Redis的设计与实现

简单动态字符串

SDS除了被用于保存字符串以外，还被用作缓冲区。AOF模块中的AOF缓冲区。

```
struct sdshdr{
   //记录buf数组中已经使用的字节的数量
   //等于sds所保存的字符串长度
   int len;
   //记录buf数组中未使用字节的数量
   int free;
   //字节数组,用于保存字符串
   char buf[];
}
```

为什么这么设计sds：

1. 获取字符长度的时间复杂度为1.
2. 杜绝缓冲区溢出
3. 减少修改字符串时带来的内存重分配次数。

SDS的空间分配策略：

1. 如果对SDS进行修改之后，SDS的长度（也就是len属性的值）将小于1MB，那么程序分配和len属性同样大小的未分配空间。
2. 如果对SDS进行修改之后，SDS的长度将大于等于1MB，那么程序会分配1MB的未使用空间。

惰性空间释放

当SDS的API需要缩短SDS保存的字符串时，程序并不会立即使用内存重分配来回收多出来的字节，而是使用free将这些字节的数量记录起来，并等待使用。

链表

因为C语言没有链表的内部数据结构，所以Redis自己实现了链表。

链表节点：

```
typedef struct listNode{
   //前置节点
   struct listNode *prev;
   //后置节点
   struct listNode *next;
   //节点的值
   void *value;
}listNode;
```

链表：

```
typedef struct list{
   //表头节点
   listNode *head;
   //表尾节点
   listNode *tail;
   //链表所包含的节点数量
   unsigned long len;
   //节点复制函数
   void *(*dup)(void *ptr);
   //节点释放函数
   void (*free)(void *ptr);
   //节点值对比函数
   int (*match)(void *ptr,void *key);
}list;
```

Redis链表的实现特性：

双端：链表节点带有prev和next指针，获取某个节点的前置节点和后置节点的复杂度都是O(1)

无环：表头节点的prev指针和表尾节点的next指针都指向null，对链表的访问以null为终点。

带表头指针和表尾指针：

带链表长度计数器

多态

字典

Redis字典所使用的哈希表由dict.h/dictht结构来定义：

```
typedef struct dictht{
   //哈希表数组
   dictEntry **table;
   //哈希表大小
   unsigned long size;
   //哈希表大小掩码，用于计算索引值
   //总是等于size-1
   unsigned long sizemask;
   //该哈希表已有节点的数量
   unsigned long used;
}dictht
```

哈希表节点：

```
typedef struct dictEntry{
   //键
   void *key;
   //值
   union{
      void *val
      uint64_tu64;
      int64_ts64;
   }v;//可以是一个指针，或者是一个uint64_t整数，又或是一个int64_t整数
   //指向下一个哈希表节点形成链表
   struct dictEntry *next;
}dictEntry;
```

字典：

Redis中的字典由dict.h/dict结构来表示：

```
typedef struct dict{
   //类型特定函数
   dictType *type;
   //私有数据
   void *privdata;
   //哈希表
   dictht ht[2];
   //rehash索引
   //当rehash不在进行时，值为-1
   int trehashhidx;
}dict;
```

type属性和privdata属性是针对不同类型的键值对，为创建多态字典而设置的。

type属性是一个指向dictType结构的指针，每个dictType保存了一簇用于操作特定类型键值对的函数，Redis会为用途不同的字典设置不同的类型特定函数，

而privdata属性则保存了需要传给那些类型特定函数的可选参数。

```
typedef struct dictType{
  //计算哈希值的函数
  unsigned int (*hashFunction)(const void *key);
  //复制键的函数
  void *(*keyDup)(void *privdata,const void *key);
  //复制值的函数
  void *(*valDup)(void *privdata,const void *obj);
  //对比键的函数
  int (*keyCompare)(void *privdata,void *key);
  //销毁值的函数
  void (*valDestructor)(void *privdata,void *obj);
}dictType;
```

ht属性是一个包含两个项的数组，数组中的每一个项都是一个dictht哈希表，一般情况下，字典只使用ht[0]哈希表，ht[1]哈希表只会在对ht[0]哈希表进行rehash时使用

