**The Annotated Redis Sources**

**by sxf (qq:360053063) v0.1620**



目 录

前序

1. 概述……………………………………………………1
2. redis 架构介绍和工作流程…………………………8

2.1 redis 整体架构和层次模型 …………………………………9

2.2 redis server组成模块和工作流程…………………………12

2.3 redis Client组成模块和工作流程…………………………15

1. redis重要功能原理简介

3.1数据引擎 ………………………………………………………16

3.1.1 Redis支持的数据类型

3.1.2 基本数据类型 (redisObject)

3.1.3 Redis的数据组织方式：

3.1.4 内部数据结构概述

3.1.5 简单动态字符串(sds)

3.1.6 双向链表(double-linked list)

3.1.7 字典(dict)

3.1.8 跳跃表（Skip List）

3.1.9 压缩列表(ziplist)

3.1.10 过期键回收

3.2 事务(transaction)

3.2.1 redis事务的特点

3.2.2 事务命令与事务队列

3.2.3 WATCH命令

3.2.4 注意事项

3.3 事件循环 ………………………………………………………54

3.4 持久化 …………………………………………………………58

3.4.1 rdb模式

3.4.2 aof模式

3.5 主从复制 ………………………………………………………71

3.5.1 文件同步

3.5.2 命令同步

3.5.3 工作流程

3.5.4 多级同步

3.5.5 主从转换

3.6 配置系统………………………………………………………87

3.6.1 配置文件

3.6.2 一些重要参数介绍

3.7 脚本系统………………………………………………………89

3.7.1 lua脚本简介

3.7.2 脚本执行流程

3.7.3 脚本安全

3.7.3 EVAL命令与EVALSHA命令

3.8 慢日志(slowlog) ……………………………………………111

3.8.1 慢日志的数据结构

3.8.2 慢日志记录过程

3.8.3 使用建议

3.9 订阅系统………………………………………………………115

3.9.1 频道结构

3.9.2 订阅链表

3.9.3 订阅模式

3.9.4 订阅流程

1. redis源码详细分析…………………………………119

4.1 redis.c

4.2 ae.c

4.3 ae\_epoll.c

4.4 ae\_select.c

4.5 ae\_kqueue.c

4.6 ae\_evport.c

4.7 t\_string.c

4.8 t\_list.c

4.9 t\_hash.c

4.10 t\_set.c

4.11 t\_zset.c

4.12 sds.c

4.13 adlist.c

4.14 dict.c

4.15 ziplist.c

4.16 intset.c

4.17 network.c

4.18 replication.c

4.19 aof.c

4.20 rdb.c

4.21 bio.c

4.22 db.c

4.23 sentinel.c

4.24 slowlog.c

1. 感悟与总结

5.1 先进经验………………………………………………………347

5.2 一些缺陷………………………………………………………349

5.1 使用忠告………………………………………………………351

5.2 未来展望………………………………………………………352

参考资料

附录A redis命令大全

附录B redis源码文件目录与说明

一些约定：

黑色为文档正文

*灰色斜体为引文(引自其他书籍和文章)*

紫色为注意点和重点

棕色为代码原文

蓝色为代码展开

绿色为代码剖析

粉色为本人对redis的一些改进思路

本分析基于redis官方最新稳定版 2.6.16

2.8增加了crc64.h notify.c

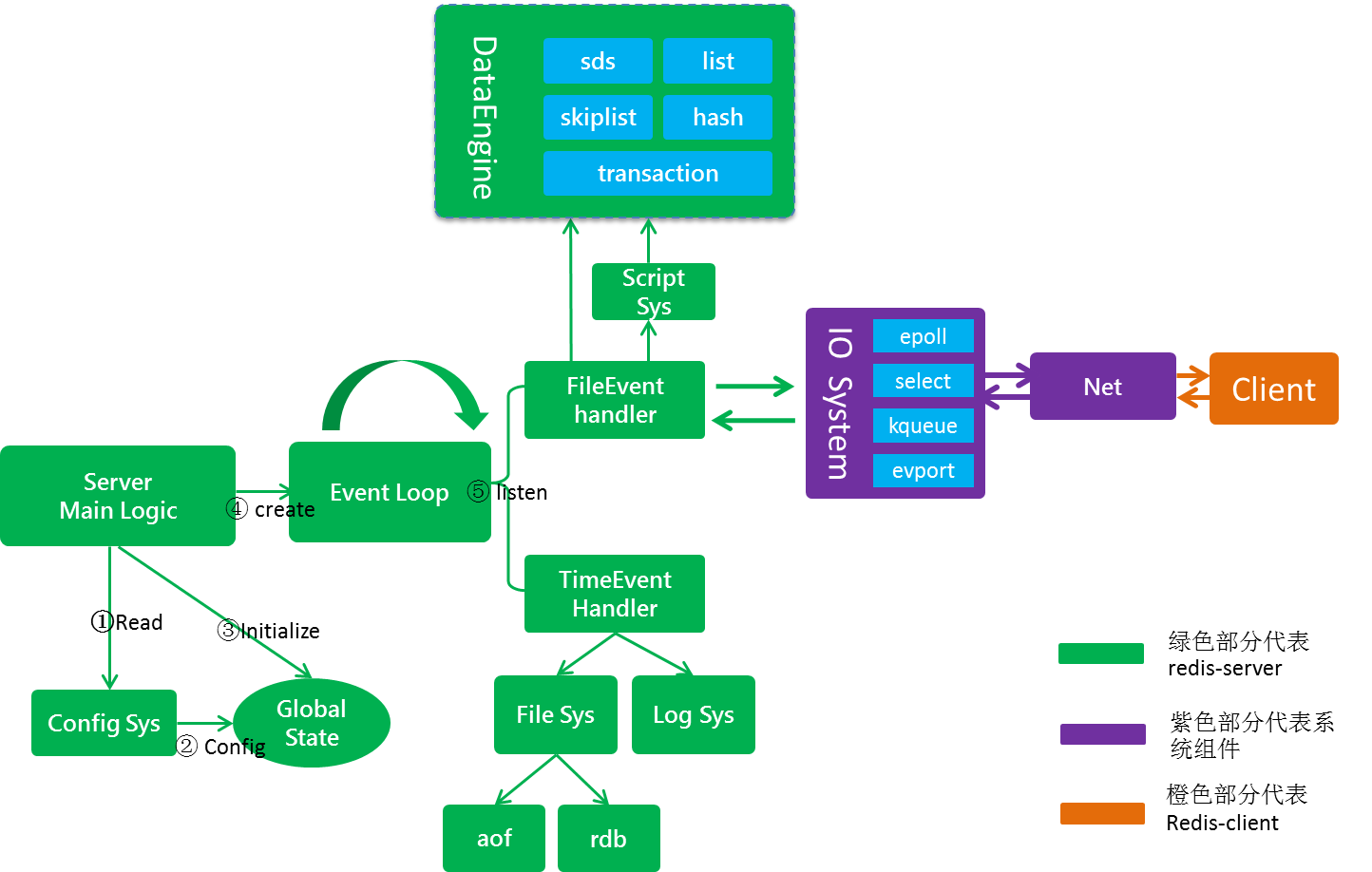
setproctitle.c

版本更新日志

|  |  |  |
| --- | --- | --- |
| 版本 | 改动 | 备注 |
| 0.1620 | 重写了真个replication的过程，并增加了“replication流程与状态变迁”图 | 有人反馈这部分写的不够清楚，看了任然搞不清楚master和slave的状态变迁细节。 |
| 0.1622 | 增加了整数结合的分析 |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

第一章redis整体架构

1.1 redisServer工作流程

****

1.2 redisClient工作流程

第二章 重要功能原理

2.1 数据引擎

2.1.1 Redis支持的数据类型

字符串

链表

哈希表

列表

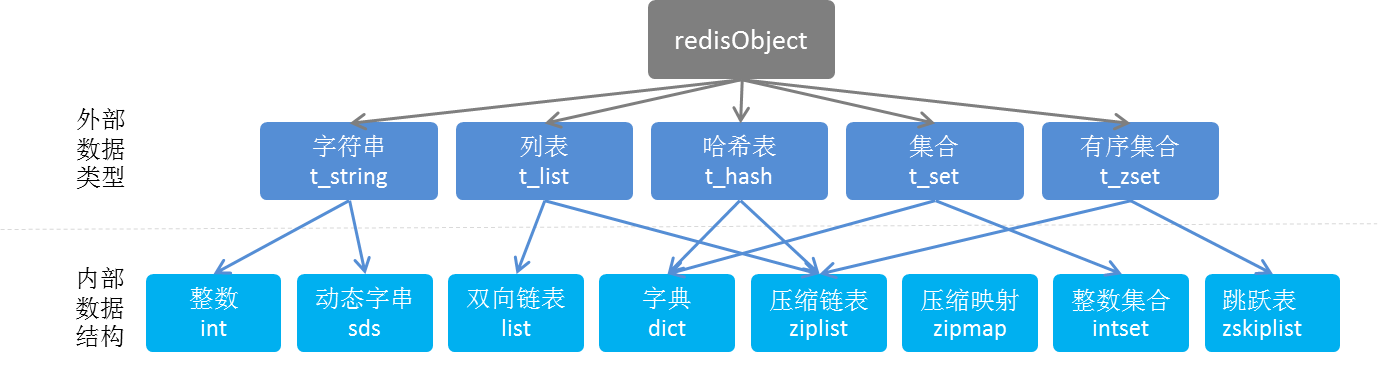
集合

有序集合

2.1.2 基本数据类型 (redisObject)

Redis支持六种数据类型，但其内部实现却是动态字符串(sds)、双端链表(list)、字典(dict)、跳跃表(skiplist)、压缩列表(ziplist)、压缩映射(zipmap)、整数集合(intset)七种数据结构(下文会详细介绍每一种数据结构)。这六种数据类型和这七种结构是怎么关连起来的？想了解这点需要了解redis的基本数据类型redisObject。

**数据类型与内部数据结构对应关系**



【注意】从 Redis 2.6 开始，没有任何数据类型是用zipmap来实现。

另外一点值得思考的是：一些命令可以用于对任意数据类型操作。例如DEL、TTL、TYPE等。但是对于不同的数据类型，这些命令的处理方式的不同的，比如删除一个列表键和删除一个字符串键的过程就不太一样。那么redis究竟如何实现用相同的命令实现不同的操作的呢？

另外，在前面介绍各个底层数据结构时有提到，Redis 的每一种数据类型，比如字符串、列表、有序集， 它们都拥有不只一种底层实现（Redis 内部称之为编码，encoding）， 这说明， 每当对某种数据类型的键进行操作时， 程序都必须根据键所采取的编码， 进行不同的操作。

Redis 的做法是让每个键都处理为一个基本类型，每个基本类型都带有数据类型属性和编码属性， 使得程序可以检查键的类型和编码， 并为它选择合适的处理方式。

typedef struct redisObject

{

unsigned type:4; // --类型

unsigned notused:2; // --未用到

unsigned encoding:4; // --编码

unsigned lru:22; // --RU 时间（相对于 server.lruclock）

int refcount; // --引用计数

void \*ptr; // --指向值的指针

} robj;

type:4;

notused:2;

encoding:4;

lru:22;

这几个值应用了C语言的位域实现，这四个属性共用一个双字(unsigned int，32bit)。

Type用了其中的4bit，lru用了其中的22bit，所以sizeof(redisObject)得出的值应该是12。

type 表示值的数据类型，可以是下面类型

#define REDIS\_STRING 0 // --字符串

#define REDIS\_LIST 1 // --链表

#define REDIS\_SET 2 // --集合

#define REDIS\_ZSET 3 // --有序集合

#define REDIS\_HASH 4 // --哈希表

encoding 表示值的编码类型，可以是下面类型

#define REDIS\_ENCODING\_RAW 0 // --字符编码

#define REDIS\_ENCODING\_INT 1 // --整数编码

#define REDIS\_ENCODING\_HT 2 // --哈希表

#define REDIS\_ENCODING\_ZIPMAP 3 // --压缩哈希

#define REDIS\_ENCODING\_LINKEDLIST 4 // --链表

#define REDIS\_ENCODING\_ZIPLIST 5 // --压缩表

#define REDIS\_ENCODING\_INTSET 6 // --整数集合

#define REDIS\_ENCODING\_SKIPLIST 7 // --跳跃表

ptr 是一个指针，指向实际保存值的数据结构，这个数据结构由 type 属性和 encoding 属性决定。举个例子，如果一个 redisObject 的 type 属性为 REDIS\_LIST ， encoding 属性为 REDIS\_ENCODING\_LINKEDLIST ，那么这个对象就是一个 Redis 列表，它的值保存在一个双端链表内，而 ptr 指针就指向这个双端链表；另一方面，如果一个 redisObject 的 type 属性为 REDIS\_HASH ， encoding 属性为 REDIS\_ENCODING\_ZIPMAP ，那么这个对象就是一个 Redis 哈希表，它的值保存在一个 zipmap 里，而 ptr 指针就指向这个 zipmap 。

【注意】redis是用纯C实现，没有用任何class，所以这个redisObject看似6种数据类型的基类，但实际上只是用一个type来标志这个基本类型所代表的究竟是哪个具体类型，encoding标志出该基本类型的内部数据结构类型，而ptr指针是把数据类型和内部实现关联起来的最关键所在。

**引用计数与对象的销毁**

Redis对基本类型的数据对象的处理方式与STL的智能指针非常相似，都采用了引用计数的方式来追踪对象的使用情况，每个 redisObject 结构都带有一个 refcount 属性，指示这个对象被引用了多少次。每当一个对象被创建这个值被初始化为1；每当robj对象被引用，Redis会给refcount的值加1；而robj对象生存期结束后，redis就会给refcount减1。当refcount为0时，redis认为没有地方再引用这个对象，redis就会释放该对象占用的内存。

**共享对象**

对于一些需要经常使用的数据对象，redis会采用一种羽量模式(Flyweight)：预先在内存中创建这些对象并在多个数据结构之间共享这些对象，避免了重复分配的麻烦， 也节约了一些 CPU 时间。我们将这些对象称为“共享对象”。

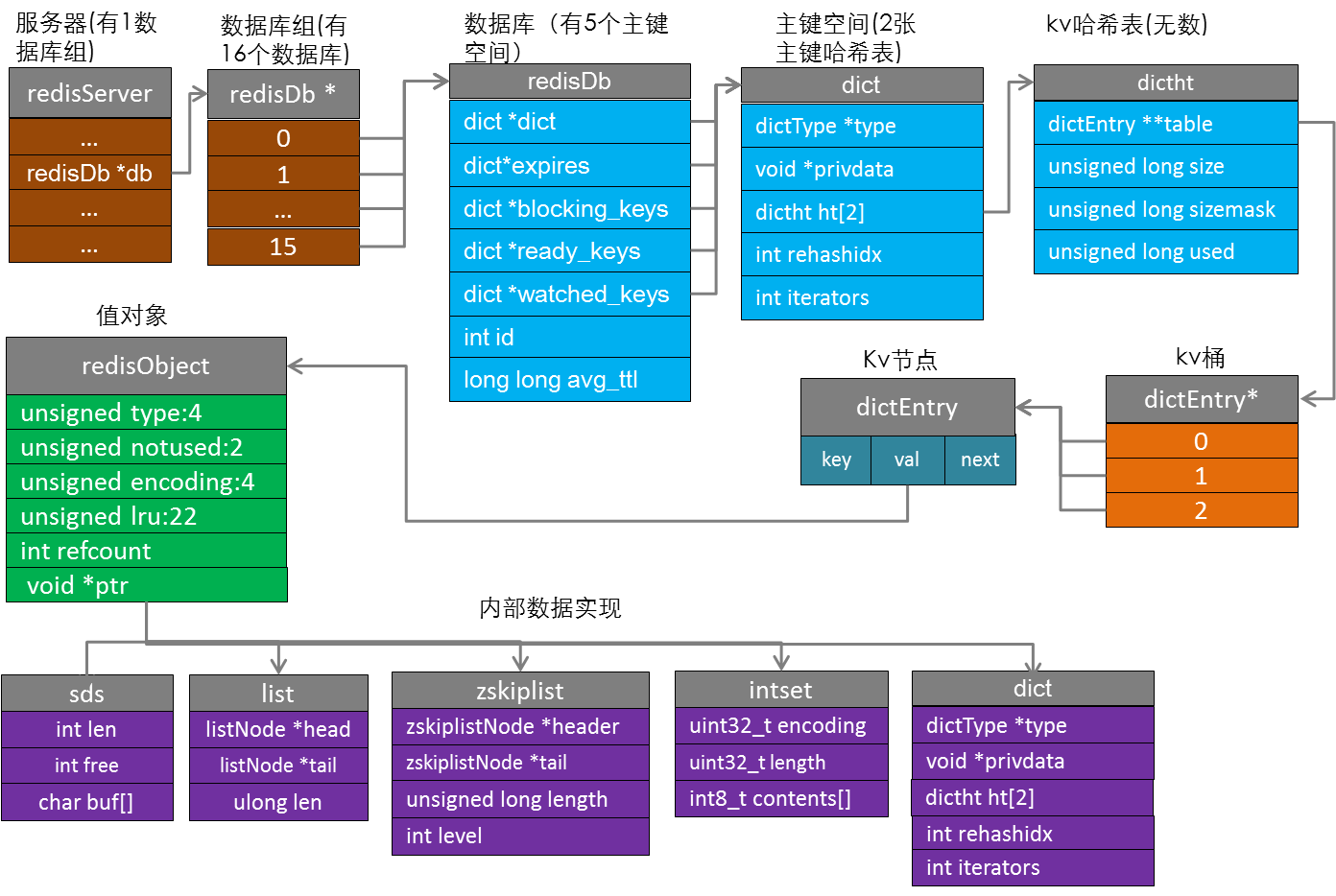
共享对象一般包括：

★各种命令的返回值，比如执行成功时返回的 OK ，执行错误时返回的 ERROR ，类型错误时返回的 WRONGTYPE ，命令入队事务时返回的 QUEUED ，等等。

★包括 0 在内，小于 REDIS\_SHARED\_INTEGERS（默认值为 10000）

的所有整数

2.1.3 Redis的数据组织方式：



2.1.4 内部数据结构

整体来说，redis的内部数据结构比较简单，一共四中：动态字符串、双向链表、哈希表、跳跃表，还有两种特殊的数据结构整形集合与压缩列表，你会发现redis并没有树结构。

简单动态字符串（Simple Dynamic String，简称sds），是redis一个内部的数据结构，几乎redis上层的char\*字符串在redis底层大都会被处理为sds。如果说hash是redis的骨架，Sds则是redis数据处理的基石。C语言的char\*字符串有两个缺点：其一、不支持追加，从“HelloWorld”再到“HelloWorld！”，C语言要重新分配内存，而且对于字符串常量来说，这些内存，一直保存在静态内存区域里，永不释放直到程序结束，而C没有垃圾回收，一些在栈内引用的字符串常量在使用结束后作为一种垃圾一直存在于内存。

【注意】这与字符数组是不同的，C语言的字符数组如果是在栈内申请的，那么它的空间是在栈内，所在区间的结束自动释放。而字符串常量则不是。再者，C语言的字符串常量每次用strlen()统计字符串长度的时候，都需要遍历整个字符串，算法复杂度是O(n)。

基于以上两点，在一个大量使用字符串常量的系统里，使用C语言的字符串是不合适的。Redis实现了自己的字符串结构**sdshdr。**

**struct sdshdr**

**{**

**int len;** // --已用长度

**int free;** // --可用长度

**char buf[];** // --字符BUF

**};**

看看sdshdr是怎么来处理追加操作的，当把“HelloWorld”赋值给sdshdr的时候，分配10+1字节的空间，这时候这个sdshdr变量的值如下。

|  |  |
| --- | --- |
| len | 10 |
| free | 0 |
| buf | HelloWorld\0 |

如果在上面的字符串追加一个“！”结尾，那么redis会重新分配2\*len的空间，并把“！”追加到原来字符串的“\0”字符之前，追加后的内存结构如下。

|  |  |
| --- | --- |
| len | 20 |
| free | 9 |
| buf | HelloWorld!\0 |

这样做的好处就是，下次，如果我把“HelloWorld！”改成“HelloWorld！！！”就不需要重新分配内存，更难能可贵的是，对于这些额外的预分配空间，如果不能被有效利用，redis在适当的时候会回收这些预分配内存，详见后面章节对sds代码的解析。

再来看，获取字符串长度的问题，由于有len属性，所以每次计算长度的时候，直接给出len值就是了，算法复杂度为O(1)。

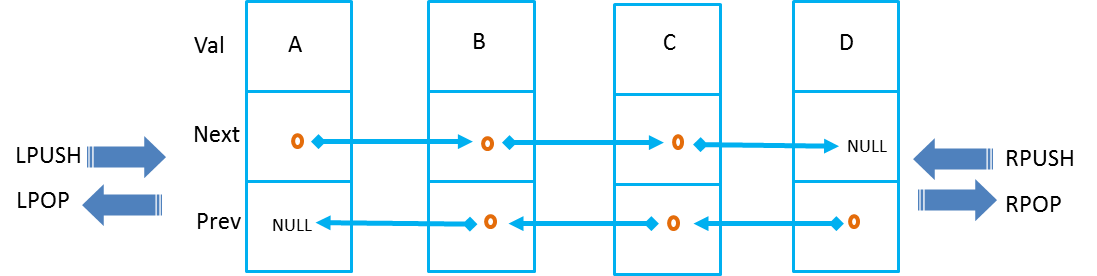
关于sds先介绍到这里，细节见后面的源码分析章节。

| 函数 | 作用 | 算法复杂度 |
| --- | --- | --- |
| sdsnewlen | 创建一个指定长度的 sds ，接受一个 C 字符串作为初始化值 | O(N) |
| sdsempty | 创建一个只包含空白字符串 "" 的 sds | O(N) |
| sdsnew | 根据给定 C 字符串，创建一个相应的 sds | O(N) |
| sdsdup | 复制给定 sds | O(N) |
| sdsfree | 释放给定 sds | O(N) |
| sdsupdatelen | 更新给定 sds 所对应 sdshdr 结构的 free 和 len | O(N) |
| sdsclear | 清除给定 sds 的内容，将它初始化为 "" | O(1) |
| sdsMakeRoomFor | 对 sds 所对应 sdshdr 结构的 buf 进行扩展 | O(N) |
| sdsRemoveFreeSpace | 在不改动 buf 的情况下，将 buf 内多余的空间释放出去 | O(N) |
| sdsAllocSize | 计算给定 sds 的 buf 所占用的内存总数 | O(1) |
| sdsIncrLen | 对 sds 的 buf 的右端进行扩展（expand）或修剪（trim） | O(1) |
| sdsgrowzero | 将给定 sds 的 buf 扩展至指定长度，无内容的部分用 \0 来填充 | O(N) |
| sdscatlen | 按给定长度对 sds 进行扩展，并将一个 C 字符串追加到 sds 的末尾 | O(N) |
| sdscat | 将一个 C 字符串追加到 sds 末尾 | O(N) |
| sdscatsds | 将一个 sds 追加到另一个 sds 末尾 | O(N) |
| sdscpylen | 将一个 C 字符串的部分内容复制到另一个 sds 中，需要时对 sds 进行扩展 | O(N) |
| sdscpy | 将一个 C 字符串复制到 sds | O(N) |

本表引自《redis设计与实现》

双向链表（Doubly Linked List），与一般的双向链表实现大同小异，没有什么值得讲的。

【结构】



【应用场景】

* 用于列表类型的底层实现
* 事务命令队列
* 服务器保存客户端的链表
* 主节点保存从节点的链表
* 订阅/发送模块保存订阅客户端链表
* 时间事件链表（time event）；
* 慢日志链表(slow log)

| **函数** | **作用** | **算法复杂度** |
| --- | --- | --- |
| listCreate | 创建新链表 | O(1) |
| listRelease | 释放链表，以及该链表所包含的节点 | O(N) |
| listDup | 创建给定链表的副本 | O(N) |
| listRotate | 取出链表的表尾节点，并插入到表头 | O(1) |
| listAddNodeHead | 将包含给定值的节点添加到链表的表头 | O(1) |
| listAddNodeTail | 将包含给定值的节点添加到链表的表尾 | O(1) |
| listInsertNode | 将包含给定值的节点添加到某个节点的之前或之后 | O(1) |
| listDelNode | 删除给定节点 | O(1) |
| listSearchKey | 在链表中查找和给定 key 匹配的节点 | O(N) |
| listIndex | 给据给定索引，返回列表中相应的节点 | O(N) |
| listLength | 返回给定链表的节点数量 | O(1) |
| listFirst | 返回链表的表头节点 | O(1) |
| listLast | 返回链表的表尾节点 | O(1) |
| listPrevNode | 返回给定节点的前一个节点 | O(1) |
| listNextNode | 返回给定节点的后一个节点 | O(1) |
| listNodeValue | 返回给定节点的值 | O(1) |

本表引自《redis设计与实现》

字典（dict），散列有两种实现方式：开散列和闭散列，Redis的hash是开散列。实在没有太多讲的，hash碰撞这些参考一下一些数据结构书籍开散列的碰撞方式即可。

**结构**

type

privedata

Ht[2]

rehashidx

iterators

table

size

sizemark

used

dictht

dict

0

1

2

dictEntry\*\*

key

dictEntry

null

val

next

null

null

table

size

sizemark

used

dictht

细心的读者可能发现，Redis的哈希实现与其他的hash实现有个不同之处，在于Redis有一个桶的数组ht[2],这个用意见下文的“重置”。

**Hash算法**

**MurmurHash2 算法** 作者Austin Appleby，MurmurHash2 算法以其高运算性能，低碰撞率而出名，由Austin Appleby创建于2008年，现已应用到Hadoop、libstdc++、Perl、[RaptorDB](http://zh.wikipedia.org/w/index.php?title=RaptorDB&action=edit&redlink=1)、nginx、[Rubinius](http://zh.wikipedia.org/w/index.php?title=Rubinius&action=edit&redlink=1) 、libmemcached等开源系统。2011年Appleby被Google雇佣，随后Google推出其变种的CityHash算法。

[注意]这个算法对你的机器行为做出如下假定:

1.我们能从任何地址读一个4字节的值而不会崩溃；

2. sizeof(int) == 4

并且有一些限制：

1.It will not work incrementally

2.对于little-endian和big-endian机器将产生不同的结果。

**unsigned int dictGenHashFunction(const void \*key, int len)**

**{**

/\* 'm' and 'r' are mixing constants generated offline.

