 aof 文件
创建一个伪造的客户端来发送从 aof 文件读取到的命令
根据写入的协议来解析读取的内容，先读取参数个数，然后依次读取参数，然后使用伪造的客户端来发送命令
如果数据比较大且启动了虚拟内存，则进行换出到虚拟内存中

总结

Aof 的设计很简单，所以逻辑也很简单明了，写入 aof 就是把会更新数据的命令保存到 aof 文件中，导出 aof 就是把这些命令重放一遍。

Replication（主从同步）

主从同步主要由两部分组成
全量同步，通过同步 rdb 文件实现
增量同步，已经完成全量同步，同步 master 端最新更新的数据

相关配置

从服务器配置

slaveof <masterip> <masterport>
配置主服务器的 ip 和 port

masterauth <master-password>
如果主服务器设置了密码，则需要配置，否则主服务器会拒绝从服务器的请求

slave-serve-stale-data yes
从服务器在进行复制时，如果有客户端来请求从服务器上的数据时，从服务器的反应策略

值可以是 yes 或者 no
yes，返回旧数据或者返回空（比如第一次同步都还没完成的情况）
no，除了 info 和 slaveof 命令，其他命令一律返回“正在同步数据错误”

repl-timeout 60
主从服务器间网络传输的超时时间配置，单位是秒

主服务器配置

repl-ping-slave-period 10
主服务多久发送一次 ping 给从服务器的时间配置，单位是秒
repl-ping-slave-period 要比从服务器配置中的 repl-timeout 值大，否则当主服务器空闲时，主从连接将会超时，redis 是通过最后一次通信时间来计算是否超时的。

全量同步

主从机的第一次同步是同步同步 rdb 文件来实现的，即 master 接收到 slave 的 sync 命令后生成一个 rdb 文件，然后把文件同步给 slave，slave 接收完 rdb 文件后，把 rdb 文件的数据导入内存。
通过一个状态机来跟踪 master 和 slave 在同步过程中的各种状态
全量同步过程中，master 扮演 server 角色，slave 扮演 client 角色。

slave

实现步骤和状态转换

slave 启动，此时 server.replstate 初始为 REDIS_REPL_NONE
配置了 slaveof 或者接收到 slaveof 命令，server.replstate 为 REDIS_REPL_CONNECT
每 100ms 左右执行一次的定时回调函数 serverCron 每执行 10 次（即大概 1000ms 左右）就会调用一次 replicationCron 函数，replicationCron 函数判断 server.replstate 为 REDIS_REPL_CONNECT 时将发起与 master 的连接，如果连接上了 master 则注册发送 sync 命令的 io 回调函数 syncWithMaster，server.replstate 转换为 REDIS_REPL_CONNECTING。
syncWithMaster 函数触发，首先发送 sync 命令给 master，然后打开临时 rdb 文件准备接受 master 同步过来的 rdb 文件，注册接受 rdb 文件的 io 回调函数 readSyncBulkPayload，server.replstate 转换为 REDIS_REPL_TRANSFER。
master 发送数据给 slave 时，回调函数 readSyncBulkPayload 被触发，开始不断地接收 master 端发送过来的 rdb 文件，直到 rdb 文件接受完毕并将接受到的数据导入内存，server.replstate 转换为 REDIS_REPL_CONNECTED。

实现细节

server.replstate 初始化为 REDIS_REPL_NONE

Redis.c：871

```
   server.replstate = REDIS_REPL_NONE;
```

在初始化配置函数 initServerConfig（redis.c：800）中对 server.replstate 进行初始化

server.replstate 转换为 REDIS_REPL_CONNECT

Config.c：194

```
      } else if (!strcasecmp(argv[0],"slaveof") && argc == 3) {
         server.masterhost = sdsnew(argv[1]);
         server.masterport = atoi(argv[2]);
         server.replstate = REDIS_REPL_CONNECT;
```

配置了 slaveof

server.replstate 转换为 REDIS_REPL_CONNECT

Replication.c：491

```
void slaveofCommand(redisClient *c) {
   if (!strcasecmp(c->argv[1]->ptr,"no") &&
      !strcasecmp(c->argv[2]->ptr,"one")) {
      if (server.masterhost) {
         sdsfree(server.masterhost);
         server.masterhost = NULL;
         if (server.master) freeClient(server.master);
         if (server.replstate == REDIS_REPL_TRANSFER)
            replicationAbortSyncTransfer();
         else if (server.replstate == REDIS_REPL_CONNECTING)
            undoConnectWithMaster();
         server.replstate = REDIS_REPL_NONE;
         redisLog(REDIS_NOTICE,"MASTER MODE enabled (user request)");
      }
   } else {
      sdsfree(server.masterhost);
      server.masterhost = sdsdup(c->argv[1]->ptr);
      server.masterport = atoi(c->argv[2]->ptr);
      if (server.master) freeClient(server.master);
      if (server.replstate == REDIS_REPL_TRANSFER)
         replicationAbortSyncTransfer();
      server.replstate = REDIS_REPL_CONNECT;
      redisLog(REDIS_NOTICE,"SLAVE OF %s:%d enabled (user request)",
         server.masterhost, server.masterport);
   }
   addReply(c,shared.ok);
}
```

接收到了 slaveof 命令

slaveof no one 是取消主从

如果不是 slaveof no one，则记下 masterhost 和 masterport，然后把 server.replstate 转换为 REDIS_REPL_CONNECT

server.replstate 转换为 REDIS_REPL_CONNECTING。

redis.c：683

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

每 100ms 左右执行一次的定时回调函数 serverCron 每执行 10 次（即大概 1000ms 左右）就会调用一次 replicationCron 函数

Replication.c：546

```
/* Check if we should connect to a MASTER */
   if (server.replstate == REDIS_REPL_CONNECT) {
      redisLog(REDIS_NOTICE,"Connecting to MASTER...");
      if (connectWithMaster() == REDIS_OK) {
         redisLog(REDIS_NOTICE,"MASTER <-> SLAVE sync started");
      }
   }
```

replicationCron 函数判断 server.replstate 为 REDIS_REPL_CONNECT 时将发起与 master 的连接

如果连接上了 master 则注册发送 sync 命令的 io 回调函数 syncWithMaster

再进入 connectWithMaster 函数看看

Replication.c：455

```
int connectWithMaster(void) {
   int fd;

   fd = anetTcpNonBlockConnect(NULL,server.masterhost,server.masterport);
   if (fd == -1) {
      redisLog(REDIS_WARNING,"Unable to connect to MASTER: %s",
         strerror(errno));
      return REDIS_ERR;
   }

   if
(aeCreateFileEvent(server.el,fd,AE_READABLE|AE_WRITABLE,syncWithMaster,NULL)
==
         AE_ERR)
   {
      close(fd);
      redisLog(REDIS_WARNING,"Can't create readable event for SYNC");
      return REDIS_ERR;
```

```
    }

    server.repl_transfer_lastio = time(NULL);
    server.repl_transfer_s = fd;
    server.replstate = REDIS_REPL_CONNECTING;
    return REDIS_OK;
}
```