Redis哈希表使用链地址法来解决键冲突。

Rehash操作：

1. 为字典的ht[1]哈希表分配空间，这个哈希表的空间大小取决于要执行的操作，以及ht[0]当前包含的键值对的数量（也即是ht[0].used属性的值）
    1. 如果当前执行的是扩展操作，那么ht[1]的大小为第一个大于等于ht[0].used*2的（2的n次方幂）
    2. 如果执行的是收缩操纵，那么ht[1]的大小为第一个大于等于ht[0].used的2的n次方幂
2. 将保存在ht[0]的所有键值对rehash到ht[1]上面：rehash指的是重新计算键的哈希值和索引值，然后将键值对放置到ht[1]哈希表的指定位置上。
3. 当ht[0]包含的所有键值对都迁移到了ht[1]之后（ht[0]变为空表），释放ht[0],将ht[1]设置为ht[0],并在ht[1]新创建一个空白哈希表，为下一次rehash做准备。

渐进式rehash

1. 为ht[1]分配空间，让字典同时持有ht[0]和ht[1]两个哈希表
2. 在字典中维持一个索引计数器变量rehashidx,并将他的值置为0，表示rehash工作正式开始
3. 在rehash进行期间，每次对字典执行添加、删除、查找或者更新操作时，线程除了执行特定的操作以外，还会顺带完成将ht[0]哈希表

在rehashidx索引上的所有键值对rehash到ht[1],当rehash工作完成之后，程序将rehashidx属性的值增加一。

4. 随着字典操作的不断执行，最终在某个时间点上，ht[0]的所有键值对都会被rehash到ht[1],这是程序将rehashidx属性的值设为-1,表示rehash工作已经完成。

跳跃表

Redis只在两个地方用到了跳跃表：一个实现有序结合键，另一个是在集群节点中用作内部数据结构。

整数集合

整数集合是集合键的底层实现之一，当一个集合只包含整数值元素，并且这个集合的元素数量不多时，Redis就会使用整数结合作为集合键的底层实现。

整数集合（intset）

```
typedef struct intset{
  //编码方式
  uint32_t encoding;
  //集合包含的元素数量
  uint32_t length;
  //保存元素的数组
  int8_t contents[];
}intset;
```

contents数组是整数集合的底层实现：整数集合的每个元素都是contents数组的一个数组项（item），各个项在数组中的大小从小到大有序的排列，并且数组中不包含任何重复项。

contents中存储什么数据取决于编码的类型。

升级

每当我们要将一个新元素添加到整数集合里面，并且新元素的类型比整数集合现有所有元素都要长时，整数集合需要先升级，然后才能将新元素添加到整数集合里面。

升级步骤：

1. 根据新元素类型，扩展整数集合的底层数组的空间大小，并为新元素分配空间
2. 将底层数组现有的所有元素都转换成与新元素相同的类型，并将类型转换后的元素放置在正确的位上，而且在放置元素的过程中，需要继续维持底层数组的有序性质不改变。

3. 将新元素添加到底层数组里面。

压缩列表

压缩列表是列表键和哈希键的底层实现之一。当一个列表键值包含少量的列表项，并且每个列表项要么就是小整数值，要么就是长度比较短的字符串，那么redis就会使用压缩列表来做列表键的底层实现。

压缩列表是Redis为了节约内存而开发的，是由一系列特殊的编码的连续内存块组成的顺序型数据结构。一个压缩列表可以包含任意多个节点，每个节点可以保存一个字节或者一个整数值。

压缩列表的节点构成：

每个压缩列表节点可以保存一个字节数组或者一个整数值，其中，字节数组可以是一下三种长度的其中一种：

1. 长度小于等于63字节的字节数组
2. 长度小于等于16383字节的字节数组
3. 长度小于等于4294967295字节的字节数组

而张数值则可以是一下六种长度的其中一种

1. 4位长，介于0到12之间的无符号整数
2. 1字节长的有符号整数
3. 3字节长的有符号整数
4. int16_t类型整数
5. int32_t类型整数
6. int64_t类型整数

每个压缩列表节点都由previous_entry_length、encoding、content三个部分组成。

previous_entry_length

节点的previous_entry_length属性以字节为单位，记录了压缩列表中前一个节点的长度。previous_entry_length属性的长度可以是1字节或者5字节。

encoding

节点的encoding属性记录了节点的content属性所保存数据的类型以及长度：

content

节点的content属性负责保存节点的值，节点值可以是一个字节数组或者整数，值的类型和长度由节点的encoding决定。

连锁更新

因为节点的变更导致previout_entry_length的长度变化，导致许多节点的长度跟着一起变化。

因为连锁更新在最坏的情况下需要对压缩列表执行N次空间重分配操作，而每次空间重分配的最坏复杂度为O(n)，所以连锁更新的最坏复杂度为O(N2)