They're not really 'magic', they just happen to work well. \*/

**uint32\_t seed = dict\_hash\_function\_seed;**

**const uint32\_t m = 0x5bd1e995;**

**const int r = 24;**

/\* Initialize the hash to a 'random' value \*/

**uint32\_t h = seed ^ len;**

/\* Mix 4 bytes at a time into the hash \*/

**const unsigned char \*data = (const unsigned char \*)key;**

**while(len >= 4)**

**{**

**uint32\_t k = \*(uint32\_t\*)data;**

**k \*= m;**

**k ^= k >> r;**

**k \*= m;**

**h \*= m;**

**h ^= k;**

**data += 4;**

**len -= 4;**

**}**

/\* Handle the last few bytes of the input array \*/

**switch(len) {**

**case 3: h ^= data[2] << 16;**

**case 2: h ^= data[1] << 8;**

**case 1: h ^= data[0]; h \*= m;**

**};**

/\* Do a few final mixes of the hash to ensure the last few

\* bytes are well-incorporated. \*/

**h ^= h >> 13;**

**h \*= m;**

**h ^= h >> 15;**

**return (unsigned int)h;**

**}**

**djb算法** 一个大小写无关的散列算法，代码很少：

**unsigned int dictGenCaseHashFunction(const unsigned char \*buf, int len)**

**{**

**unsigned int hash = (unsigned int)dict\_hash\_function\_seed;**

**while (len--)**

**{**

**hash = ((hash << 5) + hash) + (tolower(\*buf++)); //hash \* 33 + c**

**}**

**return hash;**

**}**

[注意]经过实际测试,只有当key的长度大于10字节的时候，MurmurHash的运算速度才快于DJB。而且前提是外部给出key的长。

【注意】

MurmurHash2 的更多参考资料。

<http://code.google.com/p/smhasher/> 。

[https://sites.google.com/site/murmurhash](https://sites.google.com/site/murmurhash/)/

djb 算法更多参考

<http://www.cse.yorku.ca/~oz/hash.html> 。

<http://www.cnblogs.com/rush/archive/2012/02/05/2339037.html>

**托马斯王32位混合算法(Thomas Wang's 32 bit Mix)**

unsigned int dictIntHashFunction(unsigned int key)

{

key += ~(key << 15);

key ^= (key >> 10);

key += (key << 3);

key ^= (key >> 6);

key += ~(key << 11);

key ^= (key >> 16);

return key;

}

【注意】redis虽然有该算法的代码实现，但是没有用到。

**应用场景**

* 哈希数据类型的底层实现之一(另一种是ziplist)
* 集合数据类型的底层实现之一(另一种实现是intset)
* 订阅系统

**重置**

这个是需要真正关注一下的内容。为什么要进行哈希重置？哈希表在内存中的布局都是跟进一个初始化时候一个假设的容量来分配的。随着实际的使用节点大大多于初始容量时，碰撞越来越多，每个桶上的链表长度越来越长。Hash取值的算法复杂度由O(1)退化为O(n)。

有两种办法来解决上述问题：

一种是使初始容量“足够”大，但这又会引来另外一个问题，就是hash桶的使用率太低，大量的桶处于闲置状态，导致大量的内存资源被浪费。

另一种办法就是动态增长，但这也会引发很多问题，因为一旦这个容量值变量，依附于这个值得hash算法得出的结果就完全不同，比如说原来“sxf:test”这个key用hash算法取出的桶可能是在第10250这个索引上，如果修改了这个值，可能得出的索引值变成18253。这样的结果是不能接受的，因为之前所有的key与桶的索引对应关系都被打破。Redis是采用第二种方式来解决问题的。Redis巧妙地使用了两个hash结构，正如我们分析的，哈希的重置的过程是一次缓慢的“改良”，而非一步到位的“革命”。

字典的重置(rehash)操作分为以下几个阶段:

第一阶段: 将重置标志rehashidx置为0，在ht[1]上创建一个ht[0]容量为2倍的哈希表；

此时的dict结构如下：

type

privedata

Ht[2]

rehashidx

iterators

table

size

sizemark

used

dictht

dict

0

1

2

dictEntry\*\*

key2

val

next

table

size

sizemark

used

dictht

key1

val

next

key3

dictEntry

val

next

dictEntry

dictEntry

0

1

2

dictEntry\*\*

3

4

5

key4

val

next

dictEntry

key5

val

next

dictEntry

key5

val

next

dictEntry

null

null

null

null

null

null

null

null

null

第二阶段、 把ht[0]上的数据逐步迁移到ht[1]；注意迁移是分多次逐步进行的而不是一次性完成。因为在数据量大的情况下迁移的过程需要一定时间，如果一次性迁移，用户在Redis迁移过程中不能继续正常使用Redis。随着节点的移动，字典的rehashidx 、 ht[0]->used 和 ht[1]->used 三个值也随之变化。此时的rehashidx用来记录重置进行到ht[0]的位置。

下面是rehashidx值为1时，字典的样子:

type

privedata

Ht[2]

rehashidx

iterators

table

size

sizemark

used

dictht

dict

0

1

2

dictEntry\*\*

table

size

sizemark

used

dictht

key4

val

next

dictEntry

0

1

2

dictEntry\*\*

3

4

5

null

null

key1

val

next

dictEntry

key3

val

next

dictEntry

key6

val

next

dictEntry

null

null

null

null

key2

val

next

dictEntry

null

null

key5

val

next

dictEntry

null

第三阶段、所有的节点都从 ht[0] 迁移到 ht[1]：

【注意】在重置过程中，对整个字典的key的查找，如果再ht[0]表没有找到指定key，还需要去ht[1]表查找一次。而重置过程中所有的新加节点都将添加到ht[1]表中。

type

privedata

Ht[2]

rehashidx

iterators

table

size

sizemark

used

dictht

dict

0

1

2

dictEntry\*\*

table

size

sizemark

used

dictht

key4

val

next

dictEntry

0

1

2

dictEntry\*\*

3

4

5

null

null

key1

val

next

dictEntry

key3

val

next

dictEntry

key6

val

next

dictEntry

null

null

null

null

key2

val

next

dictEntry

null

null

key5

val

next

dictEntry

null

4. 释放 ht[0] 的空间, 用ht[1]代替ht[0]，而创建一个新的哈希表，赋值给ht[1];将rehashidx 属性设置为 -1，整个重置过程到此结束。下图是重置完成后的结构:

type

privedata

Ht[2]

rehashidx

iterators

table

size

sizemark

used

dictht

dict

table

size

sizemark

used

dictht

key4

val

next

dictEntry

0

1

2

dictEntry\*\*

3

4

5

null

key1

val

next

dictEntry

key3

val

next

dictEntry

key6

val

next

dictEntry

null

null

null

key2

val

next

dictEntry

null

key5

val

next

dictEntry

null

null

**思维发散**：如果不想rehash怎么办？有其他解决方式么？效率评估如何？

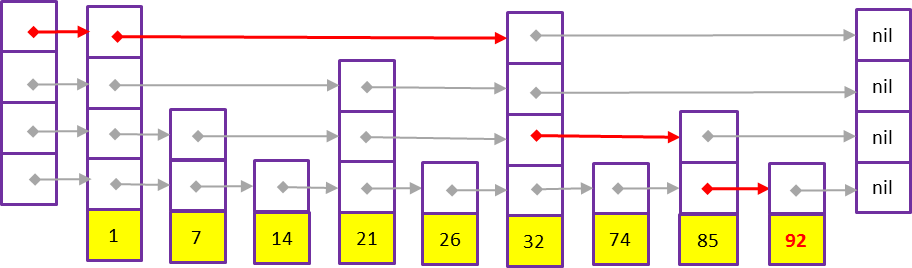
思路一：一致性哈希(DHT)？

思路二：不再简单滴将哈希值和最大容量做除法取模映射，而是将哈希值与服务器的对应关系作为元数据，交给专门的元数据管理器来管理。访问数据时，首先计算哈希值，再查找元数据管理器，获得该哈希值对应的桶索引。

这两种方法都得不偿失，增加了每次插入的负担，而hash重置并不经常进行的动作，所以比较起来，Redis的选择是比较明智的。

跳跃表（Skip List）

**结构**



**复杂度分析**

一个数据结构的好坏大部分取决于它自身的空间复杂度以及基于它一系列操作的时间复杂度。跳跃表之所以被誉为几乎能够代替平衡树，其复杂度方面自然不会落后。我们来看一下跳跃表的相关复杂度：

空间复杂度： O(n) （期望）

跳跃表高度： O(logn) （期望）

相关操作的时间复杂度：

查找： O(logn) （期望）

插入： O(logn) （期望）

删除： O(logn) （期望）

**上面斜体字内容均引自**[**http://blog.csdn.net/caoeryingzi/article/details/6018070**](http://blog.csdn.net/caoeryingzi/article/details/6018070)

**插入元素**

我们向一个跳跃表插入一个元素，需要关注两个维度的因素，一是水平的位置，另一是垂直的Level。水平的位置很好确定，因为所有有序链表也好，BST也好，就是把当前插入的节点的一个参数值(Redis的sortset的这个参数是socore)和数据结构里各元素的相应值比较，然后确定位置。

而垂直的level确定起来就比较困难，不同的决策对效率的影响也很大。可以是固定的二分法，也可以是随机决策。Redis是用随机的方式来实现的，这个实现在zset.c的zslRandomLevel()这个函数。

这个函数的实现很简单，

**int level = 1;**

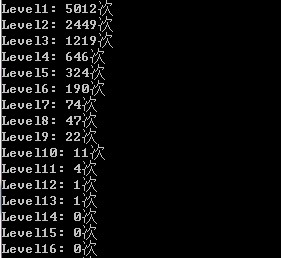
**while ( (random()&0xFFFF) < ( 0.25 \* 0xFFFF))**

**{**

**level += 1;**

**}**

这个函数是基本按照二分分布的，下面是我的Level命中测试:



**删除元素** 在跳跃表中查找到这个元素的位置，如果未找到，则退出；

如果找到，就将该元素所在整列从表中删除并将多余的“空链”删除。

**应用场景**

用来支持序集合的数据类型是Redis唯一一处使用跳跃表的地方。

其实有续集也可以用各种平衡二叉树(Red-Black Tree, AVL Tree)结构来实现。关于Redis为什么会使用跳跃表来实现，Redis的作者在回答他人的问题时，给出的一些原因，原文如下:

*There are a few reasons:*

*1) They are not very memory intensive. It's up to you basically. Changing parameters about the probability of a node to have a given number of levels will make then less memory intensive than btrees.*

*2) A sorted set is often target of many ZRANGE or ZREVRANGE operations, that is, traversing the skip list as a linked list. With this operation the cache locality of skip lists is at least as good as with other kind of balanced trees.*

*3) They are simpler to implement, debug, and so forth. For instance thanks to the skip list simplicity I received a patch (already in Redis master) with augmented skip lists implementing ZRANK in O(log(N)). It required little changes to the code.*

*About the Append Only durability & speed, I don't think it is a good idea to optimize Redis at cost of more code and more complexity for a use case that IMHO should be rare for the Redis target (fsync() at every command). Almost no one is using this feature even with ACID SQL databases, as the performance hint is big anyway.*

*About threads: our experience shows that Redis is mostly I/O bound. I'm using threads to serve things from Virtual Memory. The long term solution to exploit all the cores, assuming your link is so fast that you can saturate a single core, is running multiple instances of Redis (no locks, almost fully scalable linearly with number of cores), and using the "Redis Cluster" solution that I plan to develop in the future.*

**扩展参考**

<http://dsqiu.iteye.com/blog/1705530>

<http://blog.csdn.net/caoeryingzi/article/details/6018070>

2.1.5 事务

**简介:**传统关系数据库事务(RMDBS Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完整地执行，要么完全地不执行。 事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。通过将一组相关操作组合为一个要么全部成功要么全部失败的单元，可以简化错误恢复并使应用程序更加可靠。一个逻辑工作单元要成为事务，必须满足所谓的ACID（原子性、一致性、隔离性和持久性）属性。

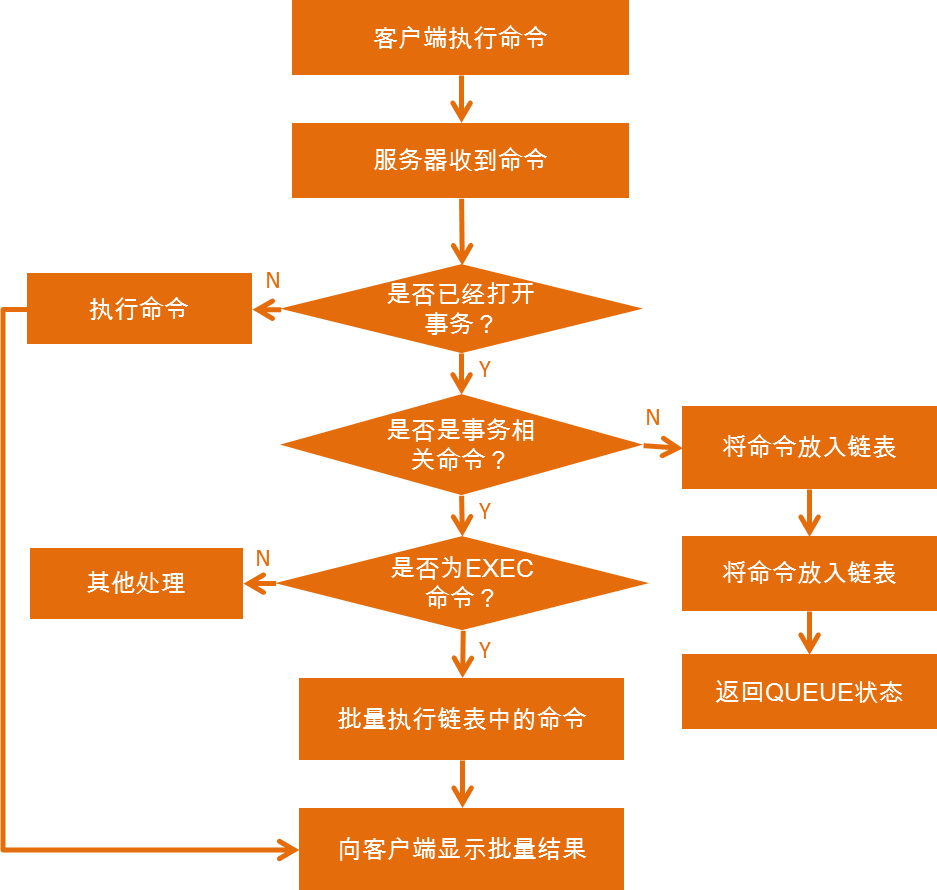
Redis的事务可以保证1)一个事务队列中的所有事务要么都不执行; 2)一个事务的命令依次执行而不被其他命令插入(原子性)。

**特点**: 与RMDBS的事务相比，Redis的事务具有下述特点:

1. Redis的事务不支持回滚，如果事务中有一条命令出现运行错误，事务里的其他命令会继续执行(包括出错命令之后的命令)，开发者需要手动复原数据。
2. 在开启AOF机制的情况下，multi和exec的之间的命令也会用一条write命令写到硬盘。如果在写时被硬中断(停电或管理员强制kill)，Redis重启时会检测到错误，可以用redis-check-aof工具修复。
3. 支持CAS(check and set)操作，watch命令可以锁定某个key，unwatch命令取消锁定，在事务执行时如果检测到watch的key被修改，事务失败。事务成功执行后，会unwatch掉所有观察的keys

【注意】作为分布式存储系统，实现事务回滚的代价很大，Redis的命令一般不会执行错误，大部分出错都是由于程序自身问题。

**流程**:



**原理**：事务开始后(multi 命令执行后)，Redis将所有命令(MULTI、EXEC、WATCH、UNWATCH、DISCARD除外)放入一个链表。当exec命令执行后，将队列中所有命令依次执行，并一次性返回输出结果。

2.2 事件循环

事件框架在高并发的网络坏境中用的越来越多了，Lighttpd、NodeJS使用了libev事件框架、Memcached使用了libevent事件框架。Redis由于力求保持简单，没有采用开源的通用事件框架，仅仅libenvet的源码就达30000多行，而整个redis代码也才30000多行，redis实现了自己的事件框架，大概500行左右的代码。

【注意】Nginx也是自己实现的事件框架。事件驱动模型的四要素:1)事件源(引发事件) 2)侦听器(监听事件) 3)处理器(处理事件) 4)映射器(事件与处理器的映射关系)

**事件源**: Redis的事件有两种:文件事件和时间事件。linux操作系统产生的事件包括:文件读写、网络收发、操作系统信号、时间事件。前两个linux统一描述为文件，信号事件Redis没有在事件库中实现（确实没这个必要），超时事件Redis只向操作系统借了个获取系统时间的系统调用。于是库中事件只有两种：文件读写和超时。看看事件循环和两个事件的定义

aeCreateEventLoop

aeCreateTimeEvent

aeCreateFileEvent

是否停止循环

BeforeSleep处理

aeProcessEvents

aeDeleteEventLoop

end

查找到距目前时间最近的一个时间事件

计算最近时间事件与现在的时间间隔tvp

aeApipoll(eventloop, tvp)

处理已监听到的文件事件记录于Eventloop.fire[]

Poll超时？

处理目前已超时的时间事件

在tvp时间内检测监听到的文件事件

是否出现相关文件事件

(AE\_READABLE

AE\_WRITEABLE)

记录文件号到fire[i].fd,记录事件到fire[i].mask

2.3 持久化

Redis的持久化可以有两种方式:rdb文件方式与aof文件方式。

2.3 .1 Rdb

Rdb的主要原理就是在某个时间点把内存中的所有数据进行快照并保存到磁盘上。rdb是Redis默认的持久化方式。

有两种方式可以触发rdb持久化：其中一种是通过时间事件触发,触发条件可以由用户在配置文件中定义，由两个参数构成：时间和改动的键数。当指定时间内更改的键数大于配置值，Redis就从当前进程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。此时的父进程继续接受处理客户端发送来的命令，而子进程开始将内存中的数据写入硬盘临时文件；直到子进程写完毕后，就会用该临时文件替换旧的rdb文件。

【注意】在使用fork进程保存数据的实现方式充分利用了类unix系统的写时复制(copy on write)技术，操作系统在创建子进程的时候，把主进程的数据复制了一份以保证子进程的操作不受影响，所以新的rdb文件存储的是执行fork一刻的内存数据，明白这点特别重要。

在所有数据写完后再把这个临时文件用原子函数rename()重命名为目标rdb文件。这种实现方式充分利用fork的。

另外一种是通过save或bgsave命令主动触发保存数据快照，区别是save这种是阻塞式的，即不会通过生成子进程而直接使用主进程来进行数据集快照的保存，后者是通过fork子进程进行保存。

2.3.2AOF

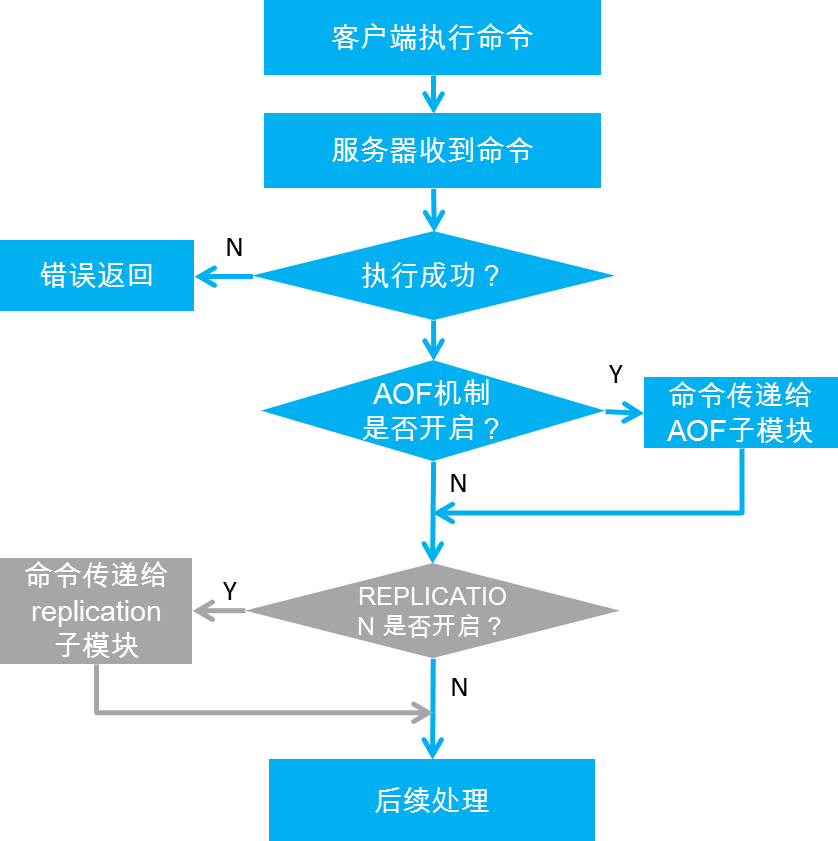
AOF(append only file)的实现原理是将每次成功执行的写操作都会记录到硬盘上。由于只是一个append到文件操作，所以写到硬盘上的操作往往非常快。但文件的形式是文本文件，所以浪费空间。当系统崩溃时，可以通过回放AOF来恢复数据。

【注意】Redis默认不开启AOF机制。

【注意】由于每次有写命令成功执行后，数据不是立即写入硬盘，而是进入系统的磁盘缓存。默认情况下每30秒执行一次同步操作，将缓存数据写入磁盘，在redis中可以通过appendfsync参数设置同步：always是每执行写入都会同步；no表示不进行同步，由系统每30秒进行一次同步；everysec是每秒进行一次同步。默认情况下Redis是采用everysec设置，即每秒同步一次。

【注意】Redis可以同时开启rdb和aof。

**流程**



**重写(Rewrite)**随着时间的推移，Redis操作的命令频繁地被记录到AOF文件中，AOF文件的体积也逐渐膨胀，这不仅占用了大量的磁盘空间，也使得从AOF恢复数据库的时间变长。为了解决上述问题，Redis实现了一个重写机制(Rewrite)——当文件大小到达一个临界值，重写机制会被启动。重写一个数据相同但体积更小的AOF文件。

重写的文件为什么会比原始AOF文件小？原因根本在于原始AOF文件冗余大量无效指令。例如：

LPUSH list A

LPUSH list B

LPUSH list C

LPUSH list D

LPOP list

LPOP list

LPOP list

这七条语句等价于一条LPUSH list A 。关于重置的细节见后面源码分析章节。

【注意】

*1.由于写操作通常是有缓冲的，所以有可能AOF操作并没有写到硬盘中，一般可以通过fsync()来强制输出到硬盘中。而fsync()的频率可以通过配置文件中的flush策略来指定，可以选择每次事件循环写操作都强制fsync或者每秒fsync至少运行一次。*

*2.当rewrite子进程开始后，父进程接受到的命令会添加到aof\_rewrite\_buf\_blocks中，使得rewrite成功后，将这些命令添加到新文件中。在rewrite过程中，原来的AOF也可以选择是不是继续添加，由于存在性能上的问题，在rewrite过程中，如果fsync()继续执行，会导致IO性能受损影响Redis性能。所以一般情况下rewrite期间禁止fsync()到旧AOF文件。这策略可以在配置文件中修改。*

*3.在rewrite结束后，在将新rewrite文件重命名为配置中指定的文件时，如果旧AOF存在，那么会unlink掉旧文件。这是就存在一个问题，处理rewrite文件迁移的是主线程，rename(oldpath, newpath)过程会覆盖旧文件，这是rename会unlink(oldfd)，而unlink操作会导致block主线程。这时，我们就需要类似libeio(http://software.schmorp.de/pkg/libeio.html)这样的库去进行异步的底层IO。作者在bio.c有一个类似的机制，通过创建新线程来进行异步操作。*

2.4**配置系统**

2.5**脚本系统**

2.6**慢日志系统**

日志就是日志，为什么叫“慢日志”？是什么慢呢？慢多少呢？带着这几个疑问，让我们来深入分析Redis的慢日志系统。

“慢”特指命令执行的速度，Redis会统计每条指令的执行时间，如果执行时间超一个参考标准，就往日志里插入一条记录。这个参考标准就是server.slowlog\_log\_slower\_than这个配置参数，这个参数的默认值是1000000微秒(即1秒)。Redis把这些执行慢的命令记录下来，方便用户跟踪Redis系统性能，以及在此基础上进行调优。

【注意】慢日志的内部实现是使用链表结构，每个节点都是slowlogEntry结构，可见Redis的慢日志根本不是存储在文件中，而是直接存储在内存中。

2.7 订阅系统

*Redis 通过* [*PUBLISH*](http://redis.readthedocs.org/en/latest/pub_sub/publish.html#publish)*、*[*SUBSCRIBE*](http://redis.readthedocs.org/en/latest/pub_sub/subscribe.html#subscribe) *等命令实现了订阅与发布模式， 这个功能提供两种信息机制， 分别是订阅/发布到频道和订阅/发布到模式， 下文先讨论订阅/发布到频道的实现， 再讨论订阅/发布到模式的实现。*

Redis用一个字典结构(dict)来实现整个订阅系统，Redis的订阅系统所有数据都放在server全局变量的dict \*pubsub\_channels里。相当于一个观察者模式。

在初始化redis服务器的时候，就给这个创建了字典结构。在这个字典结构里，频道作为key，而value是客户端链表指针。然后每次客户端在SUBSCRIBE n的时候，redis服务器把这个客户端作为一个节点添加到相应channel所对应的client链表末尾。

**结构**

Pubsub\_

channels

Channels1

Client1

Client4

Client8

null

Channels2

Channels3

…

ChannelsN

2.8 IO与网络

支持多平台: linxu(epoll)\win(select)\freebsd(kqueue)\ evport

单线程IO复用

aeEvent封装IO与定时器

默认支持10K连接AE\_SETSIZE(1024\*10)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **process** | **thread** | **IO** |
| redis | single | single | non-blocked |
| memcached | single | multi | non-blocked |
| nginx | multi | single | non-blocked |
| Apache(MPM) | multi | multi | non-blocked |

2.9 主从复制(Replication)

Redis支持一主多从结构，slave节点也可以接受其他slave节点的连接。主从连接主要有如下几个重要的用途：

1. 数据冗余备份，提高数据安全。
2. 从节点作为主节点的扩展，提供只读服务。
3. 实现主从结构后，可以用从节点者持久化，分担主节点压力。

Redis通过Replication机制完成主库(master)到从库(slave)之间的数据同步。

**文件同步与命令同步**

Master节点接受到slave节点与之相连接后，第一次的同步是文件形式的同步：master服务器会在后台启动一个进程创建rdb文件并发送给slave端，slave端收到后通过rdb文件完成对master的复制。当文件形式的同步完成后，以后的同步是依靠命令同步来完成。详细细节见后面章节关于对该功能 部分源码的分析。