注册发送 sync 命令的 io 回调函数 syncWithMaster
server.replstate 转换为 REDIS_REPL_CONNECTING

server.replstate 转换为 REDIS_REPL_TRANSFER。
Replication.c：371

```
void syncWithMaster(aeEventLoop *el, int fd, void *privdata, int mask) {
    ...
    /* Issue the SYNC command */
    if (syncWrite(fd,"SYNC \r\n",7,server.repl_syncio_timeout) == -1) {
        redisLog(REDIS_WARNING,"I/O error writing to MASTER: %s",
            strerror(errno));
        goto error;
    }

    /* Prepare a suitable temp file for bulk transfer */
    while(maxtries--) {
        snprintf(tmpfile,256,
            "temp-%d.%ld.rdb",(int)time(NULL),(long int)getpid());
        dfd = open(tmpfile,O_CREAT|O_WRONLY|O_EXCL,0644);
        if (dfd != -1) break;
        sleep(1);
    }
    if (dfd == -1) {
        redisLog(REDIS_WARNING,"Opening the temp file needed for MASTER <-> SLAVE
synchronization: %s",strerror(errno));
        goto error;
    }

    /* Setup the non blocking download of the bulk file. */
    if (aeCreateFileEvent(server.el,fd, AE_READABLE,readSyncBulkPayload,NULL)
            == AE_ERR)
    {
        redisLog(REDIS_WARNING,"Can't create readable event for SYNC");
        goto error;
    }

    server.replstate = REDIS_REPL_TRANSFER;
```

```
    server.repl_transfer_left = -1;
    server.repl_transfer_fd = dfd;
    server.repl_transfer_lastio = time(NULL);
    server.repl_transfer_tmpfile = zstrdup(tmpfile);
    return;

error:
    server.replstate = REDIS_REPL_CONNECT;
    close(fd);
    return;
}
```

首先发送 sync 命令给 master
然后打开临时 rdb 文件准备接受 master 同步过来的 rdb 文件
注册接受 rdb 文件的 io 回调函数 readSyncBulkPayload
server.replstate 转换为 REDIS_REPL_TRANSFER

server.replstate 转换为 REDIS_REPL_CONNECTED。
Replication.c：278

```
void readSyncBulkPayload(aeEventLoop *el, int fd, void *privdata, int mask) {
   ...
    /* If repl_transfer_left == -1 we still have to read the bulk length
     * from the master reply. */
    if (server.repl_transfer_left == -1) {
        if (syncReadLine(fd,buf,1024,server.repl_syncio_timeout) == -1) {
            redisLog(REDIS_WARNING,
                "I/O error reading bulk count from MASTER: %s",
                strerror(errno));
            goto error;
        }

        if (buf[0] == '-') {
            redisLog(REDIS_WARNING,
                "MASTER aborted replication with an error: %s",
                buf+1);
            goto error;
        } else if (buf[0] == '\0') {
            /* At this stage just a newline works as a PING in order to take
             * the connection live. So we refresh our last interaction
             * timestamp. */
            server.repl_transfer_lastio = time(NULL);
            return;
        } else if (buf[0] != '$') {
            redisLog(REDIS_WARNING,"Bad protocol from MASTER, the first byte is
not '$', are you sure the host and port are right?");
```

```
            goto error;
        }
        server.repl_transfer_left = strtol(buf+1,NULL,10);
        redisLog(REDIS_NOTICE,
            "MASTER <-> SLAVE sync: receiving %ld bytes from master",
            server.repl_transfer_left);
        return;
    }

    /* Read bulk data */
    readlen = (server.repl_transfer_left < (signed)sizeof(buf)) ?
        server.repl_transfer_left : (signed)sizeof(buf);
    nread = read(fd,buf,readlen);
    if (nread <= 0) {
        redisLog(REDIS_WARNING,"I/O error trying to sync with MASTER: %s",
            (nread == -1) ? strerror(errno) : "connection lost");
        replicationAbortSyncTransfer();
        return;
    }
    server.repl_transfer_lastio = time(NULL);
    if (write(server.repl_transfer_fd,buf,nread) != nread) {
        redisLog(REDIS_WARNING,"Write error or short write writing to the DB dump
file needed for MASTER <-> SLAVE synchrnonization: %s", strerror(errno));
        goto error;
    }
    server.repl_transfer_left -= nread;
    /* Check if the transfer is now complete */
    if (server.repl_transfer_left == 0) {
        if (rename(server.repl_transfer_tmpfile,server.dbfilename) == -1) {
            redisLog(REDIS_WARNING,"Failed trying to rename the temp DB into
dump.rdb in MASTER <-> SLAVE synchronization: %s", strerror(errno));
            replicationAbortSyncTransfer();
            return;
        }
        redisLog(REDIS_NOTICE, "MASTER <-> SLAVE sync: Loading DB in memory");
        emptyDb();
        /* Before loading the DB into memory we need to delete the readable
         * handler, otherwise it will get called recursively since
         * rdbLoad() will call the event loop to process events from time to
         * time for non blocking loading. */
        aeDeleteFileEvent(server.el,server.repl_transfer_s,AE_READABLE);
        if (rdbLoad(server.dbfilename) != REDIS_OK) {
            redisLog(REDIS_WARNING,"Failed trying to load the MASTER
synchronization DB from disk");
```

```
            replicationAbortSyncTransfer();
            return;
        }
        /* Final setup of the connected slave <- master link */
        zfree(server.repl_transfer_tmpfile);
        close(server.repl_transfer_fd);
        server.master = createClient(server.repl_transfer_s);
        server.master->flags |= REDIS_MASTER;
        server.master->authenticated = 1;
        server.replstate = REDIS_REPL_CONNECTED;
        redisLog(REDIS_NOTICE, "MASTER <-> SLAVE sync: Finished with success");
        /* Rewrite the AOF file now that the dataset changed. */
        if (server.appendonly) rewriteAppendOnlyFileBackground();
    }

    return;

error:
    replicationAbortSyncTransfer();
    return;
}
```

master 发送数据给 slave 时，回调函数 readSyncBulkPayload 被触发
数据协议为 multibulk 协议，因此先接受数据长度，然后接受数据
开始不断地接收 master 端发送过来的 rdb 文件，直到 rdb 文件接受完毕
将接受到的数据导入内存
server.replstate 转换为 REDIS_REPL_CONNECT

master

实现步骤和状态转换

master 启动，此时 client.replstate 初始为 REDIS_REPL_NONE
等待 slave 的发送 sync 命令，接受到 sync 命令后，如果当前没有写 rdb 的子进程，则启动一个写 rdb 的子进程，client.replstate 转换为 REDIS_REPL_WAIT_BGSAVE_END；如果当前已经有写 rdb 的子进程且有其它的 slave 正处于状态 REDIS_REPL_WAIT_BGSAVE_END，则可以与这个 slave 共用同一个 rdb 文件，client.replstate 转换为 REDIS_REPL_WAIT_BGSAVE_END；如果当前已经有写 rdb 的子进程但没有其它 slave 处于状态 REDIS_REPL_WAIT_BGSAVE_END，client.replstate 转换为 REDIS_REPL_WAIT_BGSAVE_START。
写 rdb 文件子进程结束后，判断所有 client.replstate，如果 client.replstate 为状态 REDIS_REPL_WAIT_BGSAVE_END，则打开 rdb 文件，client.replstate 转换为 REDIS_REPL_SEND_BULK，注册发送的网络 io 函数准备发送 rdb 文件；如果还有 client.replstate 的状态处于 REDIS_REPL_WAIT_BGSAVE_START，则启动一个写 rdb 文

件的子进程。然后等待写 rdb 文件子进程结束，重新进入次步骤。
开始发送 rdb 文件，直到发送完毕。client.replstate 转换为 REDIS_REPL_ONLINE。

实现细节

client.replstate 初始为 REDIS_REPL_NONE
network.c：42

```
c->replstate = REDIS_REPL_NONE;
```

在 createClient 函数中对 client.replstate 进行初始化

client.replstate 根据不同条件转换为 REDIS_REPL_WAIT_BGSAVE_START 或者转换为 REDIS_REPL_WAIT_BGSAVE_END。
Replication.c：86

```
void syncCommand(redisClient *c) {
   ...
   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) {
      ...
      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. */
         copyClientOutputBuffer(c,slave);
         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;
}
```

接收到 slave 发送过来的 sync 命令，进入到 syncCommand 函数

如果当前已经有写 rdb 的子进程且有其它的 slave 正处于状态 REDIS_REPL_WAIT_BGSAVE_END，则可以与这个 slave 共用同一个 rdb 文件，client.replstate 转换为 REDIS_REPL_WAIT_BGSAVE_END；

如果当前已经有写 rdb 的子进程但没有其它 slave 处于状态 REDIS_REPL_WAIT_BGSAVE_END，client.replstate 转换为 REDIS_REPL_WAIT_BGSAVE_START。

