**Redis**

# 缓存问题

## 缓存雪崩

指的是****大量缓存集中在一段时间内失效****，发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。

### 解决办法

这个没有完美解决办法，但可以分析用户行为，尽量让失效时间点均匀分布，设置不同的过期时间。

用加分布式锁或者分布式队列的方式保证缓存的单线程（进程）写 （eg. redis的 SETNX），从而避免失效时大量的并发请求落到底层存储系统上。在加锁方法内先从缓存中再获取一次(防止另外的线程优先获取锁已经写入了缓存)，没有再查DB写入缓存。 （当然也可以： 在没有获取锁(tryLock)的线程中一直轮询缓存，至超限时）

缓存击穿

## 缓存击穿

指的是热点key在某个特殊的场景时间内恰好失效了，恰好有大量并发请求过来了，造成DB压力。

### 解决办法

与缓存雪崩的解决方法类似： 用加锁或者队列的方式保证缓存的单线程（进程）写，在加锁方法内先从缓存中再获取一次，没有再查DB写入缓存。

还有一种比较好用的（针对缓存雪崩与缓存击穿）：

物理上的缓存是不设置超时时间的（或者超时时间比较长）， 但是在缓存的对象上增加一个属性来标识超时时间（此时间相对小）。 当获取到数据后，校验数据内部的标记时间，判定是否快超时了，如果是，异步发起一个线程（控制好并发）去主动更新该缓存。

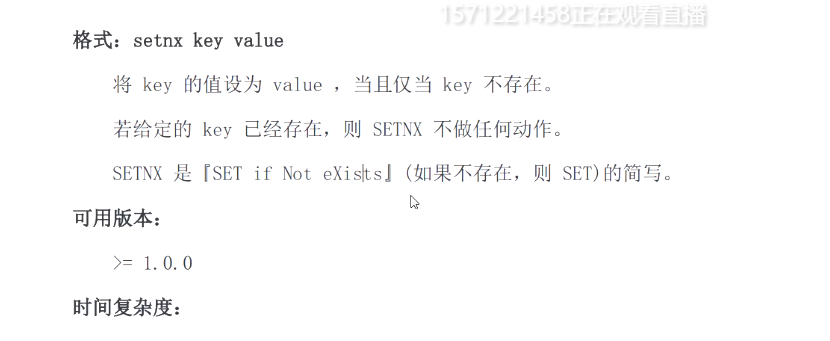
这种方式会导致一定时间内，有些请求获取缓存会拿到过期的值，看业务是否能接受而定。

## 缓存穿透

缓存穿透，是指查询一个数据库一定不存在的数据。正常的使用缓存流程大致是，数据查询先进行缓存查询，如果key不存在或者key已经过期，再对数据库进行查询，并把查询到的对象，放进缓存。如果数据库查询对象为空，则不放进缓存。

解決方法:布隆过滤器

# 分布式锁



* 放了防止redis宕机的时候，key一直存在，所以要设置超时时间。  
  设置了超时时间，如果过了超时时间，业务还没有执行完，那么key自动失效，就会有其他的线程来争夺锁，等到该线程执行完的时候，会释放锁，但是是释放的是其他线程加的锁，所以还是出现超卖的现象。
* 可以每隔一段时间(超时时间的三分之一，例如redission)检查锁是否失效，没有失效，就更新超时时间，为锁续命

## 分布式锁的问题

1. 由单机锁演进到分布式锁
2. 分布式锁可以由redis中setnx来实现
3. 使用这条命令，如果服务器宕机了，那么设置的值将一直存在

解决方法:可以设置超时时间

如果设置的超时时间过短，A线程的业务代码还没有执行完，锁就被redis删除了，此时B线程获取了锁，开始执行业务代码，刚好在这时B线程的锁就被A线程释放了，其他线程又可以获取锁继续执行，那么这还是会导致超卖问题

解决方法:可以写个定时任务，定时（最好是超时时间的三分之一）检查锁是否持有，如果持有，就延长锁的时间，可以使用redision

# 数据结构

Key设计的时候一般遵循：功能模块 表名 业务模块

## 存储上限

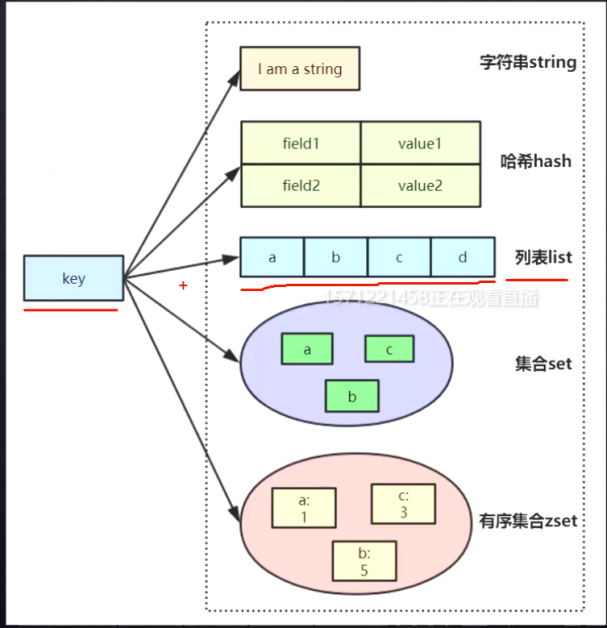
Strings类型：一个String类型的value最大可以存储512M

Lists类型：list的元素个数最多为2^32-1个，也就是4294967295个。

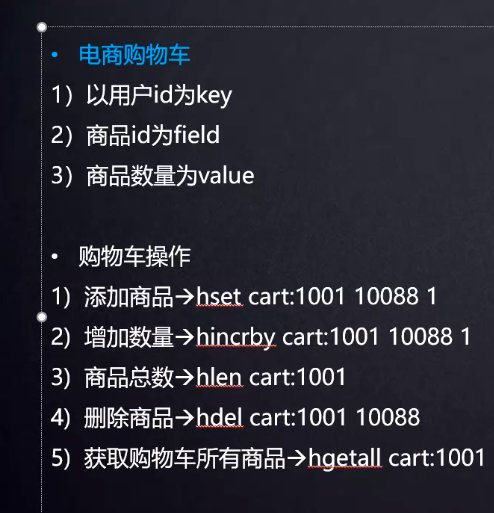
Sets类型：元素个数最多为2^32-1个，也就是4294967295个。

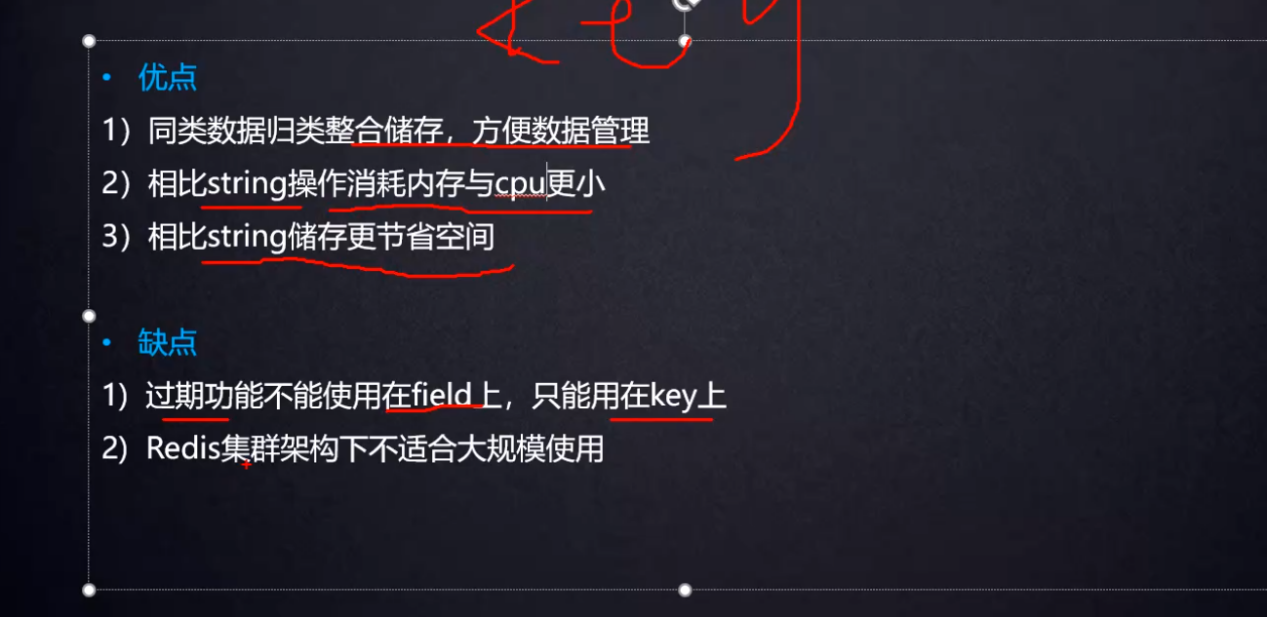
Hashes类型：键值对个数最多为2^32-1个，也就是4294967295个。

