Redis

springside edited this page on 19 Sep 2014 · 254 revisions

Pages 77

Clone this wiki locally

Clone in Desktop

版本：V3.1 2014-3-36 (@江南白衣版权所有，转载请保留出处)，针对Redis 2.8版。

1. Overview

1.1 资料

- <The Little Redis Book> ，最好的入门小册子，可以先于一切文档之前看，免费。
- 作者Antirez的博客，Antirez维护的Redis推特。
- Redis weekly redis周报。
- Redis 命令中文版， huangz同学的翻译，同时还有Redis官网几篇重要文档的翻译。
- Redis设计与实现 ，又是huangz同学的巨作，深入了解内部实现机制。
- Redis 2.6源码中文注释版 ，继续是huangz同学的大功德。

- NoSQL Fan里的Redis分类
- 《Redis in Action》 (Manning, 2013) 挺实战的一本书。

1.2 优缺点

非常非常的快，有测评说比Memcached还快(当大家都是单CPU的时候)，而且是无短板的快，读写都一般的快，所有API都差不多快，也没有MySQL Cluster、MongoDB那样更新同一条记录如Counter时慢下去的毛病。

丰富的数据结构，超越了一般的Key-Value数据库而被认为是一个数据结构服务器。组合各种结构，限制Redis用途的是你自己的想象力，作者自己捉刀写的用途入门。

因为是个人作品，Redis2.6版只有2.3万行代码，Keep it simple的死硬做法，使得普通公司而不需淘宝那个级别的文艺公司也可以吃透它。

Redis宣言就是作者的自白，我最喜欢其中的“代码像首诗”，”设计是一场与复杂性的战斗“，“Coding是一件艰苦的事情，唯一的办法是享受它。如果它已不能带来快乐就停止它。为了防止这一天的出现，我们要尽量避免把Redis往乏味的路上带。”

让人又爱又恨的单线程架构，使得代码不用处理平时最让人头痛的并发而大幅简化，也不用老是担心作者的并发有没有写对，但也带来单CPU的瓶颈，而且单线程被慢操作所阻塞时，其他请求的延时变得不确定。

那Redis不是什么?

- Redis 不是Big Data，数据都在内存中，无法以T为单位。
- 在Redis 3.0的Redis-Cluster发布并被稳定使用之前，Redis没有真正的平滑水平扩展能力。
- Redis 不支持Ad-Hoc Query，提供的只是数据结构的API，没有SQL一样的查询能力。

1.3 Feature速览

- 所有数据都在内存中。
- 五种数据结构：String / Hash / List / Set / Ordered Set。
- 数据过期时间支持。
- 不完全的事务支持。
- 服务端脚本：使用Lua Script编写，作用类似存储过程。
- PubSub：捞过界的消息一对多发布订阅功能，起码Redis-

Sentinel在使用它。

- 持久化：支持定期导出内存的Snapshot 与 记录写操作日志的Append Only File两种模式。
- Replication：Master-Slave模式，Master可连接多个只读Slave，Geographic Replication也只支持Active-Standby。
- Fail-Over：Redis-Sentinel节点负责监控Master节点，在master失效时提升slave。
- Sharding：开发中的Redis-Cluser。
- 动态配置：所有参数可用命令行动态配置不需重启，2.8版可以重新写回配置文件中，对云上的大规模部署非常合适。

1.4 八卦

- 作者是意大利的Salvatore Sanfilippo(antirez)，又是VMWare大善人聘请了他专心写Redis。
- EMC与VMWare将旗下的开源产品如Redis和Spring都整合到了孙公司Pivotal公司。
- Pivotal做的antirez访谈录，内含一切八卦，比如他的爱好是举重、跑步和品红酒。
- 默认端口6379，是手机按键上MERZ对应的号码，意大利歌女Alessia Merz是antirez和朋友们认为愚蠢的代名词。

2. 数据结构

2.1 Key

- Key 不能太长，比如1024字节，但antirez也不喜欢太短如"u:1000:pwd"，要表达清楚意思才好。他私人建议用":"分隔域，用"."作为单词间的连接，如"comment:12345:reply.to"。
- Keys，返回匹配的key，支持通配符如 "keys a*" 、 "keys a?c"，但不建议在生产环境大数据量下使用。
- SCAN命令，针对Keys的改进，支持分页查询Key。在迭代过程中，Keys有增删怎么办？要锁定写操作么？--不会，不做任何保证，撞大运，甚至同一条key可能会被返回多次。
- Sort，对集合按数字或字母顺序排序后返回或另存为list，还可以关联到外部key等。因为复杂度是最高的O(N+M*log(M))(N是集合大

小，M 为返回元素的数量)，有时会安排到slave上执行。

- Expire/ExpireAt/Persist/TTL，关于Key超时的操作。默认以秒为单位，也有p字头的以毫秒为单位的版本， Redis的内部实现见2.9过期数据清除。

2.2 String

最普通的key-value类型，说是String，其实是任意的byte[]，比如图片，最大512M。 所有常用命令的复杂度都是O(1)，普通的Get/Set方法，可以用来做Cache，存Session，为了简化架构甚至可以替换掉Memcached。

Incr/IncrBy/IncrByFloat/Decr/DecrBy，可以用来做计数器，做自增序列。key不存在时会创建并贴心的设原值为0。IncrByFloat专门针对float，没有对应的decrByFloat版本？用负数啊。

SetNx， 仅当key不存在时才Set。可以用来选举Master或做分布式锁：所有Client不断尝试使用SetNx master myName抢注Master，成功的那位不断使用Expire刷新它的过期时间。如果Master倒掉了key就会失效，剩下的节点又会发生新一轮抢夺。

其他Set指令：

- SetEx， Set + Expire 的简便写法，p字头版本以毫秒为单位。
- GetSet， 设置新值，返回旧值。比如一个按小时计算的计数器，可以用GetSet获取计数并重置为0。这种指令在服务端做起来是举手之劳，客户端便方便很多。
- MGet/MSet/MSetNx， 一次get/set多个key。
- 2.6.12版开始，Set命令已融合了Set/SetNx/SetEx三者，SetNx与SetEx可能会被废弃，这对Master抢注非常有用，不用担心setNx成功后，来不及执行Expire就倒掉了。可惜有些懒惰的Client并没有快速支持这个新指令。

GetBit/SetBit/BitOp,与或非/BitCount， BitMap的玩法，比如统计今天的独立访问用户数时，每个注册用户都有一个offset，他今天进来的话就把他那个位设为1，用BitCount就可以得出今天的总人数。

