# 概述

本文档学习记录redis3.0源码的笔记，按照Redis的启动流程可以直接从第4章开始阅读。

# 数据结构

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

对外提供的五种数据类型分别对应其内部数据结构关系如下：

* string---sds
* list----list
* set----dict，其中dict结构的键值对中value为null，类似于java中HashSet，所有key的value都是同一个object；
* zset----dict和zskiplist的组合，dict存放key，score键值对，便于查找，zskiplist存放score的顺序
* hash----dict

所有对这五种数据类型的操作命令的实现函数都位于t\_string.c、t\_list.c、t\_set.c、t\_zset.c和t\_hash.c这五个文件中。

## 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

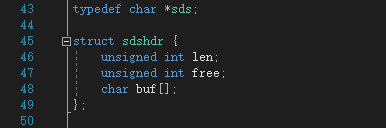


图2- 1 简单动态字符串结构体定义

## 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。

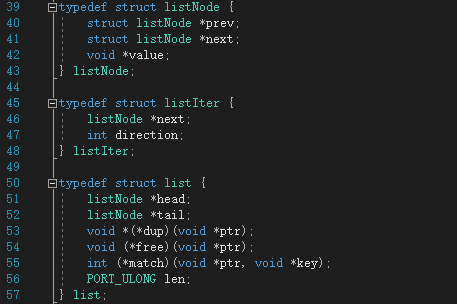


图2- 2 链表结构体定义

## 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。

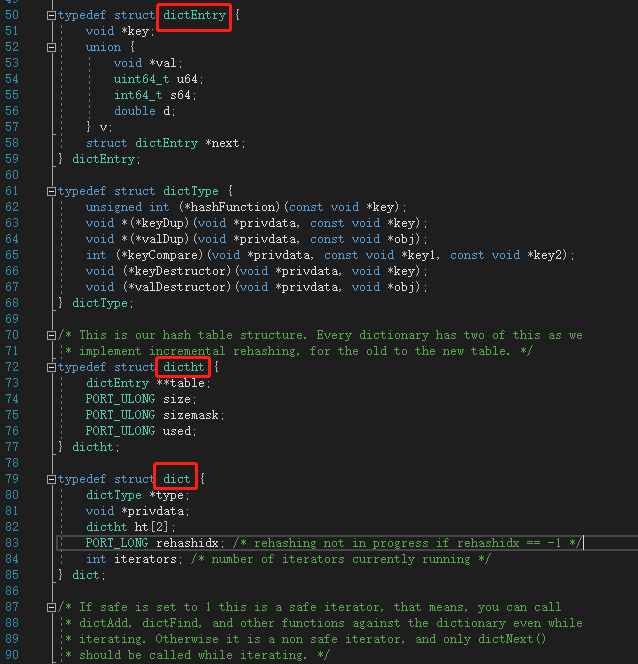


图2- 3 字典结构体定义

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。

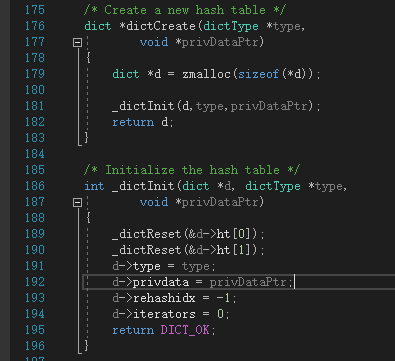


图2- 4 创建字典结构

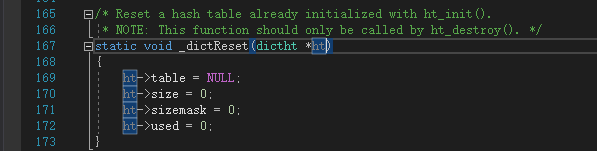


图2- 5初始化dictht

### 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：

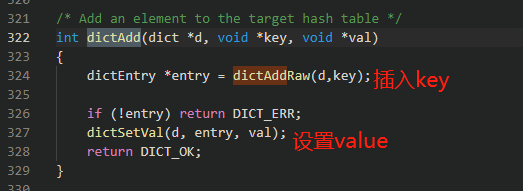


图2- 6 dict插入接口

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

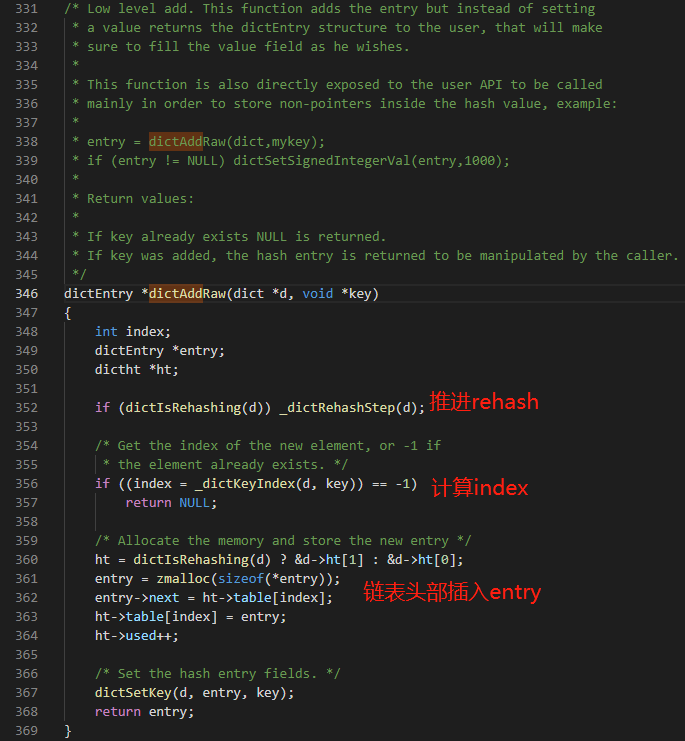


图2- 7 插入前判断

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：

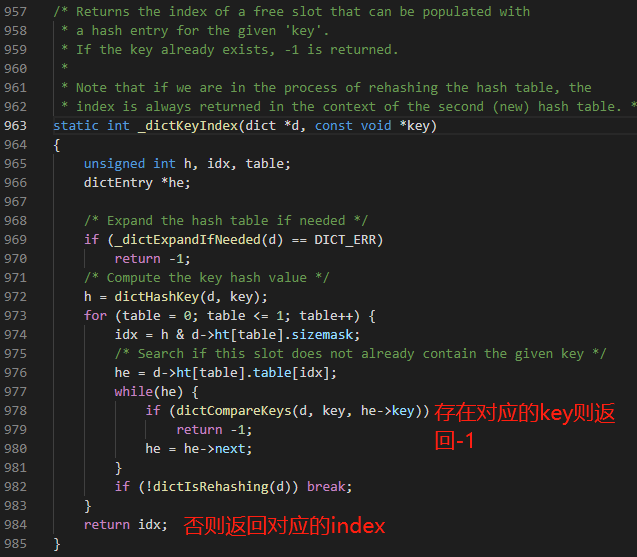


图2- 8 计算index

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：

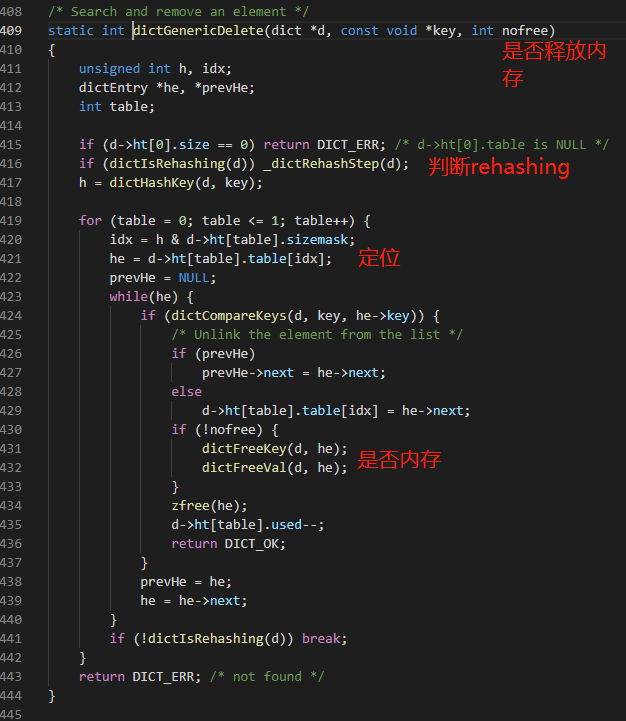


图2- 9删除操作

### 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：

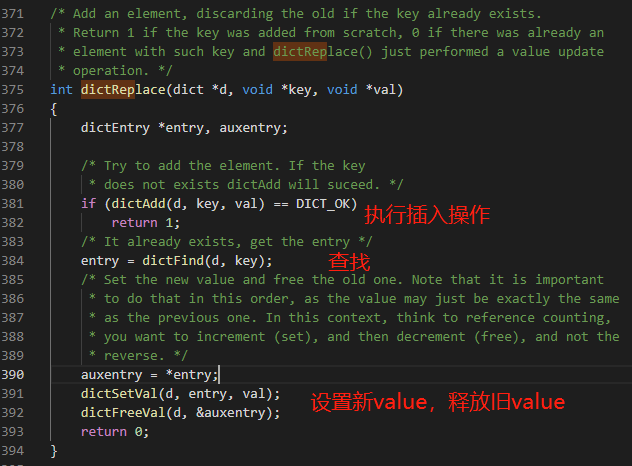


图2- 10 修改操作

### 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：

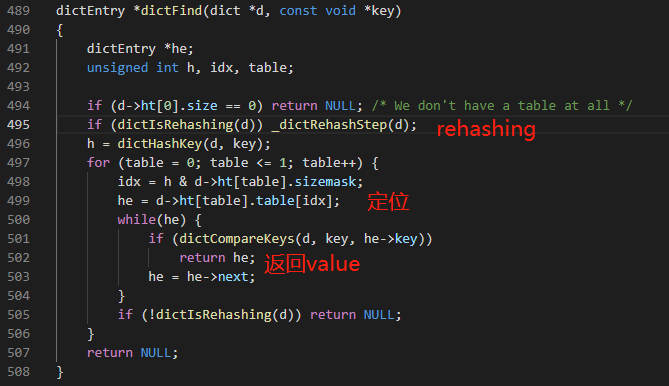