对象

Redis用到的主要数据结构：简单动态字符串（SDS）、双端链表、字典、压缩列表、整数集合等等。

Redis并没有直接使用这些数据结构来实现键值对数据库，而是基于这些数据结构创建了一个对象系统，这个系统包含**字符串对象，列表对象、哈希对象、集合对象和有序集合对象**，每种都至少用了前面一种数据结构。

此外Redis采用了基于**引用计数的内存回收机制。**

对象的类型和编码

Redis使用对象来表示数据库中的键和值，每次当我们在Redis的数据库中新创建一个键值对时，我们至少会创建两个对象，一个对象用作键值对的键（键对象），另一个对象用作键值对的值（值对象）。

Redis中的每个对象都由一个redisObject结构表示，该结构中和保存数据有关的三个属性分别是type属性、encoding属性和ptr属性：

```
typedef struct redisObject{
  //类型
  unsigned type:4;
  //编码
  unsigned encoding:4;
  //指向底层实现数据结构的指针
  // coid *ptr;
}robj;
```

类型

REDIS_STRING:字符串对象

REDIS_LIST:列表对象

REDIS_HASH:哈希对象

REDIS_SET:集合对象

REDIS_ZSET:有序集合对象

对于Redis数据库保存的键值来说，键总是一个字符串对象，而值可以是字符串对象、列表对象、哈希对象、集合对象或者有序集合对象中的一种。

编码和底层实现

对象的ptr指针指向对象的底层实现数据结构，而这些数据结构由对象的encoding属性决定。

encoding属性记录了对象所使用的编码，也即是说这个对象使用了什么数据结构作为对象的底层实现，这个属性值可以是下面常量值中的一个：

REDIS_ENCODING_INT:long类型的整数

REDIS_ENCODING_EMBSTR:embstr编码的简单动态字符串

REDIS_ENCODING_RAW:简单动态字符串

REDIS_ENCODING_HT:字典

REDIS_ENCODING_LINKEDLIST:双端链表

REDIS_ENCODING_ZIPLIST:压缩列表

REDIS_ENCODING_INTSET:整数集合

REDIS_ENCODING_SKIPLIST:跳跃表和字典

每张类型的对象都至少使用了两种不同的编码

类型	编码	对象
REDIS_STRING	REDIS_ENCODING_INT	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	使用embstr编码的简单动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_HT	使用字典实现的哈希对象
REDIS_HASH	REDIS_ENCODING_INTSET	使用整数集合实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	使用整数集合实现的集合对象

类型	编码	对象
REDIS_SET	REDIS_ENCODING_HT	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的有序结合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	使用跳跃表和字典实现的有序集合对象

字符串对象

字符串对象的编码可以是int、raw或者embstr

如果一个字符串对象保存的是整数值，并且这个整数值可以用long类型来表示，那么字符串对象会将整数值保存在字符串对象结构的ptr属性里面（将void*转换成long）,并将字符串对象编码设置为int。

如果字符串对象保存的是一个字符串值，并且这个字符串的长度大于32字节，那么字符串对象将使用一个简单动态字符串（SDS）来保存这个字符串的值，并将对象的编码设置为raw.

如果字符串对象保存的是一个字符串值，并且这个字符串的刹那高度小于等于32个字节，那么字符串对象将使用embstr编码的方式来保存这个字符串值。

另外，因为Redis没有为embstr编码的字符串对象编写任何相应的修改程序（只有int编码的字符串对象和raw编码的字符串对象有这些程序），所以embstr编码的字符串对象实际上是只读的。当我们对embstr编码的字符串对象执行修改命令时，程序会先将字符串对象在执行修改命令之后，总会变成一个raw编码的字符串对象。