Append/SetRange/GetRange/StrLen，对文本进行扩展、替换、截取和求长度，只对特定数据格式如字段定长的有用，json就没什么用。

2.3 Hash

Key-HashMap结构，相比String类型将这整个对象持久化成JSON格式，Hash将对象的各个属性存入Map里，可以只读取/更新对象的某些属性。这样有些属性超长就让它一边呆着不动，另外不同的模块可以只更新自己关心的属性而不会互相并发覆盖冲突。
另一个用法是土法建索引。比如User对象，除了id有时还要按name来查询。可以有如下的数据记录:

- (String) user:101 -> {"id":101,"name":"calvin"...}
- (String) user:102 -> {"id":102,"name":"kevin"...}
- (Hash) user:name:index-> "calvin"->101, "kevin" -> 102

底层实现是hash table，一般操作复杂度是O(1)，要同时操作多个field时就是O(N)，N是field的数量。

2.4 List

List是一个双向链表，支持双向的Pop/Push，江湖规矩一般从左端Push，右端Pop——LPush/RPop，而且还有Blocking的版本BLPop/BRPop，客户端可以阻塞在那直到有消息到来，所有操作都是O(1)的好孩子，可以当Message Queue来用。当多个Client并发阻塞等待，有消息入列时谁先被阻塞谁先被服务。任务队列系统Resque是其典型应用。
还有RPopLPush/ BRPopLPush，弹出来返回给client的同时，把自己又推入另一个list，LLen获取列表的长度。
还有按值进行的操作：LRem(按值删除元素)、LInsert(插在某个值的元素的前后)，复杂度是O(N)，N是List长度，因为List的值不唯一，所以要遍历全部元素，而Set只要O(log(N))。
按下标进行的操作：下标从0开始，队列从左到右算，下标为负数时则从右到左。

- LSet，按下标设置元素值。
- LIndex，按下标返回元素。
- LRange，不同于POP直接弹走元素，只是返回列表内一段下标的元素，是分页的最爱。
- LTrim，限制List的大小，比如只保留最新的20条消息。

复杂度也是O(N)，其中LSet的N是List长度，LIndex的N是下标的值，LRange的N是start的值+列出元素的个数，因为是链表而不是数组，所

以按下标访问其实要遍历链表，除非下标正好是队头和队尾。LTrim的N是移除元素的个数。
在消息队列中，并没有JMS的ack机制，如果消费者把job给Pop走了又没处理完就死机了怎么办?

- 解决方法之一是加多一个sorted set，分发的时候同时发到list与sorted set，以分发时间为score，用户把job做完了之后要用ZREM消掉sorted set里的job，并且定时从sorted set中取出超时没有完成的任务，重新放回list。
- 另一个做法是为每个worker多加一个的list，弹出任务时改用RPopLPush，将job同时放到worker自己的list中，完成时用LREM消掉。如果集群管理(如zookeeper)发现worker已经挂掉，就将worker的list内容重新放回主list。

2.5 Set

Set就是Set，可以将重复的元素随便放入而Set会自动去重，底层实现也是hash table。

- SAdd/SRem/SIsMember/SCard/SMove/SMembers，各种标准操作。除了SMembers都是O(1)。
- SInter/SInterStore/SUnion/SUnionStore/SDiff/SDiffStore，各种集合操作。交集运算可以用来显示在线好友(在线用户 交集 好友列表)，共同关注(两个用户的关注列表的交集)。O(N)，并集和差集的N是集合大小之和，交集的N是小的那个集合的大小*2。

2.6 Sorted Set

有序集，元素放入集合时还要提供该元素的分数，默认从小到大排列。

- ZRange/ZRevRange，按排名的上下限返回元素，正数与倒数。
- ZRangeByScore/ZRevRangeByScore，按分数的上下限返回元素，正数与倒数。
- ZRemRangeByRank/ZRemRangeByScore，按排名/按分数的上下限删除元素。
- ZCount，统计分数上下限之间的元素个数。
- ZRank/ZRevRank ，显示某个元素的正倒序的排名。
- ZScore/ZIncrby，显示元素的分数/增加元素的分数。
- ZAdd(Add)/ZRem(Remove)/ZCard(Count)，ZInsertStore(交

集)/ZUnionStore(并集)，Set操作，与正牌Set相比，少了IsMember和差集运算。

Sorted Set的实现是hash table(element->score, 用于实现ZScore及判断element是否在集合内)，和skip list(score->element,按score排序)的混合体。 skip list有点像平衡二叉树那样，不同范围的score被分成一层一层，每层是一个按score排序的链表。

ZAdd/ZRem是O(log(N))，ZRangeByScore/ZRemRangeByScore是O(log(N)+M)，N是Set大小，M是结果/操作元素的个数。可见，原本可能很大的N被很关键的Log了一下，1000万大小的Set，复杂度也只是几十不到。当然，如果一次命中很多元素M很大那谁也没办法了。

2.7 事务

用Multi(Start Transaction)、Exec(Commit)、Discard(Rollback)实现。在事务提交前，不会执行任何指令，只会把它们存到一个队列里，不影响其他客户端的操作。在事务提交时，批量执行所有指令。《Redis设计与实现》中的详述。

注意，Redis里的事务，与我们平时的事务概念很不一样：

- 它仅仅是保证事务里的操作会被连续独占的执行。因为是单线程架构，在执行完事务内所有指令前是不可能再去同时执行其他客户端的请求的。
- 它没有隔离级别的概念，因为事务提交前任何指令都不会被实际执行，也就不存在"事务内的查询要看到事务里的更新，在事务外查询不能看到"这个让人万分头痛的问题。
- 它不保证原子性——所有指令同时成功或同时失败，只有决定是否开始执行全部指令的能力，没有执行到一半进行回滚的能力。在redis里失败分两种，一种是明显的指令错误，比如指令名拼错，指令参数个数不对，在2.6版中全部指令都不会执行。另一种是隐含的，比如在事务里，第一句是SET foo bar， 第二句是LLEN foo，对第一句产生的String类型的key执行LLEN会失败，但这种错误只有在指令运行后才能发现，这时候第一句成功，第二句失败。还有，如果事务执行到一半redis被KILL，已经执行的指令同样也不会被回滚。