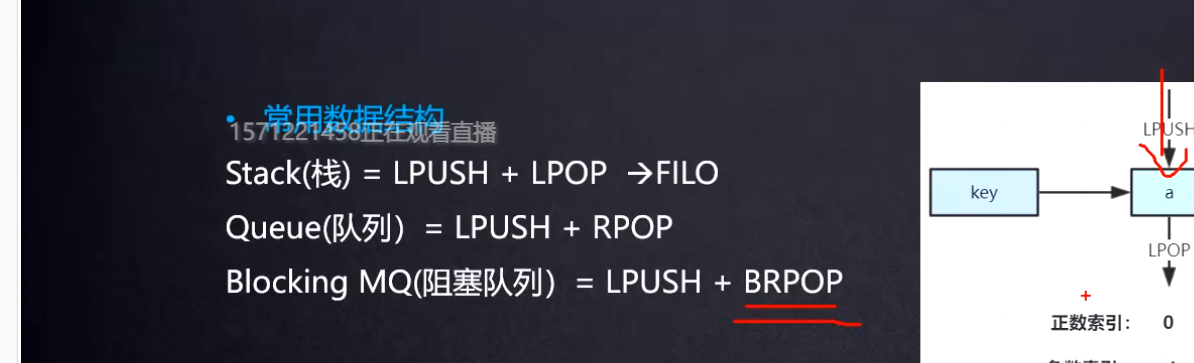
Sorted sets类型：跟Sets类型相似。

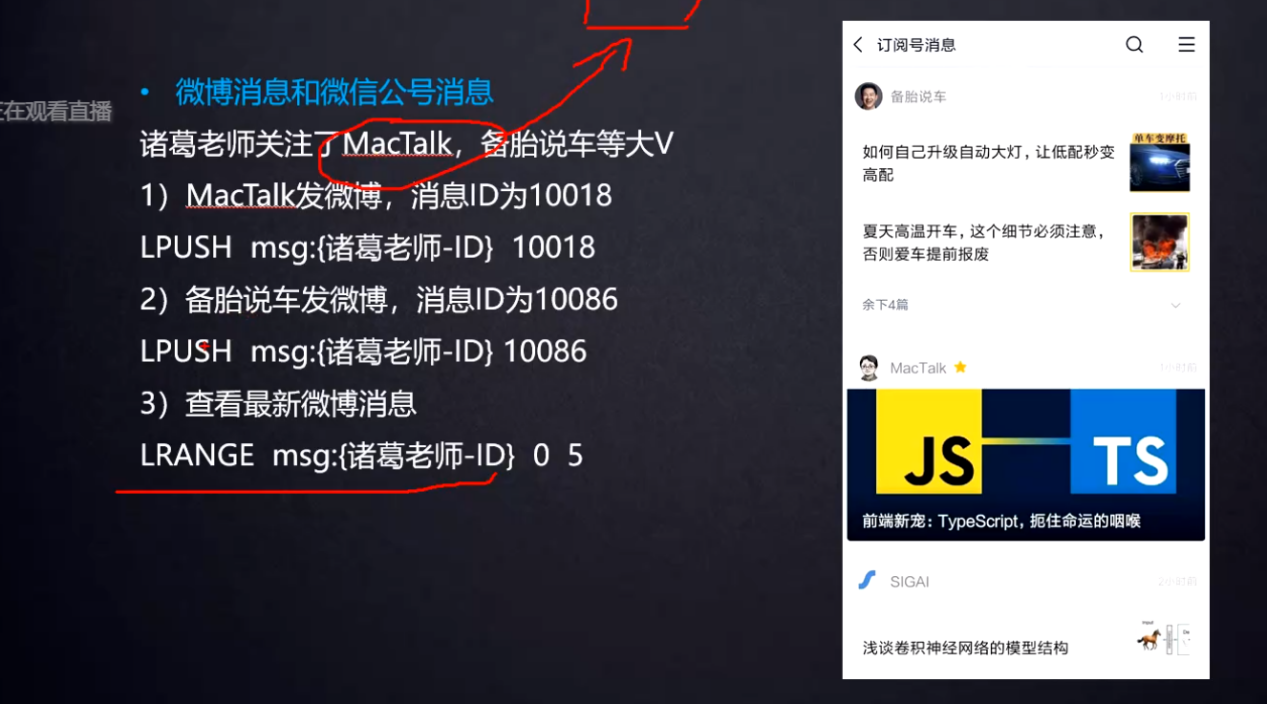


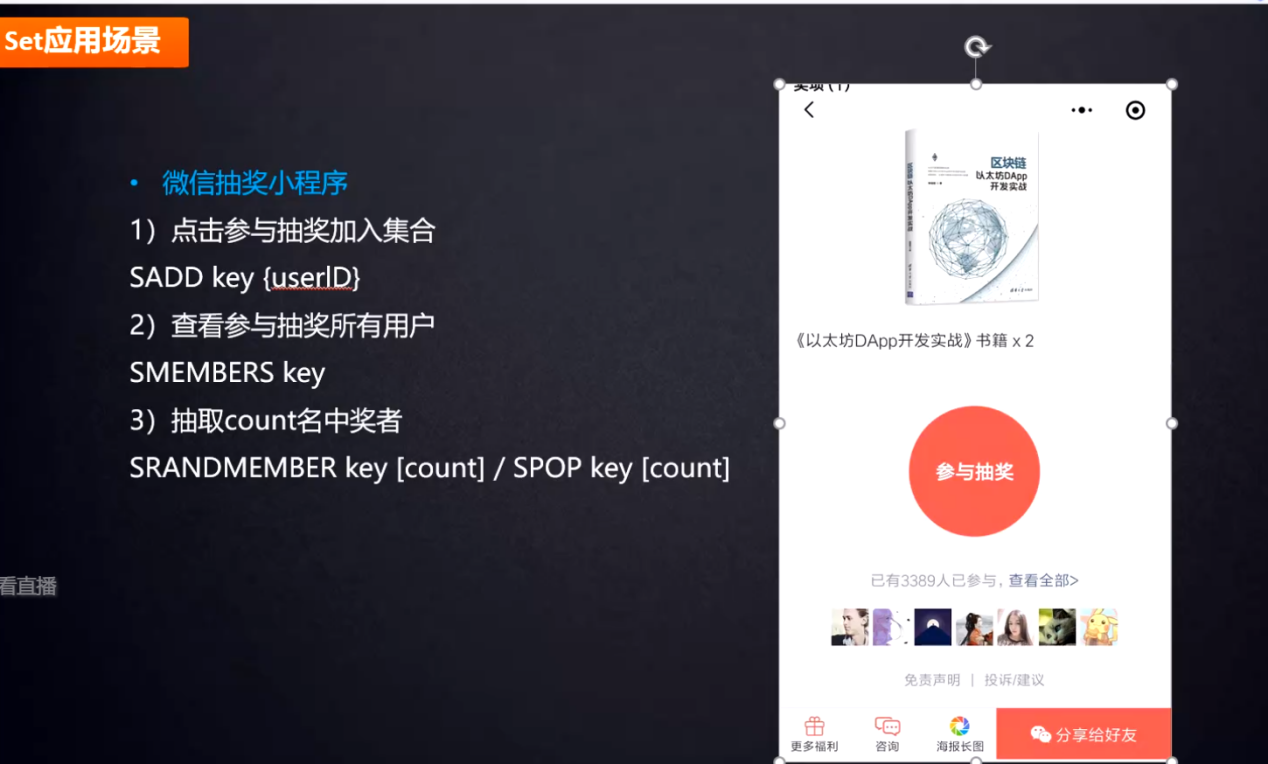
## Hash

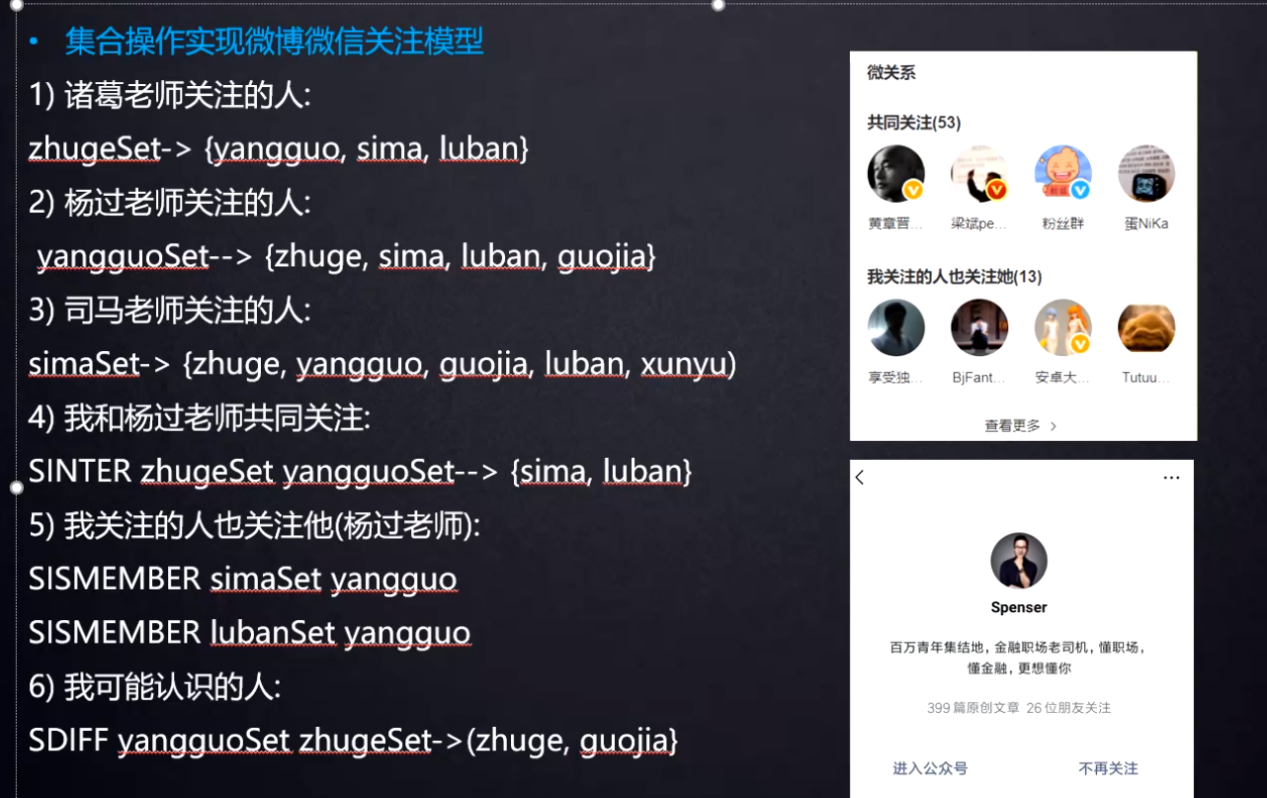


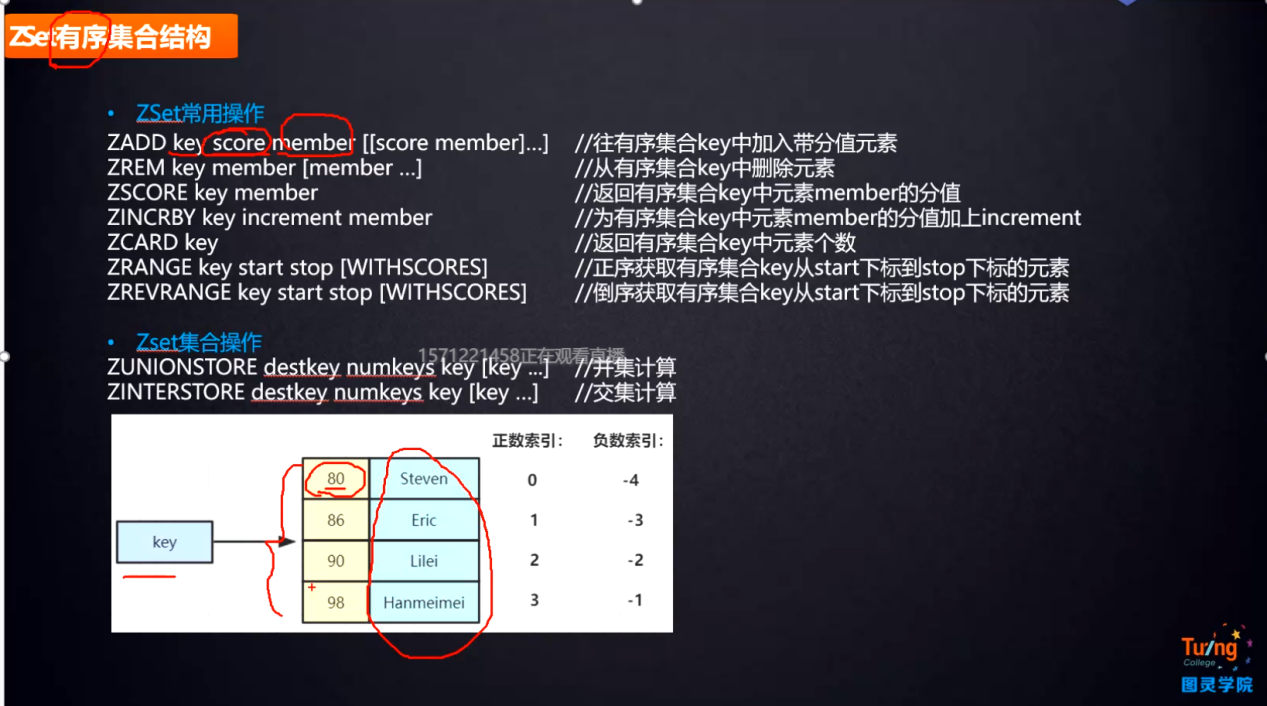






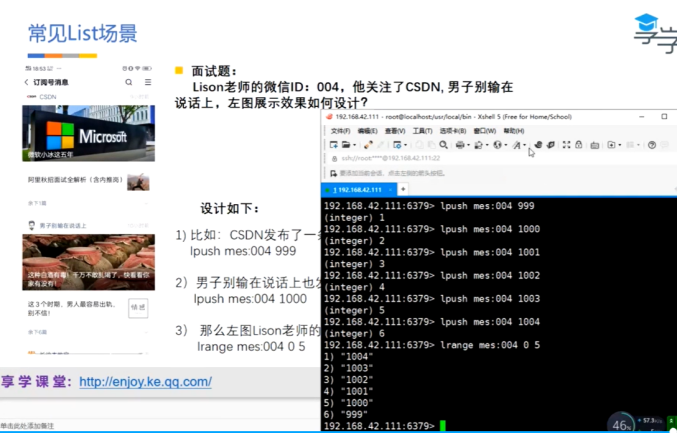






## LIst

使用场景：消息推送



## Set

抽奖:**srandmember**

## Zset

可以做热榜

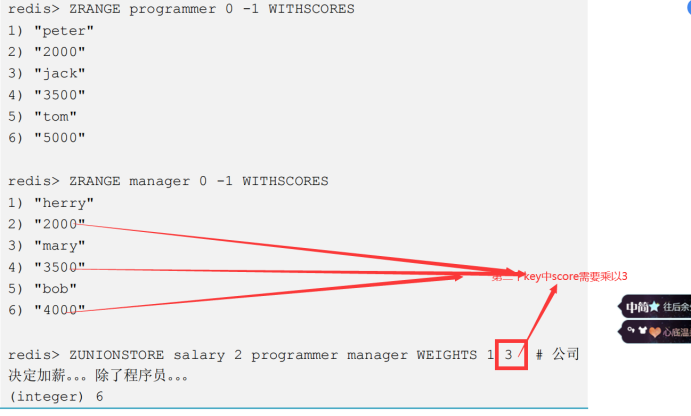
### ZUNIONSTORE

可以用来将两个key中相同的元素的score相加

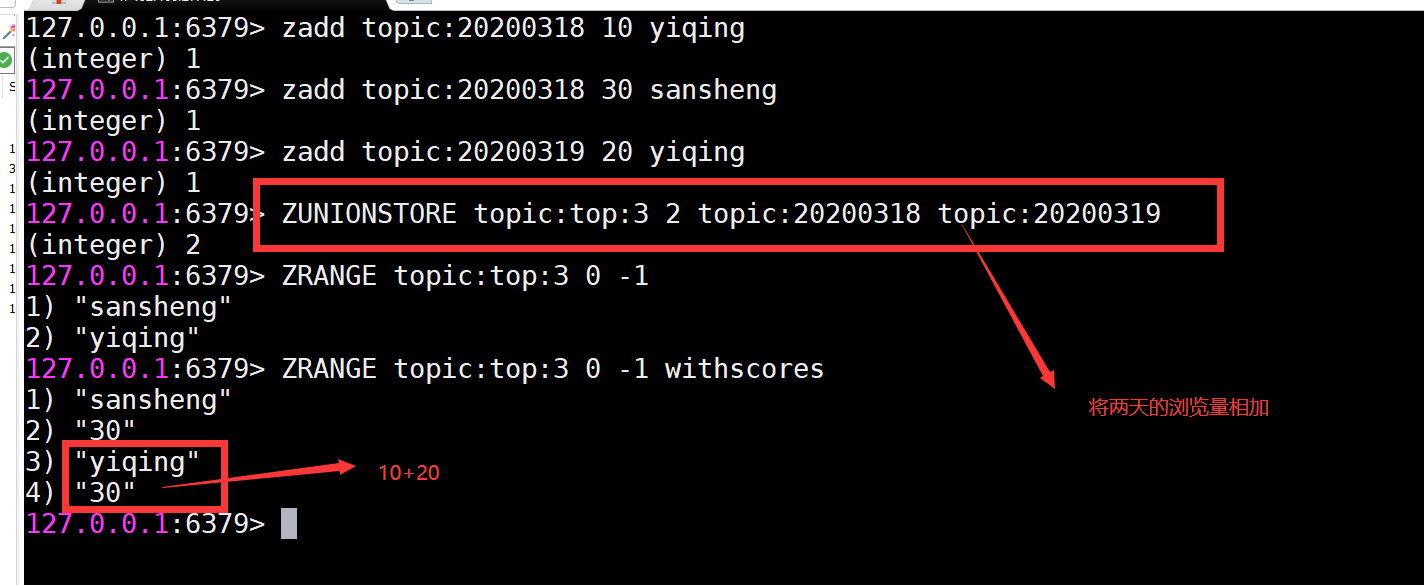
比如keya：1 one

Keyb: 2 one

用这个命令之后，就变为了3=1+2



可以用来将几天的浏览量相加

