Redis主从同步

Redis Cluster实现高可用

可以阅读：

- 1.《Redis 集群教程》
- 2.《Redis集群模式工作原理》

何使用Redis Sentinel实现高可用

建议阅读《Redis 哨兵集群实现高可用》

> *哨兵介绍*

哨兵是Redis集群机构中非常重要的一个组件，主要有以下功能

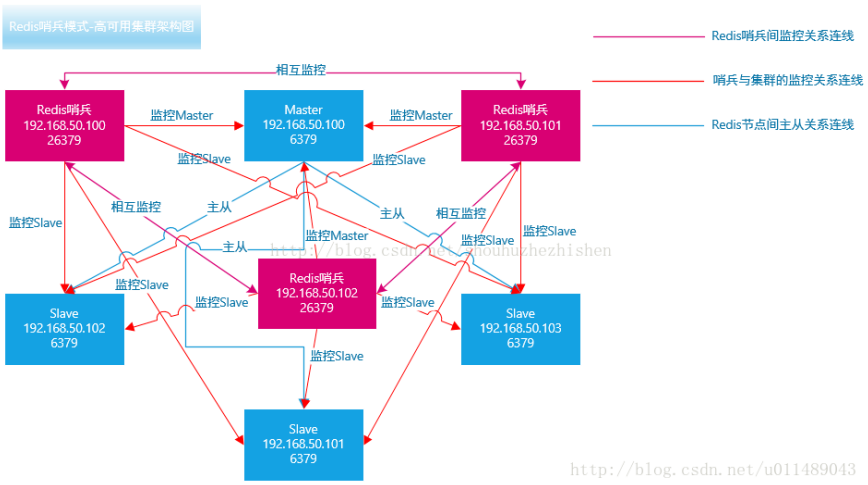
- 集群监控：负责监控Redis master和slave进行是否正常工作
- 消息通知：如果某个 redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
- 故障转移：如果 master node 挂掉了，会自动转移到 slave node 上。
- 配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。

哨兵用于实现 redis 集群的高可用，本身也是分布式的，作为一个哨兵集群去运行，互相协同工作。

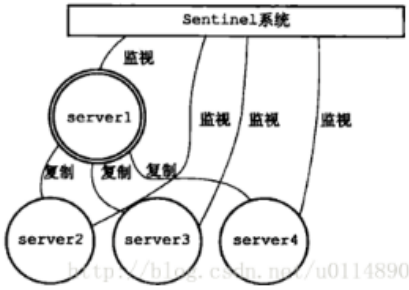
- 故障转移时，判断一个 master node 是否宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题。
- 即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了。

> *哨兵的核心知识*

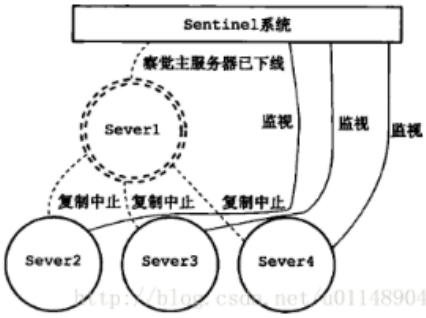
- 哨兵至少需要 3 个实例，来保证自己的健壮性。
- 哨兵 + redis 主从的部署架构，是不保证数据零丢失的，只能保证 redis 集群的高可用性。
- 对于哨兵 + redis 主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练。


Redis哨兵模式-高可用集群架构图
Redis哨兵间监控关系连线
哨兵与集群的监控关系连线
Redis节点间主从关系连线
相互监控
Redis哨兵
192.168.50.100
26379
监控Master
Master
192.168.50.100
6379
监控Master
Redis哨兵
192.168.50.101
26379
监控Slave
监控Slave
监控Slave
相互监控
主从
主从
相互监控
监控Slave
监控Slave
监控Master
监控Slave
Redis哨兵
192.168.50.102
26379
监控Slave
监控Slave
Slave
192.168.50.102
6379
Slave
192.168.50.103
6379
监控Slave
Slave
192.168.50.101
6379
http://blog.csdn.net/u011489043