列表对象

列表对象的编码可以是ziplist或者linkedList

ziplist编码的列表对象使用的是压缩列表来作为底层实现，每个压缩列表节点保存了一个列表元素。

linkedlist编码的列表对象使用的双端链表来作为底层实现，每个双端链表节点都保存了一个字符串对象，而每个字符串对象都保存了一个列表元素。

字符串对象是Redis五种类型的对象中唯一一种会被其他四种类型对象嵌套的对象。

编码转化：

当列表对象可以同时满足以下两个条件时，列表对象使用ziplist编码：

列表对象保存的所有字符串元素的长度都小于64字节。

列表对象所保存的元素数量小于512个；不能满足这两个条件的列表对象需要使用linkedlist编码。

哈希对象

哈希对象的编码可以是ziplist或者hashtable

ziplist编码的哈希对象使用了压缩列表作为底层实现，每当有新的键值对要加入到hash对象时，程序会先将保存了键的压缩列表节点推入到压缩列表表尾，然后再将保存了值的压缩列表节点推入到压缩列表表尾。

因此：

保存了同一键值对的两个节点总是紧挨在一起，保存键的节点在前，保存值的节点在后。

先添加到哈希对象中的键值对会被放在压缩列表的表头方向，而后来添加到哈希对象中的兼职会被放在压缩列表的表尾方向。

编码转换：

哈希对象保存的所有键值对的键和值的字符串长度都小于64字节；

哈希对象保存的键值对数量小于512个；不能满足这两个条件的哈希对象需要使用hashTable编码。

集合对象

集合对象的编码可以是intset或者hashtable

intset编码的集合对象使用整数集合作为底层实现，集合对象包含的所有元素都被保存在整数集合里面。

编码转换：

1. 集合对象保存的所有元素都是整数值
2. 集合对象保存的元素数量不超过 512个

不能满足这两个条件的集合对象需要使用hashtale编码。

有序集合对象

有序集合的编码可以是ziplist或者skiplist

ziplist编码的压缩列表对象使用压缩列表来作为底层实现，每个集合元素使用两个紧挨在一起的压缩列表节点来保存，第一个节点保存元素的成员，而第二个元素则保存元素的分值。

压缩列表内的集合元素按分值从小到大进行排序，分值较小的元素被放置在靠近表头的方向，而分值比较大的元素则放置在靠近表尾的方向。

内存回收

因为C语言并不具备自动内存回收的功能，所以Redis在自己的对象系统中构建了一个引用计数（reference countion）技术实现的内存回收机制，通过这一机制，程序可以通过跟踪对象的引用计数信息，在适当的时候自动释放对象并进行内存回收。

对象的引用计数信息会随着对象的使用状态而不断变化：

1. 在创建一个新对象时，引用计数的值会被初始化为1；
2. 当对象被一个新程序使用时，他的引用计数值会被增加一
3. 当对象不再别一个程序使用时，他的引用计数值会被减一。
4. 当对象引用计数值为0时，对象所占用的内存会被释放。

对象共享

Redis会共享值为0到9999的字符串对象

对象的空转时长

除了前面介绍过的type、encoding、ptr和refcount四个属性之外，redisObject结构包含的最后一个属性为lru属性，该属性记录了对象最后一次被命令程序访问的时间。

```
typedef struct redisObject{
  unsigned lru:22;
}robj;
```

数据库

Redis服务器将所有数据库都保存在服务器状态redis.h/redisServer结构的db数组中，db数组的每个项都是redis.h/redisDb结构，每个redisDb结构代表一个数据库

```
struct redisServer{
  //...
  //一个数组，保存着服务器中所有的数据库
  redisDb *db;
  //....

}
```

在初始化服务器时，程序会根据服务器状态的dbnum属性来决定创建多少个数据库：

```
struct redisServer{
  //服务器的数据库数量
  int dbnum;
}
```

切换数据库

每个redis客户端都有自己的目标数据库，每当客户端执行数据库写命令或者数据库读命令的时候，目标数据库就会成为这些命令的操作对象。

默认情况下，redis客户端的目标数据库为0号数据库，但客户端可以通过执行Select命令来切换目标数据库。

在服务器内部，客户端状态redisClient结构的db属性记录了客户端当前的目标数据库，这个属性是一个指向redisDb结构的指针

```
typedef struct redisClient{
  //...
  //记录客户端当前正在使用的数据库
  redisDb *db;
}
```

数据库键空间

Redis是一个键值对数据库服务器，服务器中每个数据库都由一个redis.h/redisDb结构表示，其中redisDb结构的dict字典保存了数据库中的所有键值对，我们将这个字典称为键空间（key space）

```
typedef struct redisDb{
  //数据库键空间，保存着数据库中所有键值对
  dict *dict;
}
```