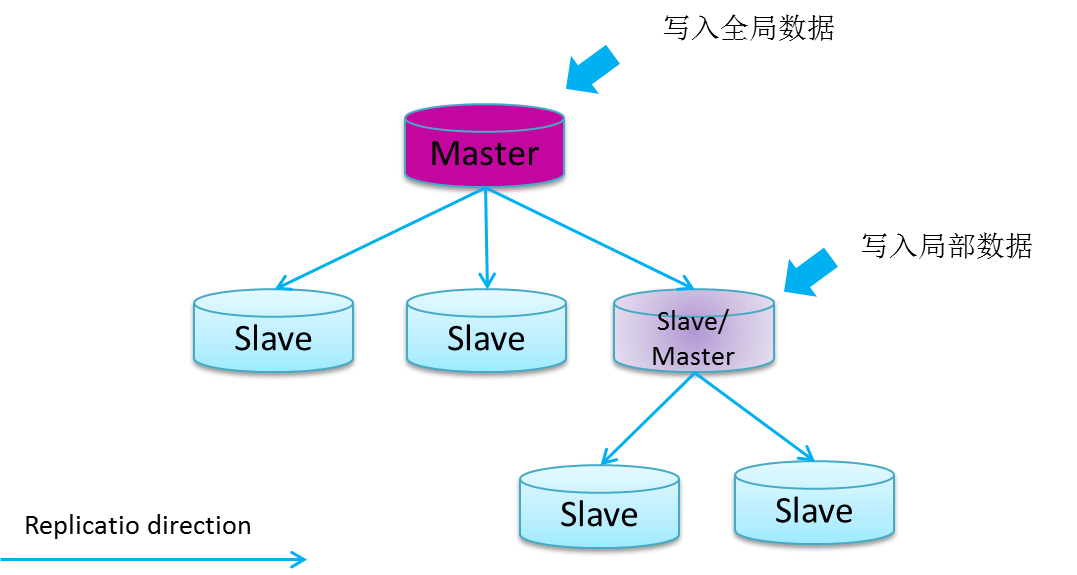
【注意】对于master来说(可能其状态是slave)，同步的过程是非阻塞的，可以继续处理其他命令，而对于slave来说第一次同步是阻塞的。

**工作流程**

我将replication过程分为五个阶段：初始阶段、连接阶段、文件保存阶段、文件同步阶段和命令同步阶段。细节参考后面章节的“主从复制(Replication)系统”。

**多级同步**

Redis支持多级同步，其结构见下图:



**主从转换**

*slaveof可以转换Redis中结点的状态。对于Slave结点，slaveof no one可以将Slave与Master断开，并将Slave转为主机。对于Master，slaveof ip port可以讲主机转为Slave。Redis的主从转换非常简单，如果同步未完成，则取消同步；否则，直接改变server.masterhost等状态。*

【注意】Replication系统算是Redis里比较复杂的系统了，但是其实现方式比起其他分布式系统来说，还是简单。详细的分析请参看后文。

第三章 redis源码分析

3.1源码文件简介

|  |  |  |
| --- | --- | --- |
| 文件 | 功能 | 备注 |
| **adlist.h/adlist.c** | 双向链表 |  |
| **ae.h/ae.c** | 事件驱动 |  |
| **ae\_epoll.c** | epoll接口, | Linux下的IO接口 |
| **ae\_kqueue.c** | kqueue接口 | Freebsd |
| **ae\_select.c** | select接口 | Windows |
| **ae\_evport.c** | event ports接口 | Illumos(OpenSolaris的衍生版本) |
| **anet.h/ anet.c** | 网络处理 | 为Server/Client通信的基础封装 |
| **aof.c** | AOF文件相关处理 |  |
| **asciilogo.h** | Redis的字符logo |  |
| **bio.h/ bio.c** | Background I/O 服务 |  |
| **bitops.c** | Bit操做函数 |  |
| **config.h/ config.c** | 配置文件解析 |  |
| **crc64.c** | crc64算法 |  |
| **db.c** | DB处理 |  |
| **debug.c** | 用于调试使用 |  |
| **dict.h/dict.c** | hash表 |  |
| **endianconv.h/endianconv.c** | 大小端转换 |  |
| **fmacros.h** | 用于Mac下的兼容性处理 |  |
| **help.h** | 命令的提示信息 |  |
| **intset.h/intset.c** | 转换为数字类型数据 |  |
| **lzf.h/lzf\_c.c** | LZF压缩算法 | 用于本地数据库的保存 |
| **lzf\_d.c** | LZF压缩算法 | 用于本地数据库的保存 |
| **lzfP.h** | LZF压缩算法 | 用于本地数据库的保存 |
| **memtest.c** |  |  |
| **migrate.c** | DUMP, RESTORE and MIGRATE 命令 |  |
| **multi.c** | 事务 |  |
| **networking.c** | 读取、解析和处理客户端命令 | 网络协议传输方法定义 |
| **object.c** | 各种对像的创建与销毁,string、list、set、zset、hash |  |
| **pqsort.h/pqsort.c** | 排序算法 |  |
| **pubsub.c** | 发布-订阅系统 |  |
| **rand.h/rand.c** | 随机数产生 |  |
| **rdb.h/rdb.c** | redis数据文件处理 |  |
| **redisassert.h** | 断言 |  |
| **redis-benchmark.c** | redis性能测试 |  |
| **redis-check-aof.c** | 用于更新日志检查的实现 |  |
| **redis-check-dump.c** | 用于本地数据库检查的实现 |  |
| **redis-cli.c** | 客户端程序 |  |
| **redis.c** | redis的主文件 | main函数 |
| **redis.h** | redis的主文件头 |  |
| **release.c** | 用于发布使用 |  |
| **replication.c** | 数据同步master-slave |  |
| **rio.h/rio.c** | Rio对象 |  |
| **scripting.c** | 脚本 |  |
| **sds.h/sds.c** | 动态字符串 |  |
| **sentinel.c** | 警戒模式 |  |
| **sha1.h/sha1.c** | sha算法的实现 |  |
| **slowlog.h/slowlog.c** | 慢日志 |  |
| **solarisfixes.h** | Solaris系统的兼容性实现 |  |
| **sort.c** | 用于list、set、zset排序 |  |
| **syncio.c** | 用于同步Socket和文件I/O操作 |  |
| **t\_hash.c** | hash类型处理 |  |
| **t\_list.c** | list类型处理 |  |
| **t\_set.c** | set类型处理 |  |
| **t\_string.c** | string类型处理 |  |
| **t\_zset.c** | 有序sort |  |
| **testhelp.h** | 一个C风格的小型测试框架 |  |
| **util.h/util.c** | 通用工具 |  |
| **version.h** | Redis版本号定义 |  |
| **ziplist.h/ziplist.c** | 压缩列表 |  |
| **zipmap.h/zipmap.c** | 压缩hash |  |
| **zmalloc.h/zmalloc.c** | 内存管理 |  |

3.2 主逻辑处理

3.2.1 main()函数

**#ifdef INIT\_SETPROCTITLE\_REPLACEMENT**

**spt\_init(argc, argv);**

**#endif**

如果定义了INIT\_SETPROCTITLE\_REPLACEMENT宏，则对库和服务器配置做一些初始化。

**setlocale(LC\_COLLATE,"");**

PROCT

**zmalloc\_enable\_thread\_safeness()**打开内存分配线程安全标志，这个函数只有一句代码即zmalloc\_thread\_safe =1，zmalloc\_thread\_safe是zmalloc.c的一个静态变量，这个值为1的情况下，redis内存的申请、释放、使用都会加锁。

【注意】此处值得一提的是，在i386CPU和AMD64CPU上，在GCC库版本号大于40100的情况下，会直接使用原子指令代替锁操作，原子指令时CPU层面的支持，无需内核锁对象介入，性能大大提高，redis针对系统环境和硬件进行效率优化。这个原子指令就是\_\_sync\_add\_and\_fetch，一个指令实现增加并获取一个值，熟悉汇编的人应该不陌生。

zmalloc\_set\_oom\_handler(redisOutOfMemoryHandler);

设置内存分配失败后的回调函数，如果不主动设置回调函数，redis将会会记录文件“zmalloc:Out of memory trying to allocate xxx bytes”，并退出线程。在这里，redis没有使用默认处理方式，而是选择记录Log，并主动报告一个OMM错误。

srand(time(NULL)^getpid()); 用当前时间随机种子值，用当前时间与进程ID进行按位异或，增大随机种子的随机概率。

gettimeofday(&tv,NULL);

dictSetHashFunctionSeed(tv.tv\_sec^tv.tv\_usec^getpid());

取得当前时间(秒和微秒)，用秒数、微秒及进程ID值三者之间的按位异或，作为hash算法(djb算法)的种子值。这个函数也只有一句代码，dict\_hash\_function\_seed,redis给这个变量提供了一个默认值5381。这个值用于hash数据结构，以后讲到hash数据结构即可。

server.sentinel\_mode = checkForSentinelMode(argc,argv);服务器是否用监视模式运行，监视模式下，redis将记录监视主节点(master nodes)的一些数据。

这个函数如果第一个参数是"redis-sentinel"，或者其他参数中有 “—sentinel”都会返还1,否则0。这句话完全可以放入initServerConfig()函数。

initServerConfig();

初始化服务器配置。配置内容很多，详细介绍见下文。

if(server.sentinel\_mode) {initSentinelConfig();initSentinel();}

initSentinelConfig();位于sentinel.c文件，设置监视端口为26379；

initSentinel();位于sentinel.c文件，清空服务器的命令表，并加入SENTINEL命令。并初始化sentinel全局变量，这个变量时一个sentinelState类型，分析到监视模式会详细深入。

if (argc >= 2){…} 这部分进行读入选项和配置文件，并修改服务器配置，如果第二个启动参数是“-v”或者”--version”， 输出当前redis版本号后退出进程; 如果第二个参数是"--help"或"-h"，则调用usage() 输出控制台服务器一些帮助信息后退出;如果第二个参数是"--test-memory"，那么根据第三个参数进行内存测试后，退出。

if(argv[j][0] != '-' || argv[j][1] != '-')

{

configfile = argv[j++];

}

记住，C程序启动后，第一个启动参数是默认的，即是你的启动路径，第2~n个参数才是你配置的。只有是第二个参数不是以“--”开头，就认为是配置文件。

while(j != argc){} 解析其他配置选项，记录到options这个sds变量，并以“ ”结尾(关于sds，详见redis内部数据结构暂时理解为字符串)。如果是以“--”开头的参数，例如 “--port 6380”这个启动参数，将给options追加入“\nport 6380 ”。

options= sdscat(options,argv[j]+2); // 这个+2目的是为了去掉”--”。

如果是以“-”开头的参数,则需要进入sdscatrepr()函数特殊处理，并追加入options变量。这里的特殊处理是指对转义字符的处理，例如“-xx\nyy\t””将被处理成“xx\\nyy\\t”。最后两句,resetServerSaveParams(); //定义于 config.c ，释放并置server.saveparams，server.saveparamslen=0。

if (configfile) server.configfile = getAbsolutePath(configfile);

resetServerSaveParams();

loadServerConfig(configfile,options);

根据配置文件和传入的选项，修改server 变量（服务器配置），如果配置文件名是以“-\0”开头，认为配置从标准输入。

loadServerConfigFromString(config);

loadServerConfigFromString(config); 这个函数非常重要，是跟进传入的字符串来完成服务器配置(也即修改server常量)

lines = sdssplitlen(config,strlen(config),"\n",1,&totlines);

在这一句根据“\n”为分割符，把config字符串分割，然后返回一个sds数组指针。然后遍历这个数组，然后根据参数做server常数的配置。

if (server.daemonize) {daemonize();}

如果服务器配置是守护形式启动，就创建守护进程Daemonize()函数里面 if ((fd = open("/dev/null",O\_RDWR, 0)) != -1){} //所有输出都放在 /dev/null下， 如果redis工作在守护线程下，配置文件的'stdout'设置为'logfile'，就不能正常记录日志。如果日志权限不足，就会进行以下行为：

initServer(); initServer()的详细详见下文

if (server.daemonize) createPidFile();

createPidFile()函数的内部实现 FILE \*fp = fopen(server.pidfile,"w")，以可写的方式打开server.pidfile配置的文件。

redisAsciiArt();

这个函数的内部实现是

#include "asciilogo.h"

char \*buf = zmalloc(1024\*16);

char \*mode = "stand alone";

if (server.cluster\_enabled) mode = "cluster";

else if (server.sentinel\_mode) mode = "sentinel";

snprintf(buf,1024\*16,ascii\_logo, REDIS\_VERSION, redisGitSHA1(),strtol(redisGitDirty(),NULL,10) > 0,(sizeof(long) == 8) ? "64" : "32",mode, server.port,(long) getpid() );

redisLogRaw( REDIS\_NOTICE|REDIS\_LOG\_RAW, buf );

zfree(buf);

打印ASCII图片(实际是日志)，就是把 logo、版本、redis\_git\_sha1、redis\_git\_dirty、系统bits、redis运行模式、服务器端口、进程ID等信息写入日志文件。ascii\_logo是一串字符串，用一些符号绘制的redis的logo，非常蛋疼。另外一个有意思的事情是，在函数里包含头文件，一个不错的思路。redisLogRaw()详细见下文。

if (!server.sentinel\_mode)

{

……

loadDataFromDisk();

……

}

如果不在后台模式下，会输出一些信息。这里只看一个函数loadDataFromDisk()，如果server.aof\_state这个配置了REDIS\_AOF\_ON，要通过loadAppendOnlyFile()载入AOF文件，否则加载RDB文件，如果RDB加载失败，则记录报错日志并退出程序，loadAppendOnlyFile()函数分析见aof.c，rdbLoad()函数分析见rdb.c。

if (server.maxmemory > 0 && server.maxmemory < 1024\*1024)

{

redisLog(…);

}

打印内存限制警告

server.el->beforesleep = beforeSleep;

beforeSleep()这个方法在Redis每次进入sleep/wait去等待监听的端口发生I/O事件之前被调用。beforeSleep函数分析，见下文。

aeMain(server.el);

启动事件主循环，这个函数内部实现为

eventLoop->stop = 0;

while (!eventLoop->stop)

{

if (eventLoop->beforesleep !=NULL)

{ eventLoop->beforesleep(eventLoop); }

aeProcessEvents(eventLoop, AE\_ALL\_EVENTS);

}

如果有需要在事件处理前执行的函数，那么其回调函数，接着执行事件aeProcessEvents()，这个函数详见ae.c。

aeDeleteEventLoop(server.el);

主循环结束后会运行到这，关闭服务器，删除事件。内部实现为

aeApiFree(eventLoop);

zfree(eventLoop->events);

zfree(eventLoop->fired);

zfree(eventLoop);

aeApiFree()在win、linux、FreeBSD都是不同的，拿linux下

aeApiState \*state = eventLoop->apidata;

close(state->epfd);

zfree(state->events);

zfree(state);

没什么难理解的，都是关闭epoll，释放事件。

3.2.2 initServerConfig()

Server是一个redisServer类型的全局变量，也是redis最重要的一个全局变量，大部分的服务器配置和一些重要状态都保存在这里。其结构详见附录。initServerConfig()函数是初始化Server的部分成员变量。这里只部分介绍，没有什么值得详细分析的。

server.arch\_bits = (sizeof(long) == 8) ? 64 : 32;

一个经典的办法，判断当前执行环境是多少位。

server.port = REDIS\_SERVERPORT;

TCP/IP连接的默认端口为REDIS\_SERVERPORT即6379，这个6379在手机上字母组合Merz对应的数字是6379，Merz是redis之父的网名，嚯嚯。

server.dbnum = REDIS\_DEFAULT\_DBNUM; 默认数据库(实际相当于数据名称空间)个数16个

server.verbosity = REDIS\_NOTICE; 日志详细级别，默认是NOTICE级别

server.maxidletime = REDIS\_MAXIDLETIME;客户端连接过期时间，默认永不过期

server.client\_max\_querybuf\_len = REDIS\_MAX\_QUERYBUF\_LEN;客户端查询缓存最大长度，默认1G

server.syslog\_ident = zstrdup("redis");系统日志识别字符串

server.syslog\_facility = LOG\_LOCAL0;

在这个函数中，对server变量进行了部分成员的初始化，其中:runid:运行该redis服务器端程序的唯一标识，即每次启动都会一个唯一ID，用来区分不同的redis服务器端程序。maxidletime:最大空闲时间，就是client连接到server时，如果超出这个值，就会被自动断开，当然，master和slave节点不包括。如果client有阻塞命令在运行，也不会断开。saveparams:这个存储的是redis服务器端程序从配置文件中读取的持久化参数

lruclock:是redis实现LRU算法所需的，每个redis object都带有一个lruclock，用来从内存中移除空闲的对象。commands:是redis命令的字符数组。sentinel\_mode:是否开启redis的哨兵模式，也就是是否监测，通知，自动错误恢复，是用来管理多个redis实例的方式。

3.2.3 initServer()

dup2(fd, STDIN\_FILENO); 赋给fd标准输入句柄

dup2(fd, STDOUT\_FILENO); 关闭标准输入，赋给fd标准输出句柄

dup2(fd, STDERR\_FILENO); 关闭标准输出，赋给fd标准错误输出句柄

if(fd >STDERR\_FILENO) close(fd); 关闭标准错误输出

signal(SIGHUP, SIG\_IGN);让系统忽略SIGHUP信号(控制终端信号)，作为守护进程运行，不会有控制终端，所以忽略掉SIGHUP信号。

signal(SIGPIPE, SIG\_IGN);让系统忽略SIGHUP信号(控制终端信号)，SIGPIPE信号是在写管道发现读进程终止时产生的信号，向已经终止的SOCK\_STREAM套接字写入也会产生此信号。redis作为server，不可避免的会遇到各种各样的client，client意外终止导致产生的信号也应该在server启动后忽略掉。

setupSignalHandlers();

setupSignalHandlers函数处理的信号分两类：1）SIGTERM。SIGTERM是kill命令发送的系统默认终止信号。也就是我们在试图结束server时会触发的信号。对这类信号，redis并没有立即终止进程，其处理行为是，设置一个server.shutdown\_asap，然后在下一次执行serverCron时，调用prepareForShutdown做清理工作，然后再退出程序。这样可以有效的避免盲目的kill程序导致数据丢失，使得server可以优雅的退出。2）SIGSEGV、SIGBUS、SIGFPE、SIGILL。这几个信号分别为无效内存引用（即我们常说的段错误），实现定义的硬件故障，算术运算错误（如除0）以及执行非法硬件指令。这类是非常严重的错误，redis的处理是通过sigsegvHandler，记录出错时的现场、执行必要的清理工作，然后kill自身。除上面提到的7个信号意外，redis不再处理任何其他信号，均保留默认操作。

if(server.syslog\_enabled)

{

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

}

如果服务器配置中启动了日志，那么就打开系统日志。openlog属于linux系统函数，如果不知道用法自行搜索。

server.current\_client = NULL; 设置当前处理的客户端为空

server.clients = listCreate(); 客户端链表，用adlist.c的listCreate()创建了链表(实际分配头结点内存)。

server.clients\_to\_close = listCreate(); 要被关闭的客户端链表，同上。

server.slaves = listCreate(); slave节点链表

server.monitors = listCreate(); monitor客户端链表

server.unblocked\_clients = listCreate(); 被取消阻塞的客户端链表

server.ready\_keys = listCreate(); 已就绪key链表

createSharedObjects(); 初始化共享对象,主要是设置redis.c里的全局对象structsharedObjectsStruct shared的属性赋初值。Redis出于性能的考虑，把一些server执行过程中经常用到的对象构造出来，放到内存中，使用到的时候直接从这里取，避免临时申请的开销。比如“+OK”反馈，错误反馈，1~10000的整数对象等等。

adjustOpenFilesLimit(); 获取最大打开文件数目，根据这个打开文件最大数，适当调整同时支持的客户端数server.maxclients 变量。

server.el = aeCreateEventLoop(server.maxclients+1024); 创建事件循环aeCreateEventLoop ()函数先创建一个结构aeEventLoop的指针变量eventLoop，并从堆中申请内存，把这个结构的一些成员赋初始值， 创建aeApiState结构指针变量state，并创建epool，并把fd赋给state->epfd，最后把state赋给eventLoop->apidata。

server.db = zmalloc(sizeof(redisDb)\*server.dbnum); 根据服务器配置分配数据库内存

if (server.port != 0)

{

server.ipfd = anetTcpServer(server.neterr,server.port,server.bindaddr);

if (server.ipfd == ANET\_ERR)

{

redisLog(REDIS\_WARNING, "Openingport %d: %s", server.port, server.neterr);

exit(1);

}

}

如果server.port设置了值，根据srver配置，创建SOCK\_STREAM套接字，并监听server.port这个端口, 如果创建失败，则结束服务器程序。

if (server.unixsocket != NULL)

{

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

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

server.unixsocketperm);

if (server.sofd == ANET\_ERR)

{

redisLog(REDIS\_WARNING, "Openingsocket: %s", server.neterr);

exit(1);

}

}

if (server.ipfd < 0 && server.sofd < 0)

{

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

exit(1);

}

建立unix socket(本地无名套接字)，流程同上。如果配置两种连接方式都没设置，服务器程序也会退出。

// 根据配置初始化数据库

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

{

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

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

过期key hash，存储会过期的key以及相应过期时间，即一对(key,time\_t)的kv组合，也是一个。

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

// --阻塞键hash

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

// --收到push命令的阻塞键hash

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

// --被WATCH命令监视的键hash

server.db[j].id = j;

// --数据库ID，数据库的id是从0开始递增。

}

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

订阅频道的hash，用来记录所有订阅的client。

server.pubsub\_patterns= listCreate();

初始化订阅-发布模式列表

server.pubsub\_patterns->free= freePubsubPattern;

发布-订阅函数的释放函数指针

server.pubsub\_patterns->match= listMatchPubsubPattern;

匹配函数

server.cronloops= 0; // --CRON执行计数

server.rdb\_child\_pid= -1; // --BGSAVE 执行指示变量

server.aof\_child\_pid= -1; // --BGREWRITEAOF 执行指示变量

aofRewriteBufferReset(); // --初始化 AOF 重写缓存

server.aof\_buf= sdsempty();

server.lastsave= time(NULL); // --最后一次成功保存的时间

server.rdb\_save\_time\_last= -1; // --结束 SAVE 的时间

server.rdb\_save\_time\_start= -1; // --开始 SAVE 的时间

server.dirty =0; // --用来后续计算server维护的数据是否有更新，如果有，需要记录aof和通知replication.

//--接下来,是一些统计变量

server.stat\_numcommands = 0; // --命令数

server.stat\_numconnections = 0; // --连接数

server.stat\_expiredkeys = 0; // --过期键

server.stat\_evictedkeys = 0;

server.stat\_starttime = time(NULL);

server.stat\_keyspace\_misses = 0;

server.stat\_keyspace\_hits = 0;

server.stat\_peak\_memory = 0;

server.stat\_fork\_time = 0;

server.stat\_rejected\_conn = 0;

memset(server.ops\_sec\_samples,0,sizeof(server.ops\_sec\_samples));

server.ops\_sec\_idx = 0;

server.ops\_sec\_last\_sample\_time = mstime();

server.ops\_sec\_last\_sample\_ops = 0;

server.unixtime = time(NULL); // 用于时间值保留，其精度为s，类似于一个缓存。redis的代码中有很多需要时间值的地方，只要其精度要求不是很高，server.unixtime又有合理的机制进行更新，就可以避免在每次需要时间值的时候执行昂贵的time系统调用。

server.lastbgsave\_status = REDIS\_OK;

server.stop\_writes\_on\_bgsave\_err = 1;

aeCreateTimeEvent(server.el, 1, serverCron, NULL, NULL);

【非常重要】创建一个ae定时事件，加到server.el->timeEventHead的头部，并将serverCron设置为这个定时事件的处理函数。这是redis的核心循环，该过程是serverCron，每秒调用次数由一个叫REDIS\_HZ的宏决定，默认是每10微秒超时，即每10微秒该ae时间事件处理过程serverCron会被过期调用。serverCron详解见下文。

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

acceptTcpHandler,NULL) == AE\_ERR)

{

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

}

创建ae文件事件，对redis的TCP或者unix socket端口进行监听，使用相应的处理函数注册。每次得到clients连接后，都会触发ae文件事件，异步接收命令。如果server.ipfd(tcp/ip文件描述,类似于windows下的handle)有值,也即tcp/ip套接字创建成功的情况下, 创建网络事件。在有连接请求进来后，acceptTcpHandler将会被调用，该函数调用accept接收连接，然后用accept函数返回的文件描述符创建一个client桩（一个redisClient对象），在server端代表连接进来的真正client。在创建client桩的时候，会将返回的这个描述符同样添加进事件监控列表，监控READABLE事件，事件发生代表着客户端发送数据过来，此时调readQueryFromClient接收客户端的query。

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);

}

}

如果server设置了aof模式做持久化，将会打开或创建对应的打开或创建 AOF 文件，保存相关的描述符。

设置内存限制。

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;

}

32位系统，如果没有显式内存设置，默认设置为3G，并把maxmemory\_policy设置为REDIS\_MAXMEMORY\_NO\_EVICTION，在程序达到最大内存限制后，拒绝后续会增大内存使用的客户端执行的命令。

if (server.cluster\_enabled)

{

clusterInit();

}

如果集群模式已打开，那么初始化集群。对于clusterInit()函数的细节分析，详见cluster.c的剖析。

scriptingInit();

初始化脚本系统。Redis的脚本系统用的是lua语言。在scriptingInit()函数主要是初始化lua环境，对于在C/C++下使用过lua脚本的人来说，非常熟悉，具体细节将在scripting.c文件分析时展开。

slowlogInit();

初始化slowlog。slowlog是redis提供的进行query分析的工具。它将执行时间长的命令统一以list形式保存在内存之中，使用者可以通过slowlog命令查看这些慢query，从而分析系统瓶颈。

bioInit();

初始化后台IO服务。

3.2.4 serverCron()函数

前面介绍过，这个函数非常重要。serverCron是一个主循环事件的回调处理函数，redis每个循环都会执行该函数。

REDIS\_NOTUSED(eventLoop);

REDIS\_NOTUSED(id);

REDIS\_NOTUSED(clientData);

REDIS\_NOTUSED这个宏很有意思，#define REDIS\_NOTUSED(V) ((void) V)

目的是为了去掉编译器对未使用的局部变量的警告。

if(server.watchdog\_period)

{

watchdogScheduleSignal(server.watchdog\_period);