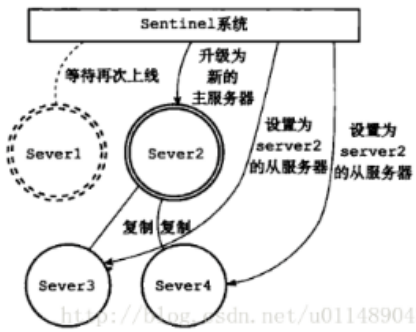
例如：



在Server1掉线后：



升级Server2为新的主服务器：



> *redis 哨兵主备切换的数据丢失问题*

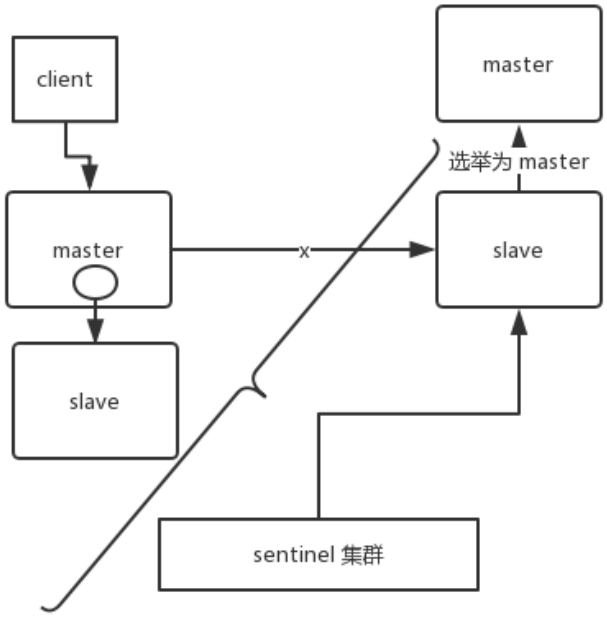
- 主备切换过程，可能导致数据丢失

  因为Master->slave的复制是异步的，所以可能有部分数据还没有复制到slave，master就宕机了，此时这部分数据就会出现丢失

- 脑裂导致数据丢失

  脑列也就是说，某个Master所在机器突然脱离了正常的网络，跟其他slave机器不能连接，但实际上master还运行着，此时哨兵可能就会认为master宕机了，然后开启选举，将其他的slave切换成master。这个时候，集群中就会出现两个master，也就是所谓的脑裂

此时虽然某个slave被切换成了master，但是可能client还没来得及切换到新的master，还继续向旧master写数据，因此旧master再次恢复的时候，会被作为一个slave挂到新的master上去，自己的数据会被清空，重新从新的master复制数据，而新的master并没有后来的clieny写入的数据，因此这部分数据就会发生丢失



> *数据丢失问题的解决方案*

进行如下配置：

```
min-slaves-to-write 1

min-slaves-max-lag 10
```

表，要求至少有 1 个 slave，数据复制和同步的延迟不能超过 10 秒。

如果说一旦所有的 slave，数据复制和同步的延迟都超过了 10 秒钟，那么这个时候，master 就不会再接收任何请求了。

- 减少异步复制数据的丢失

  有了 min-slaves-max-lag 这个配置，就可以确保说，一旦 slave 复制数据和 ack 延时太长，就认为可能 master 宕机后损失的数据太多了，那么就拒绝写请求，这样可以把 master 宕机时由于部分数据未同步到 slave 导致的数据丢失降低的可控范围内。

- 减少脑裂的数据丢失

如果一个 master 出现了脑裂，跟其他 slave 丢了连接，那么上面两个配置可以确保说，如果不能继续给指定数量的 slave 发送数据，而且 slave 超过 10 秒没有给自己 ack 消息，那么就直接拒绝客户端的写请求。因此在脑裂场景下，最多就丢失 10 秒的数据。

