**Redis高效原因：**

|  |
| --- |
| Redis本事是一个内存的数据库，将所有的数据都存在内存里面，这也是Redis高效的原因，内存的访问速度是很快的。 |

**Redis持久化出现的原因:**

|  |
| --- |
| 如果部署了Redis的服务down机，或者网络故障导致内存出现问题，这些问题出现会导致内存丢失，那么我们希望在出现这些场景时，数据任然不会丢失，比如热点数据，或者高峰时期访问比较频繁的数据，如果Redis机器down了，重启之后数据没有了，那么对应用的打击是巨大的，访问不到时，全部会穿透到数据库层，这个时候整个架构可能会雪崩；或者有一些重要数据保存在Redis中，当Redis实现我们希望这些数据能够保存，基于这些场景，Redis提供了两种持久化策略。 |

**一、redis持久化机制**

redis提供了两种持久化策略：RDB&AOF，**两种持久化策略可以同时使用，也可以使用其中一种。如果同时使用的话， 那么Redis重启时，会优先使用AOF文件来还原数据**

**①RDB**

RDB的持久化策略： 按照规则定时将内存的数据同步到磁盘

snapshot：redis在指定的情况下会触发快照

|  |
| --- |
| 1. 自己配置的快照规则：在Redis的配置文件redis.conf中  save <seconds> <changes>  save 900 1  当在900秒内被更改的key的数量大于1的时候，就执行快照  save 300 10  save 60 10000  2. save或者bgsave  save: 执行内存的数据同步到磁盘的操作，这个操作会阻塞客户端的请求  bgsave: 在后台异步执行快照操作，这个操作不会阻塞客户端的请求  3. 执行flushall的时候  清除内存的所有数据，只要快照的规则不为空，也就是第一个规则存在。那么redis会执行快照  4. 执行复制的时候 |

**快照的实现原理**

|  |
| --- |
| 1.快照是什么，存在哪里？  快照就是一个文件，存在你redis/bin下的dump.rdb，快照就是在当前状态所存储的数据  2.redis使用fork函数复制一份当前进程的副本(子进程)  3.父进程继续接收并处理客户端发来的命令，而子进程开始将内存中的数据写入硬盘中的临时文件  4.当子进程写入完所有数据后会用该临时文件替换旧的RDB文件，至此，一次快照操作完成。   注意：redis在进行快照的过程中不会修改RDB文件，只有快照结束后才会将旧的文件替换成新的，也就是说任何时候RDB文件都是完整的。 这就使得我们可以通过定时备份RDB文件来实现redis数据库的备份， RDB文件是经过压缩的二进制文件，占用的空间会小于内存中的数据，更加利于传输。 |

**RDB的优缺点**

1. 缺点

|  |
| --- |
| 使用RDB方式实现持久化，一旦Redis异常退出，就会丢失最后一次快照以后更改的所有数据。这个时候我们就需要根据具体的应用场景，通过组合设置自动快照条件的方式来将可能发生的数据损失控制在能够接受范围。如果数据相对来说比较重要，希望将损失降到最小，则可以使用AOF方式进行持久化 |

2. 优点

|  |
| --- |
| RDB可以最大化Redis的性能：父进程在保存RDB文件时唯一要做的就是fork出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无序执行任何磁盘I/O操作。同时这个也是一个缺点，如果数据集比较大的时候，fork可以能比较耗时，造成服务器在一段时间内停止处理客户端的请求； |

**实践**

1.修改redis.conf中的appendonly yes; 重启后执行对数据的变更命令， 会在bin目录下生成对应的.aof文件， aof文件中会记录所有的操作命令

2.如下两个参数可以去对aof文件做优化

auto-aof-rewrite-percentage 100; 表示当前aof文件大小超过上一次aof文件大小的百分之多少的时候会进行重写。如果之前没有重写过，以启动时aof文件大小为准

auto-aof-rewrite-min-size 64mb; 限制允许重写最小aof文件大小，也就是文件大小小于64mb的时候，不需要进行优化

**②AOF**

AOF可以将Redis执行的每一条写命令追加到硬盘文件中，这一过程显然会降低Redis的性能，但大部分情况下这个影响是能够接受的，另外使用较快的硬盘可以提高AOF的性能

**实践**

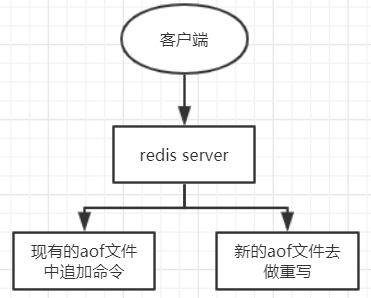
默认情况下Redis没有开启AOF（append only file）方式的持久化，可以通过appendonly参数启用，在redis.conf中找到 appendonly yes

