# 概述

本文档学习记录redis3.0源码的笔记

# 数据结构

参考<http://zhangtielei.com/posts/blog-redis-dict.html>

## 简单动态字符串

Simple Dynamic String（SDS），是一个具有长度（len）和剩余空间（free）的字符数组。结构体定义文件sds.h/sdshdr

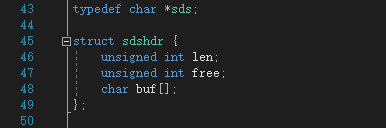


图2- 1 简单动态字符串结构体定义

## 链表

redis中定义的链表结构是一个双向链表，链表节点具有前驱pre和后继next，同时链表具有头节点head和尾节点tail。其定义文件在adlist.h。

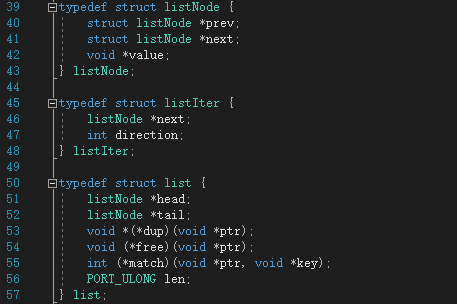


图2- 2 链表结构体定义

## 字典结构

redis中的字典结构（dict）类似于java语言中的hashmap结构，其底层实现是通过数组来存放每一个dictEntry。

dictEntry是一个键值对结构，包含一个void指针的key和一个void指针的value，同时包含一个next指针，该指针是解决hash冲突而采用的拉链法，将所有hash值一样的元素通过next指针串在一起。

dictht是包含dictEntry数组的结构，除了该数组之后，还包含数组大小size，sizemask=size-1（方便获取index，因为size是2的幂，size-1就是二进制全为1的数）。

dict结构体则是包含2个dictht元素的数组（如果没有进行rehash（即rehashidx=-1），只有ht[0]有效，如果rehashidx！=-1，处于rehash过程，所有的操作都要在两个dictht上进行）。

两个hash表最主要的作用是实现dict扩容时（rehash）的增量rehash（incremental rehash），也就是当dict负载超过load factor时，则将进行增量rehash，当该dict有增删改查操作时，需要重新计算index并将key，value存放在另一个hash表。

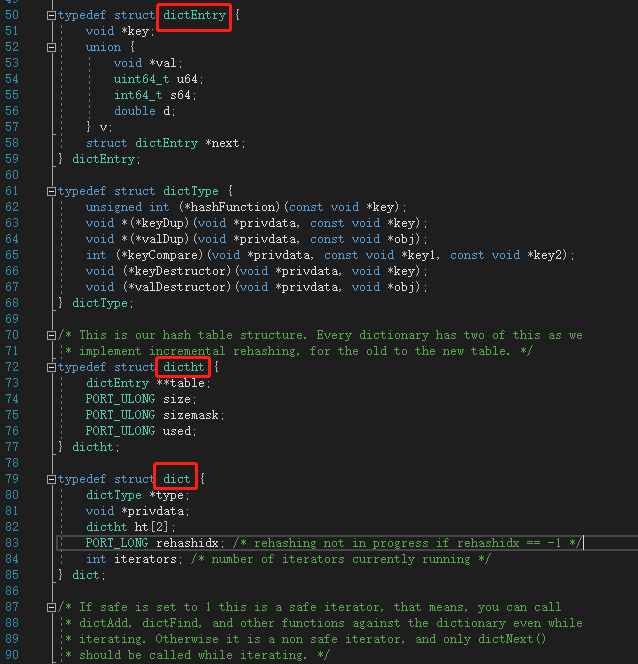


图2- 3 字典结构体定义

Rehash步骤大致如下：

1. 将rehashidx=0，表示在ht[0]中index=0的bucket中所有key需要进行rehash并转移到ht[1]中对应index中；
2. 如果有请求访问redis，先判断rehashidx是否为-1，不为-1则将ht[0]中对应rehashidx进行扩容，同时rehashidx+1，便于下一次请求；

这种做法可以将一次rehash分摊到多次请求上去，这样可以避免某一次请求因为进行rehash而导致响应延时。

有关字典的操作在dict.c文件。

### 创建

创建一个字典结构的源码如下，其主要工作是创建一个dict指针，dict中的两个hash表都未分配空间。

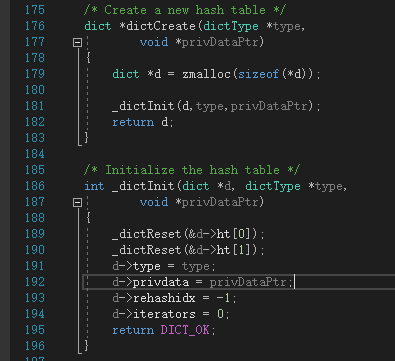


图2- 4 创建字典结构

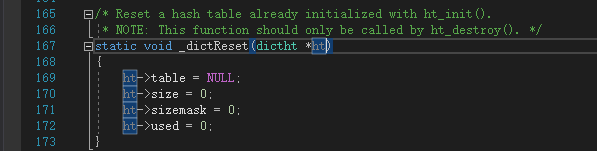


图2- 5初始化dictht

### 增

在dict中增加一个(key，value)分为了两步，第一步是先将key存放在对应bucket，然后再给Entry赋值value，代码如下：

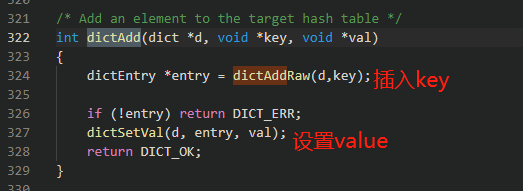


图2- 6 dict插入接口

插入key时，首先判断是否在rehashing，如果在rehash，则推进一步，并且在ht[1]中计算index，否则在ht[0]中计算index；然后再判断是否存在相同的key，存在只返回