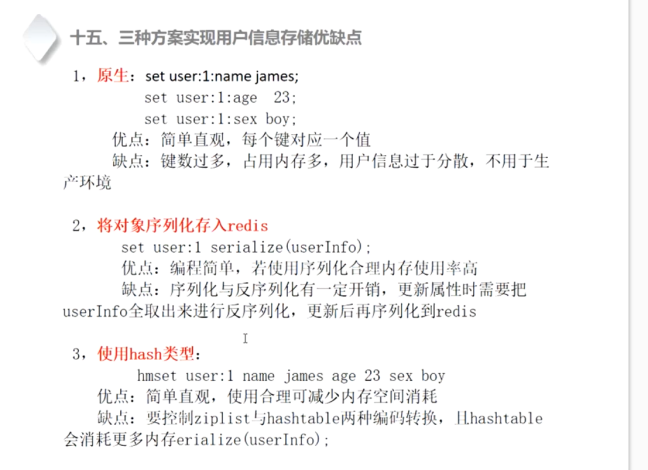


# 特性

速度快:数据存储到内存中，多路复用:单线程或单进程同时监测若干个文件描述符是否可以执行IO操作的能力。



# 数据存储的优缺点



# Redis持久化

## rdb

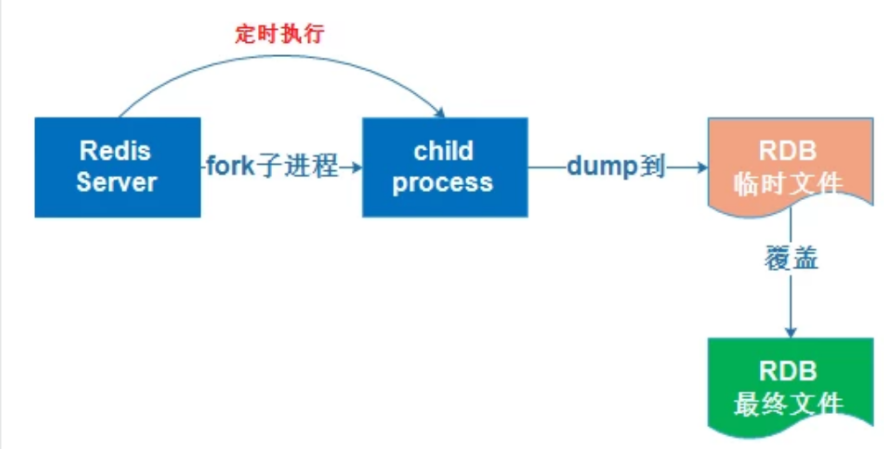
## AOF

## redis的持久化方式RDB和AOF的区别

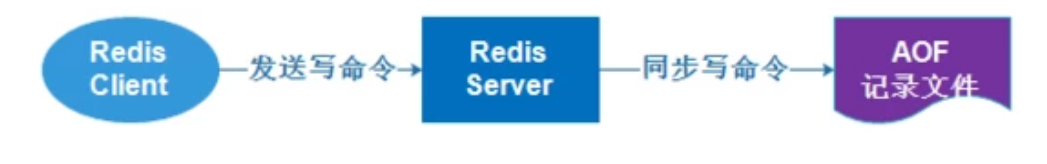
最近在项目中使用到Redis做缓存，方便多个业务进程之间共享数据。由于Redis的数据都存放在内存中，如果没有配置持久化，redis重启后数据就全丢失了，于是需要开启redis的持久化功能，将数据保存到磁盘上，当redis重启后，可以从磁盘中恢复数据。redis提供两种方式进行持久化，一种是RDB持久化（原理是将Reids在内存中的数据库记录定时dump到磁盘上的RDB持久化），另外一种是AOF持久化（原理是将Reids的操作日志以追加的方式写入文件）。那么这两种持久化方式有什么区别呢，改如何选择呢？网上看了大多数都是介绍这两种方式怎么配置，怎么使用，就是没有介绍二者的区别，在什么应用场景下使用。