图2- 11 查询操作

### 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：

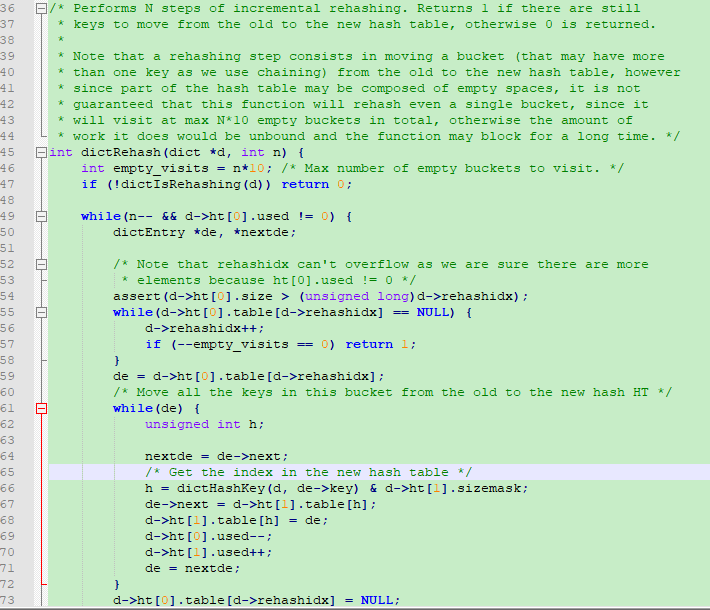


图2- 12 Rehash过程

## 跳表

参考<https://blog.csdn.net/yangbodong22011/article/details/78523885>

# 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是一个以事件为驱动的服务器，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：

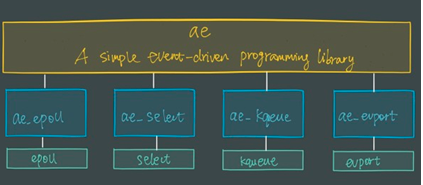


图3- 2 redis封装IO多路复用库

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。

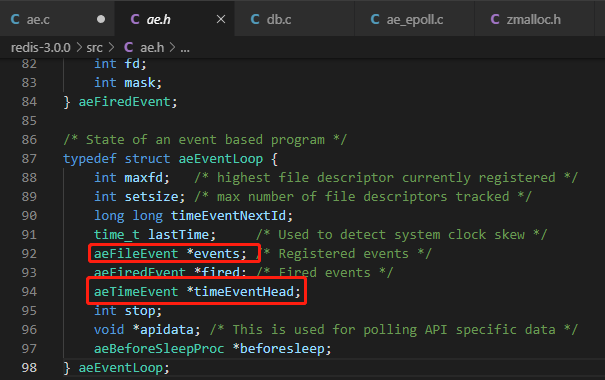


图3- 4 事件类型定义

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:文件事件数组，redis在初始化内存就会为events分配好数组大小，数组下标就是事件的fd，因此整个系统不能产生比events.size – 1还大的fd
* fired:已就绪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；

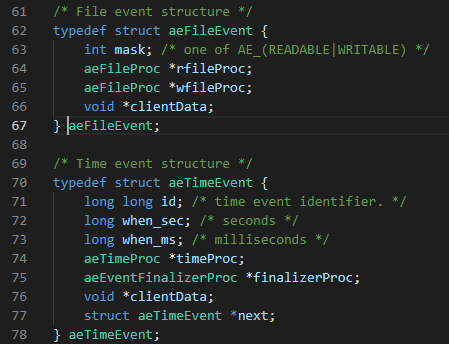


图3- 5 文件事件和时间事件类型定义

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

按照功能进行区分可以将文件事件分为连接（connection）事件，查询（包括读和写）事件。连接事件在第一阶段创建，用于监听绑定（bind）到redis上的ip和port端口，如果有connection到达，则调用连接事件对于的回调函数（accepTcpHandler），这个回调函数主要的功能是创建一个redisClient，同时创建一个查询（读）事件，该事件的回调函数是readQueryFromClient，并将该client挂在这个查询事件上，查询（读）事件处理完成后，最后会prepareClientToWrite函数中创建一个查询（写）事件，该事件的回调函数是sendReplyToClient，该函数会将prepareClientToWrite函数创建的查询（写）事件删除。

查询（读）事件会在client断开连接调用freeClient函数时将此事件删除；

查询（写）事件在执行完写操作之后就会被删除；

Redis中处理事件的主要函数依次包括：acceptTcpHandler，readQueryFromClient，processInputBuffer，processCommand，call，命令对应的xxxCommand函数，addReply，sendReplyToClient。

### 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；

Redis如何避免时间事件的死循环问题并没有给出具体的解释，可能是Redis不适合这种时间事件执行时间太长的场景。适合IO密集型而不是CPU密集型的场景。因为如果一个时间事件的执行时间（CPU占用时间）过长，会导致其下一个执行周期已经开始但是上一个执行周期还未执行结束。