如果当前没有写 rdb 的子进程，则启动一个写 rdb 的子进程，client.replstate 转换为 REDIS_REPL_WAIT_BGSAVE_END；

标识该 client 为 slave 并把发送 sync 命令的 client 加入到 server.slaves 链表中

client.replstate 根据不同条件转换为 REDIS_REPL_WAIT_BGSAVE_END 或者转换为 REDIS_REPL_SEND_BULK

redis.c：605

```
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();
        }
    }
```

等待写 rdb 子进程退出

进入 rdb 子进程退出处理函数

Rdb.c：1062

```
void backgroundSaveDoneHandler(int statloc) {
    ...
    /* Possibly there are slaves waiting for a BGSAVE in order to be served
     * (the first stage of SYNC is a bulk transfer of dump.rdb) */
    updateSlavesWaitingBgsave(exitcode == 0 ? REDIS_OK : REDIS_ERR);
```

```
}
```

rdb 子进程退出处理函数调用了 slave 处理函数

replication.c：211

```
void updateSlavesWaitingBgsave(int bgsaveerr) {
    ...
    listRewind(server.slaves,&li);
    while((ln = listNext(&li))) {
        redisClient *slave = ln->value;

        if (slave->replstate == REDIS_REPL_WAIT_BGSAVE_START) {
            startbgsave = 1;
            slave->replstate = REDIS_REPL_WAIT_BGSAVE_END;
        } else if (slave->replstate == REDIS_REPL_WAIT_BGSAVE_END) {
            struct redis_stat buf;

            if (bgsaveerr != REDIS_OK) {
                freeClient(slave);
                redisLog(REDIS_WARNING,"SYNC failed. BGSAVE child returned an
error");
                continue;
            }
            if ((slave->repldbfd = open(server.dbfilename,O_RDONLY)) == -1 ||
                redis_fstat(slave->repldbfd,&buf) == -1) {
                freeClient(slave);
                redisLog(REDIS_WARNING,"SYNC failed. Can't open/stat DB after
BGSAVE: %s", strerror(errno));
                continue;
            }
            slave->repldboff = 0;
            slave->repldbsize = buf.st_size;
            slave->replstate = REDIS_REPL_SEND_BULK;
            aeDeleteFileEvent(server.el,slave->fd,AE_WRITABLE);
            if     (aeCreateFileEvent(server.el,     slave->fd,     AE_WRITABLE,
sendBulkToSlave, slave) == AE_ERR) {
                freeClient(slave);
                continue;
            }
        }
    }
    if (startbgsave) {
        if (rdbSaveBackground(server.dbfilename) != REDIS_OK) {
            listIter li;
```

```
            listRewind(server.slaves,&li);
            redisLog(REDIS_WARNING,"SYNC failed. BGSAVE failed");
            while((ln = listNext(&li))) {
                redisClient *slave = ln->value;

                if (slave->replstate == REDIS_REPL_WAIT_BGSAVE_START)
                    freeClient(slave);
            }
        }
    }
}
```

遍历所有连上来的 slaves

如果 client.replstate 为状态 REDIS_REPL_WAIT_BGSAVE_END，则打开 rdb 文件，client.replstate 转换为 REDIS_REPL_SEND_BULK，注册发送的网络 io 函数准备发送 rdb 文件；

如果还有存在 client.replstate 的状态处于 REDIS_REPL_WAIT_BGSAVE_START 的 salve，则标识 startbgsave 为 1。

退出遍历后，如果标识 startbgsave 为 1 则启动一个写 rdb 文件的子进程。然后等待写 rdb 文件子进程结束，重新进入此函数

client.replstate 根据不同条件转换为 REDIS_REPL_SEND_BULK 或者转换为 REDIS_REPL_ONLINE

replication.c：151

```
void sendBulkToSlave(aeEventLoop *el, int fd, void *privdata, int mask) {
    ...
    if (slave->repldboff == 0) {
        /* Write the bulk write count before to transfer the DB. In theory here
         * we don't know how much room there is in the output buffer of the
         * socket, but in pratice SO_SNDLOWAT (the minimum count for output
         * operations) will never be smaller than the few bytes we need. */
        sds bulkcount;

        bulkcount = sdscatprintf(sdsempty(),"$%lld\r\n",(unsigned long long)
            slave->repldbsize);
        if (write(fd,bulkcount,sdslen(bulkcount)) != (signed)sdslen(bulkcount))
        {
            sdsfree(bulkcount);
            freeClient(slave);
            return;
        }
        sdsfree(bulkcount);
    }
    lseek(slave->repldbfd,slave->repldboff,SEEK_SET);
    buflen = read(slave->repldbfd,buf,REDIS_IOBUF_LEN);
```

```
    if (buflen <= 0) {
        redisLog(REDIS_WARNING,"Read error sending DB to slave: %s",
            (buflen == 0) ? "premature EOF" : strerror(errno));
        freeClient(slave);
        return;
    }
    if ((nwritten = write(fd,buf,buflen)) == -1) {
        redisLog(REDIS_VERBOSE,"Write error sending DB to slave: %s",
            strerror(errno));
        freeClient(slave);
        return;
    }
    slave->repldboff += nwritten;
    if (slave->repldboff == slave->repldbsize) {
        close(slave->repldbfd);
        slave->repldbfd = -1;
        aeDeleteFileEvent(server.el,slave->fd,AE_WRITABLE);
        slave->replstate = REDIS_REPL_ONLINE;
        if (aeCreateFileEvent(server.el, slave->fd, AE_WRITABLE,
            sendReplyToClient, slave) == AE_ERR) {
            freeClient(slave);
            return;
        }
        addReplySds(slave,sdsempty());
        redisLog(REDIS_NOTICE,"Synchronization with slave succeeded");
    }
}
```

首先发送 rdb 文件的长度信息
然后读取 rdb 文件内容，一次读取 16kb，然开始发送数据，并记录已发送数据大小。
如果文件没有发送完毕，则继续等网络 io 写事件触发，继续发送。
如果文件发送完毕，删除原来绑定的网络 io 写事件
client.replstate 转换为 REDIS_REPL_ONLINE
绑定新的网络 io 写事件的回调函数为 sendReplyToClient，用于增量同步，具体见增量同步。

增量同步

在第一次主从同步后，master 端的 client.replstate 为 REDIS_REPL_ONLINE，slave 端的 server.replstate 转换为 REDIS_REPL_CONNECTED
接下来 master 端的数据改变会即时通知到 slave 端
增量同步过程中，master 扮演 client 角色，slave 扮演 server 角色，与全量同步相反。

slave

replication.c：278

```
void readSyncBulkPayload(aeEventLoop *el, int fd, void *privdata, int mask) {
   ...
   /* Check if the transfer is now complete */
   if (server.repl_transfer_left == 0) {
      if (rename(server.repl_transfer_tmpfile,server.dbfilename) == -1) {
         redisLog(REDIS_WARNING,"Failed trying to rename the temp DB into
dump.rdb in MASTER <-> SLAVE synchronization: %s", strerror(errno));
         replicationAbortSyncTransfer();
         return;
      }
      redisLog(REDIS_NOTICE, "MASTER <-> SLAVE sync: Loading DB in memory");
      emptyDb();
      /* Before loading the DB into memory we need to delete the readable
       * handler, otherwise it will get called recursively since
       * rdbLoad() will call the event loop to process events from time to
       * time for non blocking loading. */
      aeDeleteFileEvent(server.el,server.repl_transfer_s,AE_READABLE);
      if (rdbLoad(server.dbfilename) != REDIS_OK) {
         redisLog(REDIS_WARNING,"Failed   trying   to   load   the   MASTER
synchronization DB from disk");
         replicationAbortSyncTransfer();
         return;
      }
      /* Final setup of the connected slave <- master link */
      zfree(server.repl_transfer_tmpfile);
      close(server.repl_transfer_fd);
      server.master = createClient(server.repl_transfer_s);
      server.master->flags |= REDIS_MASTER;
      server.master->authenticated = 1;
      server.replstate = REDIS_REPL_CONNECTED;
      redisLog(REDIS_NOTICE, "MASTER <-> SLAVE sync: Finished with success");
      /* Rewrite the AOF file now that the dataset changed. */
      if (server.appendonly) rewriteAppendOnlyFileBackground();
   }

   return;

error:
   replicationAbortSyncTransfer();
   return;
```

```
}
```

当全量同步传输的 rdb 文件接收完毕，slave 端删除了全量同步的网络 io 的读事件回调函数 readSyncBulkPayload （ replication.c:279 ） ， server.replstate 转 换 为 REDIS_REPL_CONNECTED.