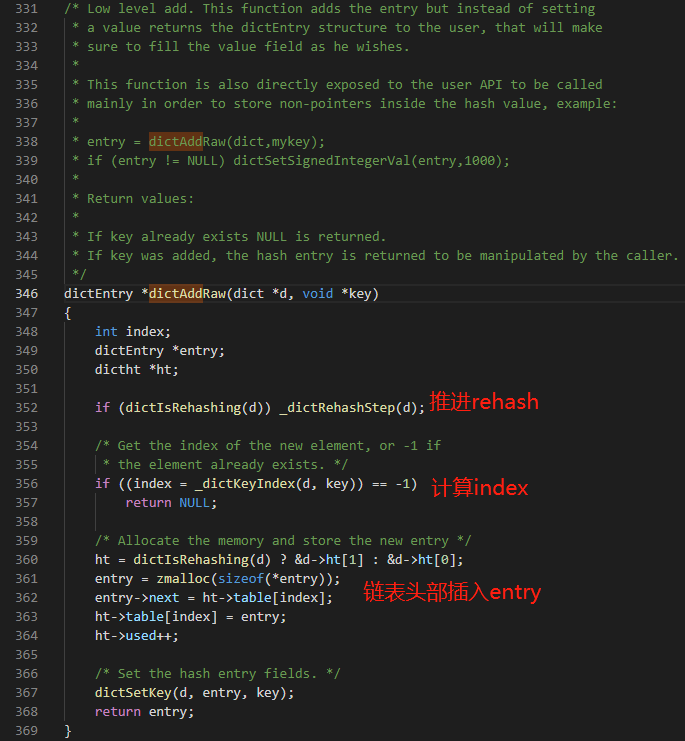


图2- 7 插入前判断

获取key对应的index时，需要先判断对应的key是否已存在，代码如下：

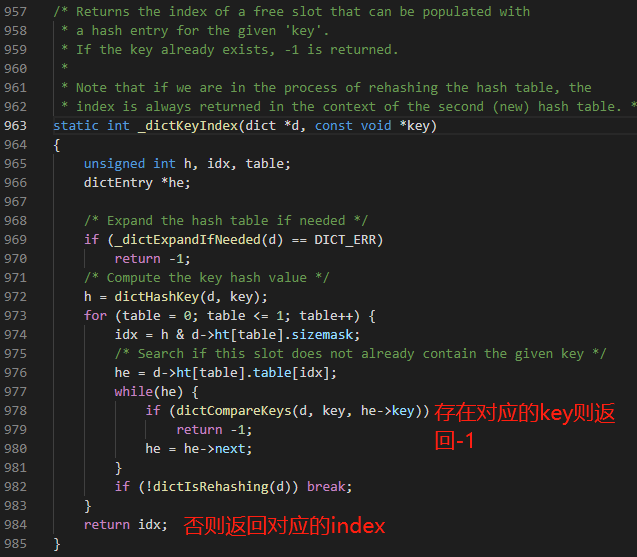


图2- 8 计算index

总结一下：

1. 是否推进rehash；
2. 判断key是否存在，存在返回-1，否则返回对应index；
3. 如果处于rehashing中，在ht[1]中插入
4. 插入时在bucket链表的头部插入

### 删

删除操作比较简单，先判断是否rehashing，然后定位，最后删除。代码如下：

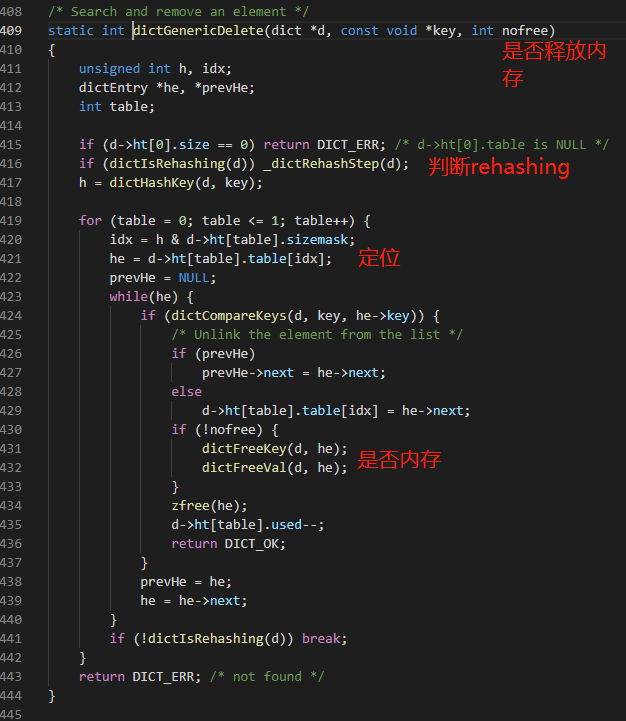


图2- 9删除操作

### 改

修改指定key值的value，先判断rehashing，然后定位，最后修改value。代码如下：

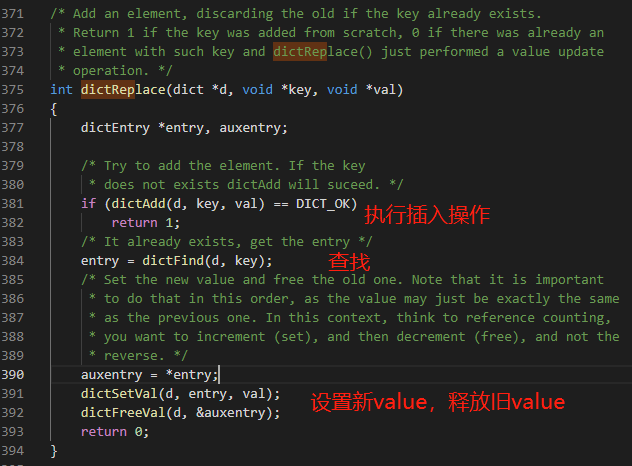


图2- 10 修改操作

### 查

查询指定key值的value，先判断rehashing，然后定位，最后返回value。代码如下：

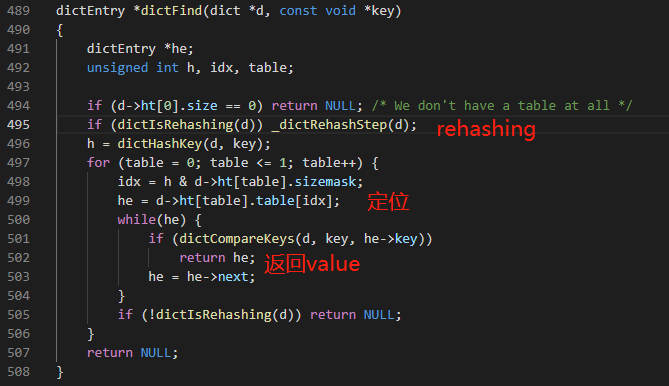


图2- 11 查询操作

### Rehash

（均摊法）Rehash操作可能存在两种情况，第一种就是需要rehash的bucket不为空，则直接推进一步，如果需要rehash的bucket为空，redis规定一次最多处理连续10个空bucket。如果本次rehash之后ht[0]中used=0，表示所有的节点都已rehash，需要先释放ht[0]，然后将ht[1]赋值给ht[0]，ht[1]设置为null。具体代码如下所示：