### 二者的区别

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。



AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。



### 二者优缺点

#### RDB存在哪些优势呢？

1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

#### RDB又存在哪些劣势呢？

1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

#### AOF的优势有哪些呢？

1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。

2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。

4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

#### AOF的劣势有哪些呢？

1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

二者选择的标准，就是看系统是愿意牺牲一些性能，换取更高的缓存一致性（aof），还是愿意写操作频繁的时候，不启用备份来换取更高的性能，待手动运行save的时候，再做备份（rdb）。rdb这个就更有些 eventually consistent的意思了。不过生产环境其实更多都是二者结合使用的。

### 如何抉择

三、RDB和AOF到底该如何选择  
1、不要仅仅使用RDB，因为那样会导致你丢失很多数据；

2、也不要仅仅使用AOF，因为那样有两个问题：

你通过AOF做冷备，没有RDB做冷备，恢复速度更快;  
第二，RDB每次简单粗暴生成数据快照，更加健壮，可以避免AOF这种复杂的备份和恢复机制的bug；  
3、综合使用AOF和RDB两种持久化机制，用AOF来保证数据不丢失，作为数据恢复的第一选择;

用RDB来做不同程度的冷备，在AOF文件都丢失或损坏不可用的时候，还可以使用RDB来进行快速的数据恢复；

# 数据淘汰策略

# Lua

## 安装教程

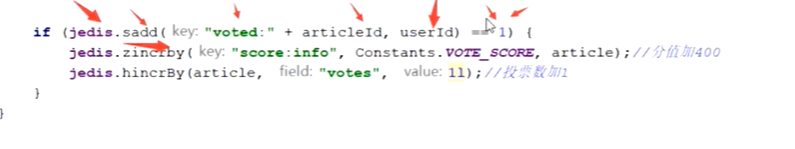
1. 到http://www.lua.org/ftp/下载包
2. 解压
3. yum install libtermcap-devel ncurses-devel libevent-devel readline-devel
4. Make linux test
5. Make install

# 应用

## 投票

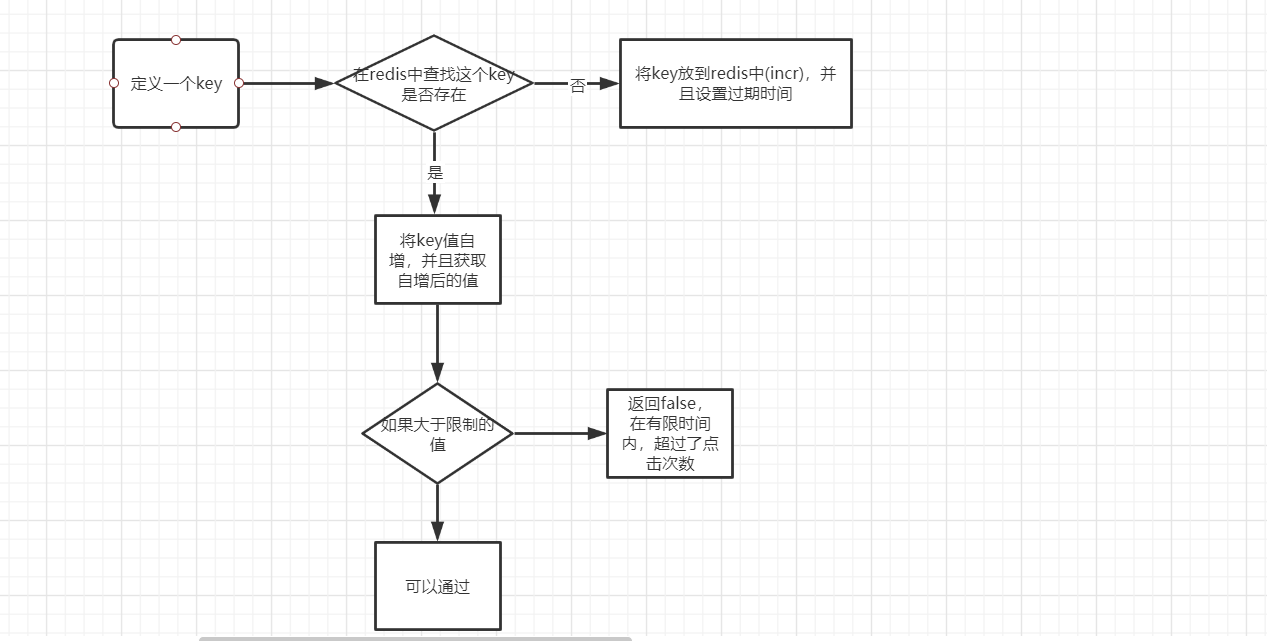
给文章投票

1. 创建文章，将文章保存到redis中
2. 给文章投票，防止一个人头多次，利用set类型来保存
3. 投一票要加分，分数需要排序，用到了zset
4. 还要记录文章的创建时间，按照时间排序，利用zset来保存



# 限流

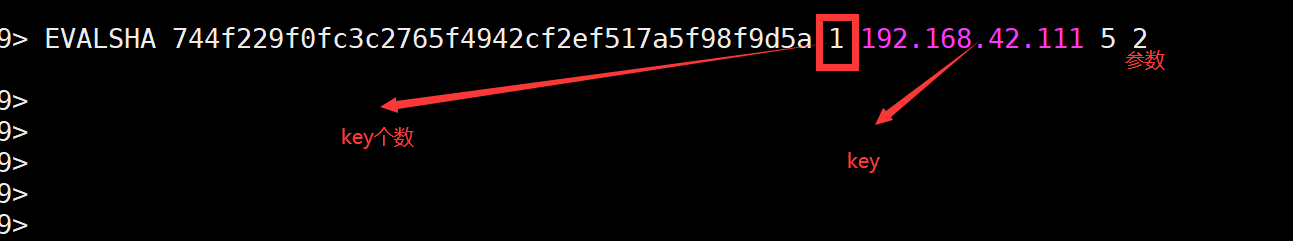
## 流程图



## 限流脚本

|  |
| --- |
| local key = KEYS[1] local limit = tonumber(ARGV[1]) local expire\_time = ARGV[2]  local is\_exists = redis.call("EXISTS", key)  if is\_exists == 1 then   if redis.call("INCR", key) > limit then  return 0  else  return 1  end  else  redis.call("SET", key, 1);  redis.call("EXPIRE", key, expire\_time)  return 1 end |

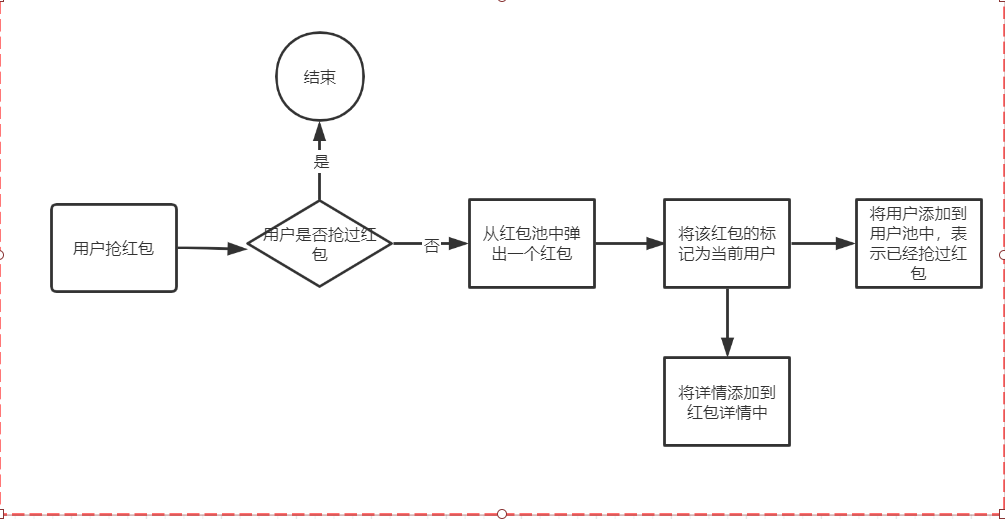
执行:EVALSHA 744f229f0fc3c2765f4942cf2ef517a5f98f9d5a 1 192.168.42.111 5 2



# 抢红包

Csdn：<https://blog.csdn.net/qq_31706095/article/details/105005493>

## 流程图



Key:hongbaoPool,hongbaoDetails,userSetForGrabbed(抢到红包的人)

参数:用户id

## Lua脚本

|  |
| --- |
| local userId = ARGV[1]  if redis.call("SISMEMBER", KEYS[3], userId) ==1 then  return nil  else  local hongbao = redis.call("RPOP", KEYS[1]);  if hongbao then  local x = cjson.decode(hongbao);  x['userId'] = userId;  local re = cjson.encode(x);  redis.call("SADD", KEYS[3], userId)  redis.call("LPUSH", KEYS[2], re)  return re  else  return nil  end  end |

## 创建红包

|  |
| --- |
| Map<String, String> map = new HashMap<>();  for (int i = 0; i < 10; i++) {  map.put("id", "rid" + i);  map.put("money", i +"");  jedisClient.lpush("hongbaoPoolKey", JSONUtils.toJSONString(map));  } |