> *sdown 和 odown 转换机制*

- sdown 是主观宕机，就一个哨兵如果自己觉得一个 master 宕机了，那么就是主观宕机
- odown 是客观宕机，如果 quorum 数量的哨兵都觉得一个 master 宕机了，那么就是客观宕机

sdown 达成的条件很简单，如果一个哨兵 ping 一个 master，超过了 is-master-down-after-milliseconds 指定的毫秒数之后，就主观认为 master 宕机了；如果一个哨兵在指定时间内，收到了 quorum 数量的其它哨兵也认为那个 master 是 sdown 的，那么就认为是 odown 了。

> *哨兵集群的自动发现机制*

哨兵互相之间的发现，是通过 redis 的 pub/sub 系统实现的，每个哨兵都会往 **sentinel**:hello 这个 channel 里发送一个消息，这时候所有其他哨兵都可以消费到这个消息，并感知到其他的哨兵的存在。

每隔两秒钟，每个哨兵都会往自己监控的某个 master+slaves 对应的 **sentinel**:hello channel 里发送一个消息，内容是自己的 host、ip 和 runid 还有对这个 master 的监控配置。

每个哨兵也会去监听自己监控的每个 master+slaves 对应的 **sentinel**:hello channel，然后去感知到同样在监听这个 master+slaves 的其他哨兵的存在。

每个哨兵还会跟其他哨兵交换对 master 的监控配置，互相进行监控配置的同步。

> *slave->master 选举算法*

如果一个 master 被认为 odown 了，而且 majority 数量的哨兵都允许主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个 slave 来，会考虑 slave 的一些信息：

```
跟 master 断开连接的时长

slave 优先级

复制 offset

run id
```

如果一个 slave 跟 master 断开连接的时间已经超过了 down-after-milliseconds 的 10 倍，外加 master 宕机的时长，那么 slave 就被认为不适合选举为 master。

```
(down-after-milliseconds * 10) +
milliseconds_since_master_is_in_SDOWN_state
```

接下来会对 slave 进行排序：

```
按照 slave 优先级进行排序，slave priority 越低，优先级就越高。

如果 slave priority 相同，那么看 replica offset，哪个 slave 复制了越多的数据，offset 越靠后，优先级就越高。

如果上面两个条件都相同，那么选择一个 run id 比较小的那个 slave。
```

> *quorum 和 majority*

每次一个哨兵要做主备切换，首先需要 quorum 数量的哨兵认为 odown，然后选举出一个哨兵来做切换，这个哨兵还需要得到 majority 哨兵的授权，才能正式执行切换。

如果 quorum < majority，比如 5 个哨兵，majority 就是 3，quorum 设置为 2，那么就 3 个哨兵授权就可以执行切换。

但是如果 quorum >= majority，那么必须 quorum 数量的哨兵都授权，比如 5 个哨兵，quorum 是 5，那么必须 5 个哨兵都同意授权，才能执行切换。

> *configuration epoch*

哨兵会对一套 redis master+slaves 进行监控，有相应的监控的配置。

执行切换的那个哨兵，会从要切换到的新 master（salve->master）那里得到一个 configuration epoch，这就是一个 version 号，每次切换的 version 号都必须是唯一的。

如果第一个选举出的哨兵切换失败了，那么其他哨兵，会等待 failover-timeout 时间，然后接替继续执行切换，此时会重新获取一个新的 configuration epoch，作为新的 version 号。

> *configuration 传播*

哨兵完成切换之后，会在自己本地更新生成最新的 master 配置，然后同步给其他的哨兵，就是通过之前说的 pub/sub 消息机制。

这里之前的 version 号就很重要了，因为各种消息都是通过一个 channel 去发布和监听的，所以一个哨兵完成一次新的切换之后，新的 master 配置是跟着新的 version 号的。其他的哨兵都是根据版本号的大小来更新自己的 master 配置的。