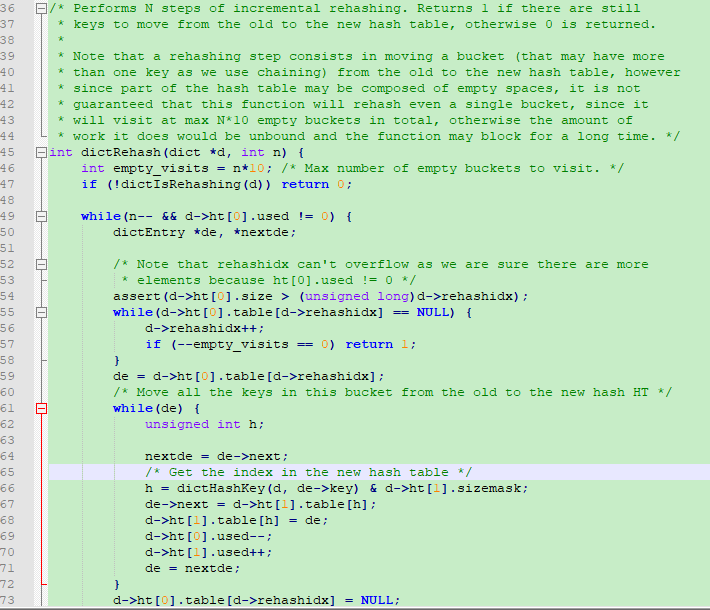


图2- 12 Rehash过程

## 跳表

# 事件

参考：

<https://draveness.me/redis-eventloop>

## 简述

* Redis提供两种事件类型，一种是文件事件，另一种是时间事件；
* ae.h头文件中定义了所有能够操作事件的函数接口，针对文件事件，redis提供了多种io复用方式，包括select、epoll，evport以及kqueue，其中select和epoll用于linux系统，evport用于Solaris系统，kqueue用于mac系统；
* 事件入口是位于ae.c文件中的aeMain方法；通过事件入口方法可以看出redis是一个以时间为驱动的服务器，主线程循环处理每个到来的不同事件；
* 如果既有fileevent需要处理，又有timeevent需要处理，redis服务先处理fileevent，但是处理fileevent前需要计算距离下一个最近的timeevent的时间差距有多大，也就是允许select函数阻塞的最大时间；
* redis的事件被称为AE事件库



图3- 1 事件循环流图示意图

## IO多路复用封装

redis封装的IO多路复用库包括如下：

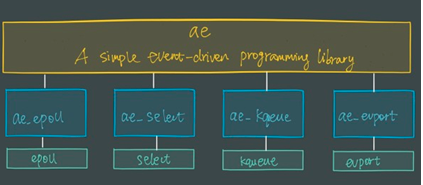


图3- 2 redis封装IO多路复用库

这种封装技术类似于Java提供的接口Interface（此处是我个人理解）。其中ae.h类似Interface，ae\_select,ae\_epoll,ae\_evport和ae\_kqueue是具体实现。Redis使用哪一个具体实现是在编译的时候决定的，通过预编译宏进行，代码如下：



图3- 3 预编译选择IO复用技术

## 类型

事件的抽象定义位于ae.h，类型名为aeEventLoop，这个类的对象相当于一个集合容器，包含文件事件aeFileEvent和时间事件aeTimeEvent，其中文件事件aeFileEvent是个数组，时间事件是个单链表。

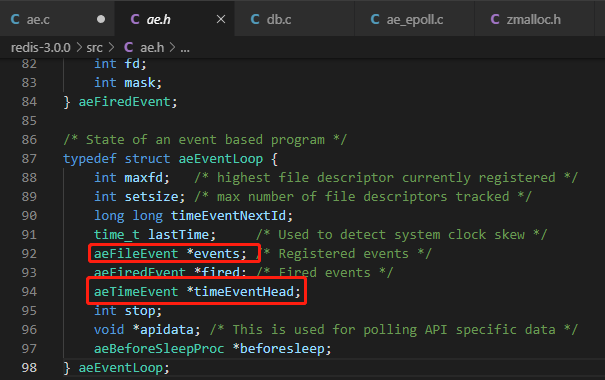


图3- 4 事件类型定义

主要字段解释：

* maxfd：目前所有注册的事件中文件描述符最大的那个fd；maxfd=-1表示无注册事件；maxfd值的修改只会在aeCreateFileEvent和aeDeleteFileEvent方法中；
* setsize：该‘容器’所能存放的最大事件个数，也是文件事件的数组大小；
* timeEventNextId:
* lastTime:记录上一次执行时间事件所处时刻；
* events:文件事件，特指某一个文件事件（此处不是链表）
* fired:待处理的文件事件数组
* timeEventHead:时间事件链表头指针；
* stop:标志字段，如果stop!=0表示无需继续处理事件（详见aeMain方法）
* apidata:
* beforesleep:函数指针，其指向的具体函数位于redis.c文件的beforSleep函数实现；

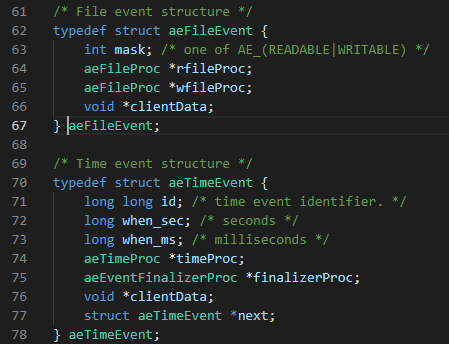


图3- 5 文件事件和时间事件类型定义

aeFileEvent主要字段：

* mask：标志字段，可取值为AE\_READABLE或者AE\_WRITABLE，故名是可写或者可读事件；
* rfileProc:用于处理可读事件的函数指针；
* wfileProc:用于处理可写事件的函数指针；
* clientdata：处理文件事件所需要的参数；

aeTimeEvent主要字段：

* id：唯一标识时间事件的id；
* when\_sec:该时间事件下次执行的秒时刻；
* when\_ms:该时间事件下次执行的毫秒时刻；
* timeProc：到点时需要调用的处理事件的函数指针；
* finalizerProc：指向时间事件被终止时需要执行的函数指针；
* clientData:时间事件处理时所需要的参数；
* next：