## 文件事件和fd之间的关系

在redis文件事件中，涉及几个概念需要理清他们之间的关系，这样理解IO多路复用起来就比较容易。

1. 文件事件中的文件描述符fd其实就是一个socket id；
2. 一个网络接口（ip，port一对值）可以产生多个socket，也就有多个fd；
3. 一个fd可能绑定多个文件事件，比如readable事件，writeable事件；
4. redis启动时在initserver方法中会监听对应的ip-port，并产生一个socket，如果有多个ip-port需要监听则会产生多个socket，每个sockt id（fd）存放在ipfd数组；
5. 然后redis会在initserver方法中遍历ipfd，为每个fd产生一个readable文件事件（其绑定的回调函数是用于创建client），这些事件以其fd为下标存放在aeEventLoop的events数组；
6. 当有远端客户端连接redis时，会触发创建client的事件就绪，创建client的过程中，会重新产生一个新的socket id，并且以该socket id绑定一个文件事件，存放在events数组。该文件事件的作用就是接受远端客户端的后续命令。

综上所述，文件事件中的fd也就是socket id（fd），每一个文件事件对于一个fd，文件事件以fd作为下标存放在events数组。

注意后续产生的fd不会存放在ipfd数组了，只有在initserver中产生的用于创建client的fd才存放在ipfd，这些ipfd即使没有远端客户端也是已存在的，但是第6步中创建的fd只有有远端客户端时才会产生。

## 执行流程

此处的执行流程并不是redis启动时的流程，而是在已经初始化好之后的事件循环流程。

### 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处理，比如做一些同步操作

if (eventLoop->beforesleep != NULL)

eventLoop->beforesleep(eventLoop);

//循环处理事件

aeProcessEvents(eventLoop, AE\_ALL\_EVENTS);

}

}

beforeSleep函数中有个比较重要的操作就是判断是否将内核态的aof\_buf刷盘到disk。

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：

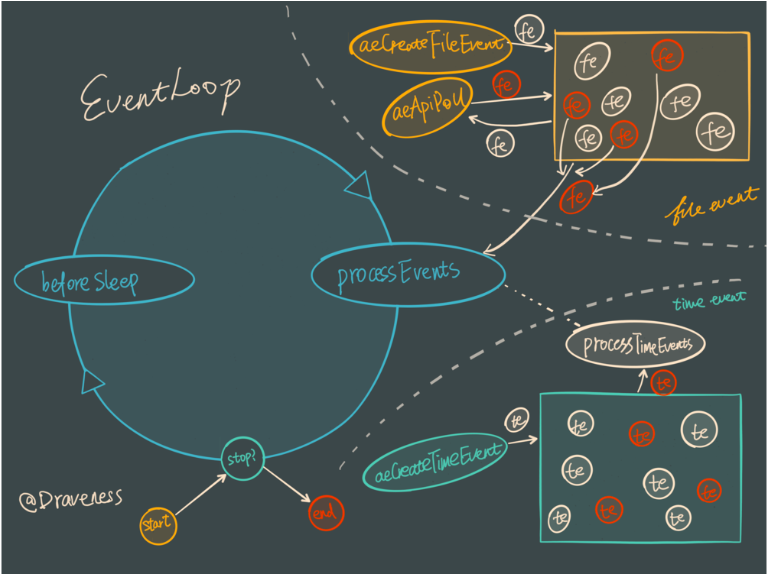


图3- 6 事件循环示意图

### 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。已就绪的fileevent会重新构造firedevent存放到fired数组。

### 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。



## 文件事件创建和删除

通过对源码的阅读理解，创建和删除文件事件主要在如下几个地方

1. 创建端口监听时，会为每一个监听的端口创建一个文件事件，该事件是Read事件，该文件事件的主要功能是为客户端创建连接（connection）时使用，文件事件的回调函数四acceptTcpHandler；
2. 创建Client（Redis中的client其实是远端客户端的代理，一个Client包含所有远端客户端的信息）时，会为其创建一个用于接收客户端请求并处理的文件事件，该事件是Read事件，文件事件的回调函数是readQueryFromClient；
3. 客户端的请求处理完成之后会为其返回对应的执行结果，此时会创建一个文件事件用于回复客户端，该事件是Write事件，该文件事件的回调函数是sendReplyToClient；
4. 回调函数sendReplyToClient成功执行（将执行结果成功从用户态写到内核态缓存）之后，会将对应的Write事件从事件数组中删除；
5. 当远端客户端断开连接（断开连接其实也是一次请求，调用readQueryFromClient）时，会解析出是断开连接，执行freeClient函数，该函数会将该Client对应的Read事件删除；

以上，2和5是一对读写事件，3和4是一对读写事件。

创建文件事件需要在io多路复用中注册这个文件描述符，比如select方式的io多路复用，会调用FD\_SET宏将对应fd的bit位设置为1，表示已经注册了该fd绑定的事件。

# 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。其中初始化阶段最重要的是完成默认配置的加载、可选配置加载以及内存空间的分配。

* 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。

此处的默认配置是按照redis作为数据库服务器角色来加载的，如果redis启动来作为sentinel，这一步加载完成之后会修改默认端口为26379，同时会修改命令集，因为redis作为数据库服务器和sentinel所能接受的命令完全不同的，因此需要修改命令集合。

### 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。其主要目的是创建一个sentinelState的对象sentinel，该对象包含了该服务器作为sentinel所需要的所有信息。

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

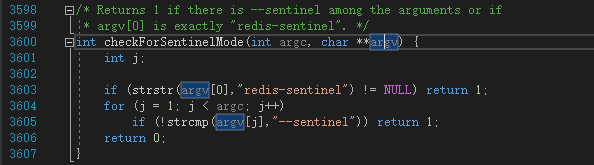


图3- 8 sentinel模式判断逻辑

### 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。

此处的配置会因为redis作为不同服务角色而不同，如果redis作为sentinel，其配置文件中每一行第一个命令都是sentinel打头，这样在解析命令的时候才知道是具体如何解析，对应的源码可以参考congfig.c/loadServerConfigFromString。

在sentinel模式下，redis加载配置项时会调用sentinel.c/sentinelHandleConfiguration方法，此方法会根据配置文件中的配置项执行对应的操作，其中最主要的操作是为配置文件中每一个被监听的服务器创建一个sentinelRedisInstance实例（因为配置文件中只需要配置master节点的信息，因此此处加载配置并生成sentinelRedisInstance实例都是对应master的实例），创建好的sentinelRedisInstance会根据不同的节点存放在一个不同dict结构中，dict的key是master名字，如果是slave或者sentinel节点，key是ip和端口拼接的字符串。

* 如果监听的是master，则存放在sentinel的masters字典
* 如果监听的是slave，则存放在其master的slaves字典
* 如果监听的是sentinel，则存放在master的sentinels字典；

这一步在sentinel.c/createSentinelRedisInstance方法中完成。

### 分配内存空间

这一步无论redis是作为数据库服务器还是sentinel服务器都需要执行这一步。

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，此时已经创建好了redis能容纳的所有aefileevent，每个aefileevent在events数组中的下标就是其fd；aeTimeEvent此时还是一个空数组，没有任何值。
* 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，将对于的sockid，也就是文件描述符存放在ipfd数组；
* 创建处理系统后台操作的周期aeTimeEvent；

