**一、前言**

因为近期项目中开始使用Redis，为了更好的理解Redis并应用在适合的业务场景，需要对Redis设计与实现深入的理解。

我分析流程是按照从main进入，逐步深入分析Redis的启动流程。同时根据Redis初始化的流程，理解Redis各个模块的功能及原理。

**二、redis启动流程**

1.初始化server变量，设置redis相关的默认值

2.读入配置文件，同时接收命令行中传入的参数，替换服务器设置的默认值

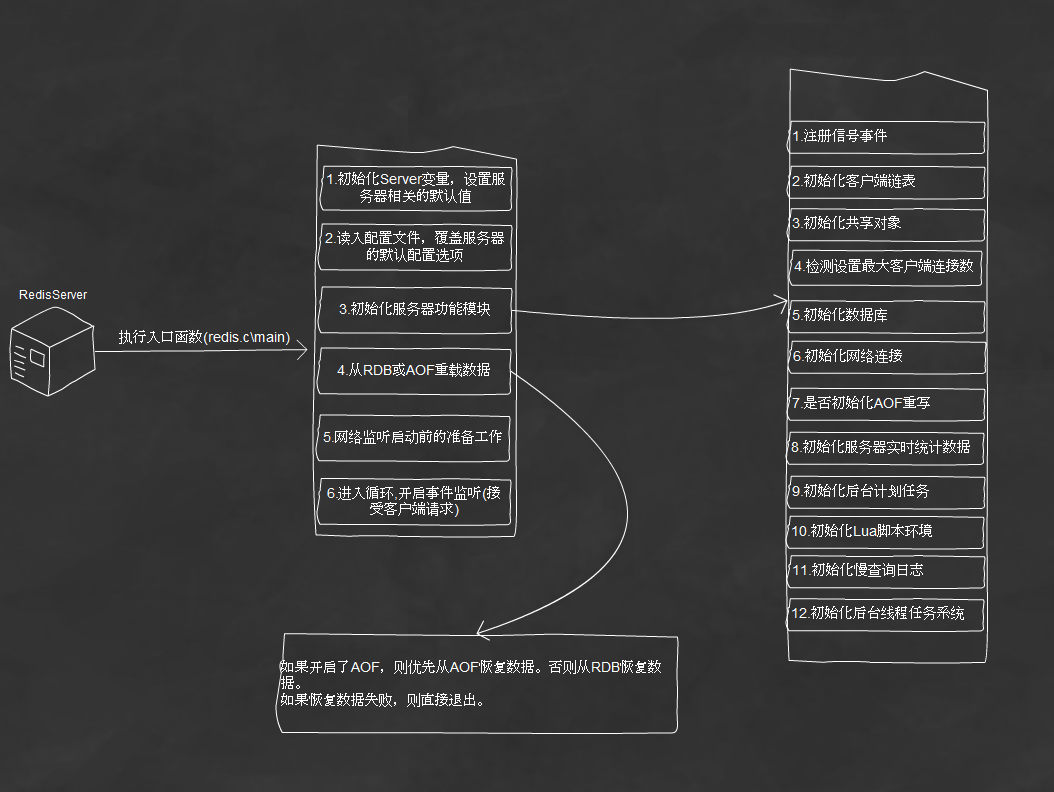
3.初始化服务器功能模块。在这一步初始化了包括进程信号处理、客户端链表、共享对象、初始化数据、初始化网络连接等

