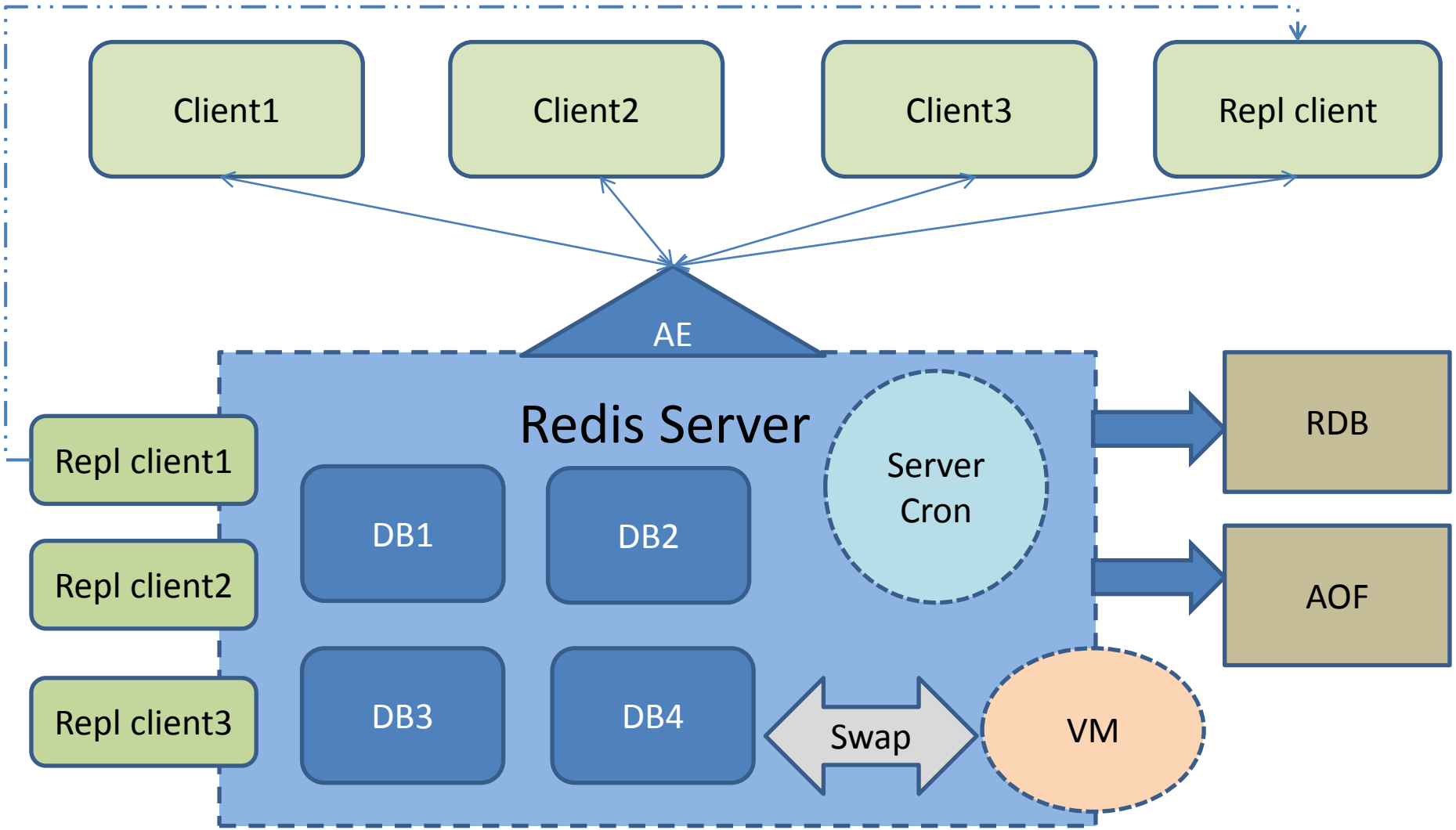
Redis 实现分析

网易杭研——胡炜

OUTLINE

- REDIS的内部实现（基于2.8.7）
  - 单线程模型的实现（AE）
  - 内存Zmalloc
  - 哈希字典的实现及操作
  - ServerCron操作
  - 阻塞操作的实现
  - 事务的实现
  - RDB
  - AOF
  - Replication