开启AOF持久化后每执行一条会更改Redis中的数据的命令后，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是apendonly.aof. 可以在redis.conf中的属性 appendfilename appendonlyh.aof修改

IMG_256

**aof重写的原理**

Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松



**同步磁盘数据**

redis每次更改数据的时候， aof机制都会讲命令记录到aof文件，但是实际上由于操作系统的缓存机制，数据并没有实时写入到硬盘，而是进入硬盘缓存。再通过硬盘缓存机制去刷新到保存到文件，这个短暂的过程还是会出现数据的丢失，可以在配置文件redis.conf中配置来避免这种情况：

# appendfsync always  每次执行写入都会进行同步，这个是最安全但是是效率比较低的方式

appendfsync everysec   每一秒执行，用的最多，安全和效率都能保证

# appendfsync no  不主动进行同步操作，由操作系统去执行，这个是最快但是最不安全的方式

**aof文件损坏以后如何修复**

服务器可能在程序正在对 AOF 文件进行写入时停机，如果停机造成了AOF文件出错（corrupt）， 那么 Redis 在重启时会拒绝载入这个AOF文件，从而确保数据的一致性不会被破坏。

当发生这种情况时，可以用以下方法来修复出错的AOF文件：

1. 为现有的AOF文件创建一个备份。

2. 使用Redis附带的redis-check-aof程序，对原来的AOF文件进行修复。

**redis-check-aof --fix**

重启Redis服务器，等待服务器载入修复后的AOF文件，并进行数据恢复。

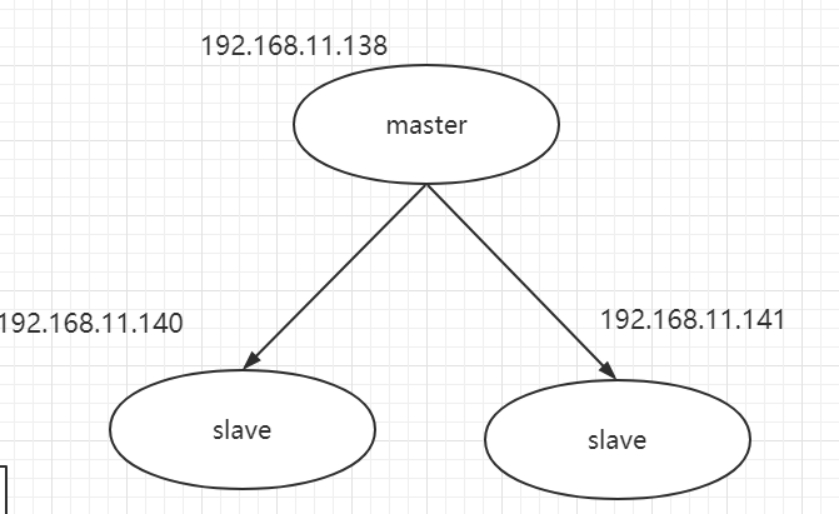
**RDB和AOF,如何选择**

一般来说,如果对数据的安全性要求非常高的话，应该同时使用两种持久化功能。如果可以承受数分钟以内的数据丢失，那么可以只使用 RDB 持久化。有很多用户都只使用 AOF 持久化， 但并不推荐这种方式：因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快 。

**两种持久化策略可以同时使用，也可以使用其中一种。如果同时使用的话， 那么Redis重启时，会优先使用AOF文件来还原数据**

**集群**

**复制（master、slave）**

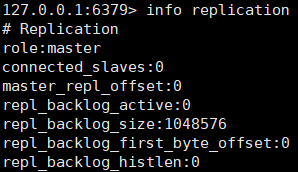


**配置过程**

修改192.168.153.130和192.168.153.131的redis.conf文件，增加slaveof masterip masterport

slaveof 192.168.153.129 6379

**查看角色，通过命令：info replication**



**实现原理**

1. slave第一次或者重连到master上以后，会向master发送一个SYNC的命令

2. master收到SYNC的时候，会做两件事

a) 执行bgsave（rdb的快照文件）

b) master会把新收到的修改命令存入到缓冲区

3.默认情况下，slave会对同步前的命令进行相应，为避免脏数据，可以修改配置文件参数:slave-serve-stale-data

缺点

没有办法对master进行动态选举,因此可以采用哨兵机制

**复制的方式**

1. 基于rdb文件的复制（第一次连接或者重连的时候）

2. 无硬盘复制:repl-diskless-sync

3. 增量复制

PSYNC master run id. offset

**哨兵机制**

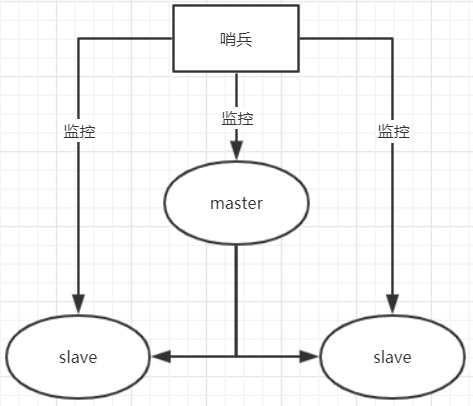
将192.168.153.129设置为主，192.168.153.130设置为从，在192.168.153.129上，配置sentinel.conf：sentinel monitor mymaster 192.168.153.129 6379 1