4.从RDB或AOF重载数据

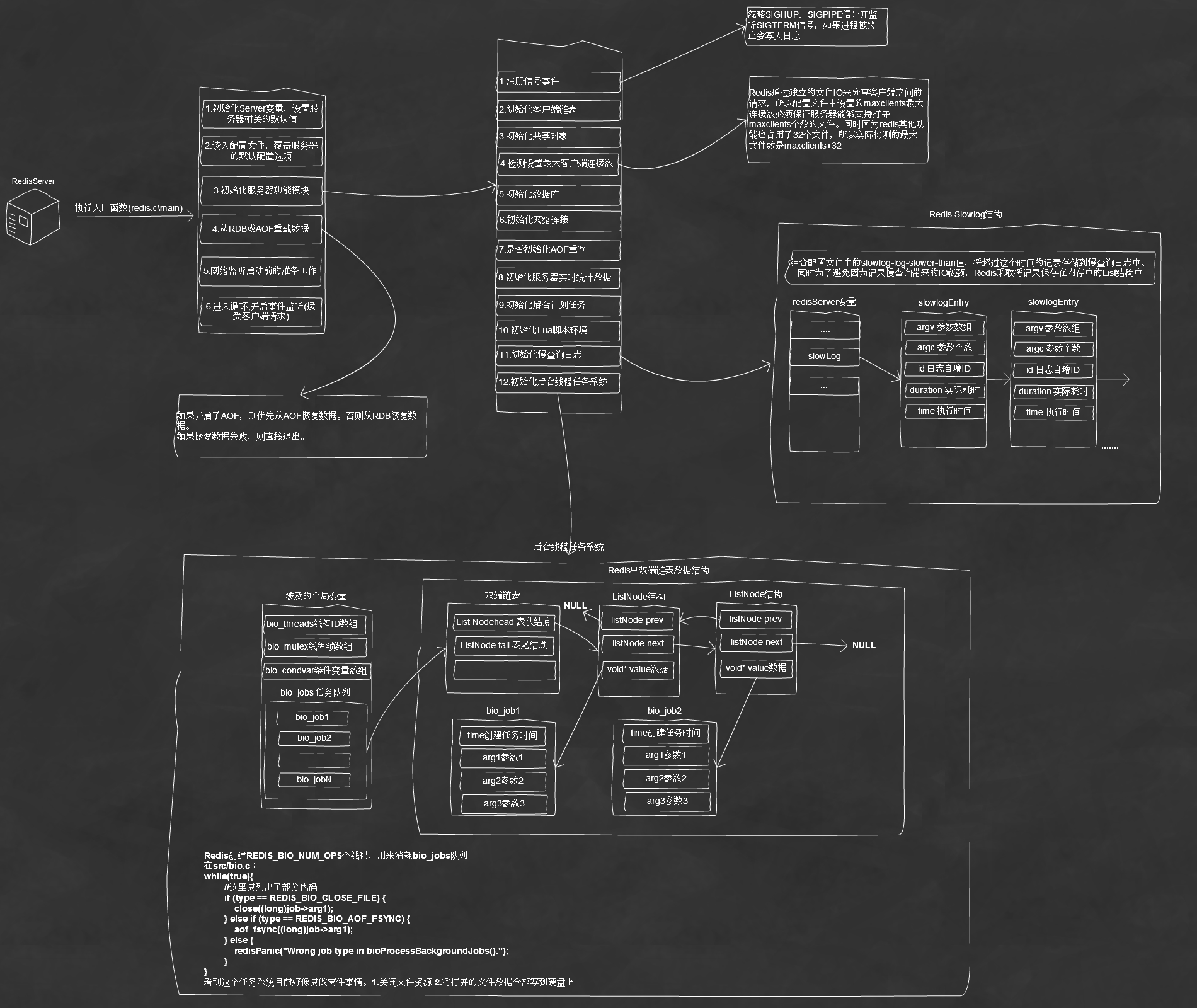
5.网络监听服务启动前的准备工作

6.开启事件监听，开始接受客户端的请求

启动的部分过程通过查看下图，会更直观。



下面是针对启动过程中，对各个模块的详细理解。（目前只分析了后台线程系统与慢查询日志系统）



**三、Redis数据持久化方案**

在使用redis时不少人都说一个问题，就是说redis宕机了怎么办？会不会数据丢失等等的问题。

现在来看看Redis提供的数据持久化解决方案，并通过原理分析优缺点。最终能得出Redis适合使用的应用场景。

**1.RDB持久化方案**

在Redis运行时，RDB程序将当前内存中的数据库快照保存到磁盘中，当Redis需要重启时，RDB程序会通过重载RDB文件来还原数据库。

从上述描述可以看出，RDB主要包括两个功能：

关于rdb的实现可以见src/rdb.c

a）保存(rdbSave)

rdbSave负责将内存中的数据库数据以RDB格式保存到磁盘中，如果RDB文件已经存在将会替换已有的RDB文件。保存RDB文件期间会阻塞主进程，这段时间期间将不能处理新的客户端请求，直到保存完成为止。