建立了一个 client，用于接收增量同步的数据，在 cretaeClient 函数中绑定增量同步的网络 io 的读事件回调函数 readQueryFromClient（networking.c:874），即 master 成为了 slave 的 client。

增量同步走命令协议，即 master 把会改变数据的命令请求转发一份到 slave

slave 对 master 发送过来的命令与其他 client 发送过来的命令处理基本类似。除了不需要回数据给 master，socket 不需要超时处理（使用主从同步特有的超时规则）等一些细节处理。

master

master 侧在判断有已经进行全量同步的 slave 的情况下，会把所有会更新数据的命令转发一份给所有 slave，用于增量同步。转发命令函数为 replicationFeedSlaves。

replication.c：10

```
void replicationFeedSlaves(list *slaves, int dictid, robj **argv, int argc) {
    listNode *ln;
    listIter li;
    int j;

    listRewind(slaves,&li);
    while((ln = listNext(&li))) {
        redisClient *slave = ln->value;

        /* Don't feed slaves that are still waiting for BGSAVE to start */
        if (slave->replstate == REDIS_REPL_WAIT_BGSAVE_START) continue;

        /* Feed slaves that are waiting for the initial SYNC (so these commands
         * are queued in the output buffer until the intial SYNC completes),
         * or are already in sync with the master. */
        if (slave->slaveseldb != dictid) {
            robj *selectcmd;

            switch(dictid) {
            case 0: selectcmd = shared.select0; break;
            case 1: selectcmd = shared.select1; break;
            case 2: selectcmd = shared.select2; break;
            case 3: selectcmd = shared.select3; break;
            case 4: selectcmd = shared.select4; break;
            case 5: selectcmd = shared.select5; break;
```

```
            case 6: selectcmd = shared.select6; break;
            case 7: selectcmd = shared.select7; break;
            case 8: selectcmd = shared.select8; break;
            case 9: selectcmd = shared.select9; break;
            default:
                selectcmd = createObject(REDIS_STRING,
                    sdscatprintf(sdsempty(),"select %d\r\n",dictid));
                selectcmd->refcount = 0;
                break;
            }
            addReply(slave,selectcmd);
            slave->slaveseldb = dictid;
        }
        addReplyMultiBulkLen(slave,argc);
        for (j = 0; j < argc; j++) addReplyBulk(slave,argv[j]);
    }
}
```

- 如果 slave 的当前 db 与 master 的当前 db 不一致，则增加选择 db 命令

把接收到的命令按 multibulk 协议发送一份给 slave

心跳包

Replication.c：521

```
void replicationCron(void) {
    ...

    /* If we have attached slaves, PING them from time to time.
     * So slaves can implement an explicit timeout to masters, and will
     * be able to detect a link disconnection even if the TCP connection
     * will not actually go down. */
    if (!(server.cronloops % (server.repl_ping_slave_period*10))) {
        listIter li;
        listNode *ln;

        listRewind(server.slaves,&li);
        while((ln = listNext(&li))) {
            redisClient *slave = ln->value;

            /* Don't ping slaves that are in the middle of a bulk transfer
             * with the master for first synchronization. */
            if (slave->replstate == REDIS_REPL_SEND_BULK) continue;
            if (slave->replstate == REDIS_REPL_ONLINE) {
                /* If the slave is online send a normal ping */
                addReplySds(slave,sdsnew("PING\r\n"));
```

```
            } else {
                /* Otherwise we are in the pre-synchronization stage.
                 * Just a newline will do the work of refreshing the
                 * connection last interaction time, and at the same time
                 * we'll be sure that being a single char there are no
                 * short-write problems. */
                if (write(slave->fd, "\n", 1) == -1) {
                    /* Don't worry, it's just a ping. */
                }
            }
        }
    }
}
```

如果达到了发送心跳包的时间间隔且有开始已经处于增量同步阶段的 slave，则发送字符串”PING\r\n”

server.cronloops 是 100ms 加一

server.repl_ping_slave_period 的单位是毫秒

超时管理

Replication.c：521

```
void replicationCron(void) {
    /* Non blocking connection timeout? */
    if (server.masterhost && server.replstate == REDIS_REPL_CONNECTING &&
        (time(NULL)-server.repl_transfer_lastio) > server.repl_timeout)
    {
        redisLog(REDIS_WARNING,"Timeout connecting to the MASTER...");
        undoConnectWithMaster();
    }

    /* Bulk transfer I/O timeout? */
    if (server.masterhost && server.replstate == REDIS_REPL_TRANSFER &&
        (time(NULL)-server.repl_transfer_lastio) > server.repl_timeout)
    {
        redisLog(REDIS_WARNING,"Timeout receiving bulk data from MASTER...");
        replicationAbortSyncTransfer();
    }

    /* Timed out master when we are an already connected slave? */
    if (server.masterhost && server.replstate == REDIS_REPL_CONNECTED &&
        (time(NULL)-server.master->lastinteraction) > server.repl_timeout)
    {
        redisLog(REDIS_WARNING,"MASTER time out: no data nor PING received...");
        freeClient(server.master);
```

```
    }

    /* Check if we should connect to a MASTER */
    if (server.replstate == REDIS_REPL_CONNECT) {
        redisLog(REDIS_NOTICE,"Connecting to MASTER...");
        if (connectWithMaster() == REDIS_OK) {
            redisLog(REDIS_NOTICE,"MASTER <-> SLAVE sync started");
        }
    }

    ...
}
```

Redis 在 slave 端会记录每次与 master 通信的时间点，当超过一定的时间没有进行通信时，slave 会认为次连接已经出现问题了，将中断与 master 建立上的连接。

并把 server.replstate 设置成 REDIS_REPL_CONNECT。

Slave 判断 server.replstate 是否为 REDIS_REPL_CONNECT，如果是则会再次尝试与 master 连接，详细见全量同步

Redis 的过期 key 的检测机制

当一个 key 过期时要删除该 key，删除的触发是有 master 控制的，master 删除该 key 后会将删除命令同步到 aof 文件和 slave。

被动检测

Redis 在查询一个 key 的时候（db.c：46，lookupKeyRead 函数；db.c：58，lookupKeyWrite 函数）会调用函数 expireIfNeeded 进行 key 是否过期的检测和处理。

另外，Redis 在执行 expire 命令（db.c：237，existsCommand 函数），keys 命令（db.c：268，keysCommand 函数）和 randomkeyCommand 命令（db.c：256，randomkeyCommand 函数）时也会调用函数 expireIfNeeded 进行 key 是否过期的检测和处理。