Watch指令，类似乐观锁，事务提交时，如果Key的值已被别的客户端

改变，比如某个list已被别的客户端push/pop过了，整个事务队列都不会被执行。

2.8 Lua Script

Redis2.6内置的Lua Script支持，可以在Redis的Server端一次过运行大量逻辑，就像存储过程一样，避免了海量中间数据在网路上的传输。

- Lua自称是在Script语言里关于快的标准，Redis选择了它而不是流行的JavaScript。
- 因为Redis的单线程架构，整个Script默认是在一个事务里的。
- Script里涉及的所有Key尽量用变量，从外面传入，使Redis一开始就知道你要改变哪些key，为了日后做水平分区做准备。如果涉及的key在不同服务器.....
- Eval每次传输一整段Script比较费带宽，可以先用Script Load载入script，返回哈希值。然后用EvalHash执行。因为就是SHA-1，所以任何时候执行返回的哈希值都是一样的。Replicate时，2.6版还是会传整段Script到Slave，2.8版改进了。
- 内置的Lua库里还很贴心的带了CJSON，可以处理json字符串。
- Script一旦执行则不容易中断，中断了也会有不可知后果，因此最好在开发环境充分测试了再上线。
- 一段用Redis做Timer的示例代码，下面的script被定期调用，从以触发时间为score的sorted set中取出已到期的Job，放到list中给Client们blocking popup。

```
-- KEYS: [1]job:sleeping, [2]job:ready
-- ARGS: [1]currentTime
-- Comments: result is the  job id
local jobs=redis.call('zrangebyscore', KEYS[1], '-inf', ARGV[1])
local count = table.maxn(jobs)

if count>0  then
  -- Comments: remove from Sleeping Job sorted set
  redis.call('zremrangebyscore', KEYS[1], '-inf', ARGV[1])

  -- Comments: add to the Ready Job list
  -- Comments: can optimize to use lpush id1,id2,... for better
performance
  for i=1,count do
    redis.call('lpush', KEYS[2], jobs[i])
  end
```

```
end
```

2.9 过期数据清除

官方文档 与 《Redis设计与实现》中的详述，过期数据的清除从来不容易，为每一条key设置一个timer，到点立刻删除的消耗太大，每秒遍历所有数据消耗也大，Redis使用了一种相对务实的做法：
当client主动访问key会先对key进行超时判断，过时的key会立刻删除。如果clien永远都不再get那条key呢？ 它会在Master的后台，每秒10次的执行如下操作： 随机选取100个key校验是否过期，如果有25个以上的key过期了，立刻额外随机选取下100个key(不计算在10次之内)。可见，如果过期的key不多，它最多每秒回收200条左右，如果有超过25%的key过期了，它就会做得更多，但只要key不被主动get，它占用的内存什么时候最终被清理掉只有天知道。

3. 性能

3.1 测试结果

- 测试环境： RHEL 6.3 / HP Gen8 Server/ 2 * Intel Xeon 2.00GHz(6 core) / 64G DDR3 memory / 300G RAID-1 SATA / 1 master(writ AOF), 1 slave(write AOF & RDB)
- 数据准备： 预加载两千万条数据，占用10G内存。
- 测试工具：自带的redis-benchmark，默认只是基于一个很小的数据集进行测试，调整命令行参数如下，就可以开100条线程(默认50)，SET 1千万次(key在0-1千万间随机)，key长21字节，value长256字节的数据。

```
redis-benchmark -t SET -c 100 -n 10000000 -r 10000000 -d 256
```

- 测试结果(TPS)： 1.SET：4.5万， 2.GET：6万 ， 3.INCR：6万，4.真实混合场景: 2.5万SET & 3万GET
- 单条客户端线程时6千TPS，50与100条客户端线程差别不大，200条时会略多。
- Get/Set操作，经过了LAN，延时也只有1毫秒左右，可以反复放心调用，不用像调用REST接口和访问数据库那样，每多一次外部访问都心痛。
- 资源监控:

1.CPU: 占了一个处理器的100%，总CPU是4%(因为总共有2CPU*6核*超

线程 = 24个处理器)，可见单线程下单处理器的能力是瓶颈。 AOF rewrite时另一个处理器占用50-70%。
2.网卡：15-20 MB/s receive, 3Mb/s send(no slave) or 15-20 MB/s send (with slave) 。当把value长度加到4K时，receive 99MB/s，已经到达千兆网卡的瓶颈，TPS降到2万。
3.硬盘：15MB/s(AOF append), 100MB/s(AOF rewrite/AOF load，普通硬盘的瓶颈),

3.2 为什么快

- 纯ANSI C编写。
- 不依赖第三方类库，没有像memcached那样使用libevent，因为libevent迎合通用性而造成代码庞大，所以作者用libevent中两个文件修改实现了自己的epoll event loop。微软的兼容Windows补丁也因为同样原因被拒了。
- 快，原因之一是Redis多样的数据结构，每种结构只做自己爱做的事，当然比数据库只有Table，MongogoDB只有JSON一种结构快了。
- 可惜单线程架构，虽然作者认为CPU不是瓶颈，内存与网络带宽才是。但实际测试时并非如此，见上。

3.3 性能调优

- 官方文档关于各种产生Latency的原因的详细分析, 中文版
- 正视网络往返时间：

1.MSet/LPush/ZAdd等都支持一次输入多个Key。
2.PipeLining模式 可以一次输入多个指令。在Jedis的实现里，所有指令先在本地的buffer中存着，直到调用sync。
3.更快的是Lua Script模式，还可以包含逻辑，直接在服务端又get又set的，见2.8 Lua Script。

- 发现执行缓慢的命令，可配置执行超过多少时间的指令算是缓慢指令(默认10毫秒，不含IO时间)，可以用slowlog get 指令查看(默认只保留最后的128条)。单线程的模型下，一个请求占掉10毫秒是件大事情，注意设置和显示的单位为微秒。
- CPU永远是瓶颈，但top看到单个CPU 100%时，就是垂直扩展的时候了。

- 持久化对性能的影响很大，见5.1持久化。
- 要熟悉各指令的复杂度，不过只要不是O(N)一个超大集合，都不用太担心。

4. 容量

4.1 最大内存

- 所有的数据都必须在内存中，原来2.0版的VM策略(将Value放到磁盘，Key仍然放在内存)，2.4版后嫌麻烦又不支持了。
- 一定要设置最大内存，否则物理内存用爆了就会大量使用Swap，写RDB文件时的速度慢得你想死。
- 多留一倍内存是最安全的。重写AOF文件和RDB文件的进程(即使不做持久化，复制到Slave的时候也要写RDB)会fork出一条新进程来，采用了操作系统的Copy-On-Write策略(子进程与父进程共享Page。如果父进程的Page-每页4K有修改，父进程自己创建那个Page的副本，不会影响到子进程，父爱如山)。留意Console打出来的报告，如"RDB: 1215 MB of memory used by copy-on-write"。在系统极度繁忙时，如果父进程的所有Page在子进程写RDB过程中都被修改过了，就需要两倍内存。
- 按照Redis启动时的提醒，设置 vm.overcommit_memory = 1 ，使得fork()一条10G的进程时，因为COW策略而不一定需要有10G的free memory。
- 其他需要考虑的内存包括：