为避免主进程阻塞，Redis提供了rdbSaveBackground函数。在新建的子进程中调用rdbSave，保存完成后会向主进程发送信号，同时主进程可以继续处理新的客户端请求。

b）读取(rdbLoad)

当Redis启动时，会根据配置的持久化模式，决定是否读取RDB文件,并将其中的对象保存到内存中。

载入RDB过程中，每载入1000个键就处理一次已经等待处理的客户端请求，但是目前仅处理订阅功能的命令(PUBLISH 、 SUBSCRIBE 、 PSUBSCRIBE 、 UNSUBSCRIBE 、 PUNSUBSCRIBE)，其他一律返回错误信息。因为发布订阅功能是不写入数据库的，也就是不保存在Redis数据库的。

RDB的缺点：

再说RDB缺点时，需要提到的是RDB有保存点的概念。在默认的redis.conf中可以看到这样的默认配置：

**[plain]** [view plaincopy](http://blog.csdn.net/a600423444/article/details/8944601)

1. #save <seconds> <changes>

**[plain]** [view plaincopy](http://blog.csdn.net/a600423444/article/details/8944601)

1. save 900 1        #如果15分钟内，有1个键被修改

**[plain]** [view plaincopy](http://blog.csdn.net/a600423444/article/details/8944601)

1. save 300 10      #如果6分钟内，有10个键被修改

**[plain]** [view plaincopy](http://blog.csdn.net/a600423444/article/details/8944601)

1. save 60 10000  #如果60秒内有10000个键被修改

意思是当满足上面任意一个条件时，将会进行快照保存。为了保证IO读写性能不会成为Redis的瓶颈，一般都会创建一个比较大的值来作为保存点。

1.此时如果保存点设置过大，就会导致宕机丢失的数据过多。保存点设置过小，又会造成IO瓶颈

2.当对数据进行保存时，可能会由于数据集过大导致操作耗时，这会导致Redis可能在短时间内无法处理客户端请求。

**2.AOF持久化方案**

以协议文本的方式，将所有对数据库进行的写入命令记录到AOF文件，达到记录数据库状态的目的。

a)保存

1.将客户端请求的命令转换为网络协议格式

2.将协议内容字符串追加到变量server.aof\_buf中

3.当AOF系统达到设定的条件时，会调用aof\_fsync(文件描述符号)将数据写入磁盘

其中第三步提到的设定条件，就是AOF性能的关键点。目前Redis支持三种保存条件机制：

1.AOF\_FSYNC\_NO：不保存

此模式下，每执行一条客户端的命令，都会将协议字符串追加到server.aof\_buf中，但不会执行写入磁盘。

写入只发生在：

     1.Redis被正常关闭

     2.Aof功能关闭

     3.系统写缓存已满，或后台定时保存操作被执行

上面三种情况都会阻塞主进程，导致客户端请求失败。

2.AOF\_FSYNC\_EVERYSECS：每一秒保存一次

由后台子进程调用写入保存，不会阻塞主进程。如果发生宕机，那么最大丢失数据会在2s以内的数据。**这也是默认的设置选项**

3.AOF\_FSYNC\_ALWAYS：每执行一个命令都保存一次

这种模式下，可以保证每一条客户端指令都被保存，保证数据不会丢失。但缺点就是性能大大下降，因为每一次操作都是独占性的，需要阻塞主进程。

b)读取

AOF保存的是数据协议格式的数据，所以只要将AOF中的数据转换为命令，模拟客户端重新执行一遍，就可以还原所有数据库状态。

读取的过程是：

1.创建模拟的客户端

2.读取AOF保存的文本，还原数据为原命令和原参数。然后使用模拟的客户端发出这个命令请求。

3.继续执行第二步，直到读取完AOF文件

AOF需要将所有的命令都保存到磁盘，那么这个文件会随着时间变得越来越大。读取也会变得很慢。

Redis提供了AOF的重写机制，帮助减少文件的大小。实现的思路是：

**[plain]** [view plaincopy](http://blog.csdn.net/a600423444/article/details/8944601)

1. LPUSH list 1 2 3 4 5

**[plain]** [view plaincopy](http://blog.csdn.net/a600423444/article/details/8944601)

1. LPOP list

**[plain]** [view plaincopy](http://blog.csdn.net/a600423444/article/details/8944601)

1. LPOP list

**[plain]** [view plaincopy](http://blog.csdn.net/a600423444/article/details/8944601)

1. LPUSH list 1

最初保存到AOF文件的将会是四条指令。但经过AOF重写后，会变成一条指令：

**[plain]** [view plaincopy](http://blog.csdn.net/a600423444/article/details/8944601)

1. LPUSH list 1 3 4 5

同时，考虑到为了在AOF重写时，不影响AOF的写入增加了AOF重写缓存的概念。

也就是说Redis在开启AOF时，除了将命令格式数据写入到AOF文件，同时也会写入到AOF重写缓存。这样AOF的写入、重写就做到了隔离，保证了重写时不会阻塞写入。