说说Redis哈希槽的概念

Redis Cluster没有使用一致性Hash，而是引入了哈希槽的概念。

Redis集群有16384（2的14次方）个哈希槽，每个key通过CRC16校验后对16384取模来决定存放到那个槽，集群的每个节点负责一部分hash槽，使用哈希槽的好处就在于可以方便的添加或移除节点，当需要增加节点时，只需要把其他节点的某些哈希槽挪到新节点就可以了，当需要移除节点时，只需要把移除节点上的哈希槽挪到其他节点就行了，在这一点上，我们以后新增或移除节点的时候不用先停掉所有的 redis 服务。

Redis最大是16384个哈希槽，所以考虑Redis集群中的每个几点都能分配到一个哈希槽，所以最多支持16384个Redis节点，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点

当你往Redis Cluster中加入一个Key时，会根据crc16(key) mod 16384计算这个key应该分布到哪个hash slot中，一个hash slot中会有很多key和value。你可以理解成表的分区，使用单节点时的redis时只有一个表，所有的key都放在这个表里；改用Redis Cluster以后会自动为你生成16384个分区表，你insert数据时会根据上面的简单算法来决定你的key应该存在哪个分区，每个分区里有很多key

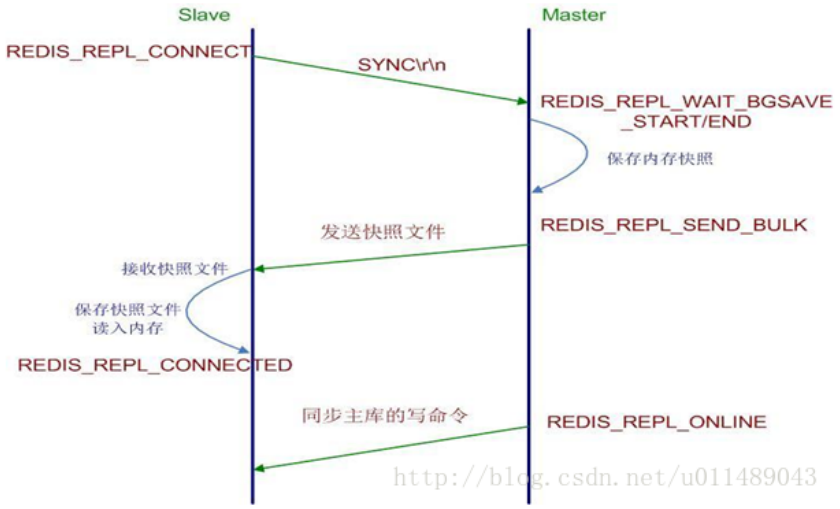
Redis Cluster主从复制模型是怎样的？

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型，每个几点都会有N-1个复制节点。

所以Redis Cluster可以说是Redis Sentinel带分片的加强版

- Redis Sentinel着眼于高可用，在master宕机时自动将slave提升为master继续提供服务。
- Redis Cluster着眼于扩展性，在单个Redis内存不足时，使用Cluster进行分片存储

主从复制过程：



- Slave端在配置文件中添加了slave of指令，于是Slave启动时读取配置文件，初始状态为REDIS_REPL_CONNECT。
- Slave端在定时任务serverCron(Redis内部的定时器触发事件)中连接Master，发送sync命令，然后阻塞等待master发送回其内存快照文件**(最新版的Redis已经不需要让Slave阻塞)**。
- Master端收到sync命令简单判断是否有正在进行的内存快照子进程，没有则立即开始内存快照，有则等待其结束，当快照完成后会将该文件发送给Slave端。
- Slave端接收Master发来的内存快照文件，保存到本地，待接收完成后，清空内存表，重新读取Master发来的内存快照文件，重建整个内存表数据结构，并将最终状态置位为 REDIS_REPL_CONNECTED状态，Slave状态机流转完成。
- Master端在发送快照文件过程中，接收的任何会改变数据集的命令都会暂时先保存在Slave网络连接的发送缓存队列里（list数据结构），待快照完成后，依次发给Slave，之后收到的命令相同处理，并将状态置位为 REDIS_REPL_ONLINE。