键空间和用户所见的数据库是直接对应的：

1. 键空间的键也就是数据库的键，每个键都是一个字符串对象。
2. 键空间的值也就是数据库的值，每个值可以是字符串对象、列表对象、哈希表对象、集合对象和有序集合对象中的任意一种Redis对象。

设置键的生存时间或过期时间

通过EXPIRE命令或者PEXPIRE命令，客户端可以以秒或者毫秒精度为数据库中的某个键设置生存时间，在经过指定的秒数或者毫秒数之后，服务器就会自动删除生存时间为0的键。

保存过期时间

redis结构的wxpires字段保存了数据库中所有键的过期时间，我们称这个字典为过期字典

1. 过期字典的键是一个指针，这个指针指向键空间中某个键对象（也就是某个数据库键）
2. 过期字典的值是一个longlong类型的整数，这个整数保存了所指向的数据库键的过期时间--一个毫秒精度的UNIX时间戳

```
typedef struct redisDb{
  //过期字典，保存着键的过期时间
  dict *expires;
}
```

过期键删除策略：

1. 定时删除：在设置键过期时间的同时，创建一个定时器，让定时器在键过期时间来临时，立即执行对键的删除操作
2. 惰性删除：放任键过期不管，但是每次从键空间中获取键时，都检查取得的键是否过期，如果过期的话，就删除该键；如果没有过期就返回该键
3. 定期删除：每隔一段时间，程序就对数据库进行一次检查，删除里面的过期键。至于要删除多少过期键，以及要检查多少数据库，由算法决定。

redis中使用惰性删除和定期删除

AOF、RDB和复制功能对过期键的处理

生成RDB文件：

在执行SAVE命令或者BGSACE命令创建一个新的RDB文件时，程序会对数据库中的键进行检查，已过期的键不会被保存到新创建的RDB文件中。

载入RDB文件：

在启动Redis服务器时，如果服务器开启了RDB功能，那么服务器将对RDB文件进行载入：

1. 如果服务器以主服务器模式运行，那么在载入RDB文件时，程序会对文件中保存的键进行检查，未过期的键会被载入到数据库中，而过期键则会被忽略，所以过期键对载入RDB的主服务器不会造成影响。
2. 如果服务器以从服务器模式运行，那么在载入RDB文件时，文件中保存的所有键，不论是否过期，都会被载入到数据库中。不过，因

为主从服务器在进行数据同步的时候，从服务器的数据库会被清空，所以一般来讲，过期键对载入RDB文件的从服务器也不会造成影响。

AOF文件写入：

当服务器以AOF持久化模式运行时，如果数据库中的某个键已经过期，但它还没有被惰性删除或是定期删除，那么AOF文件不会因为这个过期键而产生任何影响。

AOF重写：

已过期的键不会重写到数据库中

复制：

当服务器运行在复制模式下，从服务器的过期键删除动作由主服务器控制：

1. 主服务器在删除一个过期键之后，会显示的向所有从服务器发送一个DEL命令，告知从服务器删除这个过期键。
2. 从服务器在执行客户端发送的读命令时，及时碰到过期键也不会将过期键删除，而是继续像处理未过期的键一样来处理过期键
3. 从服务器只有在连接到主服务器发来的DEL命令之后，才会删除过期键。

RDB持久化

将Redis在内存中的数据库状态持久到磁盘里面，避免数据意外丢失。

RDB持久化功能所生成的RDB文件是一个经过压缩的二进制文件。

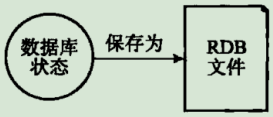

图 10-2　将数据库状态保存为 RDB 文件

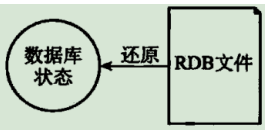

图 10-3　用 RDB 文件来还原数据库状态

RDB文件的创建和载入

有两个Redis命令可以用于生成RDB文件，一个是SAVE，另一个是BGSAVE文件。

SAVE命令会阻塞Redis服务器进程，知道RDB'文件创建完毕为止，在服务器进程阻塞期间，服务器不能处理任何命令请求。

和SAVE命令直接阻塞服务器进程的做法不同的是，BGSAVE命令会派生出一个子进程，然后由子进程负责创建RDB文件，服务器进程继续处理命令请求。

如果服务器开启了AOF持久化功能呢，那么服务器会优先使用AOF文件来还原数据库状态

只有在AOF持久化功能处于关闭状态时，服务器才会使用RDB文件来还原数据库状态。

RDB文件载入期间会一直处于阻塞状态，直到载入工作完成为止。

```
struct redis Server{
   //修改计数器
   long long dirty;
   //上一次执行保存的时间
   time_t lsatsave;
}
```