c)AOF重写流程

1.AOF重写完成会向主进程发送一个完成的信号

2.会将AOF重写缓存中的数据全部写入到文件中

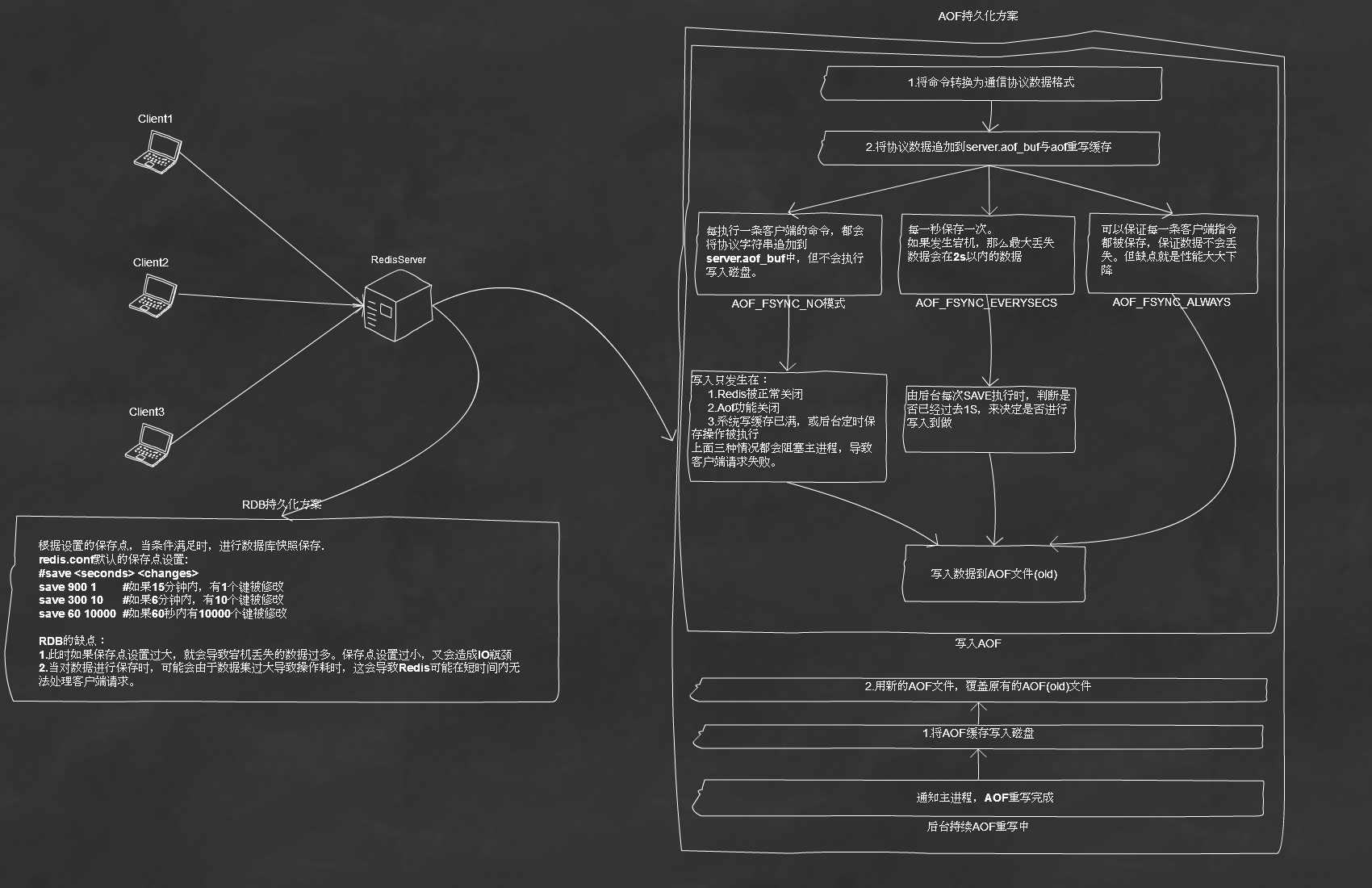
3.用新的AOF文件，覆盖原有的AOF文件。

d)AOF缺点