## 抢红包

|  |
| --- |
| public class Qianghongbao {    private static JedisPool jedisPool = new JedisPool("192.168.27.128", 6379);    private static AtomicInteger atomicInteger = new AtomicInteger(0);    public static void main(String[] args) {  String script = "local userId = ARGV[1]\n" +  "if redis.call(\"SISMEMBER\", KEYS[3], userId) == 1 then\n" +  " return nil\n" +  "else\n" +  " local hongbao = redis.call(\"RPOP\", KEYS[1]);\n" +  " if hongbao then\n" +  " local x = cjson.decode(hongbao);\n" +  " x['userId'] = userId;\n" +  " local re = cjson.encode(x);\n" +  " redis.call(\"SADD\", KEYS[3], userId)\n" +  " redis.call(\"LPUSH\", KEYS[2], re)\n" +  " return re\n" +  " else\n" +  " return nil\n" +  " end\n" +  "end";  int n = 10;  // 实现一起开始抢  CyclicBarrier cyclicBarrier = new CyclicBarrier(n);  //抢完之后，统计多少人抢到了红包  CyclicBarrier cyclicBarrier2 = new CyclicBarrier(n, new Runnable() {  @Override  public void run() {  System.out.println("共" + atomicInteger.get() + "抢到了红包");  }  });  /\* System.out.println( resource.eval(script, 3, "hongbaoPoolKey", "hongbaoDetails",  "userSetForGrabbed", "1234541"));\*/  for (int i = 0; i < n; i++) {  if (i == 19) {  System.out.println();  }  new Thread(new Runnable() {  @Override  public void run() {  Jedis resource = null;  try {  cyclicBarrier.await();  // 每个线程需要获取一个连接  resource = jedisPool.getResource();  String userId = UUID.randomUUID().toString();    Object eval = resource.eval(script, 3, "hongbaoPoolKey", "hongbaoDetails",  "userSetForGrabbed", userId);  if (eval != null) {  atomicInteger.addAndGet(1);  System.out.println(userId + "抢到了红包");  } else {  System.out.println(userId + "没有抢到红包");  }  } catch (InterruptedException e) {  e.printStackTrace();  } catch (BrokenBarrierException e) {  e.printStackTrace();  } catch (Exception e) {  e.printStackTrace();  } finally {  if (resource != null) {// 连接池最多有8个连接，用完记得关闭  resource.close();  }  try {  cyclicBarrier2.await();  } catch (InterruptedException e) {  e.printStackTrace();  } catch (BrokenBarrierException e) {  e.printStackTrace();  }  }    }  }).start();  }  }  } |

# 连接关闭

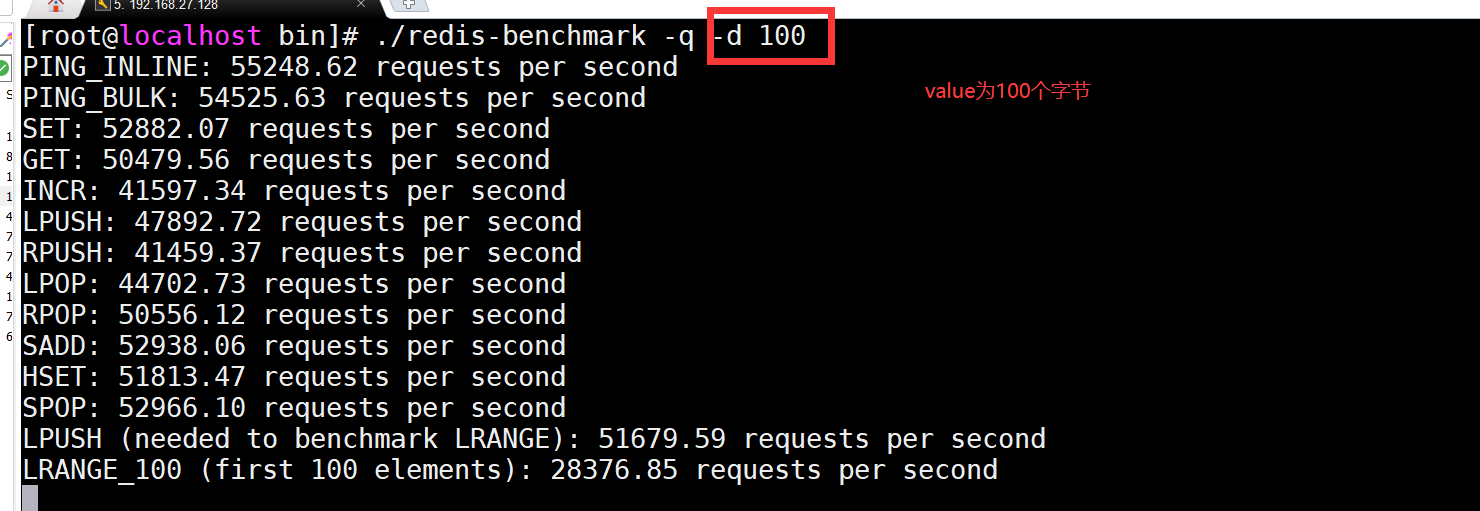
Jedis连接池默认的最大连接数为8个，用完之后记得关闭

# 测试性能

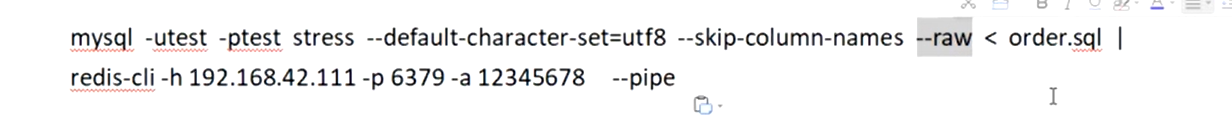
./redis-benchmark -c 100 -n 10000

100个并发连接 10000个请求

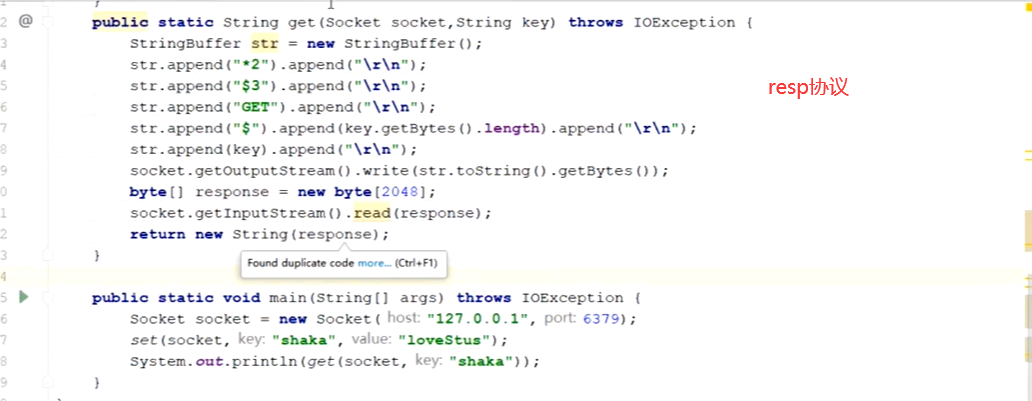
Value为100个字节的测试



# 从mysql往redis导入数据

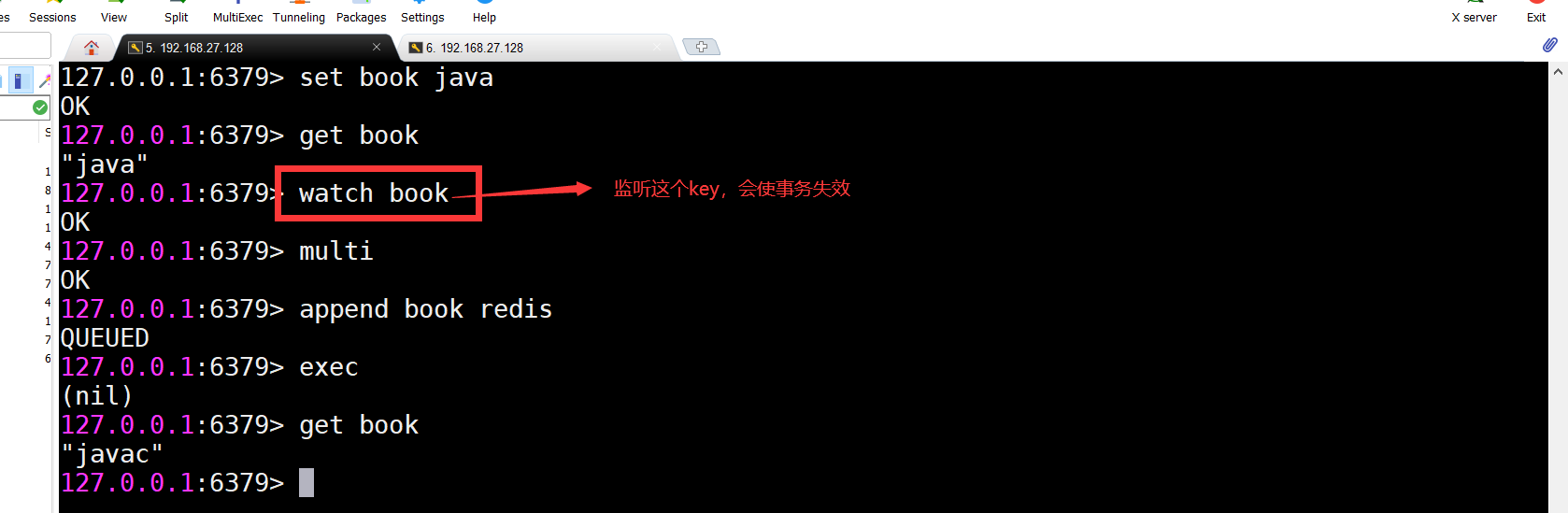


# 手写jedis

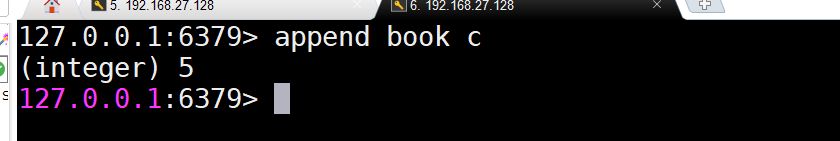


# Watch机制

会话一：监听完key之后，就去第二个会话追加



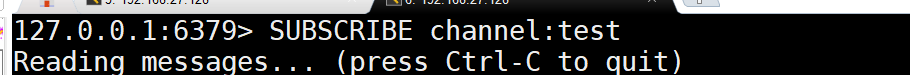
会话二



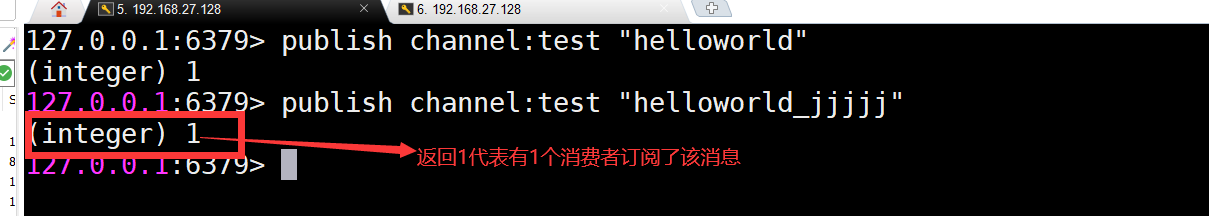
在第一个会话中查询book，发现是javac，在事务中追加的redis并没有追加进去，这是因为监听机制会使事务失效，所以没有追加进去