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. \*/

//为server.ipfd数组中所有fd绑定事件处理函数acceptTcpHandler

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

这一步只有Redis作为数据库服务器才会进行持久化数据的加载，作为sentinel无需加载。

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。

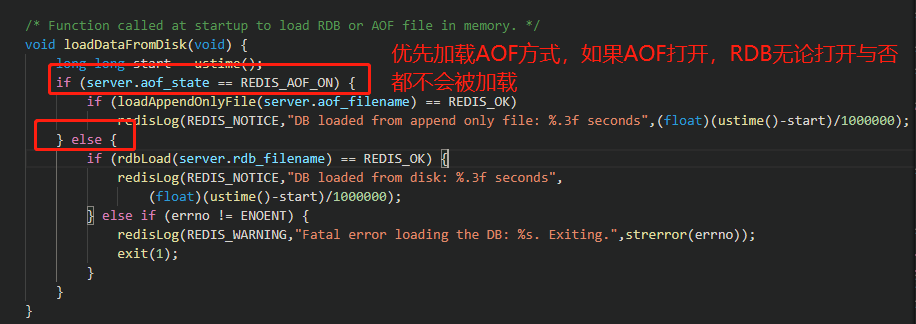


图4- 1 aof和rdb优先级

#### 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：

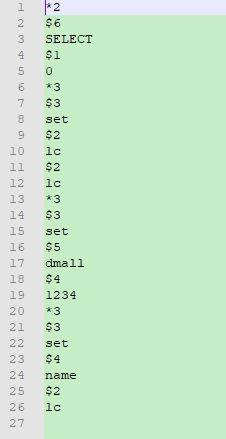


图4- 2 aof文件样本

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：

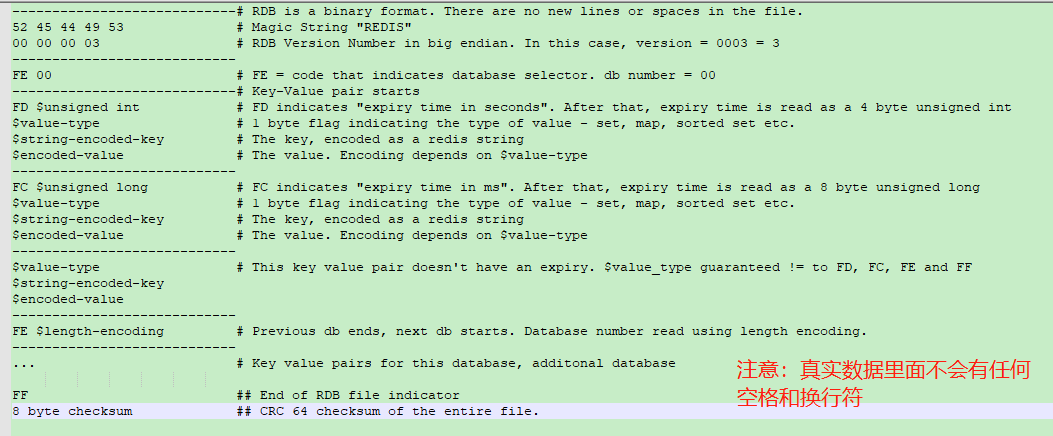


图3- 11 RDB文件格式

注：

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

### sentinel启动

启动的代码在sentinel.c/sentinelIsRunning。

这一步只有在redis作为sentinel时才会执行，其目的是为配置文件中每一个监听的master创建sentinelRedisInstance对象。

## 服务器清理

服务器清理工作很简单，就是删除aeEventLoop即可。

## 重要函数

### serverCron

该函数是个周期函数，执行频率为server.hz，其中hz可以通过配置参数hz进行配置，默认hz=10，也就是一秒钟执行10次，周期为100ms。该函数的执行周期是100ms，但是函数里面的代码块并不一定每次都会被执行，因为redis代码中定义了一个macro---run\_with\_period，这个宏定义如下，其中cronloops表示serverCron累积被执行的次数。条件判断的解释也就是当hz<=10时被宏修饰的代码块会被执行，或者如果hz>10,则100整除(1000/hz)的频率执行代码块。

#define run\_with\_period(\_ms\_) if ((\_ms\_ <= 1000/server.hz) || !(server.cronloops%((\_ms\_)/(1000/server.hz))))

比如hz=5，则100 < 1000/5,满足if调价，则被该宏修饰的代码块每次都会被执行；

如果hz=20，\_ms\_=100，则100>1000/20,if条件只有在cronloops%2==0的情况下才会为true，也就是被该宏修饰的代码块会以2\*100ms的周期执行。如果hz=30，则被该宏修饰的代码块会以3\*100ms的周期执行。通过以上分析，如果hz>10，被该宏修饰的代码块会hz/10\*100ms的周期执行。

该函数的主要功能包括如下，总结一下这些功能是为了提高redis性能而做的，并不是为了完成业务上的功能而做。

* 回收过期key；

### acceptTcpHandler

# 客户端

Redis中不管是Read命令还是Write命令，只要是要获取key（调用db.c中的lookupkeyread和lookupwrite），都需要判断这个key是已过期，如果过期需要将这个key删除，在执行读read和写write操作

## set命令执行过程

本节以redis 的set命令为例子，简单的梳理一下一个命令的大致执行过程。

1. 远程客户端（包括直接客户端或者程序客户端）输入命令：set lc lc；
2. redis服务器事件循环主体检测到有可读（Readable）事件已就绪，然后调用该事件对应的回调函数networking.c/readQueryFromClient；
3. readQueryFromClient会将server的current\_client指向当前这个事件已就绪的client，同时调用read函数将内核态数据复制到client的querybuf，调用processInputBuffer进行数据处理，最后将current\_client设置为null
4. processInputBuffer将输入的数据解析出来，并将解析出来的命令存放在client的argv里面；redis这里有两种解析方式，一种是processInlineBuffer，这种一次只解析一条命令，一种是processMultiBulkBuffer，这种会将多条命令都解析出来并逐条执行；命令的执行函数是redis.c/processCommand;
5. processCommand函数首先从命令执行表中根据输入的命令查找对应的执行函数（set命令对应的执行函数是setCommand），接下来需要做一系列的校验，只有校验通过之后才会调用该命令对应的command函数，调用命令函数封装在call函数中执行；
6. 命令执行函数主要是讲key，value存放到对应的db中，如果有过期时间expire，同时需要设置expire

void readQueryFromClient(aeEventLoop \*el, int fd, void \*privdata, int mask) {

redisClient \*c = (redisClient\*) privdata;

int nread, readlen;

size\_t qblen;

REDIS\_NOTUSED(el);

REDIS\_NOTUSED(mask);

//将当current\_client指向事件已就绪的client

server.current\_client = c;

readlen = REDIS\_IOBUF\_LEN;

/\* If this is a multi bulk request, and we are processing a bulk reply

\* that is large enough, try to maximize the probability that the query

\* buffer contains exactly the SDS string representing the object, even

\* at the risk of requiring more read(2) calls. This way the function

\* processMultiBulkBuffer() can avoid copying buffers to create the

\* Redis Object representing the argument. \*/

if (c->reqtype == REDIS\_REQ\_MULTIBULK && c->multibulklen && c->bulklen != -1

&& c->bulklen >= REDIS\_MBULK\_BIG\_ARG)

{

int remaining = (unsigned)(c->bulklen+2)-sdslen(c->querybuf);

if (remaining < readlen) readlen = remaining;

}

qblen = sdslen(c->querybuf);

if (c->querybuf\_peak < qblen) c->querybuf\_peak = qblen;

c->querybuf = sdsMakeRoomFor(c->querybuf, readlen);

//从内核态读取数据，将数据存放在c->querybuf

nread = read(fd, c->querybuf+qblen, readlen);

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) {

sdsIncrLen(c->querybuf,nread);

c->lastinteraction = server.unixtime;

if (c->flags & REDIS\_MASTER) c->reploff += nread;

server.stat\_net\_input\_bytes += nread;

} else {

server.current\_client = NULL;

return;

}

if (sdslen(c->querybuf) > server.client\_max\_querybuf\_len) {

sds ci = catClientInfoString(sdsempty(),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;

}

//处理输入数据

processInputBuffer(c);

server.current\_client = NULL;