Redis Cluster方案什么情况下会导致整个集群不可用？

有A、B、C三个节点的集群，在没有复制模型的情况下，如果节点B宕机，那么整个集群就会以为缺少5501-11000这个范围的槽而不可用。

Reis Cluster会有写操作丢失吗？为什么？

参考文档《Redis数据"丢失"讨论及规避和解决的几点总结》

Redis并不能保证数据的强一致性。写数据丢失主要有以下2个场景

- 1.程序bug或人为误操作。
- 2.因客户端缓冲区内存使用过大，导致大量键被LRU淘汰。

因客户端缓冲区的内存大小很难限制,它们消耗的内存数会计算在used_memory内；如果使用不当，导致缓冲区内存使用过大，达到maxmemory限制；（缓存场景）会导致大量的键被淘汰，最坏会把所有键清理，缓冲无键可淘汰，写入失败。相当于整个缓冲失效，对业务影响较大。

- 3.主库故障后自动重启，可能导致数据丢失。

  问题发生的现象：时间点T1,主库故障关闭了，因设置有自动重启的守护程序，时间点T2主库被重新拉起，因(T2-T1)时间间隔过小，未达到Redis集群或哨兵的主从切换判断时长；这样从库发现主库runid变了或断开过，会全量同步主库rdb清理，并清理自己的数据。而为保障性能,Redis主库往往不做数据持久化设置，那么时间点T2启动的主库，很有可能是个空实例（或很久前的rdb文件）。这种问题发生时间间隔，一般小于1分钟，可能监控告警无法感知到。

- 4.网络分区的问题，可能导致短时间的写入数据丢失。

  这种问题出现丢失数据都很少，网络分区时，Redis集群或哨兵在判断故障切换的时间窗口，这段时间写入到原主库的数据，5秒~15秒的写入量。

- 5.主从数据出现不一致，发生故障切换，从库提升为主后，导致数据丢失的情况

Redis分区是什么

分区是分割数据到多个Redis实例的处理过程，因此每个实例只保存Key的一个子集

> ***分区的优势***

- 通过利用多台计算机内存的和值，允许我们构造更大的数据库。
- 通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

> ***分区的不足***

redis的一些特性在分区方面表现的不是很好：

- 涉及多个key的操作通常是不被支持的。举例来说，当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。
- 涉及多个key的redis事务不能使用。
- 当使用分区时，数据处理较为复杂，比如你需要处理多个rdb/aof文件，并且从多个实例和主机备份持久化文件。
- 增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。

> ***分区类型***

Redis 有两种类型分区。 假设有4个Redis实例 R0，R1，R2，R3，和类似 user:1，user:2这样的表示用户的多个key，对既定的key有多种不同方式来选择这个key存放在哪个实例中。也就是说，有不同的系统来映射某个key到某个Redis服务。

- 范围分区

  最简单的分区方式是按范围分区，就是映射一定范围的对象到特定的Redis实例。

> 比如，ID从0到10000的用户会保存到实例R0，ID从10001到 20000的用户会保存到R1，以此类推。

> 这种方式是可行的，并且在实际中使用，不足就是要有一个区间范围到实例的映射表。这个表要被管理，同时还需要各 种对象的映射表，通常对Redis来说并非是好的方法。

- 哈希分区

  另外一种分区方法是hash分区。这对任何key都适用，也无需是object_name:这种形式，像下面描述的一样简单：

  - 用一个hash函数将key转换为一个数字，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。
  - 对这个整数取模，将其转化为0-3之间的数字，就可以将这个整数映射到4个Redis实例中的一个了。93024922 % 4 = 2，就是说key foobar应该被存到R2实例中。注意：取模操作是取除的余数，通常在多种编程语言中用%操作符实现。