1.AOF rewrite过程中对新写入命令的缓存(rewrite结束后会merge到新的aof文件)，留意"Background AOF buffer size: 80 MB"的字样。
2.负责与Slave同步的Client的缓存，默认设置master需要为每个slave预留不高于256M的缓存(见5.1持久化)。

- 当最大内存到达时，按照配置的Policy进行处理， 默认策略为volatile-lru，对设置了expire time的key进行LRU清除(不是按实际expire time)。如果沒有数据设置了expire time或者policy为noeviction，则直接报错，但此时系统仍支持get之类的读操作。 另外还有几种policy，比如volatile-ttl按最接近expire time的，allkeys-lru对所有key都做LRU。见Redis的文档：Redis as an LRU cache

4.2 内存占用

- 测试表明，string类型需要90字节的额外代价，就是说key 1个字节，value 1个字节时，还是需要占用92字节的长度，而上面的benchmark的记录就占用了367个字节。其他类型可根据文档自行计算或实际测试一下。
- 使用jemalloc分配内存，删除数据后，内存并不会乖乖还给操作系统而是被Redis截留下来重用到新的数据上，直到Redis重启。因此进程实际占用内存是看INFO里返回的used_memory_peak_human。
- Redis内部用了ziplist/intset这样的压缩结构来减少hash/list/set/zset的存储，默认当集合的元素少于512个且最长那个值不超过64字节时使用，可配置。
- 用make 32bit可以编译出32位的版本，每个指针占用的内存更小，但只支持最大4GB内存。

4.4 水平分区，Sharding，Partition

- Redis文档：Partitioning
- 其实，大内存加上垂直分区也够了，不一定非要沙丁一把。
- Jedis支持在客户端做分区，局限是不能动态re-sharding， 有分区的master倒了，不能减少分区必须用slave顶上。要增加分区的话，呃.....
- antire在博客里提到了Twemproxy，一个Twitter写的Proxy，但它在发现节点倒掉后，只会重新计算一致性哈希环，把数据存到别的master去，而不是集成Sentinel指向新由slave升级的master，像Memcached一样的做法也只适合做Cache的场景。

Redis-Cluster在3.0版发布，支持自动re-sharding，Redis文档：集群教程

- 采用和Hazelcast类似的算法，总共有N个分区(eg.N=1024)，每台Server负责若干个分区，而且在Server间共享此信息，而不是像Memcached那样每台Server直接在一致性哈希环上占一块地方。
- 在客户端先用一致性哈希出key 属于哪个分区，随便发给一台server，server会告诉它真正哪个Server负责这个分区，客户端缓存下来，下次还有该分区的请求就直接发到地儿了。

- Re-sharding时，会将某些分区的数据移到新的Server上，完成后各Server周知分区<->Server映射的变化，因为分区数量有限，所以通讯量不大。 在迁移过程中，客户端缓存的依然是旧的分区映射信息，原server对于已经迁移走的数据的get请求，会返回一个临时转向的应答，客户端先不会更新Cache。等迁移完成了，就会像前面那样返回一条永久转向信息，客户端更新Cache，以后就都去新server了。
- 目前问题：1. 事务和Lua脚本如果涉及到不同分区的Key如何解决，2. 创建Cluster Node的命令是Ruby写的，要有Ruby才能跑。

5. 高可用性

高可用性关乎系统出错时到底会丢失多少数据，多久不能服务。要综合考虑持久化，Master-Slave复制及Fail-Over配置，以及具体Crash情形，比如Master死了，但Slave没死。或者只是Redis死了，操作系统没死等等。

5.1 持久化

- 综述： 解密Redis持久化(中文概括版), 英文原版，《Redis设计与实现》： RDB 与 AOF。
- 很多人开始会想象两者是互相结合的，即dump出一个snapshot到RDB文件，然后在此基础上记录变化日志到AOF文件。实际上两者毫无关系，完全独立运行，因为作者认为简单才不会出错。如果使用了AOF，重启时只会从AOF文件载入数据，不会再管RDB文件。
- 正确关闭服务器：redis-cli shutdown 或者 kill，都会graceful shutdown，保证写RDB文件以及将AOF文件fsync到磁盘，不会丢失数据。 如果是粗暴的Ctrl+C，或者kill -9 就可能丢失。

5.1.1 RDB文件

- RDB是整个内存的压缩过的Snapshot，RDB的数据结构，可以配置复合的快照触发条件，默认是1分钟内改了1万次，或5分钟内改了10次，或15分钟内改了1次。
- RDB写入时，会连内存一起Fork出一个新进程，遍历新进程内存中的数据写文件，这样就解决了些Snapshot过程中又有新的写入请求进来的问题。 Fork的细节见4.1最大内存。
- RDB会先写到临时文件，完了再Rename成，这样外部程序对

RDB文件的备份和传输过程是安全的。而且即使写新快照的过程中Server被强制关掉了，旧的RDB文件还在。

- 可配置是否进行压缩，压缩方法是字符串的LZF算法，以及将string形式的数字变回int形式存储。
- 动态所有停止RDB保存规则的方法：redis-cli config set save ""

5.1.2 AOF文件

- 操作日志，记录所有有效的写操作，等于mysql的binlog，格式就是明文的Redis协议的纯文本文件。
- 一般配置成每秒调用一次fdatasync将kernel的文件缓存刷到磁盘。当操作系统非正常关机时，文件可能会丢失不超过2秒的数据(更严谨的定义见后)。 如果设为fsync always，性能只剩几百TPS，不用考虑。如果设为no，靠操作系统自己的sync，Linux系统一般30秒一次。
- AOF文件持续增长而过大时，会fork出一条新进程来将文件重写(也是先写临时文件，最后再rename，)， 遍历新进程的内存中数据，每条记录有一条的Set语句。默认配置是当AOF文件大小是上次rewrite后大小的一倍，且文件大于64M时触发。
- Redis协议，如set mykey hello， 将持久化成*3 $3 set $5 mykey $5 hello， 第一个数字代表这条语句有多少元，其他的数字代表后面字符串的长度。这样的设计，使得即使在写文件过程中突然关机导致文件不完整，也能自我修复，执行redis-check-aof即可。