### 文件事件

### 时间事件

有个特殊地方是如果处理完一个时间事件之后，需要将指针指向时间事件链表头部重新进行执行，这样做的目的是为了防止漏掉已到点的事件，比如链表第一个节点在3时刻到点执行，第二个节点在1时刻到点执行，该事件需要执行3个时钟，因此在执行完第二节点之后已经是位于4时刻，此时第一个节点已经到点可以执行了；

## 执行流程

### aeMain---事件循环体

以事件为驱动的Redis服务器的事件循环入口位于ae.c的aeMain函数。其代码结构如下：

void aeMain (aeEventLoop \*eventLoop) {

//stop不为0表示停止处理事件

eventLoop->stop = 0;

while (!eventLoop->stop) {

//beforesleep可以简单的理解就是事件处理前的预操作

if (eventLoop->beforesleep != NULL)

eventLoop->beforesleep(eventLoop);

//循环处理事件

aeProcessEvents(eventLoop, AE\_ALL\_EVENTS);

}

}

整个循环过程可以通过下图概括，其中通过虚线隔开的模块表示并非每一轮循环都要执行的部分：

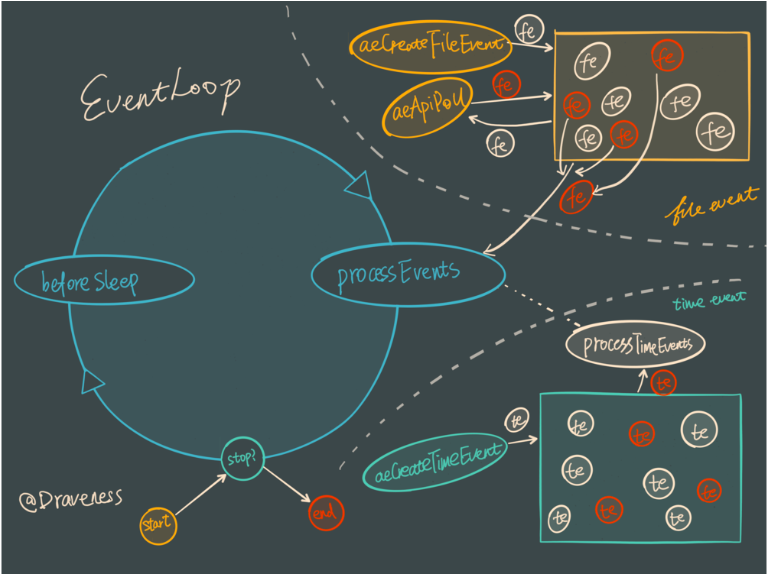


图3- 6 事件循环示意图

### aeProcessEvents—真正处理事件函数

真正处理事件的函数名是aeProcessEvents，其具体实现如下：

//flag用于表示本次是处理fileevent还是timeevent，亦或者两者都要处理，通过aeMain入口可知，AE\_ALL\_EVENTS表示两者都要处理

int aeProcessEvents(aeEventLoop \*eventLoop, int flags)

{

int processed = 0, numevents;

/\* Nothing to do? return ASAP \*/

if (!(flags & AE\_TIME\_EVENTS) && !(flags & AE\_FILE\_EVENTS)) return 0;

/\* Note that we want call select() even if there are no

\* file events to process as long as we want to process time

\* events, in order to sleep until the next time event is ready

\* to fire. \*/

//maxfd=-1表示没有fileevent需要处理，否则表示有fileevent已经注册，本次循环需要处理

//处理fileevent前，需要找到最近时间事件的shortest，计算当前时间和shortest之间的时间差距（微秒），这个时间差距用来表示select函数能够阻塞的最大时间

if (eventLoop->maxfd != -1 ||

((flags & AE\_TIME\_EVENTS) && !(flags & AE\_DONT\_WAIT))) {

int j;

aeTimeEvent \*shortest = NULL;

struct timeval tv, \*tvp;

//找到离当前时间最近的timeevent

if (flags & AE\_TIME\_EVENTS && !(flags & AE\_DONT\_WAIT))

shortest = aeSearchNearestTimer(eventLoop);

if (shortest) {

long now\_sec, now\_ms;

/\* Calculate the time missing for the nearest

\* timer to fire. \*/

aeGetTime(&now\_sec, &now\_ms);

tvp = &tv;

tvp->tv\_sec = shortest->when\_sec - now\_sec;

//计算shortest和now之间的时间差距（单位是微秒）

//这里的计算算法是比较毫秒when\_ms，如果shortest的when\_ms更小，则向秒位借一位

if (shortest->when\_ms < now\_ms) {

tvp->tv\_usec = ((shortest->when\_ms+1000) - now\_ms)\*1000;

tvp->tv\_sec --;//向秒位借位

} else {

tvp->tv\_usec = (shortest->when\_ms - now\_ms)\*1000;

}

//如果shortest-now<0表示shortest可以触发了，因此将tvp设置为0以便于通知select立即返回，不用阻塞，否则如果对于0，select可以阻塞的时间就是tvp指定的时间

if (tvp->tv\_sec < 0) tvp->tv\_sec = 0;

if (tvp->tv\_usec < 0) tvp->tv\_usec = 0;

} else {

/\* If we have to check for events but need to return

\* ASAP because of AE\_DONT\_WAIT we need to set the timeout

\* to zero \*/

if (flags & AE\_DONT\_WAIT) {

tv.tv\_sec = tv.tv\_usec = 0;

tvp = &tv;

} else {

/\* Otherwise we can block \*/

//tvp为null，通知select可以一直阻塞指导有可处理的fileevent出现为止

tvp = NULL; /\* wait forever \*/

}

}

//调用redis封装好的io多路复用库，常用的是select和epoll

//返回值是需要处理的fileevent个数，同时所有可处理的fileevent位于fired数组

numevents = aeApiPoll(eventLoop, tvp);