# 发布订阅

订阅者订阅消息

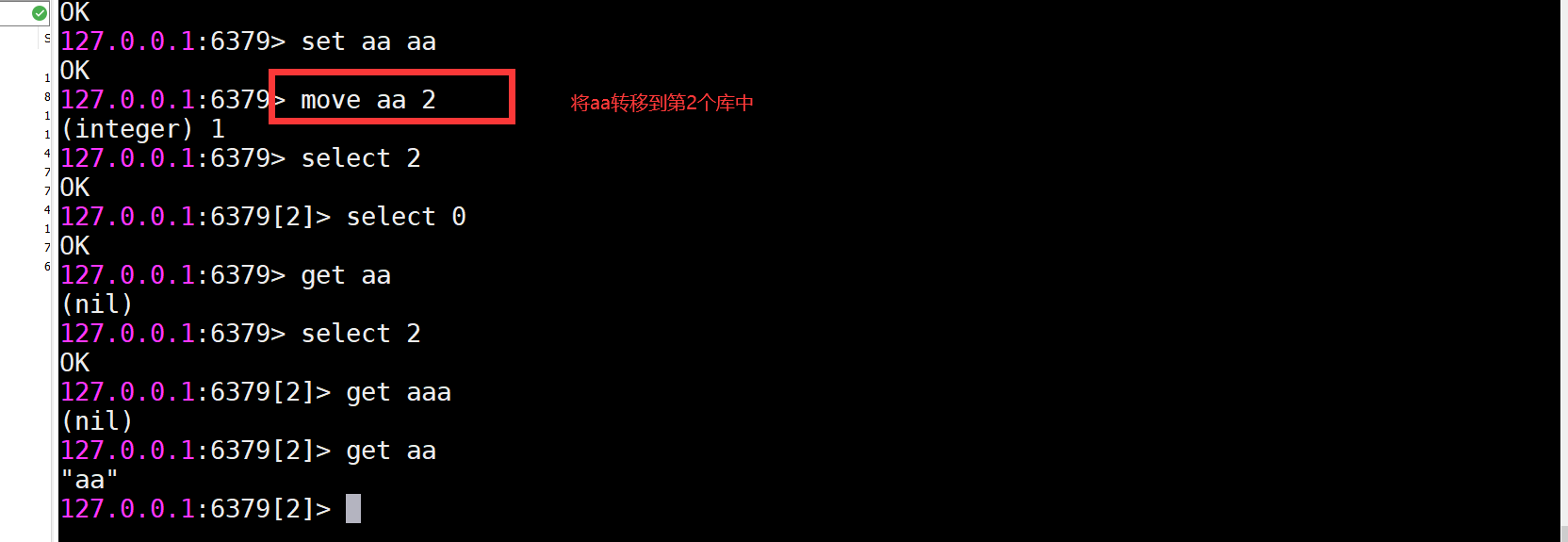


发布者发布消息

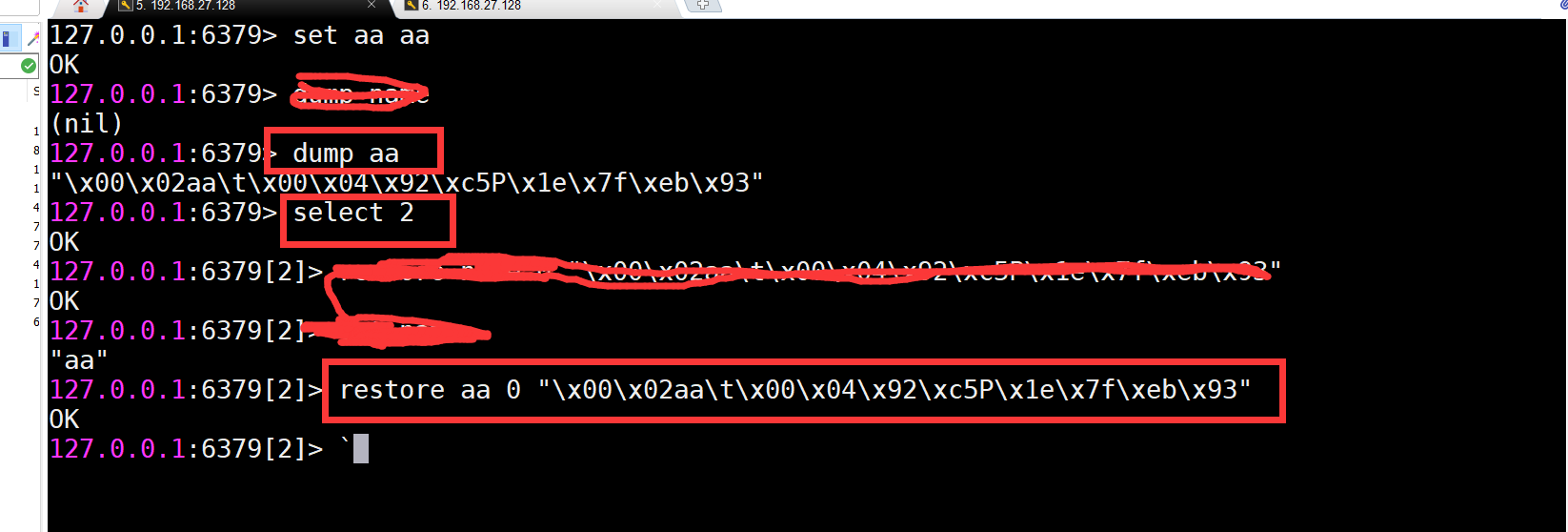


# 键的转移

1、Move key db



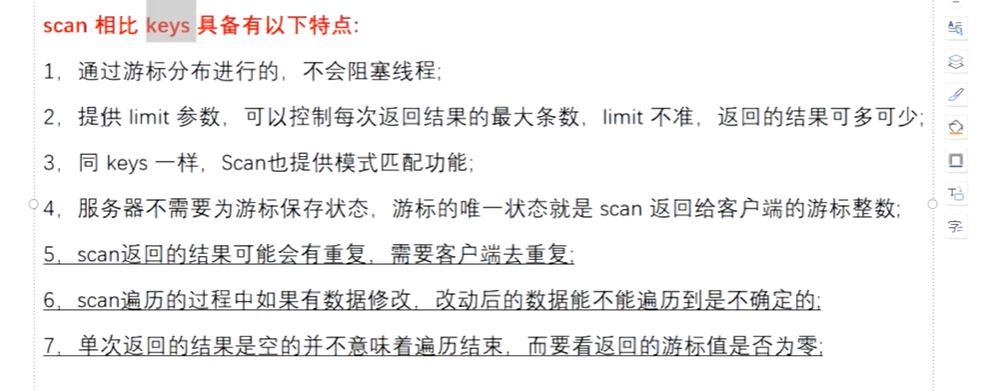
1. dump key



1. migrate



# 遍历key



# Redis高可用

## 主从拓扑

1. 一主多从

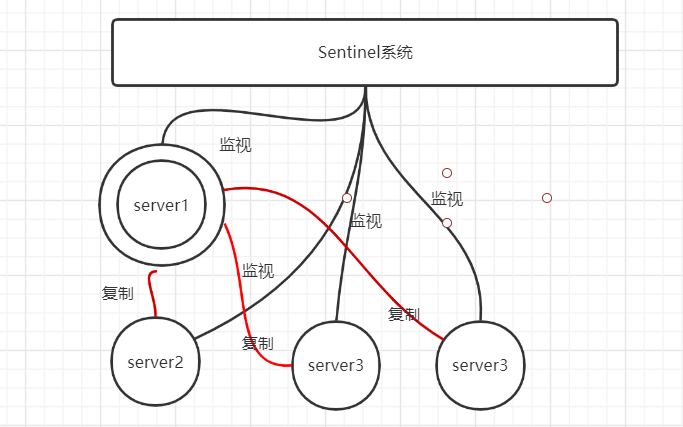
针对”读”较多的情景，”读”由较多个节点来分担，但节点越多，主节点同步到从节点的次数也越多，影响带宽，也影响主节点的稳定性。

查看主从信息

Info replication

## 哨兵

<https://www.cnblogs.com/Eugene-Jin/p/10819601.html>



### 故障转移详细流程

### 

### 作用

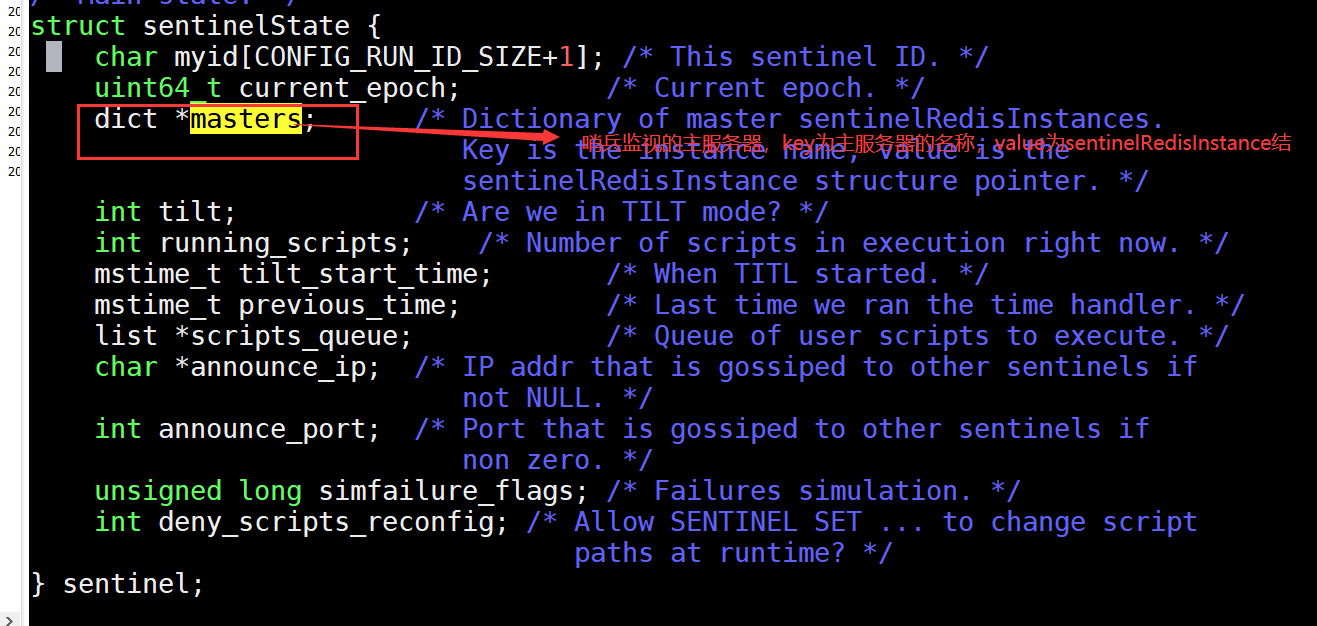
1. 可以监视任意多个主服务器，以及这些主服务器下的从服务器
2. 自动故障转移: 在主服务器下线时，自动将主服务器下的从服务升级为新的主服务器，然后由新的主服务器代替已下线的主服务器继续处理命令请求。