}

如果配置设置了server.watchdog\_period，则启动看门狗信号：如果系统在一定时间内不运行到这里，看门狗程序会触发一个SIGALRM信号到信号处理器。

structitimerval it;

it.it\_value.tv\_sec= period/1000;// --如果period为0，那么停止计时器

it.it\_value.tv\_usec= (period%1000)\*1000;

it.it\_interval.tv\_sec= 0; // --不要自动重启

it.it\_interval.tv\_usec= 0;

setitimer(ITIMER\_REAL,&it, NULL);// --以系统真实的时间来计算，它送出SIGALRM信号。

server.unixtime= time(NULL);

// --将UNIX 时间保存在服务器状态中，减少对time(NULL) 的调用，获取一个全局变量快于一个调用time(NULL)。

server.mstime= mstime();

这一句在2.6之前的版本是没有的，记录当前时间的毫秒数(是毫秒，不是微秒)。

run\_with\_period(100)trackOperationsPerSecond();

这个run\_with\_period是一个宏，看一下宏定义#define run\_with\_period(\_ms\_) if ((\_ms\_<= 1000/server.hz)|| !(server.cronloops%((\_ms\_)/(1000/server.hz))))，

如果\_ms\_<= 1000/server.hz 或者 server.cronloops不能整除\_ms\_/(1000/server.hz)的结果，就执行trackOperationsPerSecond()。

这句很不好理解，但是大致可以这么理解，按照一定周期执行trackOperationsPerSecond(), 而这个周期与server.hz有关系。而trackOperationsPerSecond()则是用来记录过去一秒命令执行情况。

updateLRUClock();

更新服务器的 LRU 时间。函数只有一句代码

server.lruclock = (server.unixtime/REDIS\_LRU\_CLOCK\_RESOLUTION)& REDIS\_LRU\_CLOCK\_MAX;

后续在执行lru淘汰策略时，作为比较的基准值。redis默认的时间精度是10s(#define REDIS\_LRU\_CLOCK\_RESOLUTION 10)，保存lruclock的变量共有22bit。换算成总的时间为1.5year（每隔1.5年循环一次）。不知为何在最初设计的时候，为lruclock只给了22bit的空间。还是读一下原始的注释吧:

*Wehave just 22 bits per object for LRU information.So we use an (eventually wrapping) LRUclock with 10 seconds resolution.2^22 bits with 10 seconds resolution ismore or less 1.5 years.Note that even if this will wrap after1.5 years it's not a problem,everything will still work but just someobject will appear younger to Redis. But for this to happen a givenobject should never be touchedfor 1.5 years.Note that you can change the resolutionaltering the REDIS\_LRU\_CLOCK\_RESOLUTION define.*

if (zmalloc\_used\_memory()> server.stat\_peak\_memory) { server.stat\_peak\_memory = zmalloc\_used\_memory();}

记录服务器启动以来的内存最高峰

size\_tzmalloc\_used\_memory(void)

{

size\_t um;

if (zmalloc\_thread\_safe)

{

#ifdefHAVE\_ATOMIC

um = \_\_sync\_add\_and\_fetch(&used\_memory,0);

#else

pthread\_mutex\_lock(&used\_memory\_mutex);

um = used\_memory;

pthread\_mutex\_unlock(&used\_memory\_mutex);

#endif

}

else

{

um = used\_memory;

}

return um;

}

if (server.shutdown\_asap)

{

if (prepareForShutdown(0) == REDIS\_OK)

{

exit(0);

}

redisLog(REDIS\_WARNING,"SIGTERMreceived but

errors trying to shut down the server,

check the logs for moreinformation");

server.shutdown\_asap = 0;

}

redis在SIG\_TERM信号的处理函数中并没有立即终止进程的执行，而是选择了标记shutdown\_asap flag，然后在serverCron中通过执行prepareForShutdown函数安全退出。prepareForShutdown()函数主要是处理了rdb、aof记录文件退出的情况，最后保存了一次rdb文件，关闭了相关的文件描述符以及删除了保存pid的文件（server.pidfile），详细见下文。

run\_with\_period(5000)

{

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

{

long long size, used,vkeys;

size = dictSlots(server.db[j].dict);

used = dictSize(server.db[j].dict);

vkeys = dictSize(server.db[j].expires);

if (used || vkeys)

{

redisLog(REDIS\_VERBOSE,"DB%d: %lld keys (%lld volatile)

in %lld slots HT.", j,used,vkeys,size);

}

}

}

每5秒输出一次redis每个数据库的统计信息: 使用的key数目、设置过期的key数目、以及当前的hashtabale的槽位数。

if (!server.sentinel\_mode)

{

run\_with\_period(5000)

{

redisLog(REDIS\_VERBOSE,

"%dclients connected (%d slaves), %zu bytes in use",

listLength(server.clients)-listLength(server.slaves),

listLength(server.slaves),

zmalloc\_used\_memory());

}

}

在非后台模式下，每5秒输出一次，client数目，slaves数目，以及总体的内存使用情况

clientsCron();

clientCron()的代码如下:

次调用clientCron例程，这是一个对server.clients列表进行处理的过程。再每次执行clientCron时，会对 server.clients进行迭代，并且保证 1/(REDIS\_HZ\*10) ofclients per call。也就是每次执行clientCron，如果clients过多，clientCron不会遍历所有clients，而是遍历一部分 clients，但是保证每个clients都会在一定时间内得到处理。处理过程主要是检测client连接是否idle超时，或者block超时，然后 会调解每个client的缓冲区大小。

int numclients = listLength(server.clients);

int iterations = numclients/(server.hz\*10);

if (iterations < 50) iterations= (numclients < 50) ? numclients : 50;

while(listLength(server.clients)&& iterations--)

{

redisClient \*c;

listNode \*head;

listRotate(server.clients);

将当前处理的客户端调到表头，这样在要删除客户端时，复杂度就是O(1)而不是O(N)

head = listFirst(server.clients);

c = listNodeValue(head);

检查客户端是否超时，如果是的话删除它的连接如果客户端正因 BLPOP/BRPOP/BLPOPRPUSH 阻塞，那么检查阻塞是否超时，是的话就退出阻塞状态

if (clientsCronHandleTimeout(c))continue;

// --释放客户端查询缓存多余的空间

if (clientsCronResizeQueryBuffer(c))continue;

}

检查连接是否超时，以及清理多余的查询缓存,最多处理50个迭代，

databasesCron();

处理数据的后台操作, databasesCron()详解见下文。

if (server.rdb\_child\_pid == -1 && server.aof\_child\_pid == -1&& server.aof\_rewrite\_scheduled)

{

rewriteAppendOnlyFileBackground();

}

如果用户在此期间，执行BGREWRITEAOF 命令的话，在后台执行AOF重写。

rewriteAppendOnlyFileBackground() 这个函数详细分析见aof.c

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);

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

{

backgroundSaveDoneHandler(exitcode,bysignal);

}

elseif (pid == server.aof\_child\_pid)

{

backgroundRewriteDoneHandler(exitcode,bysignal);

}

else

{

redisLog(REDIS\_WARNING,"Warning,

detected child with unmatched pid: %ld",(long)pid);

}

updateDictResizePolicy();

}

}

如果有server.rdb\_child\_pid或server.aof\_child\_pid配置：调用wait3获取子进程状态。此wait3为非阻塞（设置了WNOHANG flag）。注意：APUE2在进程控制章节其实挺不提倡用wait3和wait4接口的，不过redis的作者貌似对这个情有独钟。如果后台进程刚好退 出，调用backgroundSaveDoneHandler或backgroundRewriteDoneHandler进行必要的善后工作，并更新 dict resize policy（如果已经没有后台进程了，就可以允许执行resize操作了）

否则，如果没有后台的save rdb操作及rewrite操作：首先，根据saveparams规定的rdb save策略，如果满足条件，执行后台rdbSave操作；其次，根据aofrewrite策略，如果当前aof文件增长的规模，要求触发rewrite操作，则执行后台的rewrite操作。

else

{

/\* If there is not abackground saving/rewrite in progress check if we have to save/rewritenow \*/

for (j = 0; j < server.saveparamslen;j++)

{

struct saveparam \*sp= server.saveparams+j;

/\* Save if we reachedthe given amount of changes,

\* the given amount ofseconds, and if the latest bgsave was

\* successful or if, incase of an error, at least

\*REDIS\_BGSAVE\_RETRY\_DELAY seconds already elapsed. \*/

if ( server.dirty >=sp->changes && server.unixtime-server.lastsave >sp->seconds &&

(server.unixtime-server.lastbgsave\_try> REDIS\_BGSAVE\_RETRY\_DELAY || server.lastbgsave\_status == REDIS\_OK ) )

{

redisLog(REDIS\_NOTICE,"%dchanges in %d seconds. Saving...",

sp->changes, sp->seconds);

rdbSaveBackground(server.rdb\_filename);

break;

}

}

if ( server.rdb\_child\_pid== -1 &&

server.aof\_child\_pid== -1 &&

server.aof\_rewrite\_perc&&

server.aof\_current\_size> server.aof\_rewrite\_min\_size)

{

long long base = server.aof\_rewrite\_base\_size?

server.aof\_rewrite\_base\_size: 1;

long long growth =(server.aof\_current\_size\*100/base) - 100;

if (growth >= server.aof\_rewrite\_perc)

{

redisLog(REDIS\_NOTICE,"Startingautomatic rewriting of AOF on %lld%% growth",growth); rewriteAppendOnlyFileBackground();

}

}

}

if (server.aof\_flush\_postponed\_start) flushAppendOnlyFile(0);

如果在执行延时保存aof文件(从server.aof\_buf缓冲flush到磁盘)，则在此进行延时flush操作。详细分析见后面章节。

/\* Close clients that need to be closed asynchronous \*/

freeClientsInAsyncFreeQueue();

函数内部代码为

while (listLength(server.clients\_to\_close))

{

listNode \*ln = listFirst(server.clients\_to\_close);

redisClient \*c = listNodeValue(ln);

c->flags &= ~REDIS\_CLOSE\_ASAP;

freeClient(c);

listDelNode(server.clients\_to\_close,ln);

}

run\_with\_period(1000) replicationCron();

每秒执行一次replicationCron(),这个函数用来master重连和传输失败检查

run\_with\_period(100)

{

if (server.sentinel\_mode)sentinelTimer();

}

如果再监视模式下，则每100ms运行一次监视Time

server.cronloops++;

统计服务器循环计数

return 1000/server.hz;

返回 1000/server.hz，意味着server将会在100ms后重新调用这个函数

3.2.5 databasesCron()函数

databasesCron()函数主要用于在后台做一些需要逐渐实现的操作，比如key的过期，哈希重置等操作。

if (server.active\_expire\_enabled && server.masterhost == NULL)

{

activeExpireCycle(ACTIVE\_EXPIRE\_CYCLE\_SLOW);

}

activeExpireCycle()这个函数主要是用于过期值的处理， 详情见本文后面的分析。这个函数只有在active\_expire\_enabled这个配置为true，而且本库是master的的情况下执行，slave会等候由master传递的DEL消息，保证master-slave在过期值处理上的一致。redis是随机抽取过期值的，所以master和slave可能抽取不同的值，故通过DEL消息实现同步，同时这种expire机制也是不可靠的expire，即key超时后有可能不会被删除。

if (server.rdb\_child\_pid == -1 && server.aof\_child\_pid == -1) {…}

这句代码经常见，如果没有其进程进行DB存储的时候，才能进行hash重置，因为这会引起很多的写实复制内存页。

static unsigned int resize\_db = 0;

static unsigned int rehash\_db = 0;

unsigned int dbs\_per\_call = REDIS\_DBCRON\_DBS\_PER\_CALL;

unsigned int j;

if (dbs\_per\_call > server.dbnum)

{

dbs\_per\_call = server.dbnum;

}

不能超过数据库数目，默认为16。

for (j = 0; j < dbs\_per\_call; j++)

{

tryResizeHashTables(resize\_db % server.dbnum);

resize\_db++;

}

遍历每个数据库都进行Resize操作，tryResizeHashTables()的全部代码如下:

if (htNeedsResize(server.db[dbid].dict))

{

dictResize(server.db[dbid].dict);

}

if (htNeedsResize(server.db[dbid].expires))

{

dictResize(server.db[dbid].expires);

}

函数分别检查检查key空间和过期key空间。

htNeedsResize()的内部实现

long long size, used;

size = dictSlots(dict);

used = dictSize(dict);

return (size && used && size > DICT\_HT\_INITIAL\_SIZE && (used\*100/size < REDIS\_HT\_MINFILL));

size是两个字典结构的桶的数目总和，而used是两个字典结构已经使用的桶总和。

如果满足used\*100/size < REDIS\_HT\_MINFILL，就进行dictResize()。dictResize()内部实现见dict.c。

if (server.activerehashing

{

for (j = 0; j < dbs\_per\_call; j++)

{

int work\_done = incrementallyRehash(rehash\_db % server.dbnum);

rehash\_db++;

if (work\_done) {break;}

}

}

遍历每个数据库，进行rehash操作，每次调用只rehash一个。incrementallyRehash()的内部实现如下：

if (dictIsRehashing(server.db[dbid].dict))

{

dictRehashMilliseconds(server.db[dbid].dict,1);

return 1;

}

if (dictIsRehashing(server.db[dbid].expires))

{

dictRehashMilliseconds(server.db[dbid].expires,1);

return 1;

}

return 0;

dictIsRehashing ()的宏定义,dictIsRehashing(ht) ((ht)->rehashidx != -1)。dictRehashMilliseconds()细节见dict.c。

3.2.6 activeExpireCycle()函数

★★★这个函数也非常重要，牵涉到了过期主键淘汰(清除)机制，我将它比作redis的GC。Redis支持主键可以有过期时间，这种机制方便用户使用，比如一些缓存策略，过期的主键会占用大量内存资源，这就必要要求redis有及时从内存中清理失效主键。

Redis的主键淘汰有两种策略：

被动策略（passive way），在访问主键的时候，如果发现主键已经过期，就清除掉。

主动策略（active way），周期性地从过期主键空间中淘汰一部分失效的主键。

本函数则代表着redis过期主键淘汰机制的主动策略，本函数是由时间事件来驱动执行。Redis的淘汰算法是灵活适应的。如果有很少的过期key，就用很少的CPU循环；否则就会占用更多CPU循环用于淘汰计算，以防止很多可删除键空间占用大量内存。每次迭代DB数目不能超过REDIS\_DBCRON\_DBS\_PER\_CALL。

淘汰原理：遍历服务器每个数据库expires字典，从中尝试着随机抽样不超过ACTIVE\_EXPIRE\_CYCLE\_LOOKUPS\_PER\_LOOP（20）个限时主键，检查其是否过期，如果过期则删除。如果过期的限时主键个数占本次抽样个数的比例超过1/4，Redis 会认为当前数据库中的失效主键依然很多，所以它会继续进行下一轮的随机抽样和删除，直到刚才的比例低于25%才停止对当前数据库的处理，转向下一个数据 库。这里我们需要注意的是，activeExpireCycle 函数不会试图一次性处理Redis中的所有数据库，而是最多只处理 REDIS\_DBCRON\_DBS\_PER\_CALL（默认值为16），此外 activeExpireCycle 函数还有处理时间上的限制，不是想执行多久就执行多久，凡此种种都只有一个目的，那就是避免失效主键删除占用过多的CPU资源

static unsigned int current\_db = 0; /\* Last DB tested. \*/

static int timelimit\_exit = 0; /\* Time limit hit in previous call? \*/

static long long last\_fast\_cycle = 0; /\* When last fast cycle ran. \*/

unsigned int j, iteration = 0;

unsigned int dbs\_per\_call = REDIS\_DBCRON\_DBS\_PER\_CALL;// --每次最多处理DB个数

long long start = ustime(), timelimit;

注意一下这几个静态变量，current\_db,用来保存每次函数调用处理的最后一个Redis数据库的编号，因为activeExpireCycle不是一次性遍历所有数据库，而是渐进式的，下次执行本函数时，从current\_db继续开始处理。timelimit\_exit保存上一次调用本函数时，是否到达指定期限(这个期限下文有详细说明)last\_fast\_cycle用于保存上一次快速循环的执行时间。

if (type == ACTIVE\_EXPIRE\_CYCLE\_FAST)

{

if (!timelimit\_exit) return;

if (start < last\_fast\_cycle + ACTIVE\_EXPIRE\_CYCLE\_FAST\_DURATION\*2 ) return;

last\_fast\_cycle = start;

}

如果传入参数是ACTIVE\_EXPIRE\_CYCLE\_FAST，则在快速循环执行期间内不再重复执行快速循环，且在距离上次快速循环执行时间小于2倍EXPIRE\_FAST\_CYCLE\_DURATION微秒 则不执行。如果参数是ACTIVE\_EXPIRE\_CYCLE\_SLOW，则执行普通过期循环。

if (dbs\_per\_call > server.dbnum || timelimit\_exit){ dbs\_per\_call = server.dbnum; }

通常每次迭代要通常每次迭代要检测REDIS\_DBCRON\_DBS\_PER\_CALL(16)个db，有两种例外情况：1）实际数据库数量小于REDIS\_DBCRON\_DBS\_PER\_CAL；2)上一次调用activeExpireCycle()函数到达指定期限，这种情况说明数据库的过期主键很多，占用大量内存，所以需要处理全部数据库。

timelimit = 1000000\*ACTIVE\_EXPIRE\_CYCLE\_SLOW\_TIME\_PERC/server.hz/100;

timelimit\_exit = 0;

if (timelimit <= 0) timelimit = 1;

if (type == ACTIVE\_EXPIRE\_CYCLE\_FAST)

{ timelimit = ACTIVE\_EXPIRE\_CYCLE\_FAST\_DURATION;}

每次迭代执行activeExpireCycle()的指定期限(单位为微秒)，普通循环(参数是ACTIVE\_EXPIRE\_CYCLE\_SLOW)的期限是REDIS\_HZ的百分比即(1000000 \* (REDIS\_EXPIRELOOKUPS\_TIME\_PERC / 100)) / server.hz；而快速循环的指定期限是ACTIVE\_EXPIRE\_CYCLE\_FAST\_DURATION(1000微秒)。

Timelimit按照默认值计算为25000微秒即25毫秒

for (j = 0; j < dbs\_per\_call; j++) {…}

不用细讲，循环遍历固定数目的DB。看循环里面的代码：

int expired;

redisDb \*db = server.db+(current\_db % server.dbnum);

current\_db++;

此处立刻就将current\_db加一，这样可以保证即使这次无法在时间限制内删除完所有当前数据库中的失效主键，下一次调用activeExpireCycle一样会从下一个数据库开始处理，从而保证每个数据库都有被处理的机会。

do{ … }while (expired > ACTIVE\_EXPIRE\_CYCLE\_LOOKUPS\_PER\_LOOP/4);

这个循环处理当前数据库中的失效主键，直到过期的限时主键个数占本次抽样个数(20)的比例小于等于1/4，则不重复循环。看do{}里的代码:

unsigned long num, slots;

long long now, ttl\_sum;

ttl\_samples;

if ((num = dictSize(db->expires)) == 0)

{

db->avg\_ttl = 0;

break;

}

如果expires字典表大小为0，说明该数据库中没有设置失效时间的主键，直接检查下一数据库。

slots = dictSlots(db->expires); // --记录当前数据库限时主键hash的桶数

now = mstime(); // --记录当前时间

if (num && slots > DICT\_HT\_INITIAL\_SIZE &&(num\*100/slots < 1)){ break;}

如果expires字典表不为空，但是其填充率不足1%，那么随机选择主键进行检查的代价会很高，所以这里直接检查下一数据库。

expired = 0;

ttl\_sum = 0;

ttl\_samples = 0;

if (num > ACTIVE\_EXPIRE\_CYCLE\_LOOKUPS\_PER\_LOOP)

{

num = ACTIVE\_EXPIRE\_CYCLE\_LOOKUPS\_PER\_LOOP;

}

如果expires字典表中的entry个数大于ACTIVE\_EXPIRE\_CYCLE\_LOOKUPS\_PER\_LOOP，抽样次数为ACTIVE\_EXPIRE\_CYCLE\_LOOKUPS\_PER\_LOOP。

while (num--)

{

dictEntry \*de;

long long ttl;

if ((de = dictGetRandomKey(db->expires)) == NULL)

{

break;

}

// --随机获取一个设置了失效时间的主键

ttl = dictGetSignedIntegerVal(de)-now;

if ( activeExpireCycleTryExpire(db,de,now) )// --检查是否过期

{

expired++; // --如果过期，则增加统计

}

if (ttl < 0) ttl = 0;

ttl\_sum += ttl;

ttl\_samples++;

}

下面来看看activeExpireCycleTryExpire()内部实现：

long long t = dictGetSignedIntegerVal(de);

if (now > t)

{

sds key = dictGetKey(de);

robj \*keyobj = createStringObject(key,sdslen(key));

propagateExpire(db,keyobj);

通知AOF文件和客户端都删除该主键，详见db.c。

dbDelete(db,keyobj); // --删除此主键

decrRefCount(keyobj);

server.stat\_expiredkeys++;

return 1;

} else

{

return 0;

}

if (ttl\_samples)

{

long long avg\_ttl = ttl\_sum/ttl\_samples;

if (db->avg\_ttl == 0) db->avg\_ttl = avg\_ttl;

db->avg\_ttl = (db->avg\_ttl+avg\_ttl)/2;

}

给DB更新平均TTL状态

iteration++;// --每次抽样后增加iteration

if ((iteration & 0xf) == 0 && (ustime()-start) > timelimit ){ timelimit\_exit = 1;}

if (timelimit\_exit) { return; }

iteration & 0xf) == 0这句主要是检测是否到了16次抽样，每经过16次抽样就判断一次执行时间是否已经达到指定时间限制，如果已达到时间限制，则把timelimit\_exit这个静态变量设置为1，并退出函数。我很奇怪的是，这句if (timelimit\_exit)完全可以省略，直接写成if ((iteration & 0xf) == 0 && (ustime()-start) > timelimit ){ timelimit\_exit = 1; return; }

通过以上对 Redis 主键失效机制的介绍，我们知道虽然 Redis 会定期地检查设置了失效时间的主键并删除已经失效的主键，但是通过对每次处理数据库个数的限制、activeExpireCycle 函数在一秒钟内执行次数的限制、分配给 activeExpireCycle 函数CPU时间的限制、继续删除主键的失效主键数百分比的限制，Redis 已经大大降低了主键失效机制对系统整体性能的影响，但是如果在实际应用中出现大量主键在短时间内同时失效的情况还是会使得系统的响应能力降低，所以这种情 况无疑应该避免。

* 1. 事件循环

3.3.1 事件循环结构

typedef struct aeEventLoop

{

int maxfd; // --当前注册的fd最大值，初始值是-1

int setsize; // --轮循的最大文件句柄+1

long long timeEventNextId; // --下一个定时事件编号

time\_t lastTime; // --上次轮循的时间

aeFileEvent \*events; // --文件事件数组,数组大小为setsize

aeFiredEvent \*fired; // --已经轮循到的事件，可以说是epoll之后暂存在这个结构里的

aeTimeEvent \*timeEventHead; // --时间事件链表头

int stop;// --循环是否停止

void \*apidata;// --aeApiState，这是epoll/select/kqueue等用到的结构

aeBeforeSleepProc \*beforesleep; // --事件循环中每次处理事件之前调用

} aeEventLoop;

typedef struct aeFileEvent

{

int mask; // --AE\_(READABLE|WRITABLE)之一

aeFileProc \*rfileProc; // --

aeFileProc \*wfileProc;// --

void \*clientData;// --

}aeFileEvent;

typedef struct aeTimeEvent

{

long long id; // --时间事件ID

long when\_sec; // --秒

long when\_ms; // --微秒

aeTimeProc \*timeProc;// --处理函数

aeEventFinalizerProc \*finalizerProc;// --用于销毁lientData

void \*clientData;// --

struct aeTimeEvent \*next;// --next指针，为了形成事件链

}aeTimeEvent;

3.3.2 aeCreateEventLoop()函数

这个函数很重要，参数setsize是指eventloop中描述符数组的大小，也即该事件循环可以有多少个fd。这个函数的内部实现比较简单，只需关注两句：

if (aeApiCreate(eventLoop) == -1) goto err; aeApiCreate封装了不同多路复用的实现锁。如linux的epoll，sun的export,FreeBSD和Mac osx的queue，还有select。创建aeApiState，然后赋值给aeEventLoop 结构的apidata。

**for (i = 0; i < setsize; i++){ eventLoop->events[i].mask = AE\_NONE; }**

初始化所有的文件描述符的mask设置为AE\_NONE。

3.3.2 aeCreateEventLoop()函数

**if (fd >= eventLoop->setsize)**

**{**

**errno = ERANGE;**

**return AE\_ERR;**

**}**

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

**if (aeApiAddEvent(eventLoop, fd, mask) == -1){ return AE\_ERR; }**

**fe->mask |= mask;**

**if (mask & AE\_READABLE) fe->rfileProc = proc;**

**if (mask & AE\_WRITABLE) fe->wfileProc = proc;**

**fe->clientData = clientData;**

**if (fd > eventLoop->maxfd)**

**{**

**eventLoop->maxfd = fd;**

**}**

**return AE\_OK;**

3.2.4 aeCreateTiemEvent()函数

**long long id = eventLoop->timeEventNextId++;**

**aeTimeEvent \*te;**

**te = zmalloc(sizeof(\*te));**

**if (te == NULL) return AE\_ERR;**

**te->id = id;**

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

**te->timeProc = proc;**

**te->finalizerProc = finalizerProc;**

**te->clientData = clientData;**

**te->next = eventLoop->timeEventHead;**

**eventLoop->timeEventHead = te;**

**return id;**

3.2.5 aeProcessEvents()函数

1. 首先判断处理何种事件类型， 是时间类型还是文件类型又或者全部。任意单个类型都比较好理解。当处理全部类型时，先处理文件事件，再处理时间事件，那么阻塞超时设为多久呢？她会先获取注册时间事件中最快要触发的事件，这时候由于时间链表不是有序的，所以会遍历搜索，得到最小时间。如果没有时间事件，那么不会阻塞在获取文件事件触发上，立刻返回。
2. 获取阻塞时间后，就开始文件事件的触发获取。得到所有触发事件，然后遍历文件事件触发数组(eventLoop->fired)，得到fd，然后获取对应的文件事件，这里的fired已经看出，它只是个索引。调用对应的回调函数，结束文件事件的处理。
3. 最后处理到期的时间事件。遍历一次注册时间事件链，获得到期事件，处理之，这时候就有个问题，照常理应该继续检索是否有到期的事件(到期时间相同)，但是由于可能会改变eventloop的时间事件链表，所以重新从头开始检索。这是ae网络库最大的问题。在这点可以做比较容易的优化。如进行有序时间链表，虽然导致插入时间为O(n)。又比如设为跳表，可以减小插入时间为O(log(n))，但是付出更多的空间。