主动检测

单单被动检测是不够的，假设有很多过期的 key 都没有被访问，那这些垃圾数据将占用大量的内存空间。所以 redis 还使用了主动检测方式来处理过期的 key。

在 100ms 左右执行一次的 serverCron 函数有一行这样的代码

Redis.c：658

```
if (server.masterhost == NULL) activeExpireCycle();
```

redis.c：481

```
void activeExpireCycle(void) {
    int j;

    for (j = 0; j < server.dbnum; j++) {
        int expired;
        redisDb *db = server.db+j;

        /* Continue to expire if at the end of the cycle more than 25%
         * of the keys were expired. */
        do {
            long num = dictSize(db->expires);
            time_t now = time(NULL);

            expired = 0;
            if (num > REDIS_EXPIRELOOKUPS_PER_CRON)
                num = REDIS_EXPIRELOOKUPS_PER_CRON;
            while (num--) {
                dictEntry *de;
                time_t t;

                if ((de = dictGetRandomKey(db->expires)) == NULL) break;
                t = (time_t) dictGetEntryVal(de);
                if (now > t) {
                    sds key = dictGetEntryKey(de);
                    robj *keyobj = createStringObject(key,sdslen(key));

                    propagateExpire(db,keyobj);
                    dbDelete(db,keyobj);
                    decrRefCount(keyobj);
                    expired++;
                    server.stat_expiredkeys++;
                }
            }
        } while (expired > REDIS_EXPIRELOOKUPS_PER_CRON/4);
    }
}
```

遍历所有 db，在每个 db 的记录超时时间的 dict 中随机取 REDIS_EXPIRELOOKUPS_PER_CRON（现在设置为 10）个超时时间，如果发现有过期的，则对该 key 进行删除并同步到 aof 文件和所有 slave。

如果在某个 db 中随机取 REDIS_EXPIRELOOKUPS_PER_CRON（现在设置为 10）个超时时间中有超过 1/4 是过期的，则再对该 db 进行过期检测，一直到过期的比例少于 1/4。

Redis 的回包处理

Redis 的回包处理时散落在各个命令中，看起来比较零散

Redis 的回复数据可存储在两种缓存区中，一种是字符串类型的 client 的 buf 字段（buf 最多能缓存 REDIS_REPLY_CHUNK_BYTES 个字节，现这个宏定义为 5*1500），另外一种是链表，每个链表结点的值是 sds 类型，可以存储无限大数据

Redis 会先尝试用 client 的 buf 字段缓存数据，如果不能用 client 的 buf 字段缓存的，则用 client->reply 链表缓存

发送回复数据时，redis 先发送 client 的 buf 字段里边的数据，然后再发送 client->reply 链表里边的数据

绑定写事件函数

networking.c：64

```
int _installWriteEvent(redisClient *c) {
   if (c->fd <= 0) return REDIS_ERR;
   if (c->bufpos == 0 && listLength(c->reply) == 0 &&
       (c->replstate == REDIS_REPL_NONE ||
        c->replstate == REDIS_REPL_ONLINE) &&
       aeCreateFileEvent(server.el, c->fd, AE_WRITABLE,
       sendReplyToClient, c) == AE_ERR) return REDIS_ERR;
   return REDIS_OK;
}
```

- 如果还没有绑定写事件回调，则绑定

如果已经绑定了，直接返回 REDIS_OK

_installWriteEvent 函数是在 addReply 函数（networking.c：190），addReplySds 函数（networking.c：215），addReplyString 函数（networking.c：229）和 addDeferredMultiBulkLength 函数（networking.c：275）中被调用，所以命令回复数据时都至少会调用其中一个函数。

写事件回调函数

networking.c：566

```
void sendReplyToClient(aeEventLoop *el, int fd, void *privdata, int mask) {
   redisClient *c = privdata;
   int nwritten = 0, totwritten = 0, objlen;
   robj *o;
   REDIS_NOTUSED(el);
   REDIS_NOTUSED(mask);

   while(c->bufpos > 0 || listLength(c->reply)) {
      if (c->bufpos > 0) {
```

```
            if (c->flags & REDIS_MASTER) {
                /* Don't reply to a master */
                nwritten = c->bufpos - c->sentlen;
            } else {
                nwritten = write(fd,c->buf+c->sentlen,c->bufpos-c->sentlen);
                if (nwritten <= 0) break;
            }
            c->sentlen += nwritten;
            totwritten += nwritten;

            /* If the buffer was sent, set bufpos to zero to continue with
             * the remainder of the reply. */
            if (c->sentlen == c->bufpos) {
                c->bufpos = 0;
                c->sentlen = 0;
            }
        } else {
            o = listNodeValue(listFirst(c->reply));
            objlen = sdslen(o->ptr);

            if (objlen == 0) {
                listDelNode(c->reply,listFirst(c->reply));
                continue;
            }

            if (c->flags & REDIS_MASTER) {
                /* Don't reply to a master */
                nwritten = objlen - c->sentlen;
            } else {
                nwritten                        =                        write(fd,
((char*)o->ptr)+c->sentlen,objlen-c->sentlen);
                if (nwritten <= 0) break;
            }
            c->sentlen += nwritten;
            totwritten += nwritten;

            /* If we fully sent the object on head go to the next one */
            if (c->sentlen == objlen) {
                listDelNode(c->reply,listFirst(c->reply));
                c->sentlen = 0;
            }
        }
        /* Note that we avoid to send more thank REDIS_MAX_WRITE_PER_EVENT
         * bytes, in a single threaded server it's a good idea to serve
```

```
         * other clients as well, even if a very large request comes from
         * super fast link that is always able to accept data (in real world
         * scenario think about 'KEYS *' against the loopback interfae) */
        if (totwritten > REDIS_MAX_WRITE_PER_EVENT) break;
    }
    if (nwritten == -1) {
        if (errno == EAGAIN) {
            nwritten = 0;
        } else {
            redisLog(REDIS_VERBOSE,
                "Error writing to client: %s", strerror(errno));
            freeClient(c);
            return;
        }
    }
    if (totwritten > 0) c->lastinteraction = time(NULL);
    if (c->bufpos == 0 && listLength(c->reply) == 0) {
        c->sentlen = 0;
        aeDeleteFileEvent(server.el,c->fd,AE_WRITABLE);

        /* Close connection after entire reply has been sent. */
        if (c->flags & REDIS_CLOSE_AFTER_REPLY) freeClient(c);
    }
}
```

首先发送 client 的 buf 字段里边的数据
然后遍历 client->reply 链表，按顺序从头到尾把链表中的值发送给客户端
发送数据时，会记录两个值，该数据长度和已经被发送的长度
如果所有 client->reply 链表结点的数据发送完毕，则记录与客户端的最后交互时间
如果没有数据可发送了，则删除写事件。
如果 client 被设置了 REDIS_CLOSE_AFTER_REPLY 标识，则直接释放 client，REDIS_CLOSE_AFTER_REPLY 标识在协议解析失败时（networking.c：720，setProtocolError 函数），接收到 client 命令的 kill 时（networking.c：1003，clientCommand 函数）和接收到 quit 命令时（redis.c：1067，processCommand 函数）会设置该标识。

添加数据到 client 的 buf 字段

Redis 封装了很多各种场景下使用的回复函数，但底层调用的就是两种，一种是添加数据到 client 的 buf 字段，另外一种是下边的，添加回复链表结点到 client->reply 链表中

Networking.c：94

```
int _addReplyToBuffer(redisClient *c, char *s, size_t len) {
    size_t available = sizeof(c->buf)-c->bufpos;
```

```
    if (c->flags & REDIS_CLOSE_AFTER_REPLY) return REDIS_OK;

    /* If there already are entries in the reply list, we cannot
     * add anything more to the static buffer. */
    if (listLength(c->reply) > 0) return REDIS_ERR;

    /* Check that the buffer has enough space available for this string. */
    if (len > available) return REDIS_ERR;

    memcpy(c->buf+c->bufpos,s,len);
    c->bufpos+=len;
    return REDIS_OK;
}
```

如果 client 被设置了 REDIS_CLOSE_AFTER_REPLY 标识，则不再添加数据到发送缓存区，直接返回 REDIS_OK 。REDIS_CLOSE_AFTER_REPLY 标识在协议解析失败时（networking.c：720，setProtocolError 函数），接收到 client 命令的 kill 时（networking.c：1003，clientCommand 函数）和接收到 quit 命令时（redis.c：1067， processCommand 函数）会设置该标识。