综上所述，RDB的数据不实时，同时使用两者时服务器重启也只会找AOF文件。那要不要只使用AOF呢？作者建议不要，因为RDB更适合用于备份数据库(AOF在不断变化不好备份)，快速重启，而且不会有AOF可能潜在的bug，留着作为一个万一的手段。

5.1.3 读写性能

- AOF重写和RDB写入都是在fork出新进程后，遍历新进程的内存顺序写的，既不阻塞主进程继续处理客户端请求，顺序写的速度也比随机写快。
- 测试把刚才benchmark的11G数据写成一个1.3的RDB文件，或者等大的AOF文件rewrite，需要80秒，在redis-cli info中可查看。启动时载入一个AOF或RDB文件的速度与上面写入时相同，在log中可查

看。

- Fork一个使用了大量内存的进程也要时间，大约10ms per GB的样子，但Xen在EC2上是让人郁闷的239ms (KVM和VMWare貌似没有这个毛病)，各种系统的对比，Info指令里的latest_fork_usec显示上次花费的时间。
- 在bgrewriteaof过程中，所有新来的写入请求依然会被写入旧的AOF文件，同时放到buffer中，当rewrite完成后，会在主线程把这部分内容合并到临时文件中之后才rename成新的AOF文件，所以rewrite过程中会不断打印"Background AOF buffer size: 80 MB，Background AOF buffer size: 180 MB"，计算系统容量时要留意这部分的内存消耗。注意，这个合并的过程是阻塞的，如果你产生了280MB的buffer，在100MB/s的传统硬盘上，Redis就要阻塞2.8秒！！！
- NFS或者Amazon上的EBS都不推荐，因为它们也要消耗带宽。
- bgsave和bgaofrewrite不会被同时执行，如果bgsave正在执行，bgaofrewrite会自动延后。
- 2.4版以后，写入AOF时的fdatasync由另一条线程来执行，不会再阻塞主线程。
- 2.4版以后，lpush/zadd可以输入一次多个值了，使得AOF重写时可以将旧版本中的多个lpush/zadd指令合成一个，每64个key串一串。

5.1.4 性能调整

因为RDB文件只用作后备用途，建议只在Slave上持久化RDB文件，而且只要15分钟备份一次就够了，只保留save 900 1这条规则。

如果Enalbe AOF，好处是在最恶劣情况下也只会丢失不超过两秒数据，启动脚本较简单只load自己的AOF文件就可以了。代价一是带来了持续的IO，二是AOF rewrite的最后将rewrite过程中产生的新数据写到新文件造成的阻塞几乎是不可避免的。只要硬盘许可，应该尽量减少AOF rewrite的频率，AOF重写的基础大小默认值64M太小了，可以设到5G以上。默认超过原大小100%大小时重写可以改到适当的数值，比如之前的benchmark每个小时会产生40G大小的AOF文件，如果硬盘能撑到半夜系统闲时才用cron调度bgaofrewrite就好了。

如果不Enable AOF ，仅靠Master-Slave Replication 实现高可用性也可以。能省掉一大笔IO也减少了rewrite时带来的系统波动。代价是如果Master/Slave同时倒掉，会丢失十几分钟的数据，启动脚本也要比较两个Master/Slave中的RDB文件，载入较新的那个。新浪微博就选用了这种架构，见Tim的博客

5.1.5 Trouble Shooting —— Enable AOF可能导致整个Redis被Block住，在2.6.12版之前

现象描述：当AOF rewrite 15G大小的内存时，Redis整个死掉的样子，所有指令甚至包括slave发到master的ping，redis-cli info都不能被执行。

原因分析：

- 官方文档，由IO产生的Latency详细分析, 已经预言了悲剧的发生，但一开始没留意。
- Redis为求简单，采用了单请求处理线程结构。
- 打开AOF持久化功能后， Redis处理完每个事件后会调用write(2)将变化写入kernel的buffer，如果此时write(2)被阻塞，Redis就不能处理下一个事件。
- Linux规定执行write(2)时，如果对同一个文件正在执行fdatasync(2)将kernel buffer写入物理磁盘，或者有system wide sync在执行，write(2)会被block住，整个Redis被block住。
- 如果系统IO繁忙，比如有别的应用在写盘，或者Redis自己在AOF rewrite或RDB snapshot(虽然此时写入的是另一个临时文件，虽然各自都在连续写，但两个文件间的切换使得磁盘磁头的寻道时间加长），就可能导致fdatasync(2)迟迟未能完成从而block住write(2)，block住整个Redis。
- 为了更清晰的看到fdatasync(2)的执行时长，可以使用"strace -p (pid of redis server) -T -e -f trace=fdatasync"，但会影响系统性能。
- Redis提供了一个自救的方式，当发现文件有在执行fdatasync(2)时，就先不调用write(2)，只存在cache里，免得被block。但如果已经超过两秒都还是这个样子，则会硬着头皮执行write(2)，即使redis会被block住。此时那句要命的log会打印：“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.”之后用redis-cli INFO可以看到aof_delayed_fsync的值被加1。

- 因此，对于fsync设为everysec时丢失数据的可能性的最严谨说法是：如果有fdatasync在长时间的执行，此时redis意外关闭会造成文件里不多于两秒的数据丢失。如果fdatasync运行正常，redis意外关闭没有影响，只有当操作系统crash时才会造成少于1秒的数据丢失。

解决方法：

最后发现，原来是AOF rewrite时一直埋头的调用write(2)，由系统自己去触发sync。在RedHat Enterprise 6里，默认配置vm.dirty_background_ratio=10，也就是占用了10%的可用内存才会开始后台flush，而我的服务器有64G内存。很明显一次flush太多数据会造成阻塞，所以最后果断设置了sysctl vm.dirty_bytes=33554432(32M)，问题解决。

然后提了个issue，AOF rewrite时定时也执行一下fdatasync嘛， antirez三分钟后就回复了，新版中，AOF rewrite时32M就会重写主动调用fdatasync。