Redis主从复制、Redis分区、Redis集群三个之间的关系

- Redis分区是一种模式，将数据分区到不同的Redis节点上，而Redis集群的Redis Cluster 、 Twempropy、 Codis、客户端分片这四种方案是Redis分区的具体实现。
- Redis每个分区如果想要实现高可用，需要使用到Redis主从复制

Redis分区方案有哪些

Redis分区方案主要有两种类型

- 客户端分区：就是在客户端就已经决定数据会被存储到那个Redis节点或者从哪个Redis节点读取数据
  - 案例：Redis Cluster和客户端分区
- 代理分区：客户端将请求发送给代理，然后代理决定去哪个节点写数据或者读数据。代理根据分区规则决定请求哪些Redis实例，然后根据Redis的相应结果返回给客户端
  - 案例：Twemproxy和Codis

分布式Redis是前期做还是后期规模上来之后再做？

一开始就多设置几个Redis实例，例如32或者64个实例，对大多数用户来说这操作起来可能比较麻烦，但是从长久来看牺牲这点还是值得的，这样的话，当你的数据不断增长，需要更多Redis服务器时，你需要做的就是仅仅将Redis实例从一台服务迁移到另外一台服务器而已（不需要考虑重新分区问题）。一旦你添加了另外一台服务器，你需要将你一半的Redis实例从一台机器迁移到第二台机器。

Redis有哪些重要的健康指标

1. 存活数

   所有指标中最重要的就是检查Redis是否存活，可以使用ping命令判断是否存活

2. 连接数

   连接的客户端数量，可通过命令info查看（connected_clients）得到，这个值跟使用redis的服务的连接池配置关系比较大，所以在监控这个字段的值时需要注意。另外这个值也不能太大，建议不要超过5000，如果太大可能是redis处理太慢，那么需要排除问题找出原因。

另外还有一个拒绝连接数（`rejected_connections`）也需要关注，这个值理想状态是0。如果大于0，说明创建的连接数超过了`maxclients`，需要排查原因。是`redis`连接池配置不合理还是连接这个`redis`实例的服务过多等。

3. 阻塞客户端数量

   blocked_clients，一般是执行了list数据类型的BLPOP或者BRPOP命令引起的，可通过命令src/redis-cli info Clients | grep blocked_clients得到，很明显，这个值最好应该为0。

4. 使用内存峰值

   监控redis使用内存的峰值，我们都知道Redis可以通过命令config set maxmemory 10737418240设置允许使用的最大内存（强烈建议不要超过20G），为了防止发生swap导致Redis性能骤降，甚至由于使用内存超标导致被系统kill，建议used_memory_peak的值与maxmemory的值有个安全区间，例如1G，那么used_memory_peak的值不能超过9663676416（9G）。另外，我们还可以监控maxmemory不能少于多少G，比如5G。因为我们以前生产环境出过这样的问题，运维不小心把10G配置成了1G，从而导致服务器有足够内存却不能使用的悲剧。

5. 内存碎片率

   mem_fragmentation_ratio=used_memory_rss/used_memory，这也是一个非常需要关心的指标。如果是redis4.0之前的版本，这个问题除了重启也没什么很好的优化办法。而redis4.0有一个主要特性就是优化内存碎片率问题（Memory de-fragmentation）。在redis.conf配置文件中有介绍即ACTIVE DEFRAGMENTATION：碎片整理允许Redis压缩内存空间，从而回收内存。这个特性默认是关闭的，可以通过命令CONFIG SET activedefrag yes热启动这个特性。

6. 缓存命中率

keyspace_misses/keyspace_hits这两个指标用来统计缓存的命令率，keyspace_misses指未命中次数，keyspace_hits表示命中次数。keyspace_hits/(keyspace_hits+keyspace_misses)就是缓存命中率。视情况而定，建议0.9以上，即缓存命中率要超过90%。如果缓存命中率过低，那么要排查对缓存的用法是否有问题！