只有短短400行代码，所以整个事件库可以比较容易说清楚，没有特别复杂，晦涩的实现，非常适合将ae放在小项目中，没有过多的优化，使得调试比较容易。而且性能也非常不错。

**int processed = 0, numevents;**

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

如果flags标志既不是AE\_TIME\_EVENTS也不是flags & AE\_FILE\_EVENTS，则return出去。

**if (eventLoop->maxfd != -1 || ((flags & AE\_TIME\_EVENTS) && !(flags & AE\_DONT\_WAIT)) )**

**{ …}**

如果eventLoop->maxfd已经有注册fd或者 AE\_TIME\_EVENTS 状态被打开，且没有打开 AE\_DONT\_WAIT情况下进入if代码块。下面看代码块内容：

**int j;**

**aeTimeEvent \*shortest = NULL;**

**struct timeval tv, \*tvp;**

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

**{**

**shortest = aeSearchNearestTimer(eventLoop);**

**}**

如果时间事件存在的话那么根据最近可执行时间事件和现在时间的时间差来决定文件事件的阻塞时间。我们来看看aeSearchNearestTimer()的代码:

**aeTimeEvent \*te = eventLoop->timeEventHead;**

**aeTimeEvent \*nearest = NULL;**

**while(te)**

**{**

**if (!nearest || te->when\_sec < nearest->when\_sec ||**

**(te->when\_sec == nearest->when\_sec && te->when\_ms < nearest->when\_ms ) )**

**{ nearest = te; }**

**te = te->next;**

**}**

**return nearest;**

如果时间事件的链表头存在，则遍历时间事件链表，找到离现在最近的时间事件。

这种办法是比较低效的，遍历需要O(N)的时间复杂度。看一下作者在这个函数上方的注释:

*Possible optimizations (not needed by Redis so far, but...):*

*1) Insert the event in order, so that the nearest is just the head.*

*Much better but still insertion or deletion of timers is O(N).*

*2) Use a skiplist to have this operation as O(1) and insertion as O(log(N)).*

作者给出两种优化方案：一种是利用有序链表，插入的复杂度是O(N)，但是查找是O(1)；另一种是利用跳跃表，插入是O(log(n)),查找是O(1)。

从效率考虑，优化还有很有必要，尤其是第二种方法。

**if (shortest)**

**{**

**long now\_sec, now\_ms;**

**aeGetTime(&now\_sec, &now\_ms); // --取得现在的时间**

**tvp = &tv;**

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

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

**{**

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

**tvp->tv\_sec --;**

**} else**

**{**

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

**}**

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

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

**}**

**else**

**{**

**if (flags & AE\_DONT\_WAIT) { tv.tv\_sec = tv.tv\_usec = 0; tvp = &tv; }**

**else { tvp = NULL; }**

**}**

如果能找到离现在最近的时间事件,那么计算与现在时间的时间差；如果没找到则检查flags是否包含AE\_DONT\_WAIT状态，如果是则把tvp结构都清零否则让tvp等于NULL。

**numevents = aeApiPoll(eventLoop, tvp);**

当aeApiPoll返回的时候，有两种可能：1，eventLoop所关注的文件事件发生了，2，eventLoop中离现在最近的时间事件触发了；根据aeApiPoll所返回的命中的文件事件数目，我们可以查eventLoop->fired，取出命中事件的详细信息，然后调用rfileProc或wfileProc做处理；最后，调用processTimeEvents处理已经触发了的定时器事件，然后返回所有已经处理过的事件数目。

**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;**

**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++;**

**}**

至此，if{}执行完。

**if (flags & AE\_TIME\_EVENTS){ processed += processTimeEvents(eventLoop); }**

检查执行时间事件，代码详细见下文。

**return processed;**

返回处理的文件事件或时间事件数量。

*我们的事例中只有一个事件那就是通过调用aeCreateTimeEvent函数建立的。*

*记住，如果将超时时间设定为1毫秒，那么通过aeCreateTimeEvent函数建立的定时器事件很可能会被忽略。和事件循环相关连的tvp结构体被传递给ae\_epoll.c文件中的aeApiPoll函数。aeApiPoll函数在epoll描述符上进行epoll\_wait，同时想下边描述的方式激发eventLoop->fired表。*

*fd：根据掩码值此时准备好读/写的描述符。*

*mask：读/写操作可以在对应的描述符上进行。*

*aeApiPoll函数返回准备好的文件描述符个数。现在我们总结一下，当有客户端请求到来时候，aeApiPoll函数会发现并且使用一个正在监听的描述符实体和AE——READABLE掩码激发eventLoop->fired表格。*

*现在aeProcessEvents函数调用redis.c文件中已经被注册为回调函数的acceptHandler函数。acceptHandler函数在正在监听的描述符上执行accpet操作，返回一个已经和客户端建立连接的描述符。createClient函数通过调用aeCreateFileEvent函数向已经连接的描述符上添加一个文件事件。如下所示：*

if (aeCreateFileEvent(server.el, c->fd, AE\_READABLE,

readQueryFromClient, c) == AE\_ERR)

{

freeClient(c);

return NULL;

}

代码中的c指的是redisClient结构类型的变量，c->fd就是已经建立连接的描述符。然后aeProcessEvent函数调用processTimeEvents函数。

3.3.6 **processTimeEvents** ()函数

processTimeEvents从时间事件列表开始处eventLoop->timeEventHead开始迭代。对于每一个到期的时间事件，processTimeEvents调用相应的已注册的回调函数。本例中仅有一个已注册的回调函数，也就是redis.c文件中的serverCron函数。这个函数返回毫秒数，指示这个回调函数过多长时间再次调用。这写更改可以通过调用aeAddMilliSeconds函数记录，而且会在下一次循环中处理。

* 1. IO接口

3.4.1 aeApiState 结构

typedef struct aeApiState

{

int epfd;

struct epoll\_event \*events;

} aeApiState;

3.4.2 aeApiCreate()函数

**aeApiState \*state = zmalloc(sizeof(aeApiState));**

**if (!state) return -1;**

分配aeApiState\*内存空间，如果分配失败，返回失败。

**state->events = zmalloc(sizeof(struct epoll\_event)\*eventLoop->setsize);**

**if (!state->events)**

**{**

**zfree(state);**

**return -1;**

**}**

分配aeApiState的events内存空间，如果失败，释放state空间

* 1. AOF文件

AOF的相关操作，一部分是由时间事件驱动的，具体见serverCron。backgroundRewriteDoneHandler(exitcode,bysignal); 另一部分是由文件事件驱动的，大概流程是这样：客户端都的命令通过文件事件交给acceptTcpHandler()来处理，上述函数经过解析都会针对每个命令调用调用redis.c的call()函数，而在call函数中，如果该命令涉及到写操作，那么会调用progagate()来传播写操作到AOF和slaves。在progagate中，通过调用feedAppendOnlyFile()来进行。

3.6.1 aeApiState 结构

#define AOF\_RW\_BUF\_BLOCK\_SIZE (1024\*1024\*10) /\* 10 MB per block \*/

typedef struct aofrwblock

{

unsigned long used, free;

char buf[AOF\_RW\_BUF\_BLOCK\_SIZE];

} aofrwblock;

* + 1. server中AOF相关变量

**int aof\_state;** // --aof状态 REDIS\_AOF\_(ON|OFF|WAIT\_REWRITE) \*/

**int aof\_fsync;** // --文件同步策略

**char \*aof\_filename;** // --aof文件名称

**int aof\_no\_fsync\_on\_rewrite;** // --重写期间是否同步文件

**int aof\_rewrite\_perc;** // --aof重写增长百分比

**off\_t aof\_rewrite\_min\_size;** // --aof重写最少字节数

**off\_t aof\_rewrite\_base\_size;** // --上次重写aof大小

**off\_t aof\_current\_size;** // --aof当前大小

**int aof\_rewrite\_scheduled;** // --是否在BGSAVE命令结束后马上Rewrite

**pid\_t aof\_child\_pid;** // --rewrite进程的pid

**list \*aof\_rewrite\_buf\_blocks;** // --aof rewrite时暂存修改到的缓存

**sds aof\_buf;** // --AOF缓存, 事件循环中被fsync同步到硬盘上

**int aof\_fd;** // --当前选中的AOF文件的fd

**int aof\_selected\_db;** // --AOF文件当前选中的数据库ID

**time\_t aof\_flush\_postponed\_start;**  // --UNIX time of postponed AOF flush

**time\_t aof\_last\_fsync;**  // -- aof 上次fsync()的时间(UNIX time)

**time\_t aof\_rewrite\_time\_last;**  // --上次AOF rewrite耗时

**time\_t aof\_rewrite\_time\_start;**  // -- 本次AOF rewrite开始时间

**int aof\_lastbgrewrite\_status;**  // --REDIS\_OK or REDIS\_ERR \*/

**unsigned long aof\_delayed\_fsync;**  // --AOF fsync()推迟计数

**int aof\_rewrite\_incremental\_fsync;** // --fsync incrementally while rewriting?

* + 1. feedAppendOnlyFile()函数

参数有四个，cmd是命令字符串，dictid是数据库ID，argv是参数字符, argc参数数目。

**sds buf = sdsempty();**

**robj \*tmpargv[3];**

**if ( dictid != server.aof\_selected\_db )**

**{**

**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.aof\_selected\_db = dictid;**

**}**

如果命令目标数据库不是当前数据库，则先给aof文件添加一句切换到当前目标数据库的语句，并将当前数据库切换到目标数据库。

**if (cmd->proc == expireCommand || cmd->proc == pexpireCommand || cmd->proc == expireatCommand)**

**{**

**buf = catAppendOnlyExpireAtCommand(buf,cmd,argv[1],argv[2]);**

**}**

// 将 EXPIRE/PEXPIRE/EXPIREAT命令转换PEXPIREAT命令。

**else if (cmd->proc == setexCommand || cmd->proc == psetexCommand)**

**{**

**tmpargv[0] = createStringObject("SET",3);**

**tmpargv[1] = argv[1];**

**tmpargv[2] = argv[3];**

**buf = catAppendOnlyGenericCommand(buf,3,tmpargv);**

**decrRefCount(tmpargv[0]);**

**buf = catAppendOnlyExpireAtCommand(buf,cmd,argv[1],argv[2]);**

**}**

// 将 SETEX/PSETEX命令翻译为 SET和PEXPIREAT命令

**else**

**{**

**buf = catAppendOnlyGenericCommand(buf,argc,argv);**

**}**

其他不需要转换的命令或只需要通用转换(在命令队列中一句完成)的命令

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

{

server.aof\_buf = sdscatlen(server.aof\_buf,buf,sdslen(buf));

}

如果再确保配置里打开aof机制，则把buf里的内容追加到server.aof\_buf里。

**if (server.aof\_child\_pid != -1)**

**{**

**aofRewriteBufferAppend((unsigned char\*)buf,sdslen(buf));**

**}**

如果此时AOF重写(rewrite)进程正在执行重写，那么还需要将buf内容追加到AOF重写缓存(server.aof\_rewrite\_buf\_blocks)中，等AOF重写完成之前的命令执行，新追加的命令会被执行。server.aof\_rewrite\_buf\_blocks是一个内存块的链表，由于rewrite过程消耗时间可能会很长，这是需要存在内存的命令可能较多，为了避免频繁的分配新空间，Redis每次都会申请10MB的block，然后将该block添加到server.aof\_rewrite\_buf\_blocks。

* + 1. aofRewriteBufferAppend()函数

参数

unsigned char \*s ：需要追加的字符串指针

unsigned long len：需要追加的字符串长度

**listNode \*ln = listLast(server.aof\_rewrite\_buf\_blocks);**

**aofrwblock \*block = ln ? ln->value : NULL;**

取得server.aof\_rewrite\_buf\_blocks链表的尾节点，如果有值，就把尾节点的block赋值给block指针。

while(len)

{

**if (block)**

**{**

**unsigned long thislen = (block->free < len) ? block->free : len;**

**if (thislen)**

**{ memcpy(block->buf+block->used, s, thislen);**

**block->used += thislen;**

**block->free -= thislen;**

**s += thislen;**

**len -= thislen;**

**}**

**}**

如果block有值，则说明已经存在一个已经分配好的block，那么就尝试追加全部或部分字符串到已经分配的block。检查当前block剩余字节长度，如果长度不是0，则说明当前block未满，就在当前block上开始追加。

**if ( len )**

{

如果执行到此处，说明上面的block不能完全容纳需要追加的内容。还需要开辟新block。

**int numblocks;**

**block = zmalloc(sizeof(\*block));**

**block->free = AOF\_RW\_BUF\_BLOCK\_SIZE;**

**block->used = 0;**

**listAddNodeTail(server.aof\_rewrite\_buf\_blocks,block);**

分配一个10M大小的Block，并把它链接到server.aof\_rewrite\_buf\_blocks尾部。

**numblocks = listLength(server.aof\_rewrite\_buf\_blocks);**

**if (((numblocks+1) % 10) == 0)**

**{**

**int level = ((numblocks+1) % 100) == 0 ? REDIS\_WARNING :REDIS\_NOTICE;**

**redisLog(level,"Background AOF buffer size: %lu MB",**

**aofRewriteBufferSize()/(1024\*1024));**

**}**

每分配Block数目达到整10，就记录一次警告日志。

}

}

3.6.5 loadAppendOnlyFile()函数

参数：char \*filename AOF文件名

返回：如果成功返回REDIS\_OK，非致命失败(AOF文件为空)返回REDIS\_ERR，致命失败时把失败原因记录到日志，然后退出进程。

功能：看到此处，对AOF文件的工作原理应该比较清楚了，这个函数的作用就是从一个已经存在的AOF文件一条一条恢复数据库原样。在Redis以非监视模式启动时完成后，会调用loadDataFromDisk()函数来加载已有数据，在loadDataFromDisk()函数里如果检查配置里的AOF机制是打开状态，那么就调用本函数来加载数据库数据。由于AOF是按照一条接一条命令来执行的，Redis作者称这一过程为“重播”。

**struct redisClient \*fakeClient;**

**FILE \*fp = fopen(filename,"r");**

**struct redis\_stat sb;**

**int old\_aof\_state = server.aof\_state;**

**long loops = 0;**

**if (fp && redis\_fstat(fileno(fp),&sb) != -1 && sb.st\_size == 0)**

**{**

**server.aof\_current\_size = 0;**

**fclose(fp);**

**return REDIS\_ERR;**

**}**

打开的文件为空，返回错误。

**if (fp == NULL)**

**{**

**redisLog( REDIS\_WARNING,"Fatal error: can't open the append log file for reading: %s",strerror(errno));**

**exit(1);**

**}**

打开文件失败，退出程序。个人认为这一句应该提到上一个判断之前，则上面一个判断可以省略fp判断条件。

**server.aof\_state = REDIS\_AOF\_OFF;**

临时关闭AOF机制，放止在我们将要读的文件上执行事务。

**fakeClient = createFakeClient();**

这一步是AOF加载的关键：构建一个伪客户端(fake client)，然后使用这个client向server发送从aof文件中读入的命令。创建这个伪客户端的目的是能复用后事件处理的接口。

**startLoading(fp)**

更新服务器的载入状态

**while(1)**

**{**

**int argc, j;**

**unsigned long len;**

**robj \*\*argv;**

**char buf[128];**

**sds argsds;**

**struct redisCommand \*cmd;**

**if (!(loops++ % 1000))**

**{** !(loops++ % 1000),有间隔地处理外部请求,Redis经常使用这种方式

**loadingProgress(ftello(fp));**

**aeProcessEvents(server.el, AE\_FILE\_EVENTS|AE\_DONT\_WAIT);**

交给事件处理程序来处理

**}**

**if (fgets(buf,sizeof(buf),fp) == NULL)**

**{**

**if (feof(fp)){ break; }else{ goto readerr;}**

**}**

从AOF文件中取出一行到buf里，并检测出错。

if (buf[0] != '\*'){ goto fmterr; }

// --确保以界定符\*开头

argc = atoi(buf+1); // --模拟参数个数

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;**

**}**

实现虚拟客户端参数的模拟。

**cmd = lookupCommand(argv[0]->ptr);**

**if (!cmd)**

**{**

**redisLog(REDIS\_WARNING,"Unknown command '%s' reading the append only file", argv[0]->ptr);**

**exit(1);**

**}**

查找相关命令

**fakeClient->argc = argc;**

**fakeClient->argv = argv;**

**cmd->proc(fakeClient);**

借助伪客户端执行命令

**redisAssert(fakeClient->bufpos == 0 && listLength(fakeClient->reply) == 0);**

伪客户端不应该收到回复

**redisAssert((fakeClient->flags & REDIS\_BLOCKED) == 0);**

伪客户端不会进行阻塞操作等待

**/\* 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);**

**}**

if (fakeClient->flags & REDIS\_MULTI) goto readerr;

**fclose(fp);**

**freeFakeClient(fakeClient);**

**server.aof\_state = old\_aof\_state;**

**stopLoading();**

**aofUpdateCurrentSize();**

**server.aof\_rewrite\_base\_size = server.aof\_current\_size;**

**return REDIS\_OK;**

清理资源，并还原状态

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);

两个goto语句的出错标签。

3.6.6 flushAppendOnlyFile()函数

参数: int force : 如果这个值非0表示强制存储，立即写入磁盘，而非延时写入。

一共有三处调用这个函数，一处是每次事件循环之前的beforeSleep()函数，另一处是在每次时间事件循环serverCron()，这两处都是延时写入，区别是在beforeSleep()这里开启写入操作，而在serverCron()做后续写入操作。最后一处是stopAppendOnly()强制写入，发生在从aof状态开启到关闭的切换时，或者用户手动输入强制存储指令。

ssize\_t nwritten;

int sync\_in\_progress = 0;

if (sdslen(server.aof\_buf) == 0){ return; }

如果aof\_buf没有内容，则返回。

if (server.aof\_fsync == AOF\_FSYNC\_EVERYSEC)

{

sync\_in\_progress = bioPendingJobsOfType(REDIS\_BIO\_AOF\_FSYNC) != 0;

}

if (server.aof\_fsync == AOF\_FSYNC\_EVERYSEC && !force)

{

if (sync\_in\_progress)

{

if (server.aof\_flush\_postponed\_start == 0)

{

server.aof\_flush\_postponed\_start = server.unixtime;

return;

}

如果之前没有开启延时写入，此处打开延时写入，并返回。

一般是只有beforeSleep()处的调用才能执行到。

else if (server.unixtime - server.aof\_flush\_postponed\_start < 2)

{ return;}

从开启延时后还不到2秒，继续延时。

server.aof\_delayed\_fsync++;

redisLog(REDIS\_NOTICE,

"Asynchronous AOF fsync is taking too long (disk is busy?). Writing the AOF buffer without waiting for fsync to complete,

this may slow down Redis.");

记录冲洗推延次数

}

}

server.aof\_flush\_postponed\_start = 0;

执行到达这一步，写入已经不能再延时下去了，关闭延时状态，并准备把缓存数据写入磁盘。

nwritten = write(server.aof\_fd,server.aof\_buf,sdslen(server.aof\_buf));

将AOF缓存写入到文件，如果一切幸运的话，写入会原子性地完成。

if (nwritten != (signed)sdslen(server.aof\_buf))

{

这里我把作者的注释抄过来: “Ooops, we are in troubles. The best thing to do for now is aborting instead of giving the illusion that everything is working as expected.”翻译过来就是: “唉! 运行到这，我们就碰上麻烦了，现在最好的做法是马上终止程序，而不是给用户一个如预期运行的错觉！”呵呵，作者可爱但严谨。

if (nwritten == -1)

{

redisLog(REDIS\_WARNING,"Exiting on error writing

to the append-only file: %s",strerror(errno));

}

else

{

redisLog(REDIS\_WARNING,

"Exiting on short write while writing to "

"the append-only file: %s (nwritten=%ld, "

"expected=%ld)",

strerror(errno),

(long)nwritten,

(long)sdslen(server.aof\_buf));

if (ftruncate(server.aof\_fd, server.aof\_current\_size) == -1)

{

redisLog(REDIS\_WARNING, "Could not remove short write "

"from the append-only file. Redis may refuse "

"to load the AOF the next time it starts. "

"ftruncate: %s", strerror(errno));

}

}

exit(1);

}

server.aof\_current\_size += nwritten;

if ((sdslen(server.aof\_buf)+sdsavail(server.aof\_buf)) < 4000)

{

sdsclear(server.aof\_buf);

} else

{

sdsfree(server.aof\_buf);

server.aof\_buf = sdsempty();

}

在写入后，如果该缓存大小较小的话我们选择重用该空间,否则清楚aof缓存。

*在写入后，如果该缓存大小较小的话我们选择重用该空间。接下来是进行fsync的选择。这里我们看到，如果fsync策略是每次都fsync，那么会直接调用fsync来同步，如果策略是每秒同步，那么会将fsync交给IO线程进行。这里调用aof\_background\_fsync()的主要工作就是创建了异步IO的工作，将fsync交给另一个线程做。*

if (server.aof\_no\_fsync\_on\_rewrite &&

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

{ return; }

/\* Perform the fsync if needed. \*/

if (server.aof\_fsync == AOF\_FSYNC\_ALWAYS)

{

/\* aof\_fsync is defined as fdatasync() for Linux in order to avoid

\* flushing metadata. \*/

aof\_fsync(server.aof\_fd); /\* Let's try to get this data on the disk \*/

server.aof\_last\_fsync = server.unixtime;

}

else if ((server.aof\_fsync == AOF\_FSYNC\_EVERYSEC &&

server.unixtime > server.aof\_last\_fsync))

{

if (!sync\_in\_progress) aof\_background\_fsync(server.aof\_fd);

server.aof\_last\_fsync = server.unixtime;

}

3.6.7 rewriteAppendOnlyFileBackground()函数

*rewrite过程一般发生在Redis服务器端收到bgrewriteaof命令或者AOF文件膨胀到需要rewrite时进行。比较特殊的情况是，Redis在运行过程收到config命令启动AOF机制，这是也会启动rewrite。这些情况都会调用rewriteAppendOnlyFileBackground()。*

*writeAppendOnlyFileBackground()的工作主要是fork出一个子进程，然后对父进程进行AOF状态的更新。rewrite任务就交给子进程运行rewriteAppendOnlyFile()解决。*

/\* This is how rewriting of the append only file in background works:

\*

\* 1) The user calls BGREWRITEAOF

\* 2) Redis calls this function, that forks():

\* 2a) the child rewrite the append only file in a temp file.

\* 2b) the parent accumulates differences in server.aof\_rewrite\_buf.

\* 3) When the child finished '2a' exists.

\* 4) The parent will trap the exit code, if it's OK, will append the

\* data accumulated into server.aof\_rewrite\_buf into the temp file, and

\* finally will rename(2) the temp file in the actual file name.

\* The the new file is reopened as the new append only file. Profit!

\*/

**int rewriteAppendOnlyFileBackground(void)**

**{**

**pid\_t childpid;**

**long long start;**

**if (server.aof\_child\_pid != -1) return REDIS\_ERR;**

**start = ustime();**

**if ((childpid = fork()) == 0)**

**{**

**char tmpfile[256];**

**/\* Child \*/**

**if (server.ipfd > 0) close(server.ipfd);**

**if (server.sofd > 0) close(server.sofd);**

**snprintf(tmpfile,256,"temp-rewriteaof-bg-%d.aof", (int) getpid());**

**if (rewriteAppendOnlyFile(tmpfile) == REDIS\_OK)**

**{**

**size\_t private\_dirty = zmalloc\_get\_private\_dirty();**

**if (private\_dirty)**

**{**

**redisLog(REDIS\_NOTICE,**

**"AOF rewrite: %zu MB of memory used by copy-on-write",**

**private\_dirty/(1024\*1024));**

**}**

**exitFromChild(0);**

**}**

**else**

**{**

**exitFromChild(1);**

**}**

**}**

**else**

**{**

**server.stat\_fork\_time = ustime()-start;**

**if (childpid == -1)**

**{**

**redisLog(REDIS\_WARNING, "Can't rewrite**

**append only file in background: fork: %s",strerror(errno) );**

**return REDIS\_ERR;**

**}**

**redisLog(REDIS\_NOTICE,"Background append only**

**file rewriting started by pid %d",childpid);**

**server.aof\_rewrite\_scheduled = 0;**

**server.aof\_rewrite\_time\_start = time(NULL);**

**server.aof\_child\_pid = childpid;**

**updateDictResizePolicy();**

/\* We set appendseldb to -1 in order to force the next call to the

\* feedAppendOnlyFile() to issue a SELECT command, so the differences

\* accumulated by the parent into server.aof\_rewrite\_buf will start

\* with a SELECT statement and it will be safe to merge. \*/

**server.aof\_selected\_db = -1;**

**return REDIS\_OK;**

**}**

**return REDIS\_OK;**

**}**

3.6.8 rewriteAppendOnlyFile ()函数

*rewriteAppendOnlyFile()的工作就是遍历数据库，将kv键值对格式化为标准的Redis命令写入临时文件。然后强制刷到硬盘上，重命名临时文件为参数规定的名字，最后子进程退出。这里的关注点在于kv键值格式化的结果。rioWriteBulkObject,rewriteListObject,rewriteSetObject,rewriteSortedSetObject,rewriteHashObject的工作就是将5种类型对象分别解析，然后写入到硬盘。*

3.7 慢日志(slow log)系统

3.7.1 相关结构

**typedef struct slowlogEntry**

**{**

**robj \*\*argv;**

**int argc;**

**long long id; // --统一ID**

**long long duration; // --Time spent by the query, in nanoseconds.**

**time\_t time; /\* Unix time at which the query was executed. \*/**

**}**

**slowlogEntry;**

server结构里的关于慢日志相关的属性

**list \*slowlog; // --慢日志链表**

**long long slowlog\_entry\_id; // --当前日志节点ID**

**long long slowlog\_log\_slower\_than; // --记录慢日志的参考标准(单位:微秒)**

**unsigned long slowlog\_max\_len; // --慢日志链表最大节点数目**

**#define SLOWLOG\_ENTRY\_MAX\_ARGC 32 // --最长的命令参数**

**#define SLOWLOG\_ENTRY\_MAX\_STRING 128 // --最长字符串长度**

3.7.2 slowlogInit()函数

**server.slowlog = listCreate();**

**server.slowlog\_entry\_id = 0;**

**listSetFreeMethod(server.slowlog,slowlogFreeEntry);**

初始化慢日志，在Redis Server启动时调用。创建慢日志链表结构，并把slowlogFreeEntry()函数赋值给server.slowlog.free函数指针。

3.7.3 slowlogPushEntryIfNeeded ()函数

所有的客户端过来的命令，服务器收到后都会交给redis.c的call()函数来处理，在call()函数里，如果启用了slowlog，那么都会把命令极其执行时间传入这个函数，由这个函数来决定是否插入慢日志。