//遍历所有已准备好的fileevent

for (j = 0; j < numevents; j++) {

aeFileEvent \*fe = &eventLoop->events[eventLoop->fired[j].fd];

int mask = eventLoop->fired[j].mask;

int fd = eventLoop->fired[j].fd;

int rfired = 0;

     /\* note the fe->mask & mask & ... code: maybe an already processed

\* event removed an element that fired and we still didn't

\* processed, so we check if the event is still valid. \*/

//根据mask判断是read还是write操作，选择对应的handler

if (fe->mask & mask & AE\_READABLE) {

rfired = 1;

fe->rfileProc(eventLoop,fd,fe->clientData,mask);

}

if (fe->mask & mask & AE\_WRITABLE) {

if (!rfired || fe->wfileProc != fe->rfileProc)

fe->wfileProc(eventLoop,fd,fe->clientData,mask);

}

processed++;

}

}

/\* Check time events \*/

//处理timeevent

if (flags & AE\_TIME\_EVENTS)

processed += processTimeEvents(eventLoop);

return processed; /\* return the number of processed file/time events \*/

}

### aeApiPoll---筛选准备好的fileevent

该函数封装了不同系统中的IO多路复用具体实现。该函数需要详细理解Select和epoll函数的原理。

### processTimeEvents---timeevent处理函数

/\* Process time events \*/

static int processTimeEvents(aeEventLoop \*eventLoop) {

int processed = 0;

aeTimeEvent \*te;

long long maxId;

time\_t now = time(NULL);

/\* If the system clock is moved to the future, and then set back to the

\* right value, time events may be delayed in a random way. Often this

\* means that scheduled operations will not be performed soon enough.

\*

\* Here we try to detect system clock skews, and force all the time

\* events to be processed ASAP when this happens: the idea is that

\* processing events earlier is less dangerous than delaying them

\* indefinitely, and practice suggests it is. \*/

//此处有个异常情况的考虑，lasttime记录了上次处理timeevent的时间，如果now小于lasttime，表明系统时钟被修改过，因此将所有timeevent设置为马上执行，Redis如此考虑的原因是认为timeevent延迟处理的风险比提前执行的风险更大，因此提前执行以便给相关人员notify并及时处理

if (now < eventLoop->lastTime) {

te = eventLoop->timeEventHead;

while(te) {

te->when\_sec = 0;

te = te->next;

}

}

eventLoop->lastTime = now;

te = eventLoop->timeEventHead;

maxId = eventLoop->timeEventNextId-1;

while(te) {

long now\_sec, now\_ms;

long long id;

//如果id大于maxId，则跳过，为了避免死循环

if (te->id > maxId) {

te = te->next;

continue;

}

aeGetTime(&now\_sec, &now\_ms);

//执行符合时间条件的timevent

if (now\_sec > te->when\_sec ||

(now\_sec == te->when\_sec && now\_ms >= te->when\_ms))

{

int retval;

id = te->id;

//timeevent处理函数的返回值是该事件下次执行的时刻与now的时差

retval = te->timeProc(eventLoop, id, te->clientData);

processed++;

/\* After an event is processed our time event list may

\* no longer be the same, so we restart from head.

\* Still we make sure to don't process events registered

\* by event handlers itself in order to don't loop forever.

\* To do so we saved the max ID we want to handle.

\*

\* FUTURE OPTIMIZATIONS:

\* Note that this is NOT great algorithmically. Redis uses

\* a single time event so it's not a problem but the right

\* way to do this is to add the new elements on head, and

\* to flag deleted elements in a special way for later

\* deletion (putting references to the nodes to delete into

\* another linked list). \*/

//如果该timeevent需要继续执行，设置其下次执行时间，时差就是retval

if (retval != AE\_NOMORE) {

aeAddMillisecondsToNow(retval,&te->when\_sec,&te->when\_ms);

} else {

//如果该timeevent执行本次之后不再执行，则从链表删除

aeDeleteTimeEvent(eventLoop, id);

}

//从头开始遍历timeevent，因为在处理timeevent的过程中可能会产生新的timeevent，但是新的timeevent一定比maxid大，因此在这一轮不会被执行，也就避免了死循环

te = eventLoop->timeEventHead;

} else {

te = te->next;

}

}

return processed;

}

该函数处理的大致流程如所示：



图3- 7 processTimeEvents函数大致流程

举个具体事例，目前timeevent链表有3个节点，其执行时刻分别为3,4,5时刻，其id分别为3,2,1，并且id=2的节点时周期为2的周期timeevent，id=3是个定时timeevent。因此链表的timeEventNextId=4，记录此时的maxid = timeEventNextId-1 = 3，需要保存这个值，后续所有的节点都要和maxid作比较，如图所示：



此时now=4 (整个过程now不会改变)时刻进入processTimeEvent函数，因此从头开始遍历节点，第一个节点id=3，等于maxid，但是time=5>now，跳过无需执行，第二个节点id=2，小于maxid并且time=4等于now，需要执行，在执行id=2的节点时产生一个新的timeevent，通过通过aeCreateTimeEvent函数可以得出新的链表结构，同时将id=2的下一次执行时刻改为6（因为周期为2）；



id=2的节点执行完之后，从头节点id=4的节点开始重新执行，按照同样的方式，首先id=4大于maxid=3，跳过这个节点，id=3等于maxid，但是time=5>now,跳过，id=2节点大于now，跳过，id=1的节点符合要求，执行节点，由于是定时节点，id=1的节点执行之后将被删除，因此最终链表如下图所示。



# 服务端Server

## 概述

Redis服务端Server从宏观层面上来分析可以分为三个阶段，第一阶段是服务启动时的初始化，第二阶段是循环处理Event阶段，第三阶段是服务停止后的清理阶段。伪代码可以如下

int main(int argc,char\*\*argv)

{

init\_server();

aeMain();

clean\_server();

return 0;

}

其中第二阶段的aeMain执行流程已经在第3.4章节做了介绍，因此本章节主要介绍第一阶段和第三阶段。

## 服务器初始化

服务器的初始化可以细分为多个小阶段，每个阶段负责完成相应的初始化工作。

* 加载默认配置
* 初始化Sentinel（可选）
* 解析启动命令
* 加载配置（可选）
* 分配内存空间

### 加载默认配置

此阶段是给redis所有配置参数设置默认值，也就是加载redis内部设置的默认值，比如在该阶段会为dbnum（数据库个数）设置默认值16。此阶段通过调用redis.c文件中实现的initServerConfig方法进行的。