Redis 架构


Client1
Client2
Client3
Repl client
AE
Redis Server
Repl client1
Repl client2
Repl client3
DB1
DB2
DB3
DB4
Server Cron
Swap
VM
RDB
AOF

Redis 的处理模型AE 模块

- 事件类型
  - 时间类型（serverCron）
  - 文件类型（客户端请求, 复制等）
  - 两个类型不会并发，串行执行
- 好处：
  - 不需要考虑各个操作并发的情况
- 坏处：
  - 效率会有影响（时间事件如serverCron可能会被阻塞一段较长的时间）

事件的实现

- 将文件事件和时间事件注册到eventLoop中，eventloop在系统main方法中开始循环
- 从epoll/kqueue/select 中选择一种多路复用方法
- 先处理文件事件再处理时间事件(aeProcessEvent)
  - 遍历时间事件链表中找到即将触发的时间
  - 以这个时间和当前时间差为超时时间去pull文件事件，置入一个数组中，并根据读写不同的属性调用对应的回调函数
  - 遍历时间事件链表，处理到期的时间事件

Zmalloc

- 实现了zmalloc, zcalloc, zrealloc, zfree方法
- 除了mac系统之外，需要在每段内存的头部额外分配4-8字节来存储这次分配的长度
- Used_memory是一个统计变量，当前db使用了多少内存，redis可选择性的用pthread_mutex来保护这个变量的线程安全，目前貌似没看到有加锁的情况（VM被废除）

内存使用量超限

- 超过config配置的话，调用freeMemoryIfNeeded方法来释放内存（谁调用的）
  - 是否超限不能算入slave和monitor链表的大小
  - 最大内存牺牲策略
    - No-evection ，不能牺牲任何数据
    - Volatile-lru（默认），在过期数据中按lru淘汰
    - Volatile-random，在过期数据中随机淘汰
    - Allkeys-lru， 在所有数据中按lru淘汰
    - Allkeys-random ， 在所有数据中随机淘汰
    - Volatile-ttl，清除马上要过期的key

内存使用量超限

- LRU的实现方式：
- 系统维护了一个LRU时钟，每分钟自增
- 每个RedisObject有一个lru time属性，记录了上次被访问时LRU时钟时间
- LRU时间只有22字节，大约1.5年会回卷
- LRU没有诸如数据库buffer一样链表的结构
- 每次随机选取maxmemory_sample个数据，将最老的选作牺牲者
- Volatile-ttl的淘汰原则在lru的基础上加一个即将过期的判断（具体？）

RedisDB的结构

- typedef struct redisDb {
  dict *dict; /* The keyspace for this DB */
  dict *expires; /* Timeout of keys with a timeout set */
  dict *blocking_keys; /* Keys with clients waiting for data (BLPOP) */
  dict *ready_keys; /* Blocked keys that received a PUSH */
  dict *watched_keys; /* WATCHED keys for MULTI/EXEC CAS */
  int id;
  long long avg_ttl; /* Average TTL, just for stats */
  } redisDb;
- 基本单位就是dict，这里有5个dict，分别对应着数据，数据过期，block操作，事务操作

Dict 哈希字典

- 哈希字典，对应代码中的dict
- 每个dict对应两个dictht，每个dictht对应的才是一个哈希表数据结构， 两个哈希结构用于动态调整字典大小
- Dictht中保存的是一个dictEntry指针的数组
- dictEntry包括key,value的指针，并包含指向下一个dictEntry的指针
- 因此redis哈希是以开链的方式解决冲突的