1.AOF文件通常会大于相同数据集的RDB文件

2.AOF模式下性能与RDB模式下性能高低，主要取决于AOF选用的fsync模式

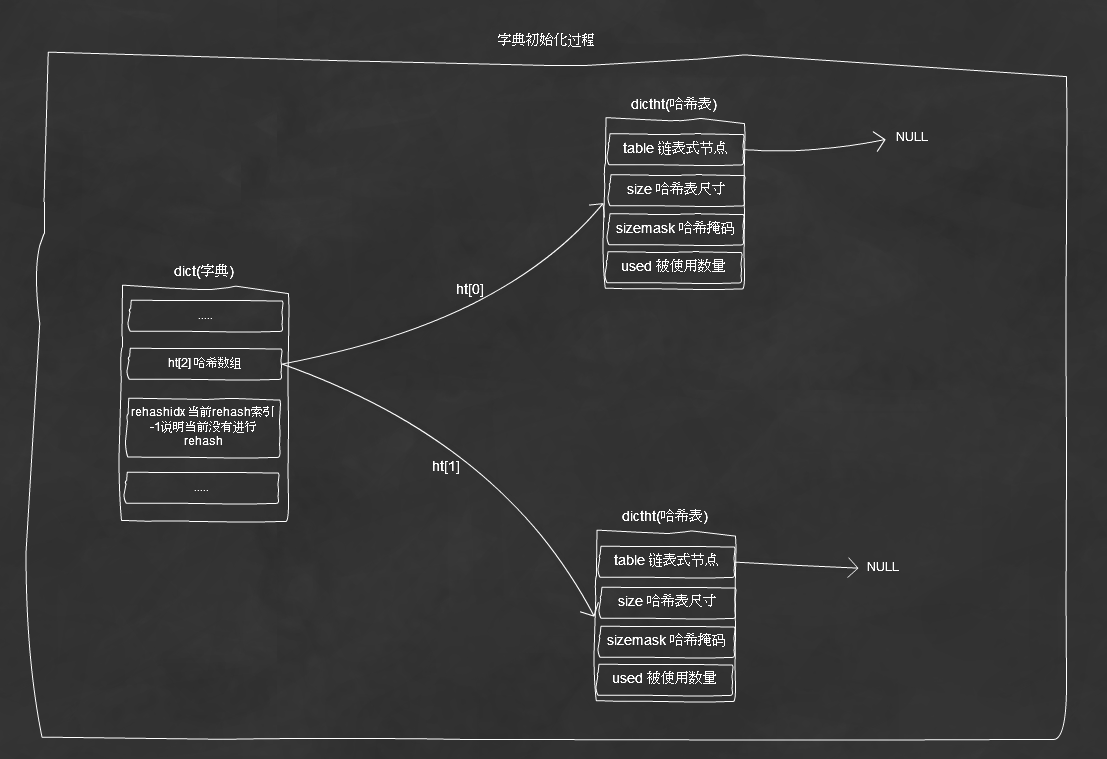
下面给出客户端请求RedisServer时，server端持久化的部分操作图解。