**if ( server.slowlog\_log\_slower\_than < 0 ) return;**

如果命令执行时间server.slowlog\_log\_slower\_than配置不正确，则返回。

**if ( duration >= server.slowlog\_log\_slower\_than )**

**{**

**listAddNodeHead(server.slowlog, slowlogCreateEntry( argv,argc,duration ) );**

**}**

如果命令执行时间大于等于server.slowlog\_log\_slower\_than，则根据这条指令创建一个slowlogEntry节点加入慢日志链表。

**while (listLength(server.slowlog) > server.slowlog\_max\_len)**

**{**

**listDelNode(server.slowlog,listLast(server.slowlog));**

**}**

如果节点数目超过server.slowlog\_max\_len，则一直删除server.slowlog的节点

直到节点数目不超过server.slowlog\_max\_len。

3.7.4 slowlogCreateEntry ()函数

参数：

**robj \*\*argv 命令**

**int argc**

**long long duration**

这个函数用于创建一个慢日志链表的节点。

**slowlogEntry \*se = zmalloc(sizeof(\*se));**

**int j, slargc = argc;**

**if (slargc > SLOWLOG\_ENTRY\_MAX\_ARGC) slargc = SLOWLOG\_ENTRY\_MAX\_ARGC;**

参数最多记录SLOWLOG\_ENTRY\_MAX\_ARGC个，超过的则被丢弃不记录。

**se->argc = slargc;**

**se->argv = zmalloc(sizeof(robj\*)\*slargc);**

**for (j = 0; j < slargc; j++)**

**｛**

**if (slargc != argc && j == slargc-1)**

**{**

**se->argv[j] = createObject(REDIS\_STRING,**

**sdscatprintf(sdsempty(),"... (%d more arguments)",**

**argc-slargc+1));**

**}**

**else**

**{**

**/\* Trim too long strings as well... \*/**

**if (argv[j]->type == REDIS\_STRING &&**

**argv[j]->encoding == REDIS\_ENCODING\_RAW &&**

**sdslen(argv[j]->ptr) > SLOWLOG\_ENTRY\_MAX\_STRING)**

**{**

**sds s = sdsnewlen(argv[j]->ptr, SLOWLOG\_ENTRY\_MAX\_STRING);**

**s = sdscatprintf(s,"... (%lu more bytes)",**

**(unsigned long)**

**sdslen(argv[j]->ptr) - SLOWLOG\_ENTRY\_MAX\_STRING);**

**se->argv[j] = createObject(REDIS\_STRING,s);**

**}**

**else**

**{**

**se->argv[j] = argv[j];**

**incrRefCount(argv[j]);**

**}**

**}**

**｝**

**se->time = time(NULL); // --记录时间**

**se->duration = duration; // --执行时间跨度(微秒)**

**se->id = server.slowlog\_entry\_id++; // --ID增1**

**return se;**

3.8 主从复制(Replication)系统 replication.c

3.8.1 Sever常量适用于Slave的一些属性

char \*masterauth; // --向主节点认证的密码

char \*masterhost; // --主节点的主机名

int masterport; // --主节点的端口

int repl\_ping\_slave\_period; // --主节点ping从节点的周期(每N秒ping一次)

int repl\_timeout; // -- Timeout after N seconds of master idle

redisClient \*master; // -- Client that is master for this slave \*/

int repl\_syncio\_timeout; // -- Timeout for synchronous I/O calls \*/

int repl\_state; // --Replication status(只有在Slave实例有用)

off\_t repl\_transfer\_size; // --master发送给slave的rdb文件字节数

off\_t repl\_transfer\_read; // --已经从master读取的rdb文件字节数

off\_t repl\_transfer\_last\_fsync\_off; // --Offset when we fsync-ed last time.

int repl\_transfer\_s; // --master (主向从同步rdb)的socket

int repl\_transfer\_fd; // --Slave -> Master SYNC temp file descriptor

char \*repl\_transfer\_tmpfile; // --Slave保存 master 发送的临时rdb文件的文件名

time\_t repl\_transfer\_lastio; // --slave端上一次收到master端传送的rdb文件的时间

int repl\_serve\_stale\_data; // --Serve stale data when link is down?

int repl\_slave\_ro; // --Slave is read only?

time\_t repl\_down\_since; // --Unix time at which link with master went down \*/

int repl\_disable\_tcp\_nodelay; // --Disable TCP\_NODELAY after SYNC?

int slave\_priority; // --Reported in INFO and used by Sentinel. \*/

3.8.2 状态

【注意】这些状态**非常重要**，是Redis在主从Replication的时候，在某一时刻主从双方各自进行何种操作(是保存rdb文件？是同步文件？是同步命令？)的判断依据。

#define REDIS\_REPL\_NONE 0 // --无激活的复制

#define REDIS\_REPL\_CONNECT 1 // --已经接收到slaveof命令，但未发出sync命令给master

#define REDIS\_REPL\_CONNECTING 2 // --正在发送ping给master

#define REDIS\_REPL\_RECEIVE\_PONG 3 // --发送ping完毕，等待PING回复

#define REDIS\_REPL\_TRANSFER 4 // --已经发出sync，但还没接收完rdb文件

#define REDIS\_REPL\_CONNECTED 5 // --已经连接到master

Master对每个Slave标记状态(master的角度记录slave的状态)

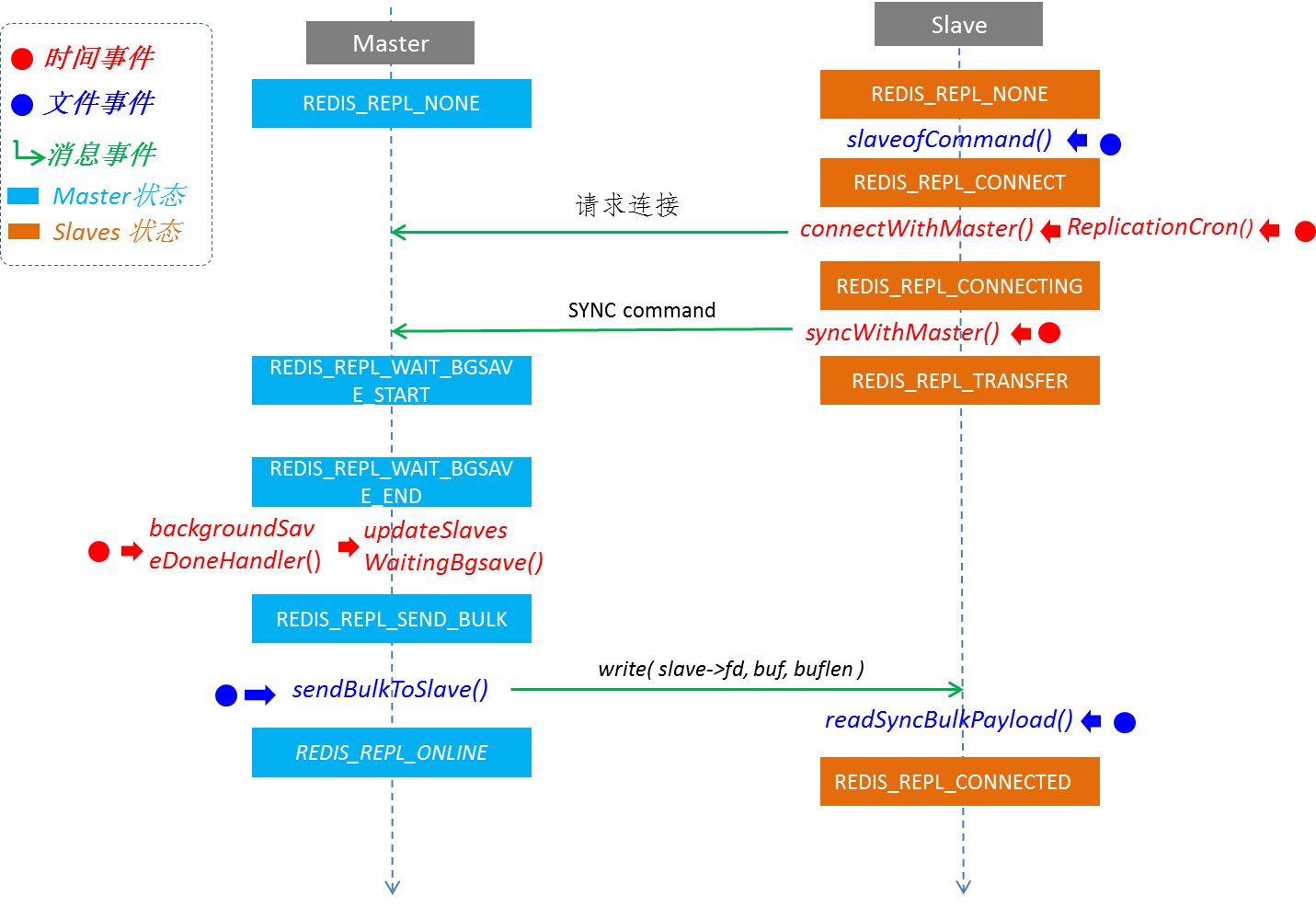
#define REDIS\_REPL\_WAIT\_BGSAVE\_START 3 // -- bgsave被抢占，等待bgsave为其持久化进行工作

#define REDIS\_REPL\_WAIT\_BGSAVE\_END 4 // --master后台写rdb文件结束

#define REDIS\_REPL\_SEND\_BULK 5 // --master正在发送rdb数据块

#define REDIS\_REPL\_ONLINE 6 // --rdb已经传输完毕

3.8.3 工作流程及状态迁移



Master和Slave状态迁移图

**阶段一 初始阶段**

Redis通过server.replstate表明Master和Slave的状态，初始时，状态均为REDIS\_REPL\_NONE。从节点的server.masterhost和server.masterport为Master的地址和端口，而主节点这两个值都为NULL。Master对于每个Slave都维护一个状态，在结构体redisClient.replstate中，对应的宏见上文。

**阶段二 建立连接**

当slave的客户端执行 slaveof ip port 这个命令，slave server从命令表(readonlyCommandTable)中找到相应的命令处理函数slaveofCommand并执行该函数。slaveofCommand函数保存了命令的IP到server.masterhost，保存port到server.masterport，并更新slave的状态为REDIS\_REPL\_CONNECT，然后Slave server返回给客户端OK结果。

【注意】虽然返回OK，但这时候实际还没有连接到Master，仅仅是做了一个记录和状态切换。

【注意】slave可以做其他slave的master，但要求这个slave必须处于REDIS\_REPL\_CONNECTED状态。

接下来，时间事件循环serverConn()调用replicationCron(),在replicationCron()函数里，发现slave状态是REDIS\_REPL\_CONNECT时就调用connectWithMaster()主动连接master。如果连接成功，slave将状态切换到REDIS\_REPL\_CONNECTING，并创建同步文件事件，处理函数为syncWithMaster()。

【注意】整个这个过程都是slave在操作，master一直维持在REDIS\_REPL\_NONE状态。

**第三阶段 保存文件**

slave调用syncWithMaster()开始与master的通信。syncWithMaster()会发送一个SYNC command给master端。然后打开一个临时文件用于存储接下来master发过来的rdb文件数据。再添加一个文件事件注册readSyncBulkPayload函数，用来处理rdb文件的数据接收，最后更新状态为**REDIS\_REPL\_TRANSFER**。

Master在收到这个SYNC command后，从readonlyCommandTable中找到相应的命令处理函数syncCommand(),并执行该函数。

【注意】如果接受sync指令的结点，本身也是Slave，且没有与任何Master建立连接，则无法响应sync命令。

【注意】如果Slave与Master之间还有其它交互命令没有处理完，也无法继续响应sync命令。

在这个函数Master检测bgsave状态，如果已经有bgsave进程，再判断是否正在响应其它Slave的sync命令，且正在dump数据（Slave的replstate为REDIS\_REPL\_BGSAVE\_END），则直接将新Slave的replstate状态置为

REDIS\_REPL\_BGSAVE\_END，并准备将dump好的数据传给新Slave。如果没有dump好的数据，则将新SLAVE的状态设为**REDIS\_REPL\_WAIT\_BGSAVE\_START**，并等待bgsave进程完成，之后再启动bgsave，为Slave dump数据。如果没有bgsave进程，则启动新的bgsave进程，并将slave的状态设置为**REDIS\_REPL\_BGSAVE\_END**。最后将新的slave加入master的server.slave列表。

【注意】这里启动bgsave就是直接调用rbd.c里的rdbSaveBackground()，功能复用。

Master的时间循环serverCron()里，如果检测到服务器开启了bgsave进程，则调用updateSlavesWaitingBgsave()，在updateSlavesWaitingBgsave()里，然后打开刚刚产生的rdb文件，并在slave的socket fd上注册一个可写文件事件，处理函数为sendBulkToSlave()。master把相应的slave状态更新为REDIS\_REPL\_SEND\_BULK。

**第四阶段 文件同步**

slave的soket可以写数据，就会激活sendBulkToSlave()函数，开始向slave的soket写rdb数据。Master会一直持续这个过程，一旦slave sockt激活，就向soket写REDIS\_IOBUF\_LEN大小的数据，直到整个文件都传送完成，把相应的slave状态更新为**REDIS\_REPL\_ONLINE**。

我们回到slave，上面讲到slave通过readSyncBulkPayload接收rdb数据，接收完整个rdb文件后，会清空整个数据库emptyDb()。然后就通过rdbLoad函数载入接收到的rdb文件，此时slave和master数据已经一直，slave的状态被更新为**REDIS\_REPL\_CONNECTED**。

**第五阶段 命令同步**

此后的同步进入命令同步了，对于master和slave都是非阻塞的。master会给所有处于REDIS\_REPL\_ONLINE状态的客户端同步成功的写操作命令。

* + 1. slaveofCommand()函数

参数: redisClient \*c: 发送命令的客户端指针

返回值: void

Slave端发起同步请求。首先，即将成为slave的redis instance收到slaveof命令或者启动时配置了slaveof选项，则执行slaveofCommand函数。

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);

cancelReplicationHandshake();

server.repl\_state = REDIS\_REPL\_NONE;

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

}

}

如果命令是slaveof no one，则取消replication。否则得到master ip和port后，判断是否已经与该master建立连接，若不是，则放弃已有的replication连接并初始化server的几个replication成员变量。

else

{

long port;

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

{ return; }

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;

}

检查是否已经连接到指定的slave。则给出警告。

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

\* we can continue. \*/

sdsfree(server.masterhost);

server.masterhost = sdsdup(c->argv[1]->ptr);

server.masterport = port;

if (server.master) freeClient(server.master);

disconnectSlaves(); // 强迫Slaves重新与Master同步

cancelReplicationHandshake();

server.repl\_state = REDIS\_REPL\_CONNECT;

redisLog(REDIS\_NOTICE,"SLAVE OF %s:%d enabled (user request)", server.masterhost, server.masterport);

}

addReply(c,shared.ok);

返回给信息给客户端

}

到此，slaveof的初始化结束，server.repl\_state的状态更新为REDIS\_REPL\_CONNECT。

*如前文rdb快照分析，当bgsave进程备份完成时，在serverCron()中会调用*

*backgroundSaveDoneHandler()处理，该函数中调用*

*updateSlavesWaitingBgsave()处理等待bgsave的Slave结点。*

*updateSlavesWaitingBgsave()检查server.slaves列表，当发现状态为*

*REDIS\_REPL\_WAIT\_BGSAVE\_START的结点时，会再次启动bgsave进程进行dump数据。*

*当发现状态为REDIS\_REPL\_WAIT\_BGSAVE\_START的结点时，会准备将数据发送给*

*Slave，并将Slave的状态修改为REDIS\_REPL\_SEND\_BULK。Master在Slave上监听*

*WRITEABLE事件，当Slave可写时，调用sendBulkToSlave()将数据发送到Slave。*

*在Master中，sendBulkToSlave()将数据从磁盘读入，通过socket发送到*

*Slave，完成后删除WRITEABLE事件，并将Slave状态改为REDIS\_REPL\_ONLINE。然后*

*再次监听Slave的WRITEABLE事件，当Slave可写时，表明Slave已经准备好开始同步指*

*令。*

**【Slave】**

*Redis可以在配置文件中指定为Slave模式，并指定Master的地址，这样的Slave结*

*点启动时即为REDIS\_REPL\_ONLINE状态，然后在replicationCron()中调用*

*syncWithMaster()与Master同步。*

*syncWithMaster()中，向Master发送sync命令，并准备接收数据的目录。然后通*

*过readSyncBulkPayload()处理Master发送的dump的数据。Slave通过syncRead()*

*和syncWrite()与Master传输数据，这两个方法底层是异步IO的，但封装成阻塞型，因此Slave第一次同步是阻塞的。*

*replicationCron()会定时Master的dump数据是否发送完成，如果长时间没有收*

*到dump数据的数据包，Slave会通过replicationAbortSyncTransfer()取消数据同*

*步。*

* + 1. replicationCron()函数

参数: 无

说明: 从节点服务器在时间事件循环ServerCron里周期性(每秒1次)调用replicationCron()。调用代码run\_with\_period(1000) replicationCron();

该函数用于主从状态的维护：检测同步数据传输是否超时，主从之间的心跳等等。

在call()中，Master会将改变了数据的命令通过replicationFeedSlaves()同

步到Slaves中。Slave以处理普通命令的流程处理这些Master发来的命令。

特别地在serverCron()中，Master会删除过期的数据，Slave则等待Master同步

DEL指令将过期数据删除。

replicationCron()中，Master会定时向所有Slave发送心跳指令，同时，

Slave的replicationCron()会通过Ping指令检查Slave与Master的连接状态，如果

Slave长时间没有收到主机的Ping，则会断开与主机的连接。

**if (server.masterhost &&**

**(server.repl\_state == REDIS\_REPL\_CONNECTING ||**

**server.repl\_state == REDIS\_REPL\_RECEIVE\_PONG) &&**

**(time(NULL)-server.repl\_transfer\_lastio) > server.repl\_timeout)**

**{**

**redisLog(REDIS\_WARNING,"Timeout connecting to the MASTER...");**

**undoConnectWithMaster();**

**}**

如果当前连接状态处于REDIS\_REPL\_CONNECTING或者REDIS\_REPL\_RECEIVE\_PONG状态下，如果当前时间距离上次同步时间超过超时时间，则撤销这个连接。

看一下的代码：

**int fd = server.repl\_transfer\_s;**

**redisAssert(server.repl\_state == REDIS\_REPL\_CONNECTING ||**

**server.repl\_state == REDIS\_REPL\_RECEIVE\_PONG);**

**aeDeleteFileEvent(server.el,fd,AE\_READABLE|AE\_WRITABLE);**

**close(fd);**

**server.repl\_transfer\_s = -1;**

**server.repl\_state = REDIS\_REPL\_CONNECT;**

注销事件委托，关闭文件，重设连接状态为REDIS\_REPL\_CONNECT，以便后面进行重连操作。

**if (server.masterhost && server.repl\_state == REDIS\_REPL\_TRANSFER &&**

**(time(NULL)-server.repl\_transfer\_lastio) > server.repl\_timeout)**

**{**

**redisLog(REDIS\_WARNING,"Timeout receiving bulk data from MASTER... If the problem persists try to set the 'repl-timeout' parameter in redis.conf to a larger value.");**

**replicationAbortSyncTransfer();**

**}**

如果当前连接状态处于REDIS\_REPL\_TRANSFER状态下，如果当前时间距离上次同步时间大于超时时间，则退出正在进行的从主服务器同步下载的数据块。跟进replicationAbortSyncTransfer()源码:

redisAssert(server.repl\_state == REDIS\_REPL\_TRANSFER);

aeDeleteFileEvent(server.el,server.repl\_transfer\_s,AE\_READABLE);

close(server.repl\_transfer\_s);

close(server.repl\_transfer\_fd);

unlink(server.repl\_transfer\_tmpfile);

zfree(server.repl\_transfer\_tmpfile);

server.repl\_state = REDIS\_REPL\_CONNECT;

if (server.masterhost && server.repl\_state == 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);

}

freeClient()见Networking.c。

if (server.repl\_state == REDIS\_REPL\_CONNECT)

{

redisLog(REDIS\_NOTICE,"Connecting to MASTER...");

if (connectWithMaster() == REDIS\_OK)

{

redisLog(REDIS\_NOTICE,"MASTER <-> SLAVE sync started");

}

}

如果从节点的server.repl\_state为REDIS\_REPL\_CONNECT，即尝试连接主节点服务器。

connectWithMaster()这个函数见后文。

if (!(server.cronloops % (server.repl\_ping\_slave\_period \* server.hz)))

{

如果连接到从节点，每隔一定周期，进行一次心跳ping，这样从节点可以计算超时。而且可以方便主节点及时发现TCP是否保持连接。

listIter li;

listNode \*ln;

listRewind(server.slaves,&li);

while((ln = listNext(&li)))

{

循环遍历从节点链表

redisClient \*slave = ln->value;

if (slave->replstate == REDIS\_REPL\_SEND\_BULK) continue;

不要ping正在与主节点同步数据的从节点

if (slave->replstate == REDIS\_REPL\_ONLINE)

{

addReplySds(slave,sdsnew("\*1\r\n$4\r\nPING\r\n"));

}

如果当前从节点处于REDIS\_REPL\_ONLINE状态，则发送一个普通ping。

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)

{

// --无须担心，这只是一个ping

}

}

}

}

3.8.7 connectWithMaster()函数

**int fd = anetTcpNonBlockConnect(NULL,server.masterhost,server.masterport);**

**if (fd == -1)**

**{**

**redisLog(REDIS\_WARNING,"Unable to connect to MASTER: %s",strerror(errno));**

**return REDIS\_ERR;**

**}**

从节点请求与主节点建立非阻塞连接，anetTcpNonBlockConnect()见anet.c。

**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;**

**}**

创建文件事件，处理函数为syncWithMaster()。

**server.repl\_transfer\_lastio = server.unixtime;**

**server.repl\_transfer\_s = fd;**

**server.repl\_state = REDIS\_REPL\_CONNECTING;**

**return REDIS\_OK;**

记录当前时间，保存连接的fd到server.repl\_transfer\_s，更新从节点连接状态为REDIS\_REPL\_CONNECTING。

3.8.8 syncWithMaster()函数

本函数是从节点在replication启动后与主节点建立主从关系的核心函数。

*在之前的connectWithMaster()后，此时slave端需要发送PING命令给master，检测master是否能回复，以此判断连接成功的socket是否为redis instance，在这种，redis一改之前全部的异步读写，使用了syncWrite，该函数向fd写入内容，如果阻塞，那么redis也会阻塞直到全部内容写入发送缓冲区。*

char tmpfile[256], \*err;

int dfd, maxtries = 5;

int sockerr = 0;

socklen\_t errlen = sizeof(sockerr);

REDIS\_NOTUSED(el);

REDIS\_NOTUSED(privdata);

REDIS\_NOTUSED(mask);

**if (server.repl\_state == REDIS\_REPL\_NONE)**

**{**

**close(fd);**

**return;**

**}**

**if (getsockopt(fd, SOL\_SOCKET, SO\_ERROR, &sockerr, &errlen) == -1){ sockerr = errno; }**

**if (sockerr)**

**{**

**aeDeleteFileEvent(server.el,fd,AE\_READABLE|AE\_WRITABLE);**

**redisLog(REDIS\_WARNING,"Error condition on socket for SYNC: %s", strerror(sockerr));**

**goto error;**

**}**

如果已经与master建立连接，就应该发送一个非阻塞的PING，我们需要在进入replication实际操作之前确认master是否可以正常回应。因为即下来的首次replication将持续大约数秒，slave将会阻塞。

if (server.repl\_state == REDIS\_REPL\_CONNECTING)

{

redisLog(REDIS\_NOTICE,"Non blocking connect for SYNC

fired the event.");

aeDeleteFileEvent(server.el,fd,AE\_WRITABLE);

删除AE\_WRITABLE事件，只保留AE\_READABLE事件。

server.repl\_state = REDIS\_REPL\_RECEIVE\_PONG;

更改slave当前的连接状态为REDIS\_REPL\_RECEIVE\_PONG。

syncWrite(fd,"PING\r\n",6,100);

发送PING，让后等待PONG回应。有超时来保证，可以不需要错误检查。

return;

}

循环中，如果状态是REDIS\_REPL\_RECEIVE\_PONG，则进行接受PONG回应。

**if (server.repl\_state == REDIS\_REPL\_RECEIVE\_PONG)**

**{**

**char buf[1024];**

**aeDeleteFileEvent(server.el,fd,AE\_READABLE);**

删除AE\_READABLE事件，我们不再需要了。

**buf[0] = '\0';**

**if (syncReadLine(fd,buf,sizeof(buf),**

**server.repl\_syncio\_timeout\*1000) == -1)**

**{**

**redisLog(REDIS\_WARNING,"I/O error reading PING reply**

**from master: %s", strerror(errno));**

**goto error;**

**}**

读取回应字符串，带有超时参数。如果失败则给出警告:“I/O错误-从master读取PING回应失败”。

**if (buf[0] != '+' && strncmp(buf,"-NOAUTH",7) != 0 &&**

**strncmp(buf,"-ERR operation not permitted",28) != 0)**

**{**

**redisLog(REDIS\_WARNING,"Error reply to**

**PING from master: '%s'",buf);**

**goto error;**

**}**

可能接收到两种回应：一种是 +PONG回应(在此我们只需要检查“+”字符)，另一种是一个认证错误。注意：旧版redis有可能回应“operation not permitted”而不是一个错误码，所以我们需要检查处理这两种出错回应。