Dict 哈希字典

- 哈希字典会动态的进行扩容或者缩容
- 不允许在后台持久化进程（RDBSave AOF Rewrite）时进行，原因是rehash会有大量的copy-on-write页
- 调整大小分为两个步骤
  - Resize
  - Rehash

Dict哈希字典ReSize

- 扩容缩容的条件
  - 扩展条件：used大于size， 并且平均每个桶的数量超过5的时候
  - 扩展的大小：size和used较大值的两倍（2次幂对齐）
  - 缩小的条件： size和used都不为0，size大于初始化大小，used* 100/size < 10， 即利用率小于10%
  - 缩小的大小： 将桶数目减少到接近于used与初始化大小中的较小值

Dict哈希字典ReHash

- 由serverCron定时时间事件来触发
- 利用两个哈希表结构，将第一个哈希中的数据读取出来插入到另外一个哈希中
- 执行分为两部分
  - serverCron会遍历所有的db进行操作，每个db大约耗时1ms，每次rehash 100个桶
  - 前台client对哈希表的增删查该会rehash当前的一个桶
- ReHash会同时操作数据dict以及expire dict
- ReHash过程中的读写操作会涉及到新旧两个哈希
  - 具体？？？

Rehash过程中的字典操作

- Dict add 在确认key值不存在后（检查ht[0] 和ht[1]）插入到新的哈希 ht[1]中
- Dict replace 先尝试dict add，失败的话，调用dict find 找到对应Value的时候调dict set
- Dict Find 按序寻找旧和的新的两个哈希 ht[0]和ht[1]，按最后找到的为准
- Dict delete 按序删除旧和新的两个哈希上的数据
- Dict next 利用iterator进行遍历，遍历完旧哈希再遍历新哈希

serverCron

- 时间事件，定时100ms左右执行一次
  - WatchDog（2.6 版本后的性能分析监控机制）
  - 更新redis 内部的时钟，包括LRU时钟
  - 更新系统的内存使用峰值
  - 响应shutdown
  - 打印一些非空db的使用情况信息（size， used， expire size）
  - 打印一些客户端信息（包括slave 信息）
  - 关闭超时客户端连接（不包括slave master ， block 客户端有单独的判断条件，一次最多50个client）
  - 回收客户端暂时不用的query buffer空间（大于32k，并且考虑峰值利用情况）， 调用zrealloc，一次最多50个client
  - 清理expire数据（一次最多16个db）
  - 启动哈希表resize rehash，并完成大部分工作（一次最多16个db）
  - 启动rdb save
  - 启动aof rewrite （系统自动检查以及用户请求）
  - 完成RDB SAVE的扫尾工作（包括同步slave中的发送RDB）
  - 完成AOF rewrite的扫尾工作（刷aof rewrite buffer）
  - 创建复制事件

阻塞操作

- BLPOP， BRPOP， BRPOP LPUSH
- 应用于空列表时会阻塞客户端，否则等效于非阻塞版本（可以用于实现分布式锁）
- 实现，利用redisDB中的blocking_keys 和系统中的ready_keys链表
  - Blocking_keys哈希表的key value分别是数据键以及阻塞客户端
  - 维持客户端的连接，但是造成客户端阻塞
  - LPUSH，RPUSH，LINSERT操作会检测blocking_keys哈希表，如果找到键值，那么创建一个ReadyList，添加到ready_keys哈希表中
  - Ready_keys链表中保存的是db和阻塞的key
  - PUSH操作之后会从Ready_keys中弹出一个元素，根据key和db去blocking list中查找阻塞客户端，并解除其block状态，并从blocking_keys中删除
  - Db中的ready_keys哈希是在muti事务中的PUSH操作重复操作同一个Key的优化实现
- 结束阻塞条件
  - 其他人修改了数据，列表推入了新的元素
  - 阻塞时间超时，被severCron断开
  - 客户端连接断开