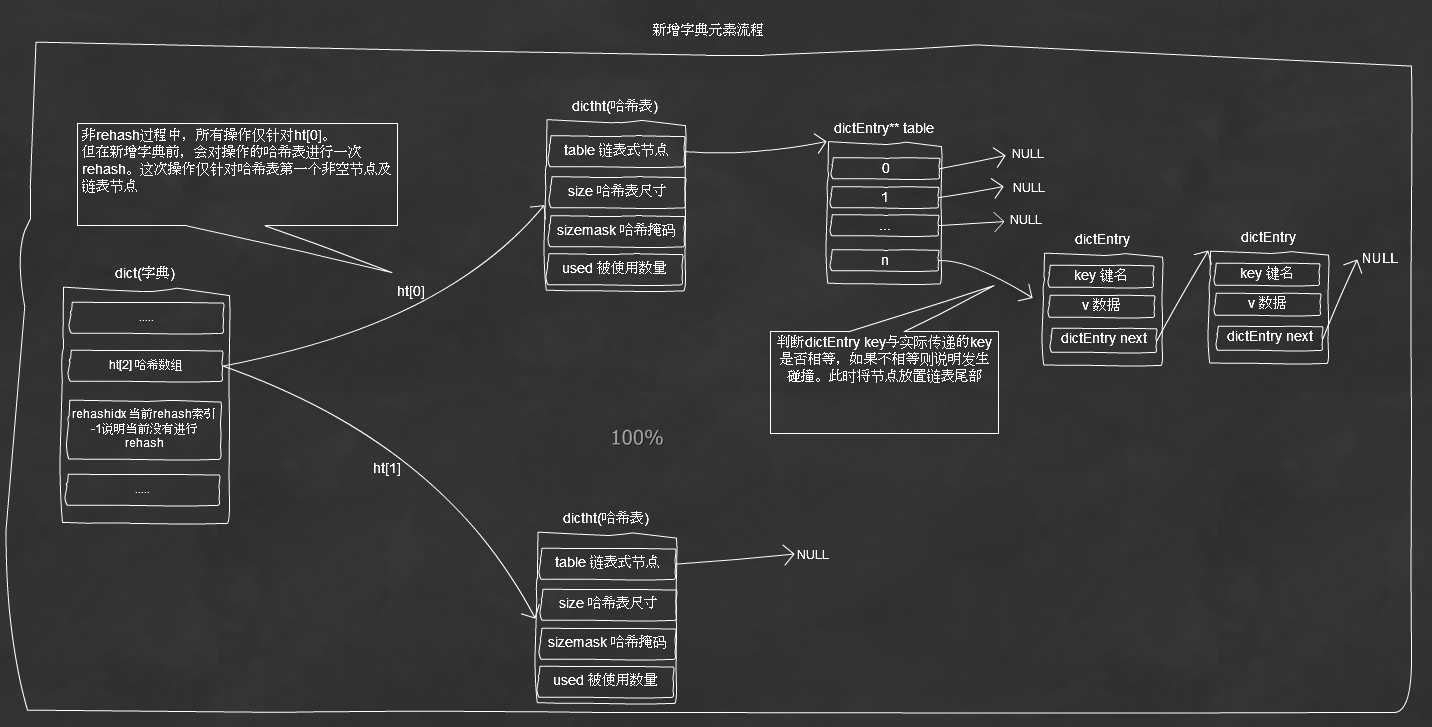
四、Redis数据库的实现

Redis是一个键值对数据库，称为键空间。实现这种KV形式的存储，Redis使用了两种数据结构类型：1、字典  
        Redis字典使用的是哈希表实现，原本不准备详细介绍Redis哈希表的实现。但发现Redis在实现哈希表时，  
提供了一个很好的rehash方案，这个方案思路很好，甚至可以衍生到其他各个应用中使用，方案的名称叫“渐进式Rehash”。  
  
        实现哈希表的方法大同小异，但为何各个开源软件总是去开发自己独有的哈希数据结构呢？  
从研究PHP内核的哈希实现与Redis哈希实现，发现应用场景决定了必须定制才能更好的发挥性能。(关于PHP哈希实现可以参看：[PHP内核中的神器之HashTable](http://blog.csdn.net/a600423444/article/details/8850617))  
a）PHP主要应用于WEB场景，在WEB场景针对单次请求数据之间是隔离的，并且哈希的数量是有限的，那么进行一次rehash也是很快的。  
所以PHP内核使用阻塞形式rehash，即rehash进行中将不能对当前哈希表进行任何操作。  
  
b）在来看Redis，常驻进程，接收客户端请求处理各项事务，并且操作的数据是相关且数据量较大的，如果使用PHP内核的那种方式就会出现：  
对哈希表进行rehash时，此时将阻塞所有客户端请求，并发性能会大大下降。

初始化字典图解：



新增字典元素图解：



Rehash执行流程：