}

## get命令执行过程

## setex命令执行过程

命令执行前的操作和set命令的执行过程类似，只不过command函数中多了设置key的过期时间。

如果一个key具有过期时间，因此在db的dict表和expires表中会各有一份key，dict中的数据是key，value键值对，expires中是key，expire键值对。

void setGenericCommand(redisClient \*c, int flags, robj \*key, robj \*val, robj \*expire, int unit, robj \*ok\_reply, robj \*abort\_reply) {

long long milliseconds = 0; /\* initialized to avoid any harmness warning \*/

if (expire) {

if (getLongLongFromObjectOrReply(c, expire, &milliseconds, NULL) != REDIS\_OK)

return;

if (milliseconds <= 0) {

addReplyErrorFormat(c,"invalid expire time in %s",c->cmd->name);

return;

}

if (unit == UNIT\_SECONDS) milliseconds \*= 1000;

}

if ((flags & REDIS\_SET\_NX && lookupKeyWrite(c->db,key) != NULL) ||

(flags & REDIS\_SET\_XX && lookupKeyWrite(c->db,key) == NULL))

{

addReply(c, abort\_reply ? abort\_reply : shared.nullbulk);

return;

}

setKey(c->db,key,val);

server.dirty++;

//如果有过期时间则在expires字典中创建一个key和到期时间的键值对

if (expire) setExpire(c->db,key,mstime()+milliseconds);

notifyKeyspaceEvent(REDIS\_NOTIFY\_STRING,"set",key,c->db->id);

if (expire) notifyKeyspaceEvent(REDIS\_NOTIFY\_GENERIC,

"expire",key,c->db->id);

addReply(c, ok\_reply ? ok\_reply : shared.ok);

}

# 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。

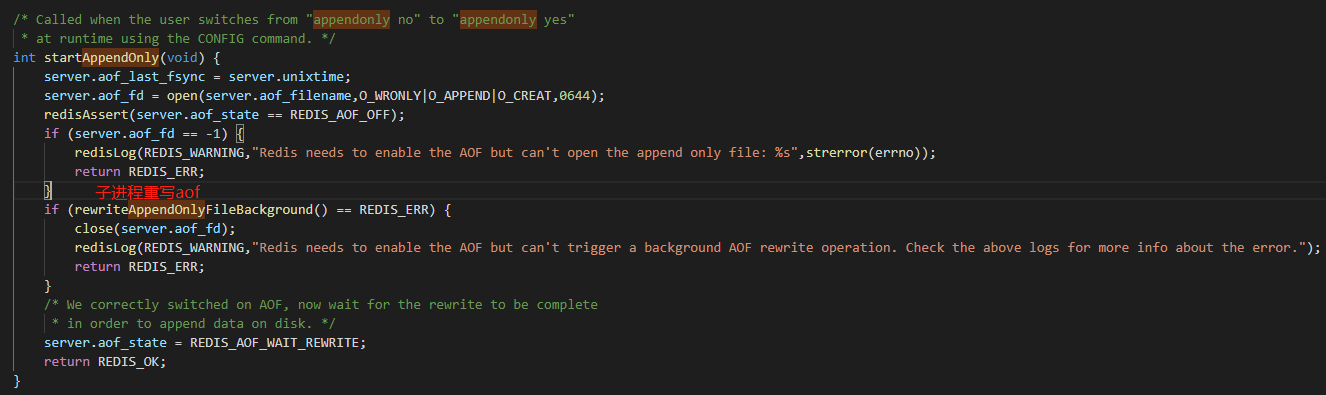


图5- 1 config命令打开aof

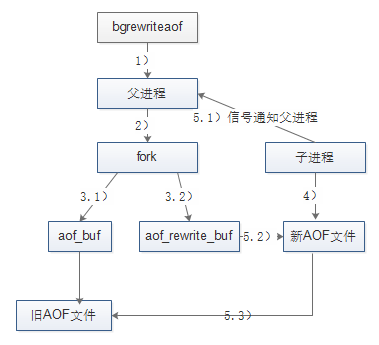


图5- 2 bgrewriteaof函数流程图

### 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。

### write类型命令写入aof过程

1. 当write类型命令执行完对应的command时（命令函数是通redis.c中的call方法触发的）；返回到call方法调用处；
2. 接着call方法后续执行，会调用propagate方法
3. propagate方法调用feedAppendOnlyFile方法
4. feedAppendOnlyFile方法调用catAppendOnlyGenericCommand方法将命令和参数转换为redis的aof文件协议格式数据，其格式详见4.2.6.1；
5. 将catAppendOnlyGenericCommand返回的格式化数据写入aof\_buf;
6. 到此命令已经写入aof\_buf，真正将aof文件刷新到disk是在下一轮事件循环之前的beforeSleep函数（redis.c中）中调用flushAppendOnlyFile。
7. flushAppendOnlyFile方法会将aof\_buf中的数据写入到内核缓冲区，至于什么时候刷盘到disk，需要看配置是no，always还是everysec

通过源码可以知道，aof刷盘频率no，always和everysec表示的是将内核态的数据刷盘到disk。Redis每一次执行完一次write类型命令都会将aof\_buf中的数据写入到内核态，但是并不一定每一次都要刷盘到disk。

### Aof重新触发条件

触发Aof rewrite的条件主要是手动输入rewrite命令，或者serverCron定时函数中判断是否满足触发条件，满足则创建子线程执行rewrite。这里同时会用上aof\_rewrite\_buff记录子线程创建时刻之后接收到的write命令

## RDB

RDB触发条件一个是手动输入save或者bgsave，另外一个就是在redis.c文件的serverCron函数中周期判断是否满足触发条件，满足之后会调用rdbSaveBackgroud来创建子线程执行持久化工作。

saveCommand主要是逐个遍历db，每个db里面遍历dict表并将每一个key,value按照对应的格式写入文件；

bgSaveCommand中父子进程一开始是共享内存空间的，如果父进程接收到写命令，则将共享内存复制一份给子进程，父进程在原空间做修改，子进程在copy的空间进行操作，这种技术依赖于操作系统的copy-on-write。

参考：<http://littledriver.net/posts/redis-persistence-1/>

# 主从复制

## 启动方式

启动主从复制功能有两种方式，一种是在从服务器的配置文件中设置slaveof参数，一种是先以一般服务器启动从服务器，然后输入slaveof命令。

## 配置启动

通过配置启动复制功能，在加载配置文件的时候为masterhost和masterport初始化值就结束， 真正触发数据复制是定期事件绑定的回调函数serverCron中调用的replicationCron方法。

## 命令启动

如果在从服务器上输入slaveof命令，将调用replication.c/slaveofCommand函数。了解该函数的主要功能前需要熟悉slaveof命令的用法。slaveof命令用法示例：

* slaveof no one，关闭从服务器复制功能，变为主服务器。
* slaveof host port，设置从服务器的主服务器

slaveofCommand源码和注释如下：

void slaveofCommand(redisClient \*c) {

    /\* SLAVEOF is not allowed in cluster mode as replication is automatically

     \* configured using the current address of the master node. \*/

//集群模式中的节点不能执行slaveof命令

    if (server.cluster\_enabled) {

        addReplyError(c,"SLAVEOF not allowed in cluster mode.");

        return;

    }

    /\* The special host/port combination "NO" "ONE" turns the instance

     \* into a master. Otherwise the new master address is set. \*/

//slave no one命令，将本服务器提升为主服务器

    if (!strcasecmp(c->argv[1]->ptr,"no") &&

        !strcasecmp(c->argv[2]->ptr,"one")) {

        if (server.masterhost) {

//这个方法主要是将server的masterhost置为null，同时断开和master的连接

            replicationUnsetMaster();

            redisLog(REDIS\_NOTICE,"MASTER MODE enabled (user request)");

        }

} else {

//slaveof host port命令

        long port;

        if ((getLongFromObjectOrReply(c, c->argv[2], &port, NULL) != REDIS\_OK))

            return;

        /\* Check if we are already attached to the specified slave \*/

//如果传入的host和port与该从服务器已经复制的主服务器一致，则直接返回

        if (server.masterhost && !strcasecmp(server.masterhost,c->argv[1]->ptr)