加载默认值必须最先执行是因为后续动作需要用到一些配置参数，如果不设置初始值会导致一些异常发生，因此需要提前设置默认值；

该阶段执行完成后，所有可配置的参数都有初始值，此时并未为Redis服务器分配用于存放data的内存空间。

### 初始化Sentinel（可选）

这一步是可选操作，需要根据启动命令或者启动参数来判断是否执行这一步。如果启动服务器采用“redis-sentinel”命令或者命令中带有“-sentinel”参数，则需要进行sentinel的初始化操作。

判断是否初始化sentinel的方法实现如下，位于redis.c/checkForSentinelMode

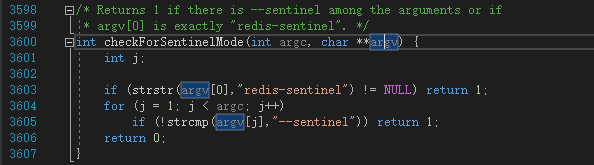


图3- 8 sentinel模式判断逻辑

### 解析启动命令

经过上面的初始化操作之后，接下来需要对启动命令中带有的参数进行解析。比如启动服务时的命令带有“--version”，“--help”等参数时，需要根据对应的参数执行相应的动作。如果启动命令指定配置文件（conf），需要解析出配置文件的路径并保存。理解这一步需要了解redis启动的所有命令。

### 加载配置（可选）

如果4.2.3解析的启动命令指定了配置文件，服务启动时需要加载指定的配置文件并为相应的配置参数进行赋值。加载配置方法如下，位于config.c/loadServerCofig。

到此，服务器所有需要的配置参数已经设置完成，包括默认的初始值和用户指定的配置参数值，而此时的redis服务器仍然没有分配存放数据的内存空间。

### 分配内存空间

该阶段涉及的细节比较多，有很多的变量需要为其分配空间，并设置空值。很多细节需要跟着源代码走才能理清，但是该阶段大的动作主要包括：

* 创建aeEventLoop对象；
* 创建db数组并为其赋初始值；
* 创建fd监听指定ip下的指定port；
* 创建aeTimeEvent数组并注册时间处理函数；

为服务器server分配空间，设置server对象中各个属性的值，主要包括如下过程：

void initServer(void) {

int j;

signal(SIGHUP, SIG\_IGN);

signal(SIGPIPE, SIG\_IGN);

setupSignalHandlers();

if (server.syslog\_enabled) {

openlog(server.syslog\_ident, LOG\_PID | LOG\_NDELAY | LOG\_NOWAIT,

server.syslog\_facility);

}

/\*\*\*part 1:为server中的一些变量分配空间\*\*\*\*/

server.pid = getpid();

server.current\_client = NULL;

server.clients = listCreate();

server.clients\_to\_close = listCreate();

server.slaves = listCreate();

server.monitors = listCreate();

server.slaveseldb = -1; /\* Force to emit the first SELECT command. \*/

server.unblocked\_clients = listCreate();

server.ready\_keys = listCreate();

server.clients\_waiting\_acks = listCreate();

server.get\_ack\_from\_slaves = 0;

server.clients\_paused = 0;

/\*\*\*\*\*\*\*\*\*end part 1\*\*\*\*/

//创建一些共享对象，比如“OK”字符串常量等等

createSharedObjects();

adjustOpenFilesLimit();

//初始化eventloop对象

server.el = aeCreateEventLoop(server.maxclients+REDIS\_EVENTLOOP\_FDSET\_INCR);

//初始化db数组

server.db = zmalloc(sizeof(redisDb)\*server.dbnum);

/\* Open the TCP listening socket for the user commands. \*/

//监听指定ip的指定port，如果未指定ip和port，则监听所有网络接口的默认端口，ipfd\_count记录了需要监听的个数，ipfd数组记录所有的文件描述符fd

if (server.port != 0 &&

listenToPort(server.port,server.ipfd,&server.ipfd\_count) == REDIS\_ERR)

exit(1);

/\* Open the listening Unix domain socket. \*/

if (server.unixsocket != NULL) {

unlink(server.unixsocket); /\* don't care if this fails \*/

server.sofd = anetUnixServer(server.neterr,server.unixsocket,

server.unixsocketperm, server.tcp\_backlog);

if (server.sofd == ANET\_ERR) {

redisLog(REDIS\_WARNING, "Opening Unix socket: %s", server.neterr);

exit(1);

}

anetNonBlock(NULL,server.sofd);

}

/\* Abort if there are no listening sockets at all. \*/

//需要监听的fd为0则直接终止

if (server.ipfd\_count == 0 && server.sofd < 0) {

redisLog(REDIS\_WARNING, "Configured to not listen anywhere, exiting.");

exit(1);

}

/\* Create the Redis databases, and initialize other internal state. \*/

//为每一个db对象分配空间

for (j = 0; j < server.dbnum; j++) {

server.db[j].dict = dictCreate(&dbDictType,NULL);

server.db[j].expires = dictCreate(&keyptrDictType,NULL);

server.db[j].blocking\_keys = dictCreate(&keylistDictType,NULL);

server.db[j].ready\_keys = dictCreate(&setDictType,NULL);

server.db[j].watched\_keys = dictCreate(&keylistDictType,NULL);

server.db[j].eviction\_pool = evictionPoolAlloc();

server.db[j].id = j;

server.db[j].avg\_ttl = 0;

}

server.pubsub\_channels = dictCreate(&keylistDictType,NULL);

server.pubsub\_patterns = listCreate();

listSetFreeMethod(server.pubsub\_patterns,freePubsubPattern);

listSetMatchMethod(server.pubsub\_patterns,listMatchPubsubPattern);

server.cronloops = 0;

server.rdb\_child\_pid = -1;

server.aof\_child\_pid = -1;

server.rdb\_child\_type = REDIS\_RDB\_CHILD\_TYPE\_NONE;

//aof重写时需要使用该buffer

aofRewriteBufferReset();

server.aof\_buf = sdsempty();

server.lastsave = time(NULL); /\* At startup we consider the DB saved. \*/

server.lastbgsave\_try = 0; /\* At startup we never tried to BGSAVE. \*/

server.rdb\_save\_time\_last = -1;

server.rdb\_save\_time\_start = -1;

server.dirty = 0;

resetServerStats();