**else**

**{**

**redisLog(REDIS\_NOTICE,"Master replied to PING,**

**replication can continue...");**

运行到这，说明已经得到成功的回应，可以开始Replication了。

**}**

**}**

**if(server.masterauth)**

**{**

**err = sendSynchronousCommand(fd,"AUTH",server.masterauth,NULL);**

**if (err)**

**{**

**redisLog(REDIS\_WARNING,"Unable to AUTH to MASTER: %s",err);**

**sdsfree(err);**

**goto error;**

**}**

**}**

如果存在验证密码，则给master发送验证密码。如果出错，则提示“Unable to AUTH to MASTER”。

**sds port = sdsfromlonglong(server.port);**

**err = sendSynchronousCommand(fd,"REPLCONF","listening-port",port, NULL);**

**sdsfree(port);**

**if (err)**

**{**

**redisLog(REDIS\_NOTICE,"(non critical): Master does not understand REPLCONF listening-port: %s", err);**

**sdsfree(err);**

**}**

发送slave的监听端口，这样master的INFO命令就能列出slave当前监听的端口。

**if (syncWrite(fd,"SYNC\r\n",6,server.repl\_syncio\_timeout\*1000) == -1)**

**{**

**redisLog(REDIS\_WARNING,**

**"I/O error writing to MASTER: %s",strerror(errno));**

**goto error;**

**}**

发起同步命令，终于可以replication了。

**while(maxtries--)**

**{**

**snprintf(tmpfile,256,"temp-%d.%ld.rdb",(int)server.unixtime,**

**(long int)getpid());**

**dfd = open(tmpfile,O\_CREAT|O\_WRONLY|O\_EXCL,0644);**

**if (dfd != -1) break;**

**sleep(1);**

**}**

准备一个临时文件来传输rdb文件块。

**if (aeCreateFileEvent(server.el,fd, AE\_READABLE,readSyncBulkPayload,NULL) == AE\_ERR)**

**{**

**redisLog(REDIS\_WARNING, "Can't create readable event for SYNC: %s (fd=%d)", strerror(errno),fd);**

**goto error;**

**}**

开始设置rbd文件块下载事件，slave端准备开始接收来自master的rdb文件。readSyncBulkPayload()函数见下文。

server.repl\_state = REDIS\_REPL\_TRANSFER;

server.repl\_transfer\_size = -1;

server.repl\_transfer\_read = 0;

server.repl\_transfer\_last\_fsync\_off = 0;

server.repl\_transfer\_fd = dfd;

server.repl\_transfer\_lastio = server.unixtime;

server.repl\_transfer\_tmpfile = zstrdup(tmpfile);

return;

更新slave连接状态及初始化其他信息。

error:

close(fd);

server.repl\_transfer\_s = -1;

server.repl\_state = REDIS\_REPL\_CONNECT;

return;

出错标记。

3.8.9 readSyncBulkPayload()函数

这个函数用于slave异步接收来自master的rdb文件块数据。

char buf[4096];

ssize\_t nread, readlen;

off\_t left;

REDIS\_NOTUSED(el);

REDIS\_NOTUSED(privdata);

REDIS\_NOTUSED(mask);

if (server.repl\_state == REDIS\_REPL\_NONE)

{

close(fd);

return;

}

如果server.repl\_transfer\_size 值为-1，我们仍需要从master的回应里读取rdb文件块的长度。

if (server.repl\_transfer\_size == -1)

{

if (syncReadLine(fd,buf,1024,server.repl\_syncio\_timeout\*1000) == -1)

{

redisLog(REDIS\_WARNING,"I/O error reading bulk count from MASTER: %s",

strerror(errno));

goto error;

}

如果读取回应消息失败，则提示后返回。

if (buf[0] == '-')

{

redisLog(REDIS\_WARNING,"MASTER aborted replication with an error: %s",buf+1);

goto error;

}

如果读出的消息包是以“-”开头，则Replication出错，退出。

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 = server.unixtime;

return;

}

如果读出的消息包是以“\0”开头，则Replication出错，退出。

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;

}

如果出的消息包是以“$”开头，则表示消息包出错，退出。

server.repl\_transfer\_size = strtol(buf+1,NULL,10);

通过一些检测，读出文件块的长度，并记录在server.repl\_transfer\_size。

redisLog(REDIS\_NOTICE,"MASTER <-> SLAVE sync: receiving %ld bytes from master",

server.repl\_transfer\_size);

记录必要的日志

return;

}

已经取得文件块大小，该fd再次可读时，则开始接收rdb内容。

**left = server.repl\_transfer\_size - server.repl\_transfer\_read;**

**readlen = (left < (signed)sizeof(buf)) ? 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 = server.unixtime;**

**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 synchronization: %s", strerror(errno));**

**goto error;**

**}**

**server.repl\_transfer\_read += nread;**

**if (server.repl\_transfer\_read >= server.repl\_transfer\_last\_fsync\_off + REPL\_MAX\_WRITTEN\_BEFORE\_FSYNC)**

**{**

**off\_t sync\_size = server.repl\_transfer\_read -server.repl\_transfer\_last\_fsync\_off;**

**rdb\_fsync\_range(server.repl\_transfer\_fd,server.repl\_transfer\_last\_fsync\_off, sync\_size);**

**server.repl\_transfer\_last\_fsync\_off += sync\_size;**

**}**

Redis定时把文件内容写入磁盘，避免堆积到最后时刻一次写入造成一个大的延时。

重复上面的接收读写过程过程，直到rdb文件完全被下载，重命名该临时文件，然后清空数据库。删除master端可读的文件事件。从rdb文件中读取完成建立数据库。

if (server.repl\_transfer\_read == server.repl\_transfer\_size)

{

if (rename(server.repl\_transfer\_tmpfile,server.rdb\_filename) == -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");

signalFlushedDb(-1);

emptyDb(); 清空数据库

加载DB到内存之前，我们需要删除master端可读文件事件，否则它会被递归调用，因为便于非阻塞加载，redLoad()会时刻调用事件循环去处理事件。

aeDeleteFileEvent(server.el,server.repl\_transfer\_s,AE\_READABLE);

if (rdbLoad(server.rdb\_filename) != 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.repl\_state = REDIS\_REPL\_CONNECTED;

redisLog(REDIS\_NOTICE, "MASTER <-> SLAVE sync: Finished with success");

现在已经完成了同步，重启AOF系统，这将引发一个AOF重写，当完成重写，后续操作会以追加到新文件。

if (server.aof\_state != REDIS\_AOF\_OFF)

{

int retry = 10;

stopAppendOnly();

while (retry-- && startAppendOnly() == REDIS\_ERR) {

redisLog(REDIS\_WARNING,"Failed enabling the AOF after successful master synchronization! Trying it again in one second.");

sleep(1);

}

if (!retry)

{

redisLog(REDIS\_WARNING,"FATAL: this slave instance finished the synchronization with its master, but the AOF can't be turned on. Exiting now.");

exit(1);

}

}

return;

error:

replicationAbortSyncTransfer();

return;

出错标记。

3.10 订阅-发布系统 (sub-pub)

在initServer()里，server.pubsub\_channels = dictCreate(&keylistDictType,NULL)这句创建了订阅-发布字典结构。

3.10.1 pubsubSubscribeChannel()函数

功能：客户端订阅某个频道

参数：

redisClient \*c 发起订阅的客户端

robj \*channel 要订阅的频道

struct dictEntry \*de;

list \*clients = NULL;

int retval = 0;

if (dictAdd(c->pubsub\_channels,channel,NULL) == DICT\_OK)

{

首先给确保订阅-发布系统里有这个频道，如果没有就添加该频道。

retval = 1;

incrRefCount(channel);

增加channel的引用计数。

de = dictFind(server.pubsub\_channels,channel);

获取目标频道频道锁对应的客户端链表

if (de == NULL)

{

如果这个链表为空，则说明还没有客户端订阅这个频道

clients = listCreate();

创建频率链表头

dictAdd(server.pubsub\_channels,channel,clients);

把这个链表作为channel对应的value加入字典结构。

incrRefCount(channel);

增加channel的引用计数

} else

{

clients = dictGetVal(de);

}

listAddNodeTail(clients,c);

把客户端加入客户端链表

}

给客户端发送各种通知

**addReply(c,shared.mbulkhdr[3]);**

**addReply(c,shared.subscribebulk);**

**addReplyBulk(c,channel);**

**addReplyLongLong(c,dictSize(c->pubsub\_channels)+listLength(c->pubsub\_patterns));**

**return retval;**

3.10.2 pubsubSubscribeChanne()函数

3.10.3 listCreate()函数

struct list \*list;

if ((list = zmalloc(sizeof(\*list))) == NULL)

{

return NULL;

}

list->head = list->tail = NULL;

list->len = 0;

list->dup = NULL;

list->free = NULL;

list->match = NULL;

return list;

这个函数很简单，申请一个链表头内存空间，

3.11 字典结构dict (dict.c)

3.11.2 重置

*渐进式 rehash 主要由 \_dictRehashStep 和 dictRehashMilliseconds 两个函数进行：*

*\_dictRehashStep 用于对数据库字典、以及哈希键的字典进行被动 rehash ；*

*dictRehashMilliseconds 则由 Redis 服务器常规任务程序（server cron job）执行，用于对数据库字典进行主动 rehash ；*

*\_dictRehashStep每次执行 \_dictRehashStep ，ht[0]->table 哈希表第一个不为空的索引上的所有节点就会全部迁移到 ht[1]->table 。在 rehash 开始进行之后（d->rehashidx 不为 -1）， 每次执行一次添加、查找、删除操作，\_dictRehashStep 都会被执行一次：*

*因为字典会保持哈希表大小和节点数的比率在一个很小的范围内， 所以每个索引上的节点数量不会很多（从目前版本的 rehash 条件来看，平均只有一个，最多通常也不会超过五个）， 所以在执行操作的同时，对单个索引上的节点进行迁移， 几乎不会对响应时间造成影响。*

*dictRehashMilliseconds 可以在指定的毫秒数内， 对字典进行 rehash 。*

*当 Redis 的服务器常规任务执行时，dictRehashMilliseconds 会被执行， 在规定的时间内， 尽可能地对数据库字典中那些需要 rehash 的字典进行 rehash ， 从而加速数据库字典的 rehash 进程（progress）。*

——引自《redis设计与实现》

3.11.3 pushGenericCommand ()函数

**int j, waiting = 0, pushed = 0;**

**robj \*lobj = lookupKeyWrite(c->db,c->argv[1]);**

**从当前数据库的主键空间里查找命令所指向的主键。**

**int may\_have\_waiting\_clients = (lobj == NULL );**

if (lobj && lobj->type != REDIS\_LIST)

{

addReply(c,shared.wrongtypeerr);

return;

}

如果

3.12 有序集合(t\_zset.c)

**有序的实现方式**

当使用**压缩列表**实现，即REDIS\_ENCODING\_ZIPLIST 编码时， 有序集将元素保存到 ziplist 数据结构里面。其中，每个有序集元素以两个相邻的 ziplist 节点表示， 第一个节点保存元素的 member， 第二个元素保存元素的 score 属性。元素按 score 值从小到大排序， 如果两个元素的 score 相同， 那么按字典序对 member进行对比排序。

【注意】当用压缩链表实现的有序集在查找/添加/删除元素都需要从头遍历，算法复杂度均为O(n)。

当使用跳跃表实现 (即编码方式为REDIS\_ENCODING\_SKIPLIST) 时，有序集合使用数据结构zset。Zset同时使用字典和跳跃表两个数据结构来保存节点元素。其中字典是以member为key，而使用

这种双结构集合了二者的各自的优点，使得

有序集可以在 O(1) 复杂度内：

◎检查给定 member 是否存在于有序集（被很多底层函数使用）；

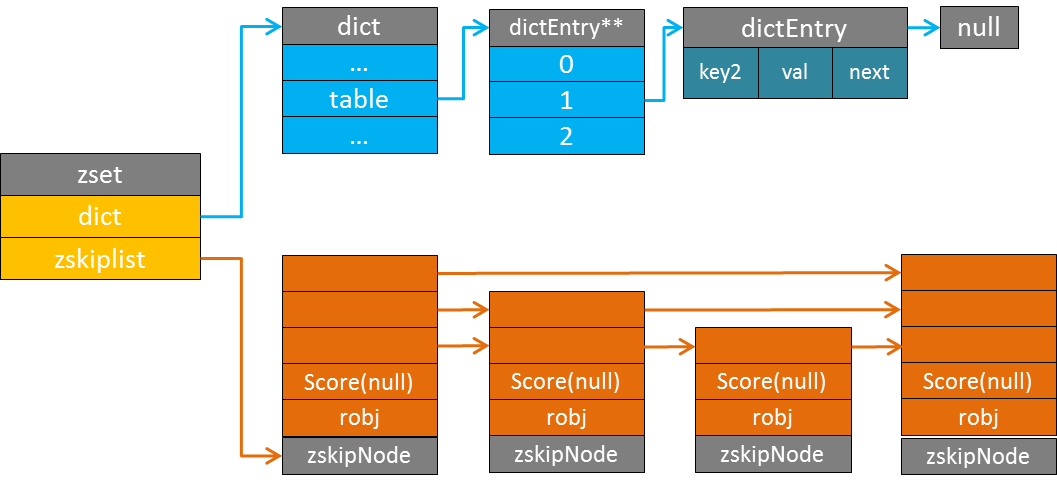
◎取出 member 对应的 score 值（实现 [ZSCORE](http://redis.readthedocs.org/en/latest/sorted_set/zscore.html#zscore) 命令）。

另一方面， 通过使用跳跃表， 可以让有序集支持以下两种操作：

◎在 O(logN) 期望时间、 O(N) 最坏时间内根据 score 对 member 进行定位（被很多底层函数使用）；

◎范围性查找和处理操作，这是（高效地）实现 [ZRANGE](http://redis.readthedocs.org/en/latest/sorted_set/zrange.html#zrange) 、 [ZRANK](http://redis.readthedocs.org/en/latest/sorted_set/zrank.html#zrank) 和 [ZINTERSTORE](http://redis.readthedocs.org/en/latest/sorted_set/zinterstore.html#zinterstore) 等命令的关键。

通过同时使用字典和跳跃表， 有序集可以高效地实现按成员查找和按顺序查找两种操作。



***zslCreate*** *创建一个zskiplist，并添加一个具有最高层数ZSKIPLIST\_MAXLEVEL（代码中定义为32）的节点来管理分层的链表。*

***zslInsert*** *插入一个节点到zskiplist，并调整每一个层级的链表都是有序的。*

***zslDelete*** *从zskiplist删除一个节点，并调整剩余节点在每个层级都是有序的。*

***zslRandomLevel*** *为新加入的节点随机产生一个不超过ZSKIPLIST\_MAXLEVEL的层数。*

**编码选择与转换**

前面提到，zset在内部有两种编码实现方式：ziplist和skiplist。那么Redis究竟是如何选择这两种数据结构的？中途会不会发生转换？

Redis选择编码方式是根据插入的第一个值来决定的。如果插入的第一个元素符合

如下两个条件，则使用ziplist实现，否则用skiplist来实现：

① server.zset\_max\_ziplist\_entries的值大于0(默认为128)；

② 元素的member长度小于server.zset\_max\_ziplist\_value(默认为64)。

Ziplist实现的zset在使用的过程中一旦发生以下情况，则会转换为skiplist：

① ziplist的元素数量大于server.zset\_max\_ziplist\_entries的值(默认128)；

② 新添加元素的member的长度大于server.zset\_max\_ziplist\_value的值(默认为64)。

3.12.1 相关结构

跳跃表节点

typedef struct zskiplistNode

{

robj \*obj;

double score; // --排序的依据

struct zskiplistNode \*backward;

struct zskiplistLevel

{

struct zskiplistNode \*forward;

unsigned int span;

}

level[];

} zskiplistNode;

zskiplistNode结构里有一个内部结构zskiplistLevel。zskiplistNode中包含一个zskiplistLevel数组**level[]**，数组的大小根据节点所在的层数(level)决定。

backward指针是为了方便向后遍历而对skip list做的改进。

跳跃表结构

typedef struct zskiplist

{

struct zskiplistNode \*header, \*tail;

unsigned long length;

int level;

} zskiplist;

typedef struct zset

{

dict \*dict;

zskiplist \*zsl;

} zset;

3.12.2 createZsetObject ()函数

robj \*createZsetObject(void)

{

zset \*zs = zmalloc(sizeof(\*zs));

申请zet结构内存。

zs->dict = dictCreate(&zsetDictType,NULL);

创建有序集合的字典结构,见dict.c讲解。

zs->zsl = zslCreate();

创建有序集合的跳跃表结构，详见下文。

robj \*o = createObject(REDIS\_ZSET,zs);

创建redisObject(robj)对象，并把zset对象复制给robj对象的ptr指针。

o->encoding = REDIS\_ENCODING\_SKIPLIST;

把robj对象的编码改为REDIS\_ENCODING\_SKIPLIST。

return o;

}

3.12.3 zslCreate ()函数

zskiplist\* zsl = zmalloc(sizeof(\*zsl));

创建zskiplist内存结构。

zsl->level = 1;

zsl->length = 0;

zsl->header = zslCreateNode(ZSKIPLIST\_MAXLEVEL,0,NULL);

初始化level为1，长度为0，并创建表头内存结构。zslCreateNode()内部代码为

**zskiplistNode \*zn = zmalloc(sizeof(\*zn)+level\*sizeof(struct zskiplistLevel));**

**zn->score = score;**

**zn->obj = obj;**

申请的内存大小为zskiplistNode的结构内存占用，外加level个zskiplistLeveld的内存结构。level默认值是ZSKIPLIST\_MAXLEVEL(32)，score初始化为0。

for ( int j = 0; j < ZSKIPLIST\_MAXLEVEL; j++)

{

zsl->header->level[j].forward = NULL;

zsl->header->level[j].span = 0;

}

zsl->header->backward = NULL;

zsl->tail = NULL;

上面都是初始化。

return zsl;

3.12.4 zslInsert()函数 ★★★

这个函数是跳跃表算法的集中体现，非常重要。zslInsert和zslDelete函数都需要首先查找到合适的位置或节点，查找的代码很简单，直接包含在了这两个函数内：

zskiplistNode \*update[ZSKIPLIST\_MAXLEVEL], \*x;

unsigned int rank[ZSKIPLIST\_MAXLEVEL]; // --更新span需要用到rank

int i, level;

redisAssert(!isnan(score));

x = zsl->header;

for (i = zsl->level-1; i >= 0; i--)

{

rank[i] = i == (zsl->level-1) ? 0 : rank[i+1];

除了rank[zsl->level-1]=0,其他的rank[i]初始时等于rank[i+1]

while (x->level[i].forward && (x->level[i].forward->score < score ||

(x->level[i].forward->score == score &&

compareStringObjects(x->level[i].forward->obj,obj) < 0)))

是从zskiplist现有的最高层开始向前，并在查找的过程中根据规则转向低层的链表继续，一直到skip list的最低层为止。同时看到redis的实现中允许相同的score存在(这时按对象的字符串进行比较），但不允许具有相同值的对象并存（集合的特性）

{

rank[i] += x->level[i].span;

x = x->level[i].forward;

}

update[i] = x;

}

这里假定集合里没有相同的key，既然Redis允许有相同的score，那么重复插入key的情况就不应该发生，在调用zslInsert()的时候，就需要判断这个key是否已经在集合里。

level = zslRandomLevel();

if (level > zsl->level)

{

for (i = zsl->level; i < level; i++)

{

rank[i] = 0;

update[i] = zsl->header;

update[i]->level[i].span = zsl->length;

}

zsl->level = level;

}

x = zslCreateNode(level,score,obj);

for (i = 0; i < level; i++)

{

x->level[i].forward = update[i]->level[i].forward;

update[i]->level[i].forward = x;

/\* update span covered by update[i] as x is inserted here \*/

x->level[i].span = update[i]->level[i].span - (rank[0] - rank[i]);

update[i]->level[i].span = (rank[0] - rank[i]) + 1;

}

/\* increment span for untouched levels \*/

for (i = level; i < zsl->level; i++)

{

update[i]->level[i].span++;

}

x->backward = (update[0] == zsl->header) ? NULL : update[0];

if (x->level[0].forward)

{ x->level[0].forward->backward = x; }

else

{ zsl->tail = x; }

zsl->length++;

return x;

查找是从zskiplist现有的最高层开始向前，并在查找的过程中根据规则转向低层的链表继续，一直到skip list的最低层为止。同时看到redis的实现中允许相同的score存在（这时按对象的字符串进行比较），但不允许具有相同值的对象并存（集合的特性）。

下面通过一个例子来说明skip list的建立过程。

按顺序执行下列语句：

***zslInsert(zsl, 5, obj1); //level=1;***

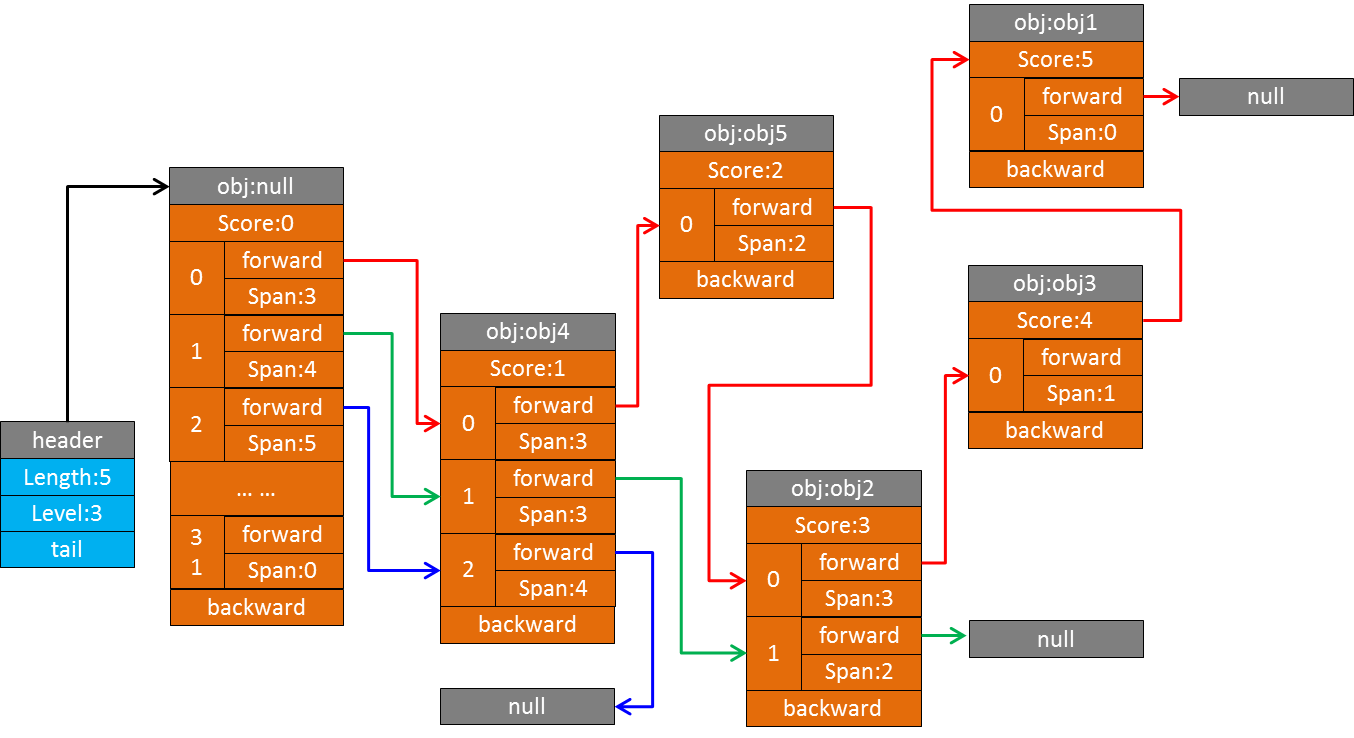
***zslInsert(zsl, 3, obj2); //level=2;***

***zslInsert(zsl, 4, obj3); //level=1;***

***zslInsert(zsl, 1, obj4); //level=3;***

***zslInsert(zsl, 2, obj5); //level=1;***

现在的zsl结构如下图所示，其中level array的数组下标是为了图例更直观，实际不占存储空间。为了保证图例的简洁，backward的指针没有画出，对应level 0红色指针相反方向的指针。



**编码的选择与切换**

集合的底层实现到底是使用字典还是使用整数集合取决于第一个添加到集合的元素的类型，如果这个元素是整数类型，那么集合就被初始化为REDIS\_ENCODING\_INTSET类型，即用整数集实现。否则编码就是REDIS\_ENCODING\_HT，用字典实现。

使用整数集合实现的集合在下列条件被触发时，集合被转换成字典类型。

①intset 保存的整数值个数超过 server.set\_max\_intset\_entries （默认值为 512 ）。

②集合里添加一个非整形元素。

3.13 压缩列表(ziplist.c)

压缩列表不是一种独立的数据结构，而是链表的一种变体，为节省内存而设计。可以存储字符串和整数值。整数字符串将被编码为整数，而不是一个字符序列。压缩列表允许在头尾都进行O(1)复杂度的push和pop操作。但是每次操作都需要对已用内存进行一次再分配，所以压缩列表的实际复杂度取决于已经使用的内存大小。

布局：

压缩列表的通常内存布局：

zlbytes

(4 bytes)

zltail

(4 bytes)

zllen

(2 bytes)

header

entry1

…

entryn

entry1

end

header

pre\_entry\_length

encoding

length

entry

end

zlend

ZIPLIST\_ENTRY\_HEAD

ZIPLIST\_ENTRY\_TAIL

ZIPLIST\_ENTRY\_END

Zlbytes：表示压缩列表占总内存的字节数

Zltail： 到压缩表尾节点的偏移量

Zllen：压缩表中节点的数量；

Zlend：用于标记ziplist的末端;

各个节点以及的。

typedef struct zlentry

{

unsigned int prevrawlensize;

unsigned int prevrawlen;

unsigned int lensize, len;

unsigned int headersize;

unsigned char encoding;

unsigned char \*p;

} zlentry;

3.13.2 ziplistNew ()函数

功能：创建压缩列表

参数: null

返回: 压缩列表地址

unsigned int bytes = ZIPLIST\_HEADER\_SIZE+1;

ZIPLIST\_HEADER\_SIZE 的大小为sizeof(uint32\_t)\*2+sizeof(uint16\_t)，

unsigned char \*zl = zmalloc(bytes);

分配压缩列表头内存

ZIPLIST\_BYTES(zl) = intrev32ifbe(bytes);

ZIPLIST\_BYTES是一个宏函数,其定义为: #define ZIPLIST\_BYTES(zl) (\*((uint32\_t\*)(zl)))

intrev32ifbe 也是一个宏函数，其定义为：

#if (BYTE\_ORDER == LITTLE\_ENDIAN)

#define intrev32ifbe(v) (v)

#else

#define intrev32ifbe(v) intrev32(v)

intrev32ifbe()是把数字的大端编码转为小端编码，如果运行机器是小端，这个宏就不做任何事情。

在x86或x64平台上，上面代码可以简化为: \*((uint32\_t\*)(zl)) = bytes。

这句的意思就是**把zl指向的前四个字节内存赋值为11**；

ZIPLIST\_TAIL\_OFFSET(zl) = intrev32ifbe(ZIPLIST\_HEADER\_SIZE);

在x86或x64平平台上上面代码可以简化为:

\*((uint32\_t\*)(zl+4) = 10;

**把zl指向的第5到第8的四个字节的内存赋值为10；**

ZIPLIST\_LENGTH(zl) = 0;

同上，\*((uint16\_t\*)((zl)+8)) = 0;

**把zl指向的第9、第10两个字节的内存赋值为0；**

zl[bytes-1] = ZIP\_END;

**把zl指向的第11个字节的内存赋值为255；**

执行完这条语句，bz在内存中的数值应该为

0b 00 00 00 0a 00 00 00 00 00 ff

return zl;