运行命令开启哨兵模式：./redis-sentinel ../sentinel.conf

IMG_260

1. 监控master和salve是否正常运行

2. 如果master出现故障，那么会把其中一台salve数据升级为master



**集群（redis3.0以后的功能）**

根据key的hash值取模 服务器的数量 。

hash

**集群的原理**

Redis Cluster中，Sharding采用slot(槽)的概念，一共分成16384个槽，这有点儿类似前面讲的pre sharding思路。对于每个进入Redis的键值对，根据key进行散列，分配到这16384个slot中的某一个中。使用的hash算法也比较简单，就是CRC16后16384取模。Redis集群中的每个node(节点)负责分摊这16384个slot中的一部分，也就是说，每个slot都对应一个node负责处理。当动态添加或减少node节点时，需要将16384个槽做个再分配，槽中的键值也要迁移。当然，这一过程，在目前实现中，还处于半自动状态，需要人工介入。Redis集群，要保证16384个槽对应的node都正常工作，如果某个node发生故障，那它负责的slots也就失效，整个集群将不能工作。为了增加集群的可访问性，官方推荐的方案是将node配置成主从结构，即一个master主节点，挂n个slave从节点。这时，如果主节点失效，Redis Cluster会根据选举算法从slave节点中选择一个上升为主节点，整个集群继续对外提供服务。这非常类似服务器节点通过Sentinel监控架构成主从结构，只是Redis Cluster本身提供了故障转移容错的能力。

slot（槽）的概念，在redis集群中一共会有16384个槽，

根据key 的CRC16算法，得到的结果再对16384进行取模。 假如有3个节点

node1  0 5460

node2  5461 10922

node3  10923 16383

节点新增

node4  0-1364,5461-6826,10923-12287

删除节点

先将节点的数据移动到其他节点上，然后才能执行删除

**市面上提供了集群方案**

1. redis shardding   而且jedis客户端就支持shardding操作  SharddingJedis ； 增加和减少节点的问题； pre shardding

3台虚拟机 redis 。但是我部署了9个节点 。每一台部署3个redis增加cpu的利用率

9台虚拟机单独拆分到9台服务器

2. codis基于redis2.8.13分支开发了一个codis-server

3. twemproxy  twitter提供的开源解决方案

redis的环境

1.密码设置-redis的配置文件中，配置：requirepass foobared

2.redis缓存的更新：

问：redis的缓存和数据库的数据如何保持原子性/一致性。

答：【先删除缓存，再更新数据库】。

问：先删除缓存，再更新数据库的话，如果存在两个并发操作：一个是更新数据库，另一个数查询缓存，更新操作在缓存删除以后，查询操作正好在查缓存，此时没有命中缓存，按照常规，缓存没有查到会直接去数据库查，而此时更新数据库的操作没有完全执行，缓存又load了老的数据，意味着后续一段时间缓存的数据都是脏数据。

答：两个不同的存储，要保证强一致性，要么使用二阶段提交，2pc或3pc这种强一致性的提交协议，或者使用算法去完成一致性，实际上，不可能在程序中为了强一致性让性能产生影响，所以一般会有解决方案是（不能完全保证，但是能将出现脏数据的概率降到最低）:【先去更新数据库，更新成功后，让缓存失效】。

还有解决方案是：【更新数据的时候，只更新缓存，不更新数据，然后同步异步调度去批量更新数据库】，好处是吗，数据完全只走缓存，不走数据库，性能很高，因为直接操作内存，但是这个过程，数据仍然不是强一致的。

3.缓存击穿：用到缓存，涉及到访问量很高时，时一定要做的

1）缓存穿透：恶意用不存在的大量key去访问，这样会直接访问数据库，数据库会down掉

解决方案2种：

A.对每个数据设置一个固定的值 key->"&&"

B.布隆过滤器：可能存在的过滤器，先内置到集合里，每次请求都会进行筛选

2）缓存击穿：在某一个高峰点，所有的key同时失效，意味着所有的请求都会进到数据库，这时也会形成瞬间压力，使得数据库崩溃

解决方案：互斥锁，当某个key的缓存失效后，不是立刻去load数据，而是使用redis提供的setnx命令设置锁，让后续的从操作变成排队等待机制，这样起到一个缓解的效果。

3）缓存失效

类似缓存击穿，key的失效时间一致导致的，或者expire过期