/\* A few stats we don't want to reset: server startup time, and peak mem. \*/

server.stat\_starttime = time(NULL);

server.stat\_peak\_memory = 0;

server.resident\_set\_size = 0;

server.lastbgsave\_status = REDIS\_OK;

server.aof\_last\_write\_status = REDIS\_OK;

server.aof\_last\_write\_errno = 0;

server.repl\_good\_slaves\_count = 0;

updateCachedTime();

/\* Create the serverCron() time event, that's our main way to process

\* background operations. \*/

//系统后台处理操作的周期timeevent，到点执行函数serverCron，该函数需要后续详细介绍，该函数返回的是一个大于0的int，表示所有执行该函数的timeevent都是周期timeevent

if(aeCreateTimeEvent(server.el, 1, serverCron, NULL, NULL) == AE\_ERR) {

redisPanic("Can't create the serverCron time event.");

exit(1);

}

/\* Create an event handler for accepting new connections in TCP and Unix

\* domain sockets. \*/

for (j = 0; j < server.ipfd\_count; j++) {

if (aeCreateFileEvent(server.el, server.ipfd[j], AE\_READABLE,

acceptTcpHandler,NULL) == AE\_ERR)

{

redisPanic(

"Unrecoverable error creating server.ipfd file event.");

}

}

if (server.sofd > 0 && aeCreateFileEvent(server.el,server.sofd,AE\_READABLE,

acceptUnixHandler,NULL) == AE\_ERR) redisPanic("Unrecoverable error creating server.sofd file event.");

/\* Open the AOF file if needed. \*/

if (server.aof\_state == REDIS\_AOF\_ON) {

server.aof\_fd = open(server.aof\_filename,

O\_WRONLY|O\_APPEND|O\_CREAT,0644);

if (server.aof\_fd == -1) {

redisLog(REDIS\_WARNING, "Can't open the append-only file: %s",

strerror(errno));

exit(1);

}

}

/\* 32 bit instances are limited to 4GB of address space, so if there is

\* no explicit limit in the user provided configuration we set a limit

\* at 3 GB using maxmemory with 'noeviction' policy'. This avoids

\* useless crashes of the Redis instance for out of memory. \*/

if (server.arch\_bits == 32 && server.maxmemory == 0) {

redisLog(REDIS\_WARNING,"Warning: 32 bit instance detected but no memory limit set. Setting 3 GB maxmemory limit with 'noeviction' policy now.");

server.maxmemory = 3072LL\*(1024\*1024); /\* 3 GB \*/

server.maxmemory\_policy = REDIS\_MAXMEMORY\_NO\_EVICTION;

}

if (server.cluster\_enabled) clusterInit();

replicationScriptCacheInit();

scriptingInit();

slowlogInit();

latencyMonitorInit();

bioInit();

}

1. 初始化db

初始化db主要为db数组分配连续的内存空间，数组大小默认为16，可以通过配置文件设置dbnum。

### 加载持久化数据

加载持久化数据方法位置redis.c/loadDataFromDisk。Redis提供两种数据持久化方案，一种是AOF，一种是RDB方式，通过代码可以知道，Redis优先加载AOF方式，只要aof方式状态为on，则采用AOF加载，否则加载RDB文件。两种加载方式逻辑需要重点关注。

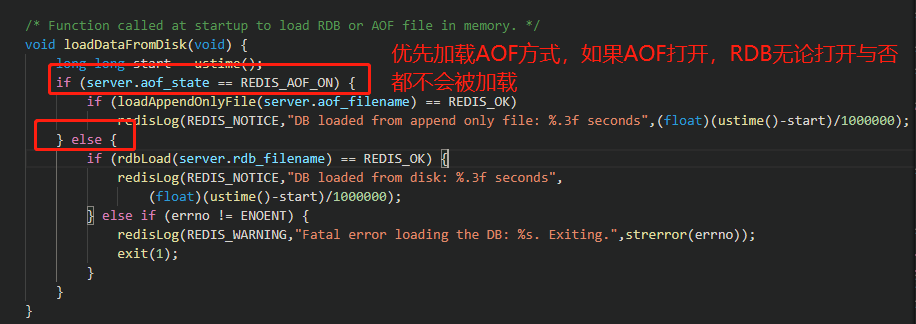


图2- 13 aof和rdb优先级

#### 加载AOF

在理解加载AOF文件时，首先需要理解AOF文件存储格式，如下是一个aof文件样本：

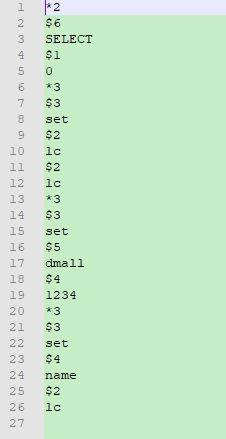


图3- 9 aof文件样本

图3- 2 对应redis命令如下：

>select 0

>set lc lc

>set dmall 1234

>set name lc

每一个redis命令行存放到aof文件时都按照如下格式进行存储

* 假设改命令行具有n个参数（包括指令，比如set，hset等）
* aof文件需要存放2\*n + 1行
* 第一行表示n，并以星号（\*）开始；
* 然后每一个参数占用2行，前一行表示该参数长度，以($)打头，后一个参数就是具体值

Aof文件加载流程图可以大致如图3- 3所示：



图3- 10 aof文件加载流程

Redis中fake client与一般的client最大的区别是其fake client的套接字编号fd为-1，因为fake client不能用于处理网络事件，只能用于载入aof文件和执行lua脚本。