7. OPS

   instantaneous_ops_per_sec这个指标表示缓存的OPS，如果业务比较平稳，那么这个值也不会波动很大，不过国内的业务比较特性，如果不是全球化的产品，夜间是基本上没有什么访问量的，所以这个字段的监控要结合自己的具体业务，不同时间段波动范围可能有所不同。

8. 持久化

   rdb_last_bgsave_status/aof_last_bgrewrite_status，即最近一次或者说最后一次RDB/AOF持久化是否有问题，这两个值都应该是"ok"。

另外，由于redis持久化时会fork子进程，且fork是一个完全阻塞的过程，所以可以监控fork耗时即latest_fork_usec，单位是微妙，如果这个值比较大会影响业务，甚至出现timeout。

9. 失效key

   如果把Redis当缓存使用，那么建议所有的key都设置了expire属性，通过命令src/redis-cli info Keyspace得到每个db中key的数量和设置了expire属性的key的属性

10. 慢日志

    通过命令slowlog get得到Redis执行的slowlog集合，理想情况下，slowlog集合应该为空，即没有任何慢日志，不过，有时候由于网络波动等原因造成set key value这种命令执行也需要几毫秒，在监控的时候我们需要注意，而不能看到slowlog就想着去优化，简单的set/get可能也会出现在slowlog中。

Redi客户端连接数一直降不下来的有关问题解决

参考文章《"Redis客户端连接数一直降不下来"的有关问题解决》

如提高Redis命中率

> *概念*

命中：可以直接通过缓存获取需要的数据

不命中：无法直接通过缓存获取想要的数据，需要再次查询数据库或者执行其他操作。

> *如何监控缓存的命中率*

Redis提供了info命令，能够获取监控服务器的状态信息

```
keyspace_hits:14414110

keyspace_misses:3228654

used_memory:433264648

expired_keys:1333536

evicted_keys:1547380
```

通过计算hits和miss，我们可以得到缓存的命中率：14414110 / (14414110 + 3228654) = 81% ，一个缓存失效机制，和过期时间设计良好的系统，命中率可以做到95%以上

> *影响缓存命中率的几个因素*

1. 业务场景和业务需求
2. 缓存的设计（粒度和策略）
3. 缓存容量和基础设施
4. 其他因素

> *提高缓存命中率的方法*

应用尽可能的通过缓存直接获取数据，并避免缓存失效。这也是比较考验架构师能力的，需要在业务需求，缓存粒度，缓存策略，技术选型等各个方面去通盘考虑并做权衡。尽可能的聚焦在高频访问且时效性要求不高的热点业务上（如字典数据、session、token），通过缓存预加载（预热）、增加存储容量、调整缓存粒度、更新缓存等手段来提高命中率。

怎么优化Redis内存占用

推荐阅读 《Redis的内存优化》

1. RedisObject对象
2. 缩减键值对象
3. 共享对象池
4. 字符串优化
5. 编码优化
6. 控制key的数量

一个Redis实例最多能存放最多能存放多少的keys？List、Set、Sorted set他们最多能存放多少元素

一个Redis实例，最多能存放2的32次方个key，任何List、Set、Sorted set都可以存放2的32次方个元素

假如Redis里面有1亿个key，其中10W个key是以某个固定的已知前缀开头的，如何把他们全部找出来？

因为Redis是单线程的，使用Keys命令获取所有key列表的时候，会造成线程阻塞，必须等执行完之后才会继续执行，这个时候如果线上查询会出现停顿的问题。

可以使用scan命令来获取所有key列表：

1. scan命令的时间复杂度也是O（N），但他是分次进行的，不会造成线程阻塞
2. scan命令提供了limit参数，可以控制每次返回的结果的最大条数
3. scan命令是增量循环，每次调用智慧返回一小部分的元素
4. scan命令返回的是一个游标，从0开始遍历到0结束遍历
5. scan的话就是遍历所有的keys
6. 其他的SCAN命令的话是SCAN选中的集合
7. scan命令返回的结果可能存在重复，需要客户端去重