如果已经开始使用链表方式来缓存回复数据时（比如，数据过大，client 的 buf 字段已经放不下了），不再使用 client 的 buf 字段来缓存回复数据，返回 REDIS_ERR。

如果 client 的 buf 字段中的可用空间已经不足于放下数据，返回 REDIS_ERR。

否则添加数据到 client 的 buf 字段中，返回 REDIS_OK

添加回复链表结点到 client->reply 链表

Redis 根据添加到 client->reply 链表中的参数的不同类型封装了 3 个添加链表结点到 client->reply 链表的函数，分别是，_addReplyObjectToList 函数，_addReplySdsToList 函数，_addReplyStringToList 函数。

Networking.c：111

```
void _addReplyObjectToList(redisClient *c, robj *o) {
    robj *tail;

    if (c->flags & REDIS_CLOSE_AFTER_REPLY) return;

    if (listLength(c->reply) == 0) {
        incrRefCount(o);
        listAddNodeTail(c->reply,o);
    } else {
        tail = listNodeValue(listLast(c->reply));

        /* Append to this object when possible. */
        if (tail->ptr != NULL &&
            sdslen(tail->ptr)+sdslen(o->ptr) <= REDIS_REPLY_CHUNK_BYTES)
```

```
        {
            tail = dupLastObjectIfNeeded(c->reply);
            tail->ptr = sdscatlen(tail->ptr,o->ptr,sdslen(o->ptr));
        } else {
            incrRefCount(o);
            listAddNodeTail(c->reply,o);
        }
    }
}
```

如果 client 被设置了 REDIS_CLOSE_AFTER_REPLY 标识，则不再添加数据到发送缓存区，直接返回。REDIS_CLOSE_AFTER_REPLY 标识在协议解析失败时（networking.c：720，setProtocolError 函数），接收到 client 命令的 kill 时（networking.c：1003，clientCommand 函数）和接收到 quit 命令时（redis.c：1067， processCommand 函数）会设置该标识。

如果 client->reply 链表是空的，则直接将对象添加到链表中

如果 client->reply 链表已经有结点，如果最后一个结点不为空且添加新的数据到尾部不会超过宏 REDIS_REPLY_CHUNK_BYTES 定义的数值（现这个宏定义为 5*1500），则把需添加的对象的数据添加到结点的数据的尾部。添加之前判断下是否还有别的地方应用了该结点，如果有的话，需先复制一份出来，因为我们即将改变该结点的数据。

如果最后一个结点为空或者改结点容纳不了要添加的结点的数组，则把要添加的对象添加到链表。

_addReplySdsToList 函数和_addReplyStringToList 函数与_addReplyObjectToList 函数类似，只是传进来的参数不同，添加时需要变成对象而已。

Redis 中的命令实现

Keys

DEL

使用

格式：DEL key [key ...]
功能：删除一个或多个 key，如果该 key 不存在则忽略
返回：被删除的 key 的个数

实现

Db.c：224

```
void delCommand(redisClient *c) {
    int deleted = 0, j;

    for (j = 1; j < c->argc; j++) {
        if (dbDelete(c->db,c->argv[j])) {
            signalModifiedKey(c->db,c->argv[j]);
            server.dirty++;
            deleted++;
        }
    }
    addReplyLongLong(c,deleted);
}
```

遍历所有传送过来的 key，在当前使用的 db 中删除该 key

调用钩子函数，通知 key 的变化，主要是用于事务（transactions）的 exec 命令

统计服务器数据的更新，主要用于写 rdb 文件的条件判断

回复客户端被删除的个数

进入 dbDelete 函数看看

Db.c

```
int dbDelete(redisDb *db, robj *key) {
    /* If VM is enabled make sure to awake waiting clients for this key:
     * deleting the key will kill the I/O thread bringing the key from swap
     * to memory, so the client will never be notified and unblocked if we
     * don't do it now. */
    if (server.vm_enabled) handleClientsBlockedOnSwappedKey(db,key);
    /* Deleting an entry from the expires dict will not free the sds of
     * the key, because it is shared with the main dictionary. */
    if (dictSize(db->expires) > 0) dictDelete(db->expires,key->ptr);
    return dictDelete(db->dict,key->ptr) == DICT_OK;
}
```

如果启动了 vm 且有被阻塞的 client 在等待该 key 换入到内存，handleClientsBlockedOnSwappedKey 会让在等待将这个 key 换入内存的 client 不再等待该 key，然后将换入任务取消，并标识 vm 中对应的 page 为 free。取消换入任务和标识为 free 是在 handleClientsBlockedOnSwappedKey-->dontWaitForSwappedKey-->decrRefCount 函数内处理。

如果该 key 设置了超时时间，也要把超时时间一并删除，redis 是使用单独的一个 dict 来存超时时间的。

删除 db 中的数据

dbDelete 函数执行完后返回到 delCommand 函数中，继续执行钩子函数 signalModifiedKey

db.c：186

```
void signalModifiedKey(redisDb *db, robj *key) {
    touchWatchedKey(db,key);
}
```

Multi.c：209

```
/* "Touch" a key, so that if this key is being WATCHed by some client the
 * next EXEC will fail. */
void touchWatchedKey(redisDb *db, robj *key) {
    list *clients;
    listIter li;
    listNode *ln;

    if (dictSize(db->watched_keys) == 0) return;
    clients = dictFetchValue(db->watched_keys, key);
    if (!clients) return;

    /* Mark all the clients watching this key as REDIS_DIRTY_CAS */
    /* Check if we are already watching for this key */
    listRewind(clients,&li);
    while((ln = listNext(&li))) {
        redisClient *c = listNodeValue(ln);

        c->flags |= REDIS_DIRTY_CAS;
    }
}
```

Db->watched_keys 是一个以 key 为键，以 watch 这个 key 的 client 链表的 dict 为值，如果 watch 的 key 有变动（client 的标识为 `REDIS_DIRTY_CAS`），该 client 接下来执行的 exec 命令将会失败。

EXISTS

使用

格式：EXISTS key
功能：判断一个可以存不存在
返回：1，如果该 key 存在；0，如果该可以不存在

实现

Db.c：237

```
void existsCommand(redisClient *c) {
    expireIfNeeded(c->db,c->argv[1]);
    if (dbExists(c->db,c->argv[1])) {
        addReply(c, shared.cone);
    } else {
```

```
        addReply(c, shared.czero);
    }
}
```

进行判断该 key 存不存在前先判断该 key 有没有超时，如果超时要先删除该 key
判断该 key 存不存在

接着进入 expireIfNeeded 函数看看

Db.c：496

```
int expireIfNeeded(redisDb *db, robj *key) {
    time_t when = getExpire(db,key);

    if (when < 0) return 0; /* No expire for this key */

    /* Don't expire anything while loading. It will be done later. */
    if (server.loading) return 0;

    /* If we are running in the context of a slave, return ASAP:
     * the slave key expiration is controlled by the master that will
     * send us synthesized DEL operations for expired keys.
     *
     * Still we try to return the right information to the caller,
     * that is, 0 if we think the key should be still valid, 1 if
     * we think the key is expired at this time. */
    if (server.masterhost != NULL) {
        return time(NULL) > when;
    }

    /* Return when this key has not expired */
    if (time(NULL) <= when) return 0;

    /* Delete the key */
    server.stat_expiredkeys++;
    propagateExpire(db,key);
    return dbDelete(db,key);
}
```

取该 key 的超时时间
如果当前正处于导入 rdb 中的数据到内存中，则先忽略，导入完后会进行处理

- 如果当前的 redis 实例是 slave，则先忽略，超时统一由 master 控制

如果没有超时返回 0
如果超时了，发送删除该 key 的命令给 slave 和添加删除命令到 aof 文件，然后删除该 key

EXPIRE

使用

格式：EXPIRE key seconds
功能：设置一个 key 的经过 seconds 秒后过期
返回：1，设置过期时间成功；0，该 key 不存在或者设置失败