事务

- Multi/discard/exec/watch
- 只有ACID中的CI两个特性
- 维护了一个队列数组，记录了操作类型及参数
- 命令数组维护在client结构体下
- WATCH命令在multi之前执行，监控键值是否被其他client修改
- redisDB中有一个watch_keys哈希表保存key和client之间的对应关系
- EXEC时会检测watch_keys 哈希
- EXEC了一半断电，AOF不完整，需要repaire
- EXEC是完整的，中间不会切换响应其他client请求

订阅发布

- SUBSCRIBE / PUBLISH
- 类似于订阅广播
- 通过server的pubsub_channel哈希表实现，key/value分别是频道/客户端链表

RDB

- RDB是Redis物化数据，保证宕机恢复的一个手段（会丢一部分最新数据）
  - 每个Redis实例只会存一份rdb文件
  - 可以通过Save以及BGSAVE 来调用
  - 分成两种，前者阻塞， 后者非阻塞
  - 主要有两个方法rdbSave以及rdbLoad

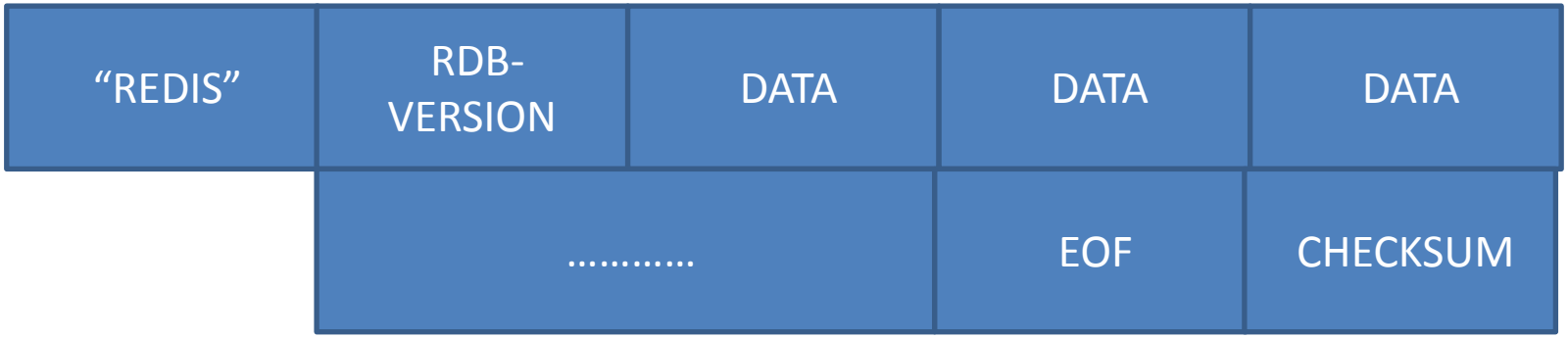
RDB

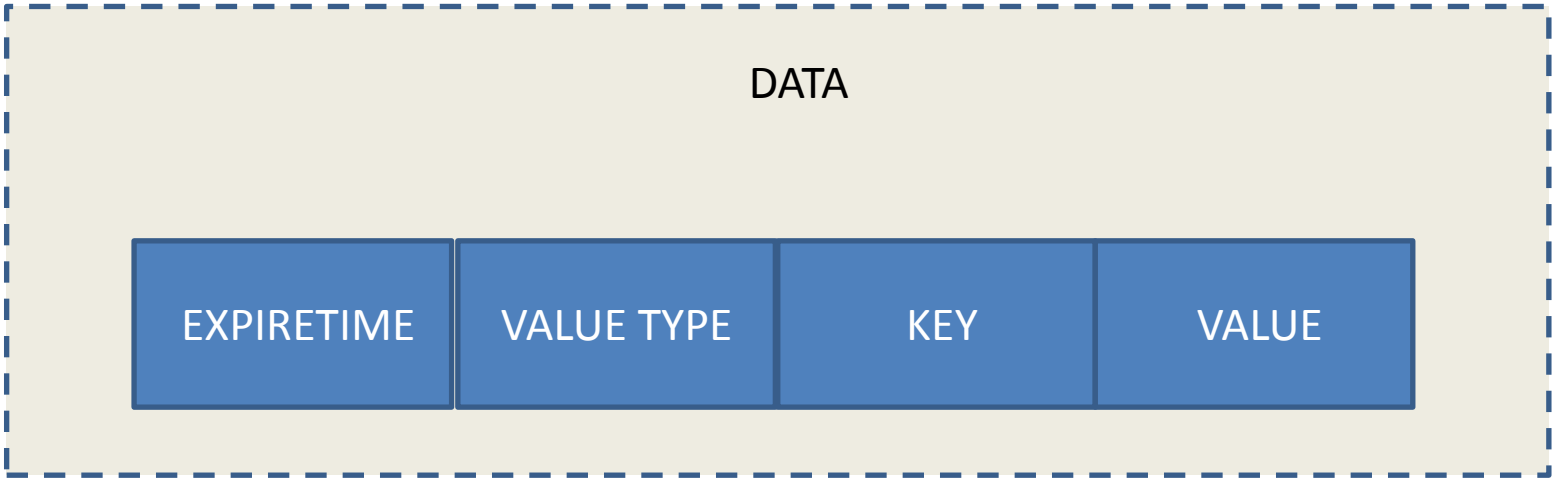
- rdbSave
  - 遍历每个db
  - 遍历每个db的dict，获取每一个dictEntry
  - 获取Key之后查询expire， 看是否过期，过期的数据舍弃
  - 将数据key, value, type, expiretime 等写入文件
  - 由于rdbSave的同时不能做rehash，因此这样的遍历是安全的
  - 遍历完之后计算checksum
  - 通过文件rename交换旧的RDB文件

RDB

- rdbLoad
  - 遍历RDB文件的每一条数据
  - 读取key, value, expiretime等信息，插入dict字典以及expire字典
  - 校验checksum

RDB文件结构


“REDIS”