5.2 Master-Slave复制

5.2.1 概述

- slave可以在配置文件、启动命令行参数、以及redis-cli执行SlaveOf指令来设置自己是奴隶。
- 测试表明同步延时非常小，指令一旦执行完毕就会立刻写AOF文件和向Slave转发，除非Slave自己被阻塞住了。
- 比较蠢的是，即使在配置文件里设了slavof，slave启动时依然会先从数据文件载入一堆没用的数据，再去执行slaveof。
- "Slaveof no one"，立马变身master。
- 2.8版本将支持PSYNC部分同步，master会拨出一小段内存来存放要发给slave的指令，如果slave短暂的断开了，重连时会从内存中读取需要补读的指令，这样就不需要断开两秒也搞一次全同步了。但如果断开时间较长，已经超过了内存中保存的数据，就还是要全同步。
- Slave也可以接收Read-Only的请求。

5.2.2 slaveof执行过程，完全重用已有功能，非常经济

- 先执行一次全同步 -- 请求master BgSave出自己的一个RDB Snapshot文件发给slave，slave接收完毕后，清除掉自己的旧数据，然后将RDB载入内存。
- 再进行增量同步 -- master作为一个普通的client连入slave，将所有写操作转发给slave，没有特殊的同步协议。

5.2.3 Trouble Shooting again

有时候明明master/slave都活得好好的，突然间就说要重新进行全同步了：

1.Slave显示：# MASTER time out: no data nor PING received...
slave会每隔repl-ping-slave-period(默认10秒)ping一次master，如果超过repl-timeout(默认60秒)都没有收到响应，就会认为Master挂了。如果Master明明没挂但被阻塞住了也会报这个错。可以适当调大repl-timeout。

2.Master显示：# Client addr=10.175.162.123:44670 flags=S oll=104654 omem=2147487792 events=rw cmd=sync scheduled to be closed ASAP for overcoming of output buffer limits.
当slave没挂但被阻塞住了，比如正在loading Master发过来的RDB，Master的指令不能立刻发送给slave，就会放在output buffer中(见oll是命令数量，omem是大小)，在配置文件中有如下配置：client-output-buffer-limit slave 256mb 64mb 60， 这是说负责发数据给slave的client，如果buffer超过256m或者连续60秒超过64m，就会被立刻强行关闭！！！ Traffic大的话一定要设大一点。否则就会出现一个很悲剧的循环，Master传输一个大的RDB给Slave，Slave努力的装载，但还没装载完，Master对client的缓存满了，再来一次。

平时可以在master执行 redis-cli client list 找那个cmd=sync，flag=S的client，注意OMem的变化。

5.3 Fail-Over

Redis-sentinel是2.6版开始加入的另一组独立运行的节点，提供自动Fail Over的支持。

- 官方文档 与 Redis核心解读–集群管理工具(Redis-sentinel)
- antirez 对 Sentinel的反驳，与下篇

5.3.1 主要执行过程

- Sentinel每秒钟对所有master，slave和其他sentinel执行Ping，redis-server节点要应答+PONG或-LOADING或-MASTERDOWN.
- 如果某一台Sentinel没有在30秒内(可配置得短一些哦)收到上述正确应答，它就会认为master处于sdown状态(主观Down)
- 它向其他sentinel询问是否也认为该master倒了（SENTINEL is-master-down-by-addr ）， 如果quonum台(默认是2)sentinel在5秒钟内都这样认为，就会认为master真是odown了(客观Down)。
- 此时会选出一台sentinel作为Leader执行fail-over, Leader会从slave中选出一个提升为master(执行slaveof no one)，然后让其他slave指向它(执行slaveof new master)。

5.3.2 master/slave 及 其他sentinel的发现

master地址在sentinel.conf里, sentinel会每10秒一次向master发送INFO，知道master的slave有哪些。 如果master已经变为slave，sentinel会分析INFO的应答指向新的master。以前，sentinel重启时，如果master已经切换过了，但sentinel.conf里master的地址并没有变，很可能有悲剧发生。另外master重启后如果没有切换成slave，也可能有悲剧发生。新版好像修复了一点这个问题，待研究。

另外，sentinel会在master上建一个pub/sub channel，名为"sentinel:hello"，通告各种信息，sentinel们也是通过接收pub/sub channel上的+sentinel的信息发现彼此，因为每台sentinel每5秒会发送一次自己的host信息，宣告自己的存在。

5.3.3 自定义reconfig脚本

- sentinel在failover时还会执行配置文件里指定的用户自定义reconfig脚本，做用户自己想做的事情，比如让master变为slave并指向新的master。
- 脚本的将会在命令行按顺序传入如下参数： <master-name> <role(leader/observer)> <state(上述三种情况)> <from-ip> <from-port> <to-ip> <to-port>
- 脚本返回0是正常，如果返回1会被重新执行，如果返回2或以上不会。 如果超过60秒没返回会被强制终止。

觉得Sentinel至少有两个可提升的地方:

- 一是如果master 主动shutdown，比如系统升级，有办法主动通知sentinel提升新的master，减少服务中断时间。
- 二是比起redis-server太原始了，要自己丑陋的以nohup sentinel > logfile 2>&1 & 启动，也不支持shutdown命令，要自己kill pid。

5.4 Client的高可用性

基于Sentinel的方案，client需要执行语句SENTINEL get-master-addr-by-name mymaster 可获得当前master的地址。 Jedis正在集成sentinel，已经支持了sentinel的一些指令，但还没发布，但sentinel版的连接池则暂时完全没有，在公司的项目里我参考网友的项目自己写了一个。

淘宝的Tedis driver，使用了完全不同的思路，不基于Sentinel，而是多写随机读， 一开始就同步写入到所有节点，读的话随便读一个还活着的节点就行了。但有些节点成功有些节点失败如何处理? 节点死掉重新起来后怎么重新同步?什么时候可以重新Ready? 所以不是很敢用。

另外如Ruby写的redis_failover，也是抛开了Redis Sentinel，基于ZooKeeper的临时方案。

Redis作者也在博客里抱怨怎么没有人做Dynamo-style 的client。

5.5 Geographic Replication

依然用Master Slave复制，支持Active-Standby模式的Geographic Replication，主要用于容灾数据恢复，或者在site1倒掉时，启动备用系统指向备库。3Scale想出了诸如用压缩的SSH隧道降低传输量等方法，可以设置远端的Slave的优先级为0，则site2上的slave永远不会被选举成master，master只会在site1的slave中产生。

6. 运维

6.1 安装

- 安装包制作：没有现成，需要自己编译，自己写rpm包的脚本，可参考utils中的install_server.sh与redis_init_script。

但RHEL下设定script runlevel的方式不一样，redis_init_script中要增加一句 "# chkconfig: 345 90 10" ，而install_server.sh可以删掉后面的那句“chkconfig --level 345 reis"