#### 加载RDB

Redis产生的RDB文件都是一个二进制文件，文件内容不存在空格，换行符等。文件结构格式如所示：

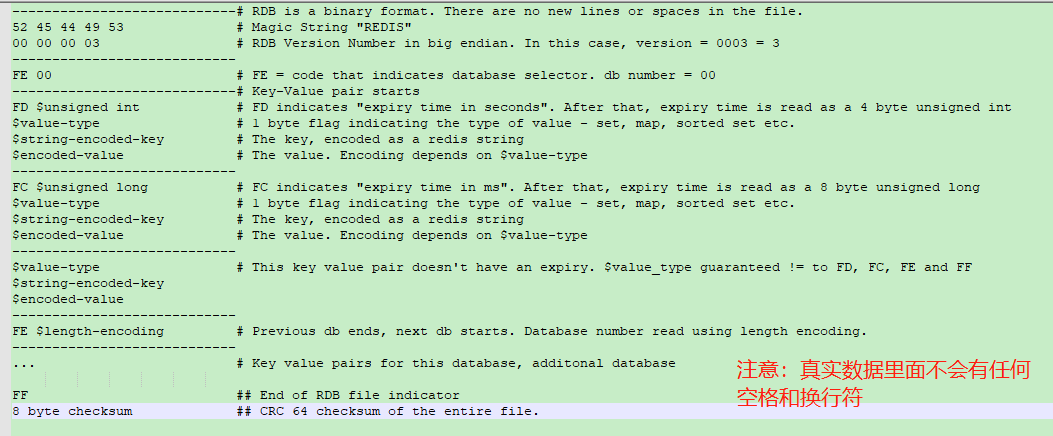


图3- 11 RDB文件格式

注：

参考地址<https://github.com/sripathikrishnan/redis-rdb-tools/wiki/Redis-RDB-Dump-File-Format>

## 服务器清理

## 重要函数

serverCron

# 客户端

# 持久化

Redis持久化包括两种方式，一种是aof（appendonlyfile），另外一种是rdb。针对这两种方式，现详细学习记录其实现原理。

参考

<https://www.cnblogs.com/kismetv/p/9137897.html#t4>

## AOF

打开或者关闭AOF持久化可以采用三种方式，分别是：

* Redis配置文件redis.conf配置appendonly yes
* Redis运行过程（run time）中通过config 命令打开，比如config set appendonly yes
* Redis运行过程（run time）中通过config 命令关闭，比如config set appendonly no

### 配置文件打开aof

通过这种方式启动aof在redis启动时检测配置文件，前提appendonly为yes，详见3.6.1。

### config命令打开aof

通过这种方式需要client发送‘config set appendonly yes’命令给服务器。服务器接收到该命令后，先解析config的指令，需要调用config.c/configSetCommd方法，该方法会根据config指令的参数调用aof.c/startAppendOnly函数。代码如下：

该方法的核心是调用rewriteAppendOnlyFileBackground函数，其主要流程如下

1. Redis父进程首先判断当前是否存在正在执行 bgsave/bgrewriteaof的子进程，如果存在则bgrewriteaof命令直接返回，如果存在bgsave命令则等bgsave执行完成后再执行。前面曾介绍过，这个主要是基于性能方面的考虑。
2. 父进程执行fork操作创建子进程，这个过程中父进程是阻塞的。
3. 父进程fork后，bgrewriteaof命令返回”Background append only file rewrite started”信息并不再阻塞父进程，并可以响应其他命令。Redis的所有写命令依然写入AOF缓冲区，并根据appendfsync策略同步到硬盘，保证原有AOF机制的正确。
4. 由于fork操作使用写时复制技术，子进程只能共享fork操作时的内存数据。由于父进程依然在响应命令，因此Redis使用AOF重写缓冲区(图中的aof\_rewrite\_buf)保存这部分数据，防止新AOF文件生成期间丢失这部分数据。也就是说，bgrewriteaof执行期间，Redis的写命令同时追加到aof\_buf和aof\_rewirte\_buf两个缓冲区。
5. 子进程根据内存快照，按照命令合并规则写入到新的AOF文件。
6. 子进程写完新的AOF文件后，向父进程发信号，父进程更新统计信息，具体可以通过info persistence查看。
7. 父进程把AOF重写缓冲区的数据写入到新的AOF文件，这样就保证了新AOF文件所保存的数据库状态和服务器当前状态一致。
8. 使用新的AOF文件替换老文件，完成AOF重写。

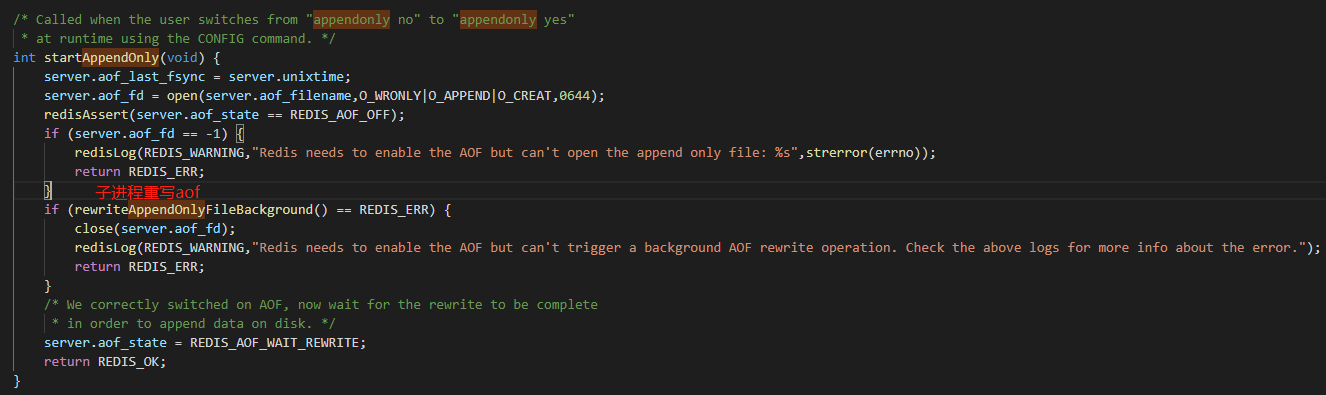


图5- 1 config命令打开aof

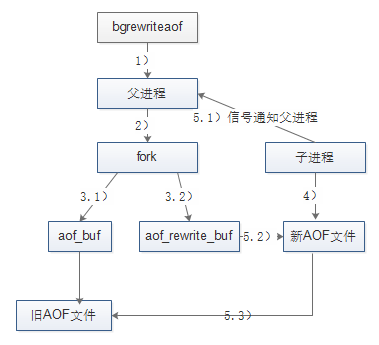


图5- 2 bgrewriteaof函数流程图

### config命令关闭aof

通过config命令关闭aof的具体实现在aof.c/stopAppendOnly，其基本思路是将aof\_buf中缓存的数据flush到磁盘，然后设置相应的server参数。

## RDB

# IO