RDB-VERSION
DATA
DATA
DATA
……
EOF
CHECKSUM


DATA
EXPIRETIME
VALUE TYPE
KEY
VALUE

AOF

- 类似于BINLOG机制，是首选的物化手段，可以做到不丢数据
- 每次常规操作之后会调用flushAppendOnlyFile来刷新aof
- Aof策略
  - 每秒fsync一次（默认）
  - 每次操作都需要fsync， 前台线程阻塞
  - 不做fsync
- Fsync排队，fsync太慢时会直接返回，打印异常log，只有切换aof on到off时会强制sync

AOF

- AOF中的内容就是网络协议过来的命令内容

```
$5
RPUSH
$4
list
$1
1
$1
2
$1
3
$1
4
*2
```

- 从AOF文件恢复数据
  - 创建一个fake client读取文件内容
  - 和普通client一样解析数据命令并执行
  - 没有网络，没有应答，也不可能block

AOF

- 当AOF文件太大的时候需要做ReWrite来替换旧的AOF文件
- AOF ReWrite的实现：
  - Fork一个子进程， 关闭所有监听
  - 主进程停止hash resize 及rehash
  - 打开一个tmp aof文件准备写入
  - 遍历每个DB前写入一条selectDB命令
  - 和RDB一样遍历每对key value
  - 跳过过期键
  - 将数据以命令的方式写入tmp aof文件中
  - 主进程中所有并发的操作内容需要写到aof rewrite buffer中
  - 在子进程写完AOF时，主进程serverCron中将aof rewrite buffer中的内容刷到新的aof文件中
  - 最后以rename的方式替换旧的aof文件

Rewrite AOF文件

- 根据不同的value type选择不同的命令：
  - String， set
  - List， rpush
  - Set，sadd
  - Zset，zadd
  - Hash， hmset