            && server.masterport == port) {

            redisLog(REDIS\_NOTICE,"SLAVE OF would result into synchronization with the master we are already connected with. No operation performed.");

            addReplySds(c,sdsnew("+OK Already connected to specified master\r\n"));

            return;

        }

        /\* There was no previous master or the user specified a different one,

         \* we can continue. \*/

//这个方法很重要，除了设置masterhost，masterport之外，如果该从服务器目前已经复制了其他master，则需要断开和master的连接，清空数据。如果该从服务器还有slaves，则断开所有的slaves，迫使它的从服务器重新同步数据

        replicationSetMaster(c->argv[1]->ptr, port);

        redisLog(REDIS\_NOTICE,"SLAVE OF %s:%d enabled (user request)",

            server.masterhost, server.masterport);

    }

    addReply(c,shared.ok);

}

以上方法总结如下：

1. 判断执行slaveof命令的服务器是否是集群中的一个节点，如果是返回；因为集群节点不能执行slaveof命令
2. 非集群节点执行slaveof命令，首先判断是否是slaveof no one，如果是置空masterhost以及断开和master的连接，置空偏移量offset，置空复制缓存；
3. 如果是slave host port命令，首先判断host，port和已有master是否完全一样，如果完全一样则直接返回，否则需要设置新的masterhost和masterport，是否原有的master，断开原有与master的连接，置空偏移量offset如果该从服务器有slaves，需要断开所有与slaves的连接，迫使slaves重新复制数据（因为offset已经被重置，则slaves节点不会执行psysc，会执行完全复制sync）

slaveof命令执行结束之后，并没有触发真正的复制数据操作，只是把复制数据所需要的信息保存了下来，真正触发数据复制是在定期事件触发的serverCron方法中，调用replicationCron方法。

## 状态机

注意：在slave和master之间任意动作都是以slave主动发起。当在服务器输入slaveof命令并连接master的过程中，此时只有slave本身会被感知为slave，而master处的感知仅仅是多了一个客户端而已。slave和master之间进行复制之前的操作类似于TCP的三次握手。

参与主从复制的服务器都对应着不同的状态，其中slave包括编号0~5的6个状态，master包括6~9的4个状态。接下来需要理清服务器在什么时候处于的不同状态（状态机转换）。

/\* Slave replication state - from the point of view of the slave. \*/

#define REDIS\_REPL\_NONE 0 /\* No active replication \*/

#define REDIS\_REPL\_CONNECT 1 /\* Must connect to master \*/

#define REDIS\_REPL\_CONNECTING 2 /\* Connecting to master \*/

#define REDIS\_REPL\_RECEIVE\_PONG 3 /\* Wait for PING reply \*/

#define REDIS\_REPL\_TRANSFER 4 /\* Receiving .rdb from master \*/

#define REDIS\_REPL\_CONNECTED 5 /\* Connected to master \*/

/\* Slave replication state - from the point of view of the master.

 \* In SEND\_BULK and ONLINE state the slave receives new updates

 \* in its output queue. In the WAIT\_BGSAVE state instead the server is waiting

 \* to start the next background saving in order to send updates to it. \*/

#define REDIS\_REPL\_WAIT\_BGSAVE\_START 6 /\* We need to produce a new RDB file. \*/

#define REDIS\_REPL\_WAIT\_BGSAVE\_END 7 /\* Waiting RDB file creation to finish. \*/

#define REDIS\_REPL\_SEND\_BULK 8 /\* Sending RDB file to slave. \*/

#define REDIS\_REPL\_ONLINE 9 /\* RDB file transmitted, sending just updates. \*/

* REDIS\_REPL\_NONE：服务器在启动时候初始化server参数的默认值时，会将repl\_state设置为该状态，或者从服务器在接收到slaveof no one命令后会置为该状态。可以理解为主从服务器的初始状态；
* REDIS\_REPL\_CONNECT:服务器启动时加载配置文件的slaveof时，会将服务器置为该状态，或者接收到slaveof host port命令后会置为该状态；
* REDIS\_REPL\_CONNECTING:该状态是replication.c/replicationCron函数中slave去连接master时发生的状态，当slave连接上master，同时创建一个fileEvent（socket，事件类型是可读可写，回调函数是replication.c/syncWithMaster）后，会进入该状态；
* REDIS\_REPL\_RECEIVE\_PONG：slave和master连接上之后并不会马上做复制操作，需要进行网络的可达性检验，通过ping-pong协议来完成。该状态是slave给master发送ping消息（必须满足一定的发送频率才会发送）之后进入该状态（master会接收到一个ping命令，执行pingCommand）。
* REDIS\_REPL\_TRANSFER：

## 状态机转换示例

本示例主要是以一个实际场景来演示master和slave之间复制时的状态转换。

服务器A，B分别各自启动，无任何master和slave，接下来在服务器B上slaveof host port，使B成为A的从服务器。

1. 服务器B接收到slaveof命令后，通过replication.c/slaveofCommand函数设置其masterhost，masterport（如果服务器B已经归属于非服务器A的slave，需要先终止之前的master连接）。到此该函数就执行完成，给发送slaveof命令的客户端返回ok。服务器B进入REDIS\_REPL\_CONNECT状态。
2. 定时事件触发执行serverCron函数（1秒执行一次），继而执行replicationCron函数，处于REDIS\_REPL\_CONNECT状态的服务器B，需要执行连接master（服务器A）的操作，这一步就是A和B之间建立了socket通信信道，同时B服务器会创建一个可读可写的文件事件，回调函数是syncWithMaster，对于服务器A来说，B服务器此时是一个一般的客户端，并且A的slaves链表还是空的。建立连接之后的B服务器处于REDIS\_REPL\_CONNECTING。
3. 服务器A和B连接成功后，会触发服务器B注册的回调函数syncWithMaster（具备可写的文件事件，如果一旦被允许就会调用其对应的回调函数执行对应的写操作，在注册写事件时就会将写的内容传递到位），由于服务器此时处于REDIS\_REPL\_CONNECTING状态，则需要给Master发送ping命令。同时会将之前创建的文件事件置为只可读事件。B服务器将进入REDIS\_REPL\_RECEIVE\_PONG状态。
4. A服务器收到ping命令之后将执行pingCommand函数，该函数主要功能就是给slave回复一个pong。
5. A服务器发送的pong会触发syncWithMaster函数，此时B服务器处于REDIS\_REPL\_RECEIVE\_PONG状态，会执行对应的代码，代码主要功能是判断是否是PONG字符串，同时要删除A和B之间socket对应的可读事件（到此该socket对应的可读可写事件都已被删除）。接收到pong回复之后，B服务器会给A服务器回复一个replconf命令（通过A,B之间的socket，不是文件事件），命令带着B服务器的port或者其他信息，并且发送这个命令是阻塞操作（当然有一个最大阻塞时间，超时将被视为异常），B服务器会等待replconf命令的回复值之后才会继续往下执行。
6. A服务器接收到replconf命令后会将replconf命令所带的key-value对保存，上一步命令所带的参数时slave的端口，会被存放在client的slave\_listening\_port；命令执行结束后返回ok；
7. 第五步说了B会等待replconf命令的回复才会继续执行。B服务器接下来就要执行真正的同步操作。构造psync命令并发送，注意此处发生psync命令会无限阻塞，没有设置最长阻塞时间。有关psync命令详见后续章节。psync命令执行完成后，会重新在A，B之间的socket上创建一个可读文件事件，回调函数是replication.c/readSyncBulkPayload（用于读取master传过来的rdb文件或者写命令），同时B服务器进入REDIS\_REPL\_TRANSFER状态。
8. A服务器生成好rdb文件后直接发送给B，将触发B服务器的文件事件调用readSyncBulkPayload，这个方法就是解析rdb文件并载入B服务器的数据库，最后B服务器进入REDIS\_REPL\_CONNECTED状态。