### 重要的属性

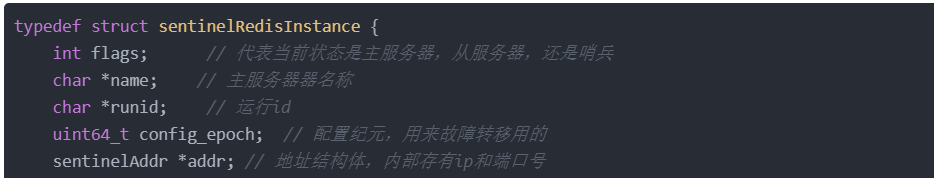
#### quorm

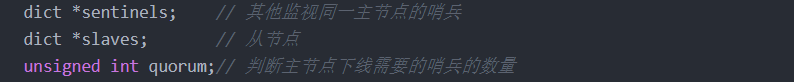
判断主服务器客观下线所需的支持投票数量

### Sentinel状态结构



sentinelRedisInstance的属性





### 如何与被监视的服务器通信

#### 创建连向主服务器的网络连接

命令连接:这个链接专门用于向主服务器发送秘书能够令，并接收命令恢复。

订阅链接:这个连接专门用于订阅主服务器的\_sentinel\_:hello频道。

**为什么有两个连接？**

**在redis目前的发布与订阅功能中，被发送的信息都不会保存在Redis服务器里面，如果在信息发送时，想要接收信息的客户端不在线或者断线，那么这个客户端就会丢失这条信息，想要接收信息的客户端不在线或者断线，那么这个客户端就会丢失这条信息。因此，为了不丢失\_sentinel\_:hello频道的任何信息，Sentinel必须专门用一个订阅链接来接收该频道的信息。**

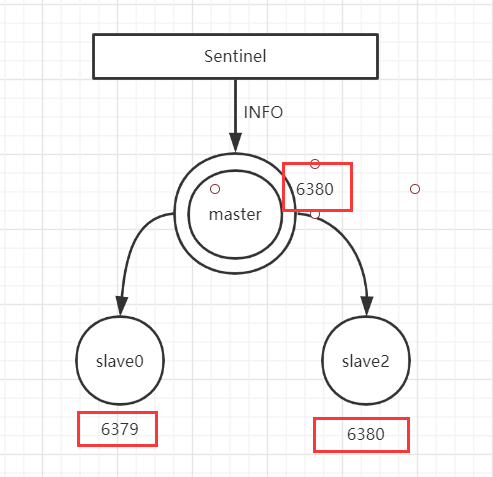
**另一方面，除了订阅频道之外，Sentinel还必须向主服务器发送命令，以此来与主服务器进行通信，所以Sentinel还必须向主服务器发送命令，依次来与主服务器进行通信，所以Sentinel还必须向主服务器创建命令连接。**

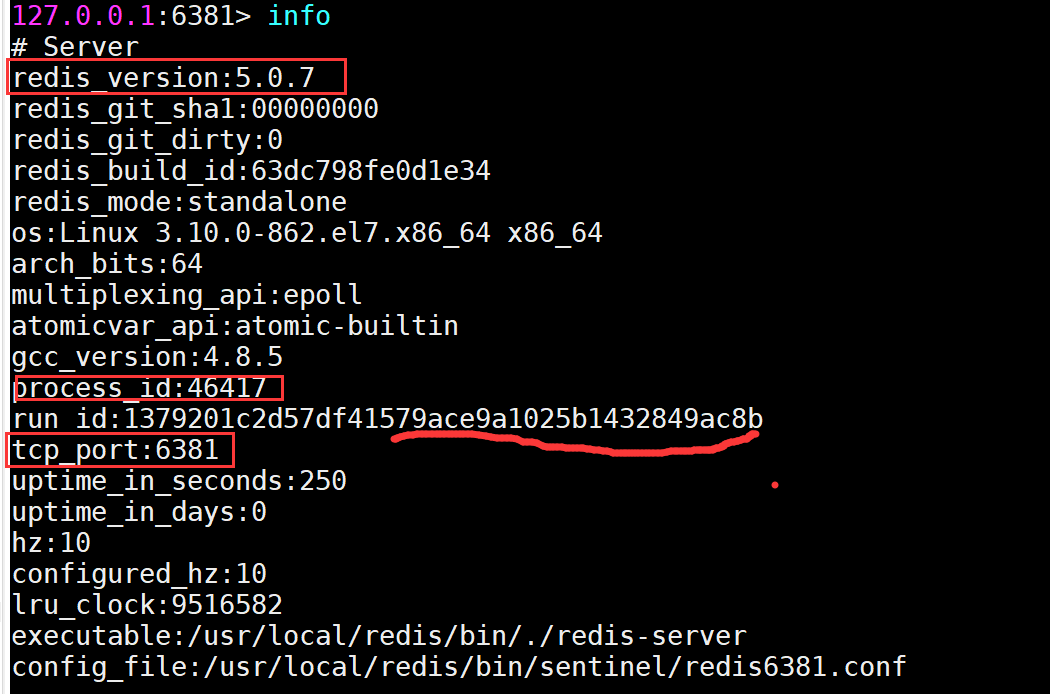
**因为Sentinel需要与多个实例创建多个网络连接，所以Sentinel使用的是异步连接。**

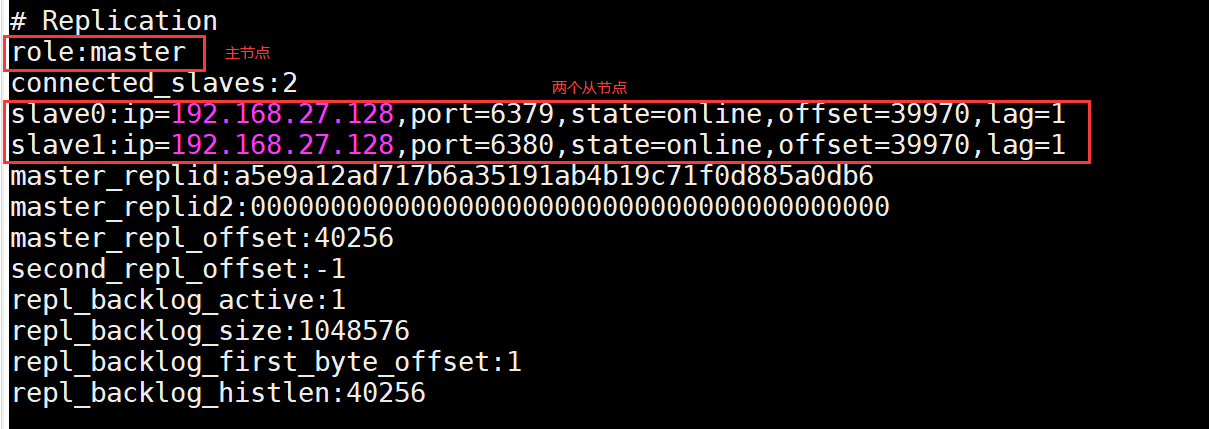
#### 获取主服务器信息

Sentinel将以每10秒一次的频率，通过命令连接向被监视的主服务器发送INFO命令，并通过分析INFO命令的回复来获取主服务器的当前信息。

举个例子，假设如下图所示，主服务器master有三个从服务slave0，slave1，slave2，并且一个Sentinel正在连接主服务器，那么Sentinel将持续地向主服务器发送INFO命令，并获得类似于以下内容的回复:







通过分析主服务器返回的INFO信息，Sentinel可以获取以下两方面的信息:

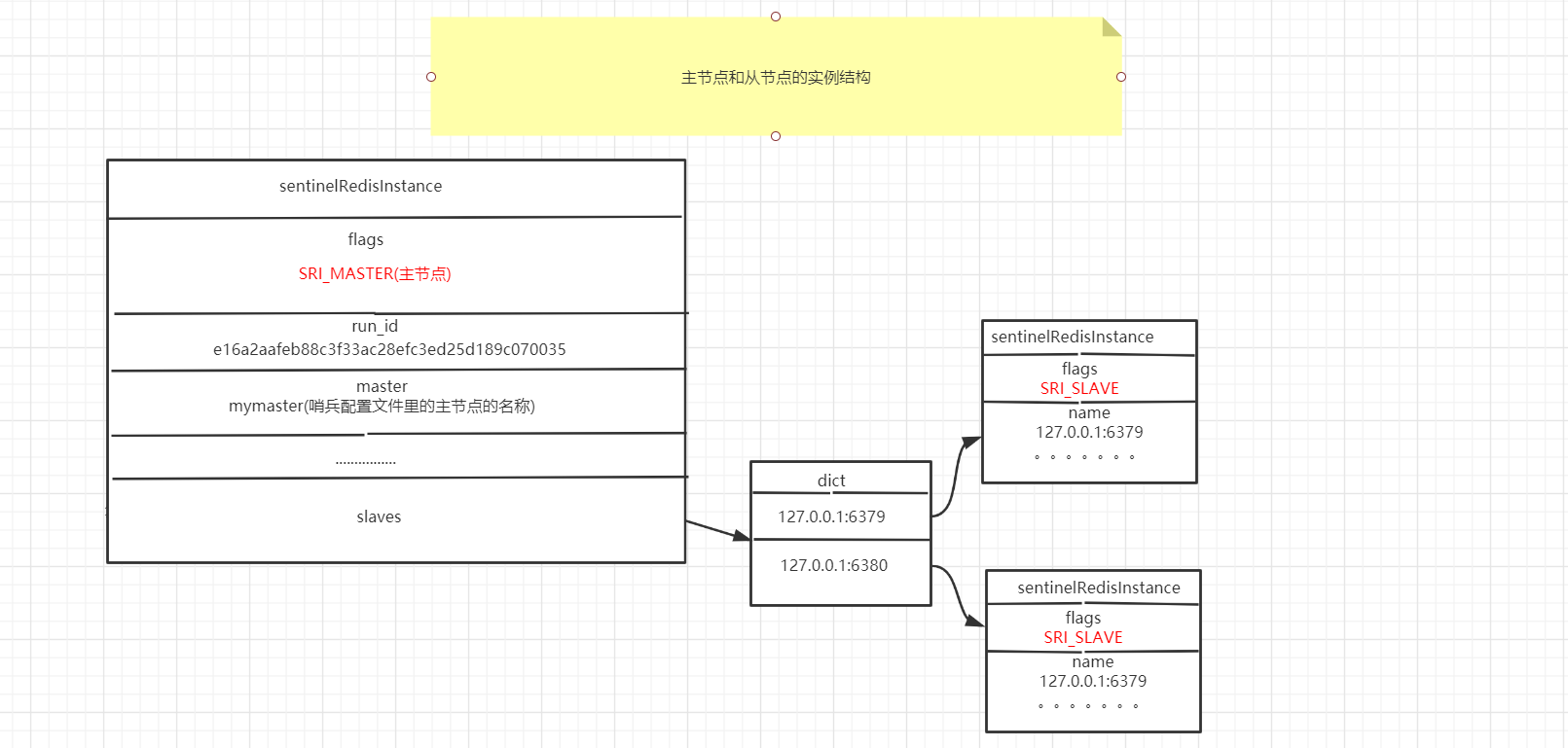
* 一方面是关于主服务器的信息，包括run\_id域记录的服务器运行id，以及role域记录的服务器角色。
* 另一方面是关于主服务器下的从服务器的信息，每个从服务器都由一个“slave”开头的行记录，每行的ip=域记录了从服务器的IP地址，而port=域则记录了从服务器的端口号。根据这些IP地址和端口号，Sentinel无需用户提供从服务器的地址信息，就可以自动发现从服务器。

根据run\_id和role域记录的信息，Sentinel就可以对主服务器的实例结构进行更新，例如，主服务器重启之后，它的运行id就会和实例结构之前的运行id不同，Sentinel检测到这一情况之后，就会对实例结构的运行id进行更新。

将主服务器重启之后，run\_id如下图，上图中的run\_id不一样



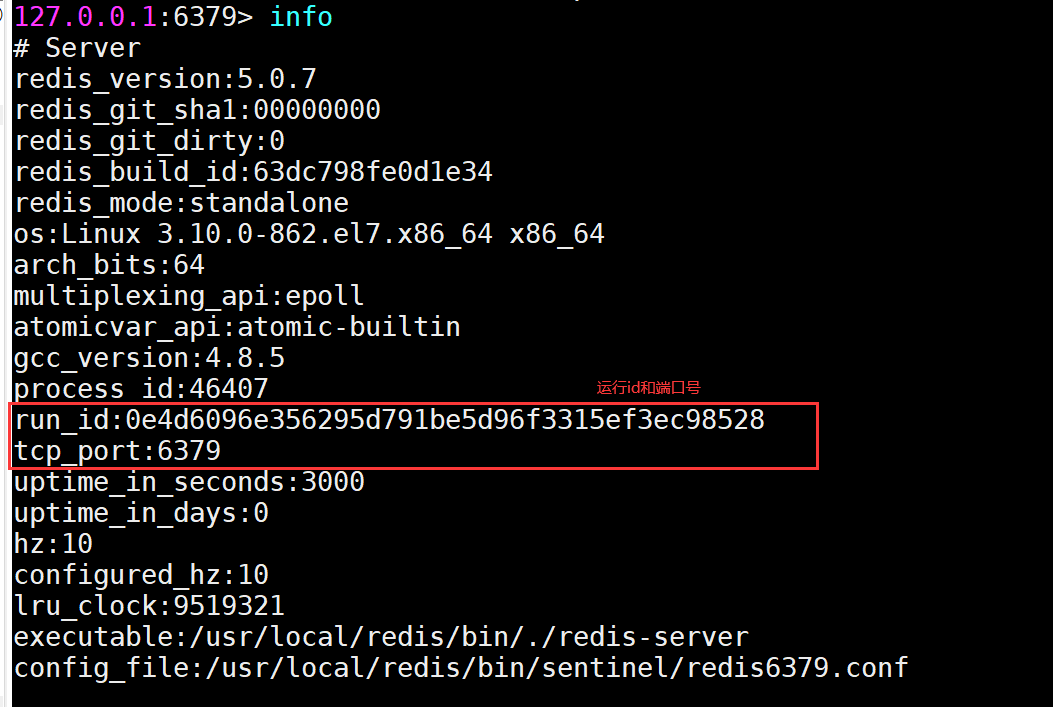
主节点和从节点的实例结构：

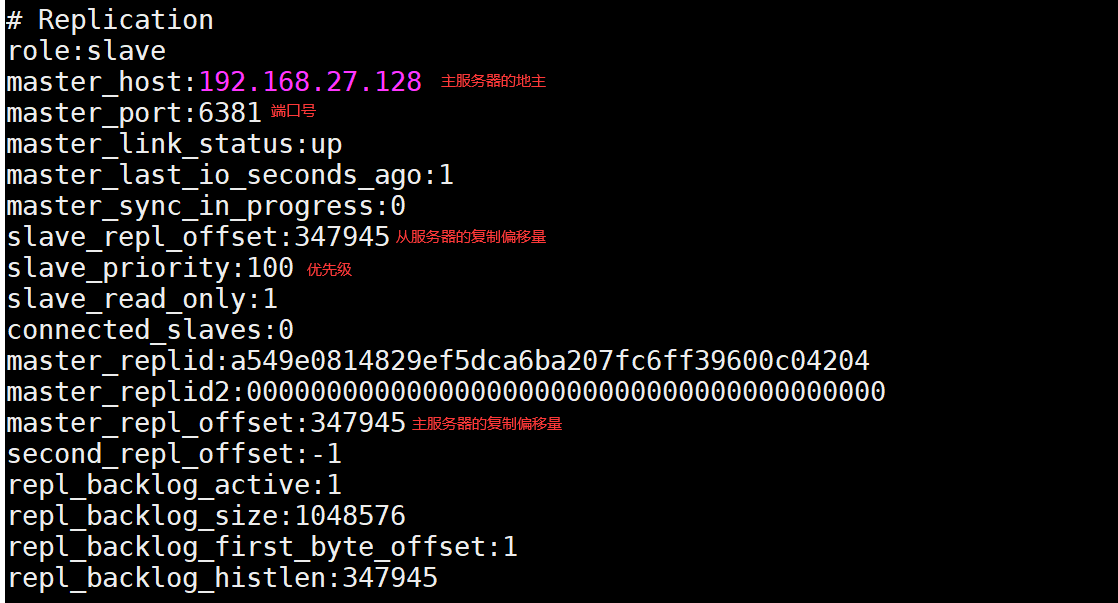


#### 获取从服务器信息

当Sentinel发现主服务器有新的从服务器时，Sentinel除了会为这个新的从服务器创建相应的实例结构之外，Sentinel还会创建连接到从服务器的命令连接和订阅连接。

在创建命令连接之后，Sentinel会以每10秒一次的频率通过命令连接向从服务器发送INFO命令，并获得类似于以下的回复:

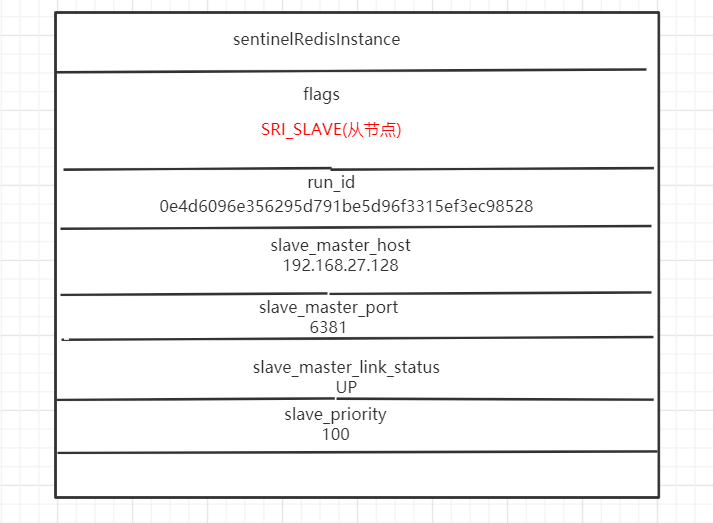




根据INFO命令的回复，Sentinel会提取出以下信息：

* 从服务器的运行ID
* 从服务器的role
* 主服务器的IP地址，以及主服务器的端口号
* 主从服务器的连接状态master\_lin\_status
* 从服务器的优先级
* 从服务器的复制偏移量

根据这些信息，Sentinel会对从服务器的实例结构进行更新。下图就是实例结构更新之后的样子：



#### 向主服务器和从服务器发送信息

在默认的情况下，Sentinel会以每两秒一次的频率，通过命令连接向所有被监视得到主服务器和从服务器发送以下格式的命令

PUBLISH \_sentinel\_:hello "192.168.27.128,26379,431b10ecf2c82a0c02e8d2bfa9bd1e176a031984,0,mymaster,192.168.27.128,6381,0"

红色字体是哨兵的信息：第一个参数代表ip地址，第二个参数代表端口号，第三个参数代表运行id，第四个参数代表当前配置纪元。

后面的参数是被监视服务器的信息。

#### 创建连向其他Sentinel的命令连接

当Sentinel通过频道信息发现一个新的Sentinel使，它不仅会为新Sentinel在sentinel是字典中创建相应的是实例结构，还会创建一个连向新sentinel的命令连接，而新sentinel也同样会创建连向这个Sentinel的命令连接，最终监视同一主服务器的多个Sentinel将形成相互连接的网络：SentinelA有连向SentinelB的命令连接，SentinelB也同样有，连向SentinelA的命令连接。