  除了string之外，其他类型默认一次最多写64项，超过了会拆分

  过期信息会重写成set expire 命令

后台进程和前台操作的并发

- BGSAVE及AOFREWRITE子进程fork 会拷贝内存
- Copy-on-write 的机制保证不会占用太多的内存
- RDB-SAVE 以及 AOF-REWRITE只需要读取原hash表内容，不存在并发的问题

- 前台推送中心的数据，8G的库做RDB-SAVE时只会额外多出三四百兆的内存

VM

- 早期版本中的REDIS有VM 功能
- 用于内存放不下数据，redis将冷数据放到外存中，当需要读的时候将数据从外存换入内存
- VM中只是将Value存到外存中，Key永驻内存
- 外存按页管理，通过一个bitmap进行管理（已用和未用）
- 一个value 需要存储到外存的连续页面中（寻页算法）
- Values swap计算方法：
  - swapbility = idle-time * log(estimated size)
- 根据不同的配置文件中的线程数可以选择是blocking方式的swap或者是后台线程方式swap
- 尝试100次，从哈希里选取随机的key， 每个db里尝试5个key， 选择swapbility最大的那个key，将其换出到磁盘上
- 每个redisObj对象有一个状态标识，标识其存储位置
- IO线程的任务
  - REDIS_IOJOB_LOAD：将给定key的对象从交换空间load回主存
  - REDIS_IOJOB_PREPARE_SWAP:计算将val指向的对象交换到交换空间所需要的页数，计算的结果放在page域
  - REDIS_IOJOB_DO_SWAP:将val指向的对象交换到交换空间

Replication

- 复制目前的Redis中有两种：
  - SYNC 全量
  - PSYNC 增量 2.8版本中引入
- 当配置完主备关系后，从库会立即向主库发起一个SYNC命令 syncWithMaster方法
  - 主库起一个BGSAVE进程，备份一个rdb文件
  - serverCr将rdb发送到从库
  - 后台备份的同时将前台的操作记录到backlog缓冲区中
  - 从库调用rdbload将数据从rdb文件中载入，然后再执行命令缓冲区中的命令

Replication

- Backlog是一个全局的环状缓冲区
- 每次主库上的操作都会将操作写入backlog buffer
- 从库一直在读取backlog buffer中的内容
  - 每个client将backlog buffer中对应内容拷贝到自己的output buffer中

Replication

- 增量复制PSYNC是2.8的新特性
- 防止网络一断全部重新BGSAVE导致负载过重的情况
- 每次尝试先做增量复制，不行再做全量复制
  - 客户端会缓存一个master 的client，称之为cache master，cache master 在每次master连接断开时会做保存，在尝试PSYNC时会检测cache master是否匹配
  - 向master发送一个PSYNC命令
  - 如果master 返回一个FULLRESYNC，那么可以解析reply中的runid（独立复制的标识）和offset（backlog偏移），丢弃cache master，一切重新开始同步
  - 如果master 返回一个CONTINUE，那么说明可以继续同步
  - 如果时需要FULLRESYNC， 那么创建一个新的tmp.rdb文件用来接收master 传输过来的rdb文件
  - 创建一个readSyncBulkPayLoad文件事件，分批读入数据
  - 将这个SLAVE加入到slave链表中

Replication

- 主机收到SYNC/PSYNC命令的处理流程
  - 如果从库想要做FULLREPLICATION，那么会发送一个run_id 为?的请求，否则会尝试做PSYNC
  - 检测run_id 是否匹配
  - 解析slave请求，获取psync offset 偏移
  - 判断slave 请求中的offeset，是否还存在于master上的backlog buffer中
  - 如果PSYNC失败，开始一个FULLRESYNC操作
  - 如果后台有一个正在进行的BGSAVE，查看是否可以利用（即是否有其他SALVE调用了BGSAVE之后在等待BGSAVE完成，如果是别的SLAVE在调用BGSAVE，可以拷贝那个SLAVE 对应client的OUTPUT BUFFER到当前的SLAVE）
  - 自己起一个BGSAVE子进程来保存RDB文件

QA