当服务器成功执行一个数据库修改命令之后，程序就会对dirty计数器进行更新：命令修改的多少次数据库，dirty计数器的值就增加多少。

RDB文件结构

图 10-10　RDB 文件结构

RDB文件的最开头是REDIS部分，这个部分长度为5个字节，保存着“REDIS”五个字符。通过这五个字符，程序可以在载入文件时，快速检查所在如的文件是否是RDB文件。

db_version长度为4个字节，他的值是一个字符串表示的整数，这个整数记录了RDB文件的版本号。

databases部分包含着零个或任意多个数据库，以及各个数据库中的键值对数据。

EOF常量的长度为1字节，这个常量标志着RDB文件正文内容的结束，当读入程序遇到这个值的时候，他知道所有数据库的所有键值对都已经载入完毕了。

check_sum是一个8字节长的无符号整数，保存着一个校验和，这个校验和是程序通过对REDIS、db_version、databases、EOF四个部分的内容的计算得出。服务器在载入RDB文件时，会将载入数据所计算出的校验和与check_sum所记录的校验和进行对比，以此来检查RDB文件是否有出错或者损坏的情况出现。

AOF持久化

AOF与RDB通过保存数据库中的键值对来记录数据库状态不同，AOF持久化是通过保存服务器所执行的写命令来记录数据库状态的。

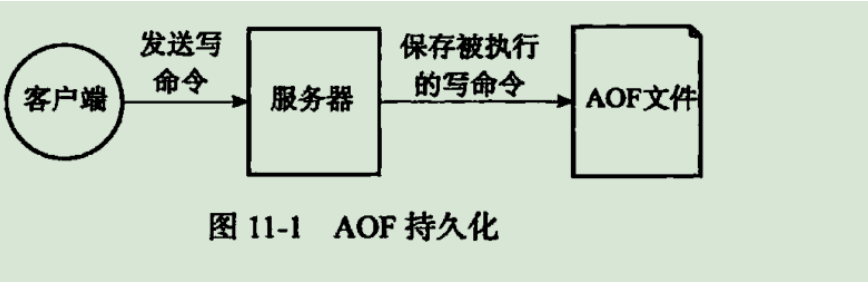

图 11-1　AOF 持久化

AOF持久化功能的实现

AOF持久化功能的实现可以分为命令追加、文件写入、文件同步三个步骤。

命令追加；

当AOF持久化功能处于打开状态时，服务器在执行完一个写命令之后，会以协议的格式将被执行的写命令追加到服务器状态的aof_buf缓冲区末尾

```
struct redisServer{
   //AOF缓冲区
   sds aof_buf;
}
```

AOF文件的写入和同步

Redis服务器的进程就是一个事件循环（loop），这个循环中文件事件负责接收客户端的命令请求，以及向客户端发送命令恢复，而事件事件则负责执行向serverCron函数这样需要定时运行的函数。

因为服务器在处理文件事件时可能会执行写命令，是的一些内容被追加到aof_buf缓冲区里面，所以在服务器每次结束一个事件之前，他都会调用flushAppendOnlyFile函数，考虑是否需要将aof_buf缓冲区中的内容写入AOF文件里面。

AOF文件的载入与数据还原

因为AOF文件里面包含了重建数据库状态的所有写命令，所以服务器只要读入并且重新执行一遍AOF文件里面保存的写命令，就可以还原服务器关闭之前的数据库状态。

AOF重写。

复制

旧版复制功能的实现

Redis的复制功能主要分为同步和命令传播两个操作：

1. 同步操作用于将从服务器的数据库状态更新至主服务当前所处的数据库状态
2. 命令传播操作则用于在主服务器的数据库状态被修改，导致主从服务器的数据库状态出现不一致时，让主从服务器的数据库重新回到一致状态。