- 云服务：Redis Cloud，在Amazon、Heroku、Windows Azure、

App Frog上提供云服务，供同样部署在这些云上的应用使用。其他的云服务有GarantiaData，已被redis-cloud收购。另外还有Redis To Go, OpenRedis, RedisGreen。

- CopperEgg统计自己的用户在AWS上的数据库部署：mysqld占了50%半壁江山, redis占了18%排第二, mongodb也有11%, cassandra是3%，Oracle只有可怜的2%。
- Chef Recipes：brianbianco/redisio，活跃，同步更新版本。

6.2 部署模型

- Redis只能使用单线程，为了提高CPU利用率，有提议在同一台服务器上启动多个Redis实例，但这会带来严重的IO争用，除非Redis不需要持久化，或者有某种方式保证多个实例不会在同一个时间重写AOF。
- 一组sentinel能同时监控多个Master。
- 有提议说环形的slave结构，即master只连一个slave，然后slave再连slave，此部署有两个前提，一是有大量的只读需求需要在slave完成，二是对slave传递时的数据不一致性不敏感。

6.3 配置

约30个配置项，全都有默认配置，对redif.conf默认配置的修改见附录1。

6.3.1 三条路

- 可以配置文件中编写。
- 可以在启动时的命令行配置，redis-server --port 7777 --slaveof 127.0.0.1 8888。
- 云时代大规模部署，把配置文件满街传显然不是好的做法， 可以用redis-cli执行Config Set指令， 修改所有的参数，达到维护人员最爱的不重启服务而修改参数的效果，而且新里还可以执行 Config Rewrite 将改动写回到文件中，原配置文件里有的项会就地更改，新的项而且不是默认值的，写在文件最后。如果写入过程中crash，所有修改都不会发生。

6.3.2 安全保护

- 在配置文件里设置密码：requirepass foobar。
- 禁止某些危险命令，比如残暴的FlushDB，将它rename成""：

rename-command FLUSHDB ""。

6.4 监控与维护

综述： Redis监控技巧

6.4.1 监控指令

Info指令将返回非常丰富的信息。 着重监控检查内存使用，是否已接近上限，used_memory是Redis申请的内存，used_memory_rss是操作系统分配给Redis的物理内存，两者之间隔着碎片，隔着Swap。 还有重点监控 AOF与RDB文件的保存情况，以及master-slave的关系。Statistic信息还包括key命中率，所有命令的执行次数，所有client连接数量等，CONFIG RESETSTAT 可重置为0。

Monitor指令可以显示Server收到的所有指令，主要用于debug，影响性能，生产环境慎用。

SlowLog 检查慢操作(见2.性能)。

6.4.2 Trouble Shooting支持

- 日志可以动态的设置成verbose/debug模式，但不见得有更多有用的log可看,verbose还会很烦的每5秒打印当前的key情况和client情况。指令为config set loglevel verbose。
- 最爱Redis的地方是代码只有2.3万行，而且编码优美，而且huangz同学还在原来的注释上再加上了中文注释——Redis 2.6源码中文注释版 ，所以虽然是C写的代码，虽然有十年没看过C代码，但这几天trouble shooting毫无难度，一看就懂。
- Trobule shotting的经历证明antirez处理issue的速度非常快(如果你的issue言之有物的话)，比Weblogic之类的商业支持还好。

6.4.3 持久化文件维护

- 如果AOF文件在写入过程中crash，可以用redis-check-aof修复，见5.1.2
- 如果AOF rewrite和 RDB snapshot的过程中crash，会留下无用的临时文件，需要定期扫描删除。

6.4.4 三方工具

官网列出了如下工具，但暂时没发现会直接拿来用的：

- Redis Live，基于Python的web应用，使用Info和Monitor获得系统情况和指令统计分析。 因为Monitor指令影响性能，所以建议用cron

定期运行，每次偷偷采样两分钟的样子。

- phpRedisAdmin，基于php的Web应用，目标是MysqlAdmin那样的管理工具，可以管理每一条Key的情况，但它的界面应该只适用于Key的数量不太多的情况，Demo。
- Redis Faina，基于Python的命令行，Instagram出品，用户自行获得Monitor的输出后发给它进行统计分析。由于Monitor输出的格式在Redis版本间不一样，要去github下最新版。
- Redis-rdb-tools 基于Python的命令行，可以分析RDB文件每条Key对应value所占的大小，还可以将RDB dump成普通文本文件然后比较两个库是否一致，还可以将RDB输出成JSON格式，可能是最有用的一个了。
- Redis Sampler，基于Ruby的命令行，antirez自己写的，统计数据分布情况。

7. Java Driver

7.1 Driver选择

各个Driver好像只有Jedis比较活跃，但也5个月没提交了，也是Java里唯一的Redis官方推荐。

Spring Data Redis的封装并不太必要，因为Jedis已足够简单，没有像Spring Data MongoDB对MongoDB java driver的封装那样大幅简化代码，顶多就是加强了一点点点pipeline和transaction状态下的coding，禁止了一些此状态下不能用的命令。而所谓屏蔽各种底层driver的差异并不太吸引人，因为我就没打算选其他几种driver。有兴趣的可以翻翻它的JedisConnection代码。

所以，SpringSide直接在Jedis的基础上，按Spring的风格封装了一个JedisTemplate，负责从池中获取与归还Jedis实例，处理异常。

7.2 Jedis的细节

Jedis基于Apache Commons Pool做的连接池，默认MaxActive最大连接数只有8，必须重新设置。而且MaxIdle也要相应增大，否则所有新建的连接用完即弃，然后会不停的重新连接。

另外Jedis设定了每30秒对所有连接执行一次ping，以发现失效的连接，这样每30秒会有一个拿不到连接的高峰。但效果如何需要独立分析。比

如系统高峰之后可能有一长段时间很闲，而且Redis Server那边做了Timeout控制会把连接断掉，这时候做idle checking是有意义的，但30秒一次也太过频繁了。否则关掉它更好。
Jedis的blocking pop函数，应用执行ExecutorService.shutdownNow()中断线程时并不能把它中断，见讨论组。两个解决方法：

- 不要用不限时的blocking popup，传多一个超时时间参数，如5秒。
- 找地方将调用blocking popup的jedis保存起来，shutdown时主动调用它的close。

7.3 Redis对Client端连接的处理

- Redis默认最大连接数是一万。
- Redis默认不对Client做Timeout处理，可以用timeout 项配置，但即使配了也不会非常精确。

8. Windows的版本

Windows版本方便对应用的本地开发调试，但Redis并没有提供，好在微软提供了一个依赖LibUV实现兼容的补丁，https://github.com/MSOpenTech/redis ，但redis作者拒绝合并到master中，微软只好苦憋的时时人工同步。 目前的稳定版是2.6版本，支持Lua脚本。
因为github现在已经没有Download服务了，所以编译好的可执行文件藏在这里：