实现

Db.c：570

```
void expireCommand(redisClient *c) {
   expireGenericCommand(c,c->argv[1],c->argv[2],0);
}
```

Db.c：528

```
void expireGenericCommand(redisClient *c, robj *key, robj *param, long offset)
{
   dictEntry *de;
   long seconds;

   if (getLongFromObjectOrReply(c, param, &seconds, NULL) != REDIS_OK) return;

   seconds -= offset;

   de = dictFind(c->db->dict,key->ptr);
   if (de == NULL) {
      addReply(c,shared.czero);
      return;
   }
   /* EXPIRE with negative TTL, or EXPIREAT with a timestamp into the past
    * should never be executed as a DEL when load the AOF or in the context
    * of a slave instance.
    *
    * Instead we take the other branch of the IF statement setting an expire
    * (possibly in the past) and wait for an explicit DEL from the master. */
   if (seconds <= 0 && !server.loading && !server.masterhost) {
      robj *aux;

      redisAssert(dbDelete(c->db,key));
      server.dirty++;

      /* Replicate/AOF this as an explicit DEL. */
```

```
        aux = createStringObject("DEL",3);
        rewriteClientCommandVector(c,2,aux,key);
        decrRefCount(aux);
        signalModifiedKey(c->db,key);
        addReply(c, shared.cone);
        return;
    } else {
        time_t when = time(NULL)+seconds;
        setExpire(c->db,key,when);
        addReply(c,shared.cone);
        signalModifiedKey(c->db,key);
        server.dirty++;
        return;
    }
}
```

expireCommand 函数中只是简单调用了 expireGenericCommand 函数

取出客户端传送过来的超时时间

查找该 key，如果该可以不存在，返回 0

如果设置的超时时间小于等于 0，删除该 key，并把 expire 命令转变成删除命令，在调用命令的函数（redis.c：1035，call 函数）处用于与 aof 文件和 slaves 同步

如果设置的超时时间大于 0，则设置该 key 的超时时间，在调用命令的函数（redis.c：1035，call 函数）会同步给 aof 文件和 slaves

EXPIREAT

使用

格式：EXPIREAT key timestamp

功能：设置一个 key 的到时间点 timestamp 后过期

返回：1，设置过期时间成功；0，该 key 不存在或者设置失败

实现

Db.c：574

```
void expireatCommand(redisClient *c) {
    expireGenericCommand(c,c->argv[1],c->argv[2],time(NULL));
}
```

与 expire 命令类似，具体见 expire 命令

KEYS

使用

格式：KEYS pattern
功能：获取所有匹配 pattern 的 key
返回：匹配 pattern 的 key
警告：keys 命令可能严重影响性能，在生产环境中请谨慎使用

实现

Db.c：268

```
void keysCommand(redisClient *c) {
    dictIterator *di;
    dictEntry *de;
    sds pattern = c->argv[1]->ptr;
    int plen = sdslen(pattern), allkeys;
    unsigned long numkeys = 0;
    void *replylen = addDeferredMultiBulkLength(c);

    di = dictGetIterator(c->db->dict);
    allkeys = (pattern[0] == '*' && pattern[1] == '\0');
    while((de = dictNext(di)) != NULL) {
        sds key = dictGetEntryKey(de);
        robj *keyobj;

        if (allkeys || stringmatchlen(pattern,plen,key,sdslen(key),0)) {
            keyobj = createStringObject(key,sdslen(key));
            if (expireIfNeeded(c->db,keyobj) == 0) {
                addReplyBulk(c,keyobj);
                numkeys++;
            }
            decrRefCount(keyobj);
        }
    }
    dictReleaseIterator(di);
    setDeferredMultiBulkLength(c,replylen,numkeys);
}
```

addDeferredMultiBulkLength 会将预留出第一个回复链接结点，因为现在还不知道 db 中有多少个 key，所以无法设置回复数据数量，于是先预留出一个回复链表结点，并把链表结点的值设置为 null。然后遍历当前 db，取出所有 key 的进行匹配，凡是匹配的 key 则进行添加到回复链接中
addReplyBulk 会先判断回复链表是否为空，不为空则往回复链表中添加，然后判断最后一个链表结点的

值是否为 null，为 null 则新加一个结点到链表尾部。在 addDeferredMultiBulkLength 将第一个回复链接结点的值设置为 null 了，所以 addReplyBulk 会添加一个链表结点到尾部。具体可参考 redis 的回包处理。

最后设置回复数据数量，设置后如果发现回复链表下一个结点有数据，会将下一个结点的数据合并过来

MOVE

使用

格式：MOVE key db
功能：把一个 key 从当前数据库移到参数 db 指定的目标数据库中
返回：1，移动成功；0，移动失败（可能当前数据库中没有这个 key 或者目标数据库中已经有这个 key）

实现

Db.c：394

```
void moveCommand(redisClient *c) {
    robj *o;
    redisDb *src, *dst;
    int srcid;

    /* Obtain source and target DB pointers */
    src = c->db;
    srcid = c->db->id;
    if (selectDb(c,atoi(c->argv[2]->ptr)) == REDIS_ERR) {
        addReply(c,shared.outofrangeerr);
        return;
    }
    dst = c->db;
    selectDb(c,srcid); /* Back to the source DB */

    /* If the user is moving using as target the same
     * DB as the source DB it is probably an error. */
    if (src == dst) {
        addReply(c,shared.sameobjecterr);
        return;
    }

    /* Check if the element exists and get a reference */
    o = lookupKeyWrite(c->db,c->argv[1]);
```

```
    if (!o) {
        addReply(c,shared.czero);
        return;
    }

    /* Return zero if the key already exists in the target DB */
    if (lookupKeyWrite(dst,c->argv[1]) != NULL) {
        addReply(c,shared.czero);
        return;
    }
    dbAdd(dst,c->argv[1],o);
    incrRefCount(o);

    /* OK! key moved, free the entry in the source DB */
    dbDelete(src,c->argv[1]);
    server.dirty++;
    addReply(c,shared.cone);
}
```

保存源 db 的指针和 id
获取目标 db 的 id，如果目标 db 不存在，返回错误
如果当前源 db 与目标 db 一样，则返回错误
在源 db 中查找该 key，如果源 db 中不存在该 key，返回 0
在目标 db 中查找该 key，如果源 db 中已经存在该 key，返回 0
在目标 db 添加该 key
在源 db 中删除该 key
返回 1

OBJECT

使用

格式：OBJECT subcommand key
功能： 查看一个 key 对应的 val 对象的信息，subcommand 有 REFCOUNT，ENCODING，IDLETIME；REFCOUNT 用于查看 key 对应的 val 对象的引用次数；ENCODING 用于查看 key 对应的 val 对象的编码；IDLETIME 用于查看 key 对应的 val 对象的多久没有被读或写的时间，返回单位是秒。
返回：根据 subcommand 不同返回不同的值。

实现

Object.c：493

```
void objectCommand(redisClient *c) {
    robj *o;

    if (!strcasecmp(c->argv[1]->ptr,"refcount") && c->argc == 3) {
        if ((o = objectCommandLookupOrReply(c,c->argv[2],shared.nullbulk))
                == NULL) return;
        addReplyLongLong(c,o->refcount);
    } else if (!strcasecmp(c->argv[1]->ptr,"encoding") && c->argc == 3) {
        if ((o = objectCommandLookupOrReply(c,c->argv[2],shared.nullbulk))
                == NULL) return;
        addReplyBulkCString(c,strEncoding(o->encoding));
    } else if (!strcasecmp(c->argv[1]->ptr,"idletime") && c->argc == 3) {
        if ((o = objectCommandLookupOrReply(c,c->argv[2],shared.nullbulk))
                == NULL) return;
        addReplyLongLong(c,estimateObjectIdleTime(o));
    } else {
        addReplyError(c,"Syntax error. Try OBJECT
(refcount|encoding|idletime)");
    }
}
```