3.13.3 \_\_ziplistInsert ()函数

static unsigned char \*\_\_ziplistInsert(unsigned char \*zl, unsigned char \*p, unsigned char \*s, unsigned int slen)

参数:

size\_t curlen = intrev32ifbe(ZIPLIST\_BYTES(zl)), reqlen, prevlen = 0;

size\_t offset;

int nextdiff = 0;

unsigned char encoding = 0;

long long value = 123456789;

初始化以防警告，用一个魔数，方便检测出某些情况下我们没赋值就使用。

zlentry entry, tail;

如果 p 之后不是没有节点（不是插入到末端）那么取出节点相关资料，以及 prevlen

if ( p[0] != ZIP\_END )

{

entry = zipEntry(p);

prevlen = entry.prevrawlen;

}

else

{

unsigned char \*ptail = ZIPLIST\_ENTRY\_TAIL(zl);

获取列表最后一个节点（表尾）的地址

if (ptail[0] != ZIP\_END)

{

prevlen = zipRawEntryLength(ptail);

}

如果地址之后不是末端（也即是，列表至少有一个节点）, 保存 ptail 指向的节点的空间长度。

}

查看将新值是否支持编码为整数，如果可以的话返回1 ，并将新值保存到value，编码形式保存到 encoding

if (zipTryEncoding(s,slen,&value,&encoding))

{

reqlen = zipIntSize(encoding);

可以保存为整数，那么继续计算保存它所需的空间

}

else

{

不能编码为整数，直接使用字符串长度

reqlen = slen;

}

reqlen += zipPrevEncodeLength(NULL,prevlen);

计算编码前面节点所需的长度(prevlen)

reqlen += zipEncodeLength(NULL,encoding,slen);

计算编码当前节点所需的长度(slen)

/\* When the insert position is not equal to the tail, we need to

\* make sure that the next entry can hold this entry's length in

\* its prevlen field. \*/

nextdiff = (p[0] != ZIP\_END) ? zipPrevLenByteDiff(p,reqlen) : 0;

如果添加的位置不是表尾，那么必须确定后继节点的 prevlen 空间足以保存新节点的编码长度

zipPrevLenByteDiff 的返回值有三种可能：

1）新旧两个节点的编码长度相等，返回 0

2）新节点编码长度 > 旧节点编码长度，返回 5 - 1 = 4

3）旧节点编码长度 > 新编码节点长度，返回 1 - 5 = -4

offset = p-zl;

保存偏移量，因为重分配空间有可能改变zl的内存地址

zl = ziplistResize(zl,curlen+reqlen+nextdiff);

重分配空间，并更新长度属性和表尾，新空间长度 = 现有长度 + 新节点所需长度 + 编码新节点长度所需的长度差 O(N)

p = zl+offset;

更新p的指针

if (p[0] != ZIP\_END)

如果新节点不是添加到列表末端，那么它后面就有其他节点，因此，我们需要移动这部分节点

{

/\* Subtract one because of the ZIP\_END bytes \*/

memmove(p+reqlen,p-nextdiff,curlen-offset-1+nextdiff);

向右移动移原有数据，为新节点让出空间，O(N)的复杂度。

zipPrevEncodeLength(p+reqlen,reqlen);

将本节点的长度编码至下一节点

ZIPLIST\_TAIL\_OFFSET(zl) = intrev32ifbe(intrev32ifbe(ZIPLIST\_TAIL\_OFFSET(zl))+reqlen);

更新压缩表的表尾偏移量

/\* When the tail contains more than one entry, we need to take

\* "nextdiff" in account as well. Otherwise, a change in the

\* size of prevlen doesn't have an effect on the \*tail\* offset. \*/

有需要的话，将nextdiff 也加上到zltail上。

tail = zipEntry(p+reqlen);

if (p[reqlen+tail.headersize+tail.len] != ZIP\_END)

{

ZIPLIST\_TAIL\_OFFSET(zl) =intrev32ifbe(intrev32ifbe(ZIPLIST\_TAIL\_OFFSET(zl))+nextdiff);

}

}

else

{

ZIPLIST\_TAIL\_OFFSET(zl) = intrev32ifbe(p-zl);

更新 ziplist 的 zltail 属性，现在新添加节点为表尾节点

}

/\* When nextdiff != 0, the raw length of the next entry has changed, so

\* we need to cascade the update throughout the ziplist \*/

if (nextdiff != 0)

{

offset = p-zl;

zl = \_\_ziplistCascadeUpdate(zl,p+reqlen); // O(N^2)

p = zl+offset;

}

下面写入数据到节点

p += zipPrevEncodeLength(p,prevlen);

编码上一节点的长度，并向后移动指针

p += zipEncodeLength(p,encoding,slen);

编码本节点的长度和类型，并向后移动指针

if (ZIP\_IS\_STR(encoding))

{

memcpy(p,s,slen); 如果是字符串，直接用memcpy拷贝入内容到节点

}

else

{

zipSaveInteger(p,value,encoding);

}

ZIPLIST\_INCR\_LENGTH(zl,1);

更新节点数量

return zl;

返回压缩列表指针。

3.14 整数集合(ziplist.c)

整数集合是集合的底层实现之一，当集合中的元素是整数，而且元素个数不是很多时，底层实现选择用整数集合(intset)。整数集合的元素是有序(从小到大)且不重复的。

typedef struct intset

{

uint32\_t encoding; *// --元素编码类型*

uint32\_t length; // --元素个数

int8\_t contents[]; // --元素数组,元素的容器

} intset;

encoding 的类型有：

#define INTSET\_ENC\_INT16 (sizeof(int16\_t)) 16位整型

#define INTSET\_ENC\_INT32 (sizeof(int32\_t)) 32位整型

#define INTSET\_ENC\_INT64 (sizeof(int64\_t)) 64位整型

robj \*setTypeCreate(robj \*value)

{

if (isObjectRepresentableAsLongLong(value,NULL) == REDIS\_OK)

{

return createIntsetObject();

}

return createSetObject();

}

当创建一个集合时，会调用到t\_set.c的setTypeCreate()的函数，该函数首先用isObjectRepresentableAsLongLong()判断一下插入的这个数是不是一个整型，如果是整型就用createIntsetObject()创建整数集合，否则就用createSetObject()创建字典。isObjectRepresentableAsLongLong()里注意调用util.c的string2ll()函数来判断一个字符串究竟是否可以转换为整数。

3.14.2 string2ll()函数

**参数**:

const char \*s, 目标字符串指针

size\_t slen, 目标字符串长度

long long \*value 转换后的值

**返回值：**

0表示字符串不可被转为整数，1表示可以。

**源码:**

const char \*p = s; // --指向目标字符串

size\_t plen = 0; // --已经处理的字符数

int negative = 0; // --是否为负数

unsigned long long v; // --转换后的

if (plen == slen){ return 0; }

如果字符串长度为0，则直接返回0。

if (slen == 1 && p[0] == '0')

{

if (value != NULL) \*value = 0;

return 1;

}

特别情况，只有一位且为0，直接把value赋值0，然后返回1。

if (p[0] == '-')

{

negative = 1;

p++; plen++;

if (plen == slen){ return 0; }

}

如果首字符是负号，则标记为负数，如果全部字符串只是一个负号，则返回0。

if (p[0] >= '1' && p[0] <= '9')

{

v = p[0]-'0'; // --通过减去字符“0”的ascii值，把字符的ascii值变成数值。

p++; // --字符串指针向后跳一个字符

plen++;// --处理的字符数增加1。

}

else if (p[0] == '0' && slen == 1)

{

\*value = 0;

return 1;

}

else

{

return 0;

}

只有两种条件，一个字符串有可能转为整数值，不满足下面两个条件直接返回0:

① 首字符必须是1-9

② 整个串是0字符

这第二个条件与函数开头的判断重复了。

while (plen < slen && p[0] >= '0' && p[0] <= '9')

{

if ( v > ( ULLONG\_MAX / 10 ) )

{

return 0;

}

如果如果 v \* 10 > ULLONG\_MAX，则值溢出，直接返回0。

v \*= 10;

if ( v > ( ULLONG\_MAX - ( p[0]-'0' ) ) )

{

return 0;

}

如果如果v + (p[0]-'0') > ULLONG\_MAX，则值溢出，直接返回0。

**v += p[0]-'0';**

**p++;**

**plen++;**

**}**

**if (plen < slen){ return 0; }**

如果不是每个字符都被成功转换，则返回0。

**if ( negative)**

**{**

**if ( v > ((unsigned long long)(-(LLONG\_MIN+1))+1) ){ return 0; }**

如果结果值小于最小阙值，返回0。

**if (value != NULL){ \*value = -v; }**

将结果取反后复制value。

**}**

**else**

**{**

**if (v > LLONG\_MAX){ return 0; }**

**if (value != NULL){ \*value = v; }**

**}**

3.14.3 createIntsetObject()函数

这个函数很简单，一个四行代码。

intset \*is = intsetNew();

创建一个intset对象。看一下intsetNew()的实现，也比较简单:

intset \*is = zmalloc(sizeof(intset));

is->encoding = intrev32ifbe(INTSET\_ENC\_INT16);

is->length = 0;

return is;

分配intset对象的内存空间，将该对象的编码设为INTSET\_ENC\_INT16，并初始化元素长度后返回该对象的地址。

robj \*o = createObject(REDIS\_SET,is);

创建一个基本类型对象，并把刚创建的intset对象的地址赋值给基本类型对象的ptr指针。

o->encoding = REDIS\_ENCODING\_INTSET;

将基本类型对象的编码类型设置为REDIS\_ENCODING\_INTSET。

return o;

返回这个基本对象的地址

*【注意】Contents 数组的 int8\_t 类型声明比较容易让人误解，实际上， intset 并不使用 int8\_t 类型来保存任何元素，结构中的这个类型声明只是作为一个占位符使用：在对 contents 中的元素进行读取或者写入时，程序并不是直接使用 contents 来对元素进行索引，而是根据 encoding 的值，对 contents 进行类型转换和指针运算，计算出元素在内存中的正确位置。在添加新元素，进行内存分配时，分配的空间也是由 encoding 的值决定。*

3.14.4 t\_set.c::saddCommand()函数

robj \*set;

int j, added = 0;

set = lookupKeyWrite(c->db,c->argv[1]);

根据key名称到key空间查找相应的set。

if (set == NULL)

{

set = setTypeCreate(c->argv[2]);

dbAdd(c->db,c->argv[1],set);

}

如果没有这个set，就创建一个set。setTypeCreate()函数上面介绍过。

else

{

if (set->type != REDIS\_SET)

{

addReply(c,shared.wrongtypeerr);

return;

}

如果基本数据对象的类型不是REDIS\_SET，就退出。

}

for (j = 2; j < c->argc; j++)

{

c->argv[j] = tryObjectEncoding(c->argv[j]);

if (setTypeAdd(set,c->argv[j])) added++;

循环加入参数里的所有给出元素。

}

if (added) signalModifiedKey(c->db,c->argv[1]);

每当数据库的key被改变，为了实现“WATCH”机制。

详细见db.c。

server.dirty += added;

记录一下从上次保存后的更改次数。

【改进】server.dirty += added这一句完全可以放入if{}语句里，可以省略一次计算。

addReplyLongLong(c,added);

3.14.5 t\_set.c:: setTypeAdd()函数

long long llval;

if (subject->encoding == REDIS\_ENCODING\_HT)

{

if (dictAdd(subject->ptr,value,NULL) == DICT\_OK)

{

incrRefCount(value);

return 1;

}

}

如果目标集合的编码是REDIS\_ENCODING\_HT(即集合底层以字典方式实现)，就直接给字典添加键-值元素，并增加值得引用次数。

else if ( subject->encoding == REDIS\_ENCODING\_INTSET )

{

if (isObjectRepresentableAsLongLong(value,&llval) == REDIS\_OK )

{

uint8\_t success = 0;

subject->ptr = intsetAdd(subject->ptr,llval,&success);

if (success)

{

if (intsetLen(subject->ptr) > server.set\_max\_intset\_entries)

{ setTypeConvert(subject,REDIS\_ENCODING\_HT); }

return 1;

}

}

在目标集合编码是REDIS\_ENCODING\_INTSET(即集合底层以intset方式实现)。

首先判断要插入的值是否为整数，如果是，则先把值插入到整数结合，然后比较当前整数集合元素数与server.set\_max\_intset\_entries值比较，如果超过这个值，则把集合底层实现从字典转换成整数集合。

else

{

setTypeConvert(subject,REDIS\_ENCODING\_HT);

redisAssertWithInfo(NULL,value,dictAdd(subject->ptr,value,NULL)

== DICT\_OK);

incrRefCount(value);

return 1;

}

如果要插入的值是否为整数则需要把目标集合转换为以字典来实现，然后把元素插入字典容器。

}

else

{

redisPanic("Unknown set encoding");

}

return 0;

如果目标集合的编码形式既非REDIS\_ENCODING\_HT，也非REDIS\_ENCODING\_INTSET，则返回错误。

3.14.6 t\_set.c:: setTypeConvert()函数

**返回**:null

**参数**：

robj \*setobj 目标集合对象指针

int enc 目标集合要转向的编码类型，实际上这个参数是多余，只能把REDIS\_ENCODING\_INTSET转换成REDIS\_ENCODING\_HT。

**功能**：转换一个set的编码类型。

setTypeIterator \*si;

redisAssertWithInfo( NULL,setobj,setobj->type == REDIS\_SET && setobj->encoding == REDIS\_ENCODING\_INTSET );

if ( enc == REDIS\_ENCODING\_HT )

{

int64\_t intele;

dict \*d = dictCreate(&setDictType,NULL);

robj \*element;

创建新的字典对象。

dictExpand(d,intsetLen(setobj->ptr));

用集合的大小，预先扩展字典，防止哈希重置。

si = setTypeInitIterator(setobj);

创建集合的迭代器。

while (setTypeNext(si,NULL,&intele) != -1)

{

element = createStringObjectFromLongLong(intele);

redisAssertWithInfo(NULL,element,dictAdd(d,element,NULL) == DICT\_OK);

}

迭代整个集合，根据取出的整数元素创建sds对象，并加入到字典里。

setTypeReleaseIterator(si);

释放集合的迭代器。

setobj->encoding = REDIS\_ENCODING\_HT;

更改编码。

zfree(setobj->ptr);

setobj->ptr = d;

释放原来的整数集合内存空间，然后把字典对象指针赋值给新集合键对象的ptr指针。

}

else

{

redisPanic("Unsupported set conversion");

}

**内存管理**

内存管理对于rendis这样的系统来说自然非常重要。Redis的内存管理功能在Zmalloc.h 和Zmalloc.c两个文件来实现。Redis屏蔽了不同平台的差异，对外提供统一封装的接口

Redis用宏定义来巧妙地实现底层平台的差异。

相关代码如下：

#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) || (TC\_VERSION\_MAJOR > 1)

#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))

#include <jemalloc/jemalloc.h>

#if (JEMALLOC\_VERSION\_MAJOR == 2 && JEMALLOC\_VERSION\_MINOR >= 1) || (JEMALLOC\_VERSION\_MAJOR > 2)

#define HAVE\_MALLOC\_SIZE 1

#define zmalloc\_size(p) je\_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

#ifndef ZMALLOC\_LIB

#define ZMALLOC\_LIB "libc"

#endif

#ifndef HAVE\_MALLOC\_SIZE

size\_t zmalloc\_size(void \*ptr);

#endif

如果系统中有google的tc\_malloc库，那么使用使用tc\_malloc的接口体改malloc接口；如果系统中有facebook的jemalloc库，则用je\_mallo接口代替malloc接口；如果当前系统是Mac系统则使用<malloc/malloc.h>中的内存分配接口。其他情况，在每一段分配好的空间前头，同时多分配一个定长的字段，用来记录分配的空间大小。

Redis优先尝试选择google的tc\_malloc来进行管理，如果系统中没有facebook的jemalloc库，则用je\_mallo接口代替malloc接口；如果当前系统是Mac系统则使用<malloc/malloc.h>中的内存分配接口。其他情况，在每一段分配好的空间前头，同时多分配一个定长的字段，用来记录分配的空间大小。

tc\_malloc是google开源处理的一套内存管理库，是用C++实现的，主页在这里。TCMalloc给每个线程分配了一个线程局部缓存。小分配可以直接由线程局部缓存来满足。需要的话，会将对象从中央数据结构移动到线程局部缓存中，同时定期的垃圾收集将用于把内存从线程局部缓存迁移回中央数据结构中。这篇文章里对TCMalloc有个详细的介绍。

jemalloc 也是一个内存创管理库，其创始人Jason Evans也是在FreeBSD很有名的开发人员，参见这里。Jemalloc聚集了malloc的使用过程中所验证的很多技术。忽略细节，从架构着眼，最出色的部分仍是arena和thread cache。

读者一定会有疑问系统不是有了malloc 吗，为什么还有这样的内存管理库？？ 由于经典的libc的分配器碎片率为较高，可以查看这篇文章的分析，关于内存碎片不太了解的童鞋请参考这里， malloc 和free 怎么工作的参考这里。 关于ptmalloc,tcmalloc和jemalloc内存分配策略的一篇总结不错的文章，请点这里。

下面介绍redis封装的内存管理相关函数，src/zmalloc.h有相关声明。

[?](http://www.2cto.com/database/201405/302960.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | void \*zmalloc(size\_t size);//malloc  void \*zcalloc(size\_t size);//calloc  void \*zrealloc(void \*ptr, size\_t size);/realloc  void zfree(void \*ptr);//free  char \*zstrdup(const char \*s);  size\_t zmalloc\_used\_memory(void);  void zmalloc\_enable\_thread\_safeness(void);  void zmalloc\_set\_oom\_handler(void (\*oom\_handler)(size\_t));  float zmalloc\_get\_fragmentation\_ratio(void);  size\_t zmalloc\_get\_rss(void);  size\_t zmalloc\_get\_private\_dirty(void);  void zlibc\_free(void \*ptr); |

现在主要介绍下redis内存分配函数 void \*zmalloc(size\_t size)，其对应的声明形式如下：

[?](http://www.2cto.com/database/201405/302960.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | void \*zmalloc(size\_t size) {      void \*ptr = malloc(size+PREFIX\_SIZE);        if (!ptr) zmalloc\_oom\_handler(size);  #ifdef HAVE\_MALLOC\_SIZE      update\_zmalloc\_stat\_alloc(zmalloc\_size(ptr));      return ptr;  #else      \*((size\_t\*)ptr) = size;      update\_zmalloc\_stat\_alloc(size+PREFIX\_SIZE);      return (char\*)ptr+PREFIX\_SIZE;  #endif  } |

[阅读](http://book.2cto.com/)源码我们发现有个PREFIX\_SIZE 宏，其宏定义形式如下：

[?](http://www.2cto.com/database/201405/302960.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\* zmalloc.c \*/  #ifdef HAVE\_MALLOC\_SIZE  #define PREFIX\_SIZE (0)  #else  #if defined(\_\_sun)  #define PREFIX\_SIZE (sizeof(long long))  #else  #define PREFIX\_SIZE (sizeof(size\_t))  #endif  #endif |

结合src/zmalloc.h有相关宏声明，我们发现，因为 tc\_malloc 、je\_malloc 和 Mac平台下的 malloc 函数族提供了计算已分配空间大小的函数（分别是tc\_malloc\_size, je\_malloc\_usable\_size和malloc\_size），所以就不需要单独分配一段空间记录大小了。在linux和sun平台则要记录分配空间大小。对于linux，使用sizeof(size\_t)定长字段记录；对于sun [系统](http://www.2cto.com/os/)，使用sizeof(long long)定长字段记录，其对应源码中的 PREFIX\_SIZE 宏。

PREFIX\_SIZE 有什么用呢？

为了统计当前进程到底占用了多少内存。在 zmalloc.c 中，有一个静态变量：

[?](http://www.2cto.com/database/201405/302960.html)

|  |  |
| --- | --- |
| 1 | static size\_t used\_memory = 0; |

这个变量它记录了进程当前占用的内存总数。每当要分配内存或是释放内存的时候，都要更新这个变量(当然可以是线程安全的)。因为分配内存的时候，需要指定分配多少内存。但是释放内存的时候，（对于未提供malloc\_size函数的内存库）通过指向要释放内存的指针是不能知道释放的空间到底有多大的。这时候，上面提到的PREFIX\_SIZE就起作用了，可以通过其中记录的内容得到空间的大小。（不过在linux系统上也有相应的函数获得分配内存空间的大小,参见这里）。

通过zmalloc的源码我们可以发现，其分配空间代码为void \*ptr = malloc(size+PREFIX\_SIZE); 显然其分配空间大小为:size+PREFIX\_SIZE ，对于使用tc\_malloc或je\_malloc的情况或mac系统，其 PREFIX\_SIZE 为0。当分配失败时有相应的出错处理 。

前面我们已经说过redis通过使用used\_memory 的变量来统计当前进程到底占用了多少内存，因此在分配和释放内存时我们需要紧接着更新used\_memory 的相应值，对应到redis源码中为：

[?](http://www.2cto.com/database/201405/302960.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #ifdef HAVE\_MALLOC\_SIZE      update\_zmalloc\_stat\_alloc(zmalloc\_size(ptr));      return ptr;  #else      \*((size\_t\*)ptr) = size;      update\_zmalloc\_stat\_alloc(size+PREFIX\_SIZE);      return (char\*)ptr+PREFIX\_SIZE;  #endif |

上面的代码有事宏预处理 #ifdef HAVE\_MALLOC\_SIZE 显然是上面我们说过的利用的tc\_malloc je\_malloc Mac等提供malloc\_size函数的情形，我们可以很容易得知分配内存的大小通过统一化的malloc\_size函数即可。但是对于没有提供malloc\_size功能的函数，redis是怎么处理的呢？看上面的源码 #else下面的代码即是其实现，其对应的内存结构如下：

|  |  |
| --- | --- |
| prefix-size | memory size |

分配的内存前加一个固定大小的prefis-size空间，用于记录该段内存的大小，size所占据的内存大小是已知的，为size\_t类型的长度，因此通过\*((size\_t\*)ptr) = size; 即可对当前内存块大小进行指定。每次分配内存后，返回的实际地址指针为指向memorysize的地址( (char\*)ptr+PREFIX\_SIZE; )，通过该指针，可以很容易的计算出实际内存的头地址，从而释放内存。

redis通过update\_zmalloc\_stat\_alloc(\_\_n,\_\_size) 和 update\_zmalloc\_stat\_free(\_\_n) 这两个宏负责在分配内存或是释放内存的时候更新used\_memory变量。update\_zmalloc\_stat\_alloc定义如下：

[?](http://www.2cto.com/database/201405/302960.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #define update\_zmalloc\_stat\_alloc(\_\_n) do { \      size\_t \_n = (\_\_n); \      if (\_n&(sizeof(long)-1)) \_n += sizeof(long)-(\_n&(sizeof(long)-1)); \      if (zmalloc\_thread\_safe) { \          update\_zmalloc\_stat\_add(\_n); \      } else { \          used\_memory += \_n; \      } \  } while(0) |

redis把这个更新操作写成宏的形式主要是处于效率的考虑。

上面的代码中

A，if (\_n&(sizeof(long)-1)) \_n += sizeof(long)-(\_n&(sizeof(long)-1));  
主要是考虑对齐问题，保证新增的\_n 是 sizeof(long)的倍数。

B, if (zmalloc\_thread\_safe) { \  
update\_zmalloc\_stat\_add(\_n); \

}

如果进程中有多个线程存在，并保证线程安全zmalloc\_thread\_safe，则在更新变量的时候要加锁。 通过宏HAVE\_ATOMIC选择相应的同步机制。

zmalloc\_calloc、zmalloc\_free等的实现就不仔细介绍了详情参见源码。

最后讲解下 zmalloc\_get\_rss()函数。  
这个函数用来获取进程的RSS。神马是RSS？全称为Resident Set Size，指实际使用物理内存（包含共享库占用的内存）。在linux系统中，可以通过读取/proc/pid/stat文件系统获取，pid为当前进程的进程号。读取到的不是byte数，而是内存页数。通过系统调用sysconf(\_SC\_PAGESIZE)可以获得当前系统的内存页大小。 获得进程的RSS后，可以计算目前数据的内存碎片大小，直接用rss除以used\_memory。rss包含进程的所有内存使用，包括代码，共享库，堆栈等。 哪来的内存碎片？上面我们已经说明了通常考虑到效率，往往有内存对齐等方面的考虑，所以，碎片就在这里产生了。相比传统glibc中的malloc的内存利用率不是很高一般会使用别的内存库系统。在redis中默认的已经不使用简单的malloc了而是使用 jemalloc， 在源文件src/Makefile下有这样一段代码:

[?](http://www.2cto.com/database/201405/302960.html)

|  |  |
| --- | --- |
| 1 | ifeq ($(uname\_S),Linux) MALLOC=jemalloc |

可以知道在linux系统上默认使用jemalloc, 在redis发布的源码中有相关的库 deps/jemalloc 。

总的来说 redis则完全自主分配内存，在请求到的时候实时根据内建的算法分配内存，完全自主控制内存的管理。简单即是没吧，不过功能确实强大。

【函数】

| 函数 | 作用 | 算法复杂度 |
| --- | --- | --- |
| intsetNew | 创建 intset | O(1) |
| intsetAdd | 添加新元素（不升级） | O(N) |
| intsetUpgradeAndAdd | 添加新元素（升级） | O(N) |
| \_intsetGet | 按索引获取元素 | O(1) |
| \_intsetSet | 按索引设置元素 | O(1) |
| intsetSearch | 查找元素，返回索引 | O(lgN) |
| intsetRemove | 删除元素 | O(N) |

第四章 关于redis的一些感悟

**Redis的一些先进经验总结：**

1. KISS：无时不刻体现了KISS原则。
2. 自我改良，哈希重置(Rehash)，AOF重写(AOF Rewrite)都体现了这种思想。

改良的代价远远小于革命，对现有系统冲击非常有限。在保证现有系统的可用情况下，通过自我迭代式的改良，量变到质变，使得系统进化。

1. 事件驱动
2. 出神入化的指针使用：

**Redis当前存在的一些缺陷**

1. 内存吞噬：慢日志、
2. 单线程 redis虽然宣称主从复制无阻塞，但是，由于redis使用单线程服务，而和slave的交互由处理线程统一处理，因此，对性能有影响。在slave第一次和master做同步时，如果master快照文件较大，则快照文件的传输将耗费较长时间，文件传输过程中master无法提供服务。
3. 一些具体实现效率不高：如时间事件的处理，每次都是从时间事件链表里取出一个触发期距离当前时最近的一个，然后做处理，算法复杂度为O(n)。
4. 只支持单向同步
5. 不支持集群