- https://github.com/MSOpenTech/redis/tree/2.6/bin/release
- https://github.com/MSOpenTech/redis/tree/2.8/bin/release

9. 单元测试、集成测试

NoSQL Unit 是使用了Redis的项目的福音，它提供三个功能： 1. 嵌入式的Jedis实例，用于单元测试。在springside-extension的JedisTemplateTest里使用了它。 1. ManagedRedis，可控制已安装在机器上的redis，可用集成测试。将在下个迭代试用。 1. 将数据定义在json文件里，可以在测试时装载数据或校验redis中的数据。
但嵌入式的Redis，不能模仿Lua脚本。而ManagedRedis也不支持Windows上的Redis。

10. 成功案例

注：下文中的链接都是网站的架构描述文档。

Twitter和新浪微博， 都属于将Redis各种数据结构用得出神入化的那种，如何发布大V如奥巴马的消息是它们最头痛的问题。

Tumblr： 11亿美刀卖给Yahoo的图片日志网站，22 台Redis server，每台运行8 - 32个实例，总共100多个Redis实例在跑。有着Redis has been completely problem free and the community is great的崇高评价。Redis在里面扮演了八爪鱼多面手的角色：

- Dashboard的海量通知的存储。
- Dashboard的二级索引。
- 存储海量短链接的HBase前面的缓存。
- Gearman Job Queue的存储。
- 正在替换另外30台memcached。

Instagram ， 曾经，Redis powers their main feed, activity feed, sessions system, and other services。但可惜目前已迁往Cassandra，说新架构只需1/4的硬件费用，是的，就是那个导致Digg CTO辞职的Canssandra。

Flickr , 依然是asynchronous task system and rudimentary queueing system。之前Task system放在mysql innodb，根本，撑不住。

The Others：

- Pinterest，混合使用MySQL、Membase与Redis作为存储。
- Youporn.com，100%的Redis，MySQL只用于创建新需求用到的sorted set，300K QPS的大压力。
- 日本微信 ，Redis在前负责异步Job Queue和O(n)的数据，且作为O(n*t)数据的cache，HBase在后，负责O(n*t)数据， n是用户，t是时间。
- StackOverflow ，2 Redis servers for distribute caching，好穷好轻量。
- Github，任务系统Resque的存储。
- Digg，用来做页面计数器之类的。
- Discourge，号称是为下一个十年打造的论坛系统， We use Redis for our job queue, rate limiting, as a cache and for transient

data，刚好和我司的用法一样。

- 情色网站 YouPorn，使用 Redis 进行数据存储，Redis 服务器每秒处理30万个页面请求，每小时会记录8-15GB数据。

11. In SpringSide

extension modules项目封装了常用的函数与场景，showcase example的src/demo/redis目录里有各场景的benchmark测试。

11.1 Jedis Template

典型的Spring Template风格，和JdbcTemplate，HibernateTemplate一样，封装从JedisPool获取与归还Connecton的代码，有带返回值与无返回值两种返回接口。同时，对最常用的Jedis调用，直接封装了一系列方法。

11.2 Scheduler与Master Elector

Scheduler实现了基于Redis的高并发单次定时任务分发。具体选型见Scheduler章节。

Master Elector基于redis setNx()与expire()两个api实现，与基于Zookeeper，Hazelcast实现的效果类似。

11.3 Showcase中的Demo

计有Session，Counter，Scheduler 与 Master Elector四款。

12. What is new in Redis 2.8

- 全新的Sentinel实现，Setinel会在自己配置文件持久化谁是最新master，也会让每台redis-server持久化新的mater，重新起来的节点如果错误的把自己当成maser，sentinel也会发出slaveof的指令纠正它，原来要自行实现的脚本都可以去掉了。
- CONFIG REWRITE将命令行CONFIG SET动态改变的配置写回到配置文件。
- SCAN命令，分页匹配遍历所有的Key。
- EVALSHA，用sha1执行的Lua脚本可被直接Replicate，不需要翻译成完整的Lua脚本再replicate，节约了Replicate的带宽。
- 提升了Expired keys的收集算法，在CPU繁忙时也不会累积大量超时的Key??
- 说了很久的半同步，Slave短时间断开后不需要做全同步。

- 支持IPv6，并可绑定多个IP地址。
- 进程名会带上端口号和是否子进程的标识，ps时能分得清一台机器上的多个redis，或者是fork出来的bgrewrite/bgrewriteaof字进程了。

其他用不上的new feature：

- PUBSUB 收取keyspace 更新信息.
- Masters can stop accepting writes if not enough slaves with a given maximum latency are connected. 默认为0.

附录

附录1： 对redis.conf默认配置的修改

Master上

- maxmemory，设置为可用内存的一半.
- logfile stdout -> /var/log/redis/redis.log ，指定日志文件
- dir ./ -> /var/lib/redis，指定持久化文件及临时文件目录.
- (可选)daemonize no -> yes ，启动daemonize模式，注意如果用daemon工具启动redis-server时设回false。
- appendonly no->yes，打开AOF文件.
- auto-aof-rewrite-percentage 100, 综合考虑硬盘大小，可接受重启加载延时等尽量的大，减少AOF rewrite频率.
- auto-aof-rewrite-min-size 64mb，同上，起码设为5G. *(可选)注释掉RDB的所有触发规则，在Master不保存RDB文件。
- client-output-buffer-limit slave 256mb 64mb 60. 考虑Traffic及Slave同步是RDB加载所需时间，正确设置避免buffer撑爆client被关掉后又要重新进行全同步。
- 安全配置，可选。

Slave上

- 设置RDB保存频率，因为RDB只作为Backup工具，设置为30分钟保存一次就够了save 1800 1。
- repl-timeout 60, 适当加大比如120，避免master实际还没倒掉就认为master倒了。
- (可选)slaveof 设置master地址，也可动态设定。

附录2：版本变更历史

- 3.1版 20 2014-3-24 增加Redis 2.8内容
- 3.0.1版-3.0.3版 2013-8-1-2014-1-24，在微博发布后反应良好，持续修改。
- 3.0版 2013-6-29，在公司Workshop后修订，提高wiki的可读性而不只是简单的记录知识点。

附录3：其他参考资料

- Redis的几个认识误区 by Tim yang。