同步

从服务器对主服务器的同步操作需要通过主服务器发送SYNC命令来完成，以下是SYNC命令执行的步骤：

1. 从服务器向主服务器发送SYNC命令。
2. 收到SYNC命令的主服务器执行BGSAVE，在后台生成一个RDB文件，并使用一个缓冲区记录从现在开始执行的所有写命令。
3. 当主服务器的BGSDAVE命令执行完毕时，主服务器会将BGSAVE命令生成的RDB的文件发送给从服务器，从服务器接受并载入这个RDB文件，将自己的数据库状态更新至主服务器执行BGSAVE命令时的数据库状态。
4. 主服务器将记录在缓冲区里面的所有写命令发送给从服务器，从服务器执行这些写命令，将自己的数据库状态更新 至主服务器数据库当前所处的状态。

命令传播

当主服务执行了写命令导致与从服务器状态不一致之后，会将导致主从服务器不一致的那条指令发送给从服务器执行，当从服务器执行了相同的命令之后，主从服务器将再次回到一致状态。

旧版实现对于从服务器断线重连会浪费大量资源做一次BGSAVE导致效率低效。

新版复制功能的实现

从Redis2.8版本开始，使用PSYNC命令代替SYNC命令来执行复制时的同步操作。

PSYNC命令具有完整重同步（Full resynchronization）和部分重同步（partial resynchronization）两种模式

> *其中完整同步用于处理初次复制的情况：完整的同步的执行步骤和SYNC命令的执行步骤完全一样，他们都是通过让主服务器创建并发送RDB文件，以及向服务器发送保存在缓冲区里面的写命令来进行同步。*
>
> *而部分重同步则用于处理断线后重复复制情况：当从服务器在断线后重新连接主服务器时，如果条件允许，主服务器可以将主从服务器连接断开期间执行的命令发送给从服务器，从服务器只要接受并执行这些命令，就可以将数据库更新至主服务器当前所处的状态。*

部分重同步的实现

部分重同步主要由以下三个部分构成：

> *主服务器的复制偏移量（Replication Offset）和从服务器的复制偏移量*
>
> *主服务器的复制积压缓冲区*
>
> *服务器的运行ID*

复制偏移量

执行复制的双方--主服务器和从服务器会分别维护一个复制偏移量

主服务器每次向从服务器传播N个字节数据时，就将自己的复制偏移量的值加上N。

从服务器每次接收到主服务器传来的N个字节数据时，就将自己的复制偏移量加上N。

复制积压缓冲区

复制积压缓冲区是由主服务器维护的一个固定长度（fixed-size）先进先出（FIFO）队列，默认大小是1MB。

当从服务器重新连接上主服务器时，从服务器会通过PSYNC命令将自己的复制偏移量offset发送给服务器，主服务器会根据这个复制偏移量来决定对从服务器执行何种同步操作：

1. 如果offset偏移量之后的数据（也即是偏移量offset+1开始的数据）仍然存在于复制积压缓冲区里面，那么主服务器将对从服务器执行部分重同步操作。
2. 相反，如果offset偏移量之后的数据已经不存在与复制积压缓冲区，那么主服务器将对服务器执行完整的同步操作。

服务器运行ID

每个Redis服务器，不论是主服务器还是从服务器都会有自己的运行ID。

运行ID在服务器启动的时候自动生成，由40个随机的十六进制字符组成。

当从服务器对主服务器进行初次复制时，主服务器会将自己的运行ID传送给从服务器，而从服务器则会将这个运行ID保存起来。

当从服务器断线并重新连上一个主服务器时，从服务器将向当前连接的主服务器发送之前保存的运行ID。

> *如果从服务器保存的运行ID和当前连接的主服务器的运行ID相同，那么说明从服务器断线之前复制的就是当前连接的这个主服务器，主服务器可以继续尝试执行部分同步操作。*

> *相反的，如果从服务器保存的运行ID和当前连接的主服务器的运行ID并不相同，那么说明从服务器断线之前，连接的不是当前连接的这个主服务器，主服务器将对从服务器执行完整的同步操作。*

Sentinel

Sentinel（哨兵）是Redis的高可用解决方案：由一个或多个Sentinel实例组成系统可以监视任意多个主服务器，以及这些主服务器属下的所有从服务器，并在被监视的主服务器进入下线状态时，自动将下线主服务器属下的某个从服务器升级为新的主服务器，然后由新的主服务器代替已下线的主服务器执行命令请求。

集群

节点

一个Redis集群通常是由多个节点组成，在刚开始的时候，每个节点都是相互独立的，他们处于一个只包含自己的集群中。