## PSYNC命令

PSYNC命令只有在Redis 2.8以上版本才支持。如果低于这个版本，使用的是sync命令，这个命令就是完成重同步。并且发送psync命令的slave采用的是阻塞是发送，也就是发送了psync命令之后会等待其结果之后才会继续执行。

PSYNC命令主要用于第一次主从复制，或者断线重连之后的重复制。

slave服务器发送PSYNC命令的代码位于replication.c/slaveTryParticalResynchronization方法中。PSYNC命令分为两种情况，一种是完整重同步，一种是部分重同步。如果是完整重同步，PSYNC命令格式为psync ? -1，如果是部分重同步，PSYNC命令格式为psync masterid offset。

* 完整重同步：第一次从主服务器复制数据、或者master id被修改，或者offset已无法定位到数据，则需要主从服务器之间进行完整重同步。也就是将主服务器的数据库生成rdb文件发生给从服务器执行；
* 部分重同步：slave直接从master的缓存中获取未同步的写命令并执行。

Master服务器接收到psync命令之后执行syncCommand函数（psync和sync命令执行同一个函数）。如果参数是? -1,则表示需要完整重同步，master会执行bgsave，通过子线程来生成rdb文件。子线程是否完成rdb文件的生成，是在master的serverCron进行轮询的（需要检测子线程的返回码），代码如下，backgroundsavedonehandler的工作是为master的每一个slave返回rdb文件。

/\* Check if a background saving or AOF rewrite in progress terminated. \*/

    if (server.rdb\_child\_pid != -1 || server.aof\_child\_pid != -1) {

        int statloc;

        pid\_t pid;

        if ((pid = wait3(&statloc,WNOHANG,NULL)) != 0) {

            int exitcode = WEXITSTATUS(statloc);

            int bysignal = 0;

            if (WIFSIGNALED(statloc)) bysignal = WTERMSIG(statloc);

//执行bgsave完成工作之后的操作

            if (pid == server.rdb\_child\_pid) {

                backgroundSaveDoneHandler(exitcode,bysignal);

            } else if (pid == server.aof\_child\_pid) {

//aof rewrite完成之后的操作

                backgroundRewriteDoneHandler(exitcode,bysignal);

            } else {

                redisLog(REDIS\_WARNING,

                    "Warning, detected child with unmatched pid: %ld",

                    (long)pid);

            }

            updateDictResizePolicy();

        }

} else {

。。。。。。

}

如果master监测到psync只需要执行部分重同步，则会为该slave发送offset之后的所有数据，同时将该client加入到slaves链表。

psync命令的返回值有三种：

* psync\_fullresync:如果返回这个值，表明需要完成重同步，slave会创建一个可读文件事件等待master的rdb文件；此处需要注意的是master解析出需要fullresync后，给slave发送fullresync，同时返回的是执行出错，使得master不能直接返回，会接着执行startBgsaveForReplication。
* psync\_continue;如果返回这个值，slave会创建一个可读文件事件，其回调函数是readQueryFromClient，这个函数就是读取一般命令的回调函数。Master给slave返回continue之后，会将复制缓冲器中滞后的信息（全是写命令）一起打包发给slave，slave就是通过readQueryFromClient执行这打包消息的。
* psync\_not\_supported:

## 命令传播

所谓命令传播，就是完成主从复制第一阶段的同步之后，主从服务器已经达到一致状态，但是当客户端修改主服务器的数据库时，为了使从服务器和主服务器保持一致，需要将主服务器执行的写命令发送给从服务器，这个阶段就是命令传播。

命令传播在redis代码中是通过如下方式，每一次执行的命令都会调用call方法，call方法中会调用propagate函数，propagate会调用相应的函数，如果开启了aof，会将写命令写入aof buff，同时会为每一个slave传播该命令，调用replicationfeedslaves。

## 心跳监测

心跳检测除了检测网络连接情况之外，还有个主要的作用就是判断master和slave的offset是否一致，如果不一致，master会给slave发送缺少的数据。

发送的命令是replconf ack offset

# sentinel哨兵

参考：<http://redisbook.com/preview/sentinel/init_sentinel.html>

## 启动方式

将redis父亲启动为sentinel的命令如下：

$ redis-sentinel /path/to/your/sentinel.conf

或者

$ redis-server /path/to/your/sentinel.conf --sentinel

如果服务器以sentinel的模式启动，会创建一个全局唯一的sentinel对象保存有关哨兵的信息（服务器的一般信息仍然用server对象保存），这个sentinel对象中有一个masters的字典表，其中每一个master有一个slaves字典表，同时每个master有个sentinels的字典表。

## sentinel配置

通过启动命令可以知道，如果redis服务器要作为sentinel哨兵，需要向服务器传递所需要的配置信息。典型的配置信息如下所示（每一行的注释是我本文自己加的，不是配置文件中的信息）：

sentinel monitor mymaster 127.0.0.1 6379 2

#监听127.0.0.1:6379的master，取名mymaster，最后面的2表示sentinel集群中如果有2个sentinel进程认为该master失效了，则才会重新选举master。因为一个sentinel可能会因为网络问题误认为该master无效。

sentinel down-after-milliseconds mymaster 60000

#down-after-milliseconds用于判断sentinel给mymaster发送周期心跳报文最长的等待时间，单位毫秒。如果超过这个时间未收到心跳反馈信息或者期间返回错误信息，则主观认为该master已失效，需要给其他sentinel发送信息并收集其他sentinel的信息。如果达到2个sentinel主观认为该master已失效，则sentinel集群中所有sentinel都会客观认为该master失效，需要选举新的master

sentinel failover-timeout mymaster 180000

sentinel parallel-syncs mymaster 1

sentinel monitor resque 192.168.1.3 6380 4

sentinel down-after-milliseconds resque 10000

sentinel failover-timeout resque 180000

sentinel parallel-syncs resque 5

sentinel配置只需要配置master节点信息，因为slave节点信息可以从master中获取。同时需要注意的是sentinel的配置文件是会动态修改的，如果某个master宕机，其某个slave被选为master后，配置文件会将之前的master配置删除，加入新被选中的slave信息写入配置文件。

## 配置加载

通过两种方式启动sentinel都需要传入配置文件，配置文件的加载和一般服务器加载配置文件都是一样的，只是sentinel的每一个配置项前面都有个sentinel。加载配置就是为server配置参数的过程，其中加载monitor时，会为每一个被监测的master创建一个sentinelRedisInstance。直观的理解就是这个每一个被监测的对象（包括master、slave已经sentinel）都对应一个sentinelRedisInstance。

## 执行流程

sentinel的配置加载完成之后，已经建立好了所有需要监听的master对应的sentinelRedisInstance，存放在masters字典中。接下来就等着执行周期函数serverCron中的sentinelTimer方法，这个方法就是sentinel的主要操作。sentinelTimer方法代码如下

void sentinelTimer(void) {

sentinelCheckTiltCondition();

//遍历所有的master

    sentinelHandleDictOfRedisInstances(sentinel.masters);

    sentinelRunPendingScripts();

    sentinelCollectTerminatedScripts();

    sentinelKillTimedoutScripts();

    /\* We continuously change the frequency of the Redis "timer interrupt"

     \* in order to desynchronize every Sentinel from every other.