根据发送过来的 subcommand 走不同分支

统一调用 `objectCommandLookupOrReply` 来获取 `key` 对应的 `val` 对象

根据子命令返回对应信息

接下来看看获取对象函数

Object.c：484

```
robj *objectCommandLookupOrReply(redisClient *c, robj *key, robj *reply) {
    robj *o = objectCommandLookup(c,key);

    if (!o) addReply(c, reply);
    return o;
}
```

Object.c：476

```
robj *objectCommandLookup(redisClient *c, robj *key) {
    dictEntry *de;

    if (server.vm_enabled) lookupKeyRead(c->db,key);
    if ((de = dictFind(c->db->dict,key->ptr)) == NULL) return NULL;
    return (robj*) dictGetEntryVal(de);
}
```

获取 key 对应的 val 对象，返回 val 对象，然后在 `objectCommand` 函数中会根据子命令获取对应信息。

再看看 `estimateObjectIdleTime` 函数

Object.c：465

```
unsigned long estimateObjectIdleTime(robj *o) {
    if (server.lruclock >= o->lru) {
        return (server.lruclock - o->lru) * REDIS_LRU_CLOCK_RESOLUTION;
    } else {
        return ((REDIS_LRU_CLOCK_MAX - o->lru) + server.lruclock) *
                    REDIS_LRU_CLOCK_RESOLUTION;
    }
}
```

REDIS_LRU_CLOCK_RESOLUTION 现定义为 10

PERSIST

使用

格式：PERSIST key
功能：把一个有过期时间的 key 的过期时间删掉，即变成永不过期
返回：1，成功删掉 key 的过期时间；0，key 不存在或者 key 本来就没有过期时间

实现

Db.c：589

```
void persistCommand(redisClient *c) {
    dictEntry *de;

    de = dictFind(c->db->dict,c->argv[1]->ptr);
    if (de == NULL) {
        addReply(c,shared.czero);
    } else {
        if (removeExpire(c->db,c->argv[1])) {
            addReply(c,shared.cone);
            server.dirty++;
        } else {
            addReply(c,shared.czero);
        }
    }
}
```

如果查找不到该 key，返回 0
如果成功删除该 key，返回 1，删除失败，返回 0

RANDOMKEY

使用

格式：RANDOMKEY
功能：获取当前 db 中的一个随机 key
返回：随机 key；或者空如果当前 db 为空

实现

Db.c：256

```
void randomkeyCommand(redisClient *c) {
    robj *key;

    if ((key = dbRandomKey(c->db)) == NULL) {
        addReply(c,shared.nullbulk);
        return;
    }

    addReplyBulk(c,key);
    decrRefCount(key);
}
```

Strings

Hashes

Lists

Sets

Sorted sets

Pub/sub

Transactions

Scripting

Connection

Servers

Reids 配置分析

看源码，学算法

Lzf 字符串压缩算法

http://oldhome.schmorp.de/marc/liblzf.html

Sha1 算法

http://svn.ghostscript.com/jbig2dec/trunk/sha1.h
http://svn.ghostscript.com/jbig2dec/trunk/sha1.c

看源码，学编程

结构体中的位操作

Redis.h：251

```
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(redisObjct) minus sizeof(redisObjectVM), so using
     * Redis without VM active will not have any overhead. */
} robj;
```

变长参数的使用

Sds.c：188

```
sds sdscatvprintf(sds s, const char *fmt, va_list ap) {
    va_list cpy;
    char *buf, *t;
    size_t buflen = 16;
```

```
    while(1) {
        buf = zmalloc(buflen);
#ifdef SDS_ABORT_ON_OOM
        if (buf == NULL) sdsOomAbort();
#else
        if (buf == NULL) return NULL;
#endif
        buf[buflen-2] = '\0';
        va_copy(cpy,ap);
        vsnprintf(buf, buflen, fmt, cpy);
        if (buf[buflen-2] != '\0') {
            zfree(buf);
            buflen *= 2;
            continue;
        }
        break;
    }
    t = sdscat(s, buf);
    zfree(buf);
    return t;
}

sds sdscatprintf(sds s, const char *fmt, ...) {
    va_list ap;
    char *t;
    va_start(ap, fmt);
    t = sdscatvprintf(s,fmt,ap);
    va_end(ap);
    return t;
}
```

相关参考地址：

http://blog.csdn.net/bigwhite20xx/article/details/2412880

http://linux.about.com/library/cmd/blcmdl3_va_start.htm

Linux 中的管道通信编程

Vm.c：42

```
void vmInit(void) {
    ...
    int pipefds[2];
    ...
    if (pipe(pipefds) == -1) {
        redisLog(REDIS_WARNING,"Unable to intialized VM: pipe(2): %s. Exiting."
```

```
            ,strerror(errno));
        exit(1);
    }
    server.io_ready_pipe_read = pipefds[0];
    server.io_ready_pipe_write = pipefds[1];
    ...
}
```

Linux 中的多线程编程

Vm 模块的实现中用到了 linux 多线程编程

Vm.c：42

```
void vmInit(void) {
    ...
    pthread_mutex_init(&server.io_mutex,NULL);
    pthread_mutex_init(&server.io_swapfile_mutex,NULL);
    ...
    /* LZF requires a lot of stack */
    pthread_attr_init(&server.io_threads_attr);
    pthread_attr_getstacksize(&server.io_threads_attr, &stacksize);
    ...
    pthread_attr_setstacksize(&server.io_threads_attr, stacksize);
    ...
}
```

Linux 中的文件锁

Vm.c：42

```
void vmInit(void) {
    ...
    struct flock fl;


    ...
    /* Lock the swap file for writing, this is useful in order to avoid
     * another instance to use the same swap file for a config error. */
    fl.l_type = F_WRLCK;
    fl.l_whence = SEEK_SET;
    fl.l_start = fl.l_len = 0;
    if (fcntl(server.vm_fd,F_SETLK,&fl) == -1) {
        redisLog(REDIS_WARNING,
            "Can't lock the swap file at '%s': %s. Make sure it is not used by another
Redis instance.", server.vm_swap_file, strerror(errno));
```

```
        exit(1);
    }
    ...
}
```

Linux 中的多进程编程

Rdb 模块的实现中用到了多进程编程

Linux 中的 epoll 网络事件处理

Redis 的网络事件库很好地封装了网络 io，epoll 是其中的一种实现

编程技巧

类型偏移计算的巧妙利用

intset.c：25

```
static int64_t _intsetGetEncoded(intset *is, int pos, uint8_t enc) {
    int64_t v64;
    int32_t v32;
    int16_t v16;

    if (enc == INTSET_ENC_INT64) {
        memcpy(&v64,((int64_t*)is->contents)+pos,sizeof(v64));
        memrev64ifbe(&v64);
        return v64;
    } else if (enc == INTSET_ENC_INT32) {
        memcpy(&v32,((int32_t*)is->contents)+pos,sizeof(v32));
        memrev32ifbe(&v32);
        return v32;
    } else {
        memcpy(&v16,((int16_t*)is->contents)+pos,sizeof(v16));
        memrev16ifbe(&v16);
        return v16;
    }
}
```

巧妙利用了不同类型的偏移计算

`((int64_t*)is->contents)+pos`，偏移 pos 个 int64_t 大小的字节数

`((int32_t*)is->contents)+pos`，偏移 pos 个 int32_t 大小的字节数

`((int16_t*)is->contents)+pos`，偏移 pos 个 int16_t 大小的字节数

确定系统大小端

见 sha1.c，不过这个宏有点复杂
感叹实现者为了效率付出的劳动

实现自己的数据结构

参考

http://redis.io/
http://pauladamsmith.com/articles/redis-under-the-hood.html
http://pauladamsmith.com/blog/2011/03/redis_get_set.html
http://antirez.com/post/redis-virtual-memory-story.html
http://hi.baidu.com/hins_pan/blog/item/11819838876cfae93a87cebe.html
http://antirez.com/post/redis-persistence-demystified.html
http://antirez.com/post/a-few-key-problems-in-redis-persistence.html