     \* This non-determinism avoids that Sentinels started at the same time

     \* exactly continue to stay synchronized asking to be voted at the

     \* same time again and again (resulting in nobody likely winning the

     \* election because of split brain voting). \*/

    server.hz = REDIS\_DEFAULT\_HZ + rand() % REDIS\_DEFAULT\_HZ;

}

方法主体是调用sentinelHandleDictOfRedisInstance，其参数就是所有的master实例，该方法就是遍历每个master及其所有slaves和sentinel执行周期性操作，并且该方法会修改serverCron的执行频率，随机生成一个10~19之间的数。

void sentinelHandleDictOfRedisInstances(dict \*instances) {

    dictIterator \*di;

    dictEntry \*de;

    sentinelRedisInstance \*switch\_to\_promoted = NULL;

    /\* There are a number of things we need to perform against every master. \*/

    di = dictGetIterator(instances);

    while((de = dictNext(di)) != NULL) {

        sentinelRedisInstance \*ri = dictGetVal(de);

//每个实例的周期性操作

        sentinelHandleRedisInstance(ri);

        if (ri->flags & SRI\_MASTER) {

//如果是master，递归遍历所有slaves和sentinel

            sentinelHandleDictOfRedisInstances(ri->slaves);

            sentinelHandleDictOfRedisInstances(ri->sentinels);

            if (ri->failover\_state == SENTINEL\_FAILOVER\_STATE\_UPDATE\_CONFIG) {

                switch\_to\_promoted = ri;

            }

        }

    }

    if (switch\_to\_promoted)

        sentinelFailoverSwitchToPromotedSlave(switch\_to\_promoted);

    dictReleaseIterator(di);

}

周期性操作主要包括：

* 连接实体sentinelRedisInstance，包括命令连接和发布订阅连接，命令连接用于发送（info，ping），发布订阅channel是（\_\_sentinel\_\_:hello）
* 根据条件发送info，ping或者发布信息到channel，每次只能执行一个命令的执行；
* 检查是否主观下线subject-down，主观下线就是该sentinel和监听的instance失联则被认定为主观下线；
* 检查是否客观下线object-down：客观下线就是当满足认定实例失败的sentinel个数大于等于quorum，则认为该instance客观下线

void sentinelHandleRedisInstance(sentinelRedisInstance \*ri) {

    /\* ========== MONITORING HALF ============ \*/

/\* Every kind of instance \*/

//检查连接

sentinelReconnectInstance(ri);

//根据时间条件发送命令（info，ping，pub）

    sentinelSendPeriodicCommands(ri);

    /\* ============== ACTING HALF ============= \*/

    /\* We don't proceed with the acting half if we are in TILT mode.

     \* TILT happens when we find something odd with the time, like a

     \* sudden change in the clock. \*/

    if (sentinel.tilt) {

        if (mstime()-sentinel.tilt\_start\_time < SENTINEL\_TILT\_PERIOD) return;

        sentinel.tilt = 0;

        sentinelEvent(REDIS\_WARNING,"-tilt",NULL,"#tilt mode exited");

    }

/\* Every kind of instance \*/

//判断该实例是否失联（主观下线）

    sentinelCheckSubjectivelyDown(ri);

    /\* Masters and slaves \*/

    if (ri->flags & (SRI\_MASTER|SRI\_SLAVE)) {

        /\* Nothing so far. \*/

    }

    /\* Only masters \*/

if (ri->flags & SRI\_MASTER) {

//判断是否客观下线

        sentinelCheckObjectivelyDown(ri);

//如果客观下线，则执行failover

        if (sentinelStartFailoverIfNeeded(ri))

            sentinelAskMasterStateToOtherSentinels(ri,SENTINEL\_ASK\_FORCED);

        sentinelFailoverStateMachine(ri);

        sentinelAskMasterStateToOtherSentinels(ri,SENTINEL\_NO\_FLAGS);

    }

}

# IO 多路复用

## 概述

理解unix中的io多路复用的前提是了解unix的五种io模型。分别是

* 阻塞io
* 非阻塞io
* io多路复用
* 信号驱动io
* 异步io

## select多路复用

了解select首先要了解select函数中参数所需要的类型fd\_set。fd\_set是一个unsigned long类型的数组，数组长度是1024/(8\*sizeof(unsigned long)),其中通过一个位来关联一个事件event，因此最多关联1024个事件。

fd\_set简单的理解就是一个bitmap，bit总位数为1024。

fd\_set中1024不仅仅是限制的文件描述符个数，也是文件描述符的最大值。因为fd\_set中是通过fd作为bitmap的下标获取值。

select函数被调用是需要传递两个fd\_set，其中一个是可读事件集合，另一个是可写事件集合。

通过源码来理解select的用法。

static int aeApiPoll(aeEventLoop \*eventLoop, struct timeval \*tvp) {

    aeApiState \*state = eventLoop->apidata;

    int retval, j, numevents = 0;

    memcpy(&state->\_rfds,&state->rfds,sizeof(fd\_set));

    memcpy(&state->\_wfds,&state->wfds,sizeof(fd\_set));

//select，第一个参数指定最大fd+1，第二个可读fd集合，第三个可写fd集合，第四个异常fd集合，第五个表示select阻塞事件，如果为null表示立即返回

    retval = select(eventLoop->maxfd+1,

                &state->\_rfds,&state->\_wfds,NULL,tvp);

if (retval > 0) {

//遍历redis中当前所有的fileevent，如果事件是readable，从\_rfds判断该事件是否已就绪，否则从\_wfds判断是否就绪，如果就绪，将事件的fd和mask构造成aeFiredEvent存放在fires数组，fires数组存放所有就绪的文件事件，mask表示文件时可读还是可写

        for (j = 0; j <= eventLoop->maxfd; j++) {

            int mask = 0;

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

            if (fe->mask == AE\_NONE) continue;

            if (fe->mask & AE\_READABLE && FD\_ISSET(j,&state->\_rfds))

                mask |= AE\_READABLE;

            if (fe->mask & AE\_WRITABLE && FD\_ISSET(j,&state->\_wfds))

                mask |= AE\_WRITABLE;

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

            eventLoop->fired[numevents].mask = mask;

            numevents++;

        }

    }

    return numevents;

}

* select函数执行之后，没有准备就绪的fd对应bit会置为0；
* 使用select的io多路复用必须再增加一个array保存原始的fd数组，这个数组用于select之后作为原数据和select返回的fd\_set做对比判断；

## poll多路复用

这种方式的多路复用Redis没有使用，但是可以加强了解。poll多路复用的原理和select类似，只是poll没有fd个数限制，因为poll监测的是一个pollfd类型的数组

struct pollfd{  
 int fd;  
 short events;  
 short revents;

}

函数头

int poll (struct pollfd \*fd, unsigned int nfds, int timeout);

其中fd指针是个数组，nfds是需要监测的个数（数组大小可能比nfds大）。poll函数返回之后需要逐个遍历pollfd对象中的revents，判断事件是否已就绪。

## epoll多路复用

epoll其底层是通过一个文件描述符管理多个文件描述符，其中涉及三个接口，分别是：

* epoll\_create:
* epoll\_ctl:
* epoll\_wait:

epoll中涉及的epoll\_event事件是linux内核能够识别的事件，在调用epoll\_wait函数之后，需要将events数组中的epoll\_event转换为上层应用能识别的事件。

### epoll\_create

函数头：int epoll\_create(int size);

size并不是限制epoll能监测的最大的描述符个数，而是建议内核初始分配内部数据结构的个数；函数执行成功之后返回一个新的文件描述符，这个描述符用于管理系统需要监听的所有事件对应的fd。通俗的理解就是该函数创建了一个容器，容器的句柄handler是函数返回值fd，通过这个fd可以直接对应到本容器。

### epoll\_ctl

函数头：int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event\* events);

本函数用于向epoll容器增加、删除或者修改文件描述符。其中epfd是epoll\_create返回的epoll容器句柄，op操作类型，包括增加、删除，修改，fd是被监测的文件描述符

### epoll\_wait

函数头：int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout);

epfd是epoll容器句柄，events数组是存放就绪的fd对应的epoll\_event，maxevents表示最多返回就绪事件个数，timeout是设置函数阻塞时间。epoll\_event结构体中存在一个描述事件类型的字段，比如读，写，错误等类型。

返回值表示已就绪的fd个数，其对应的epoll\_event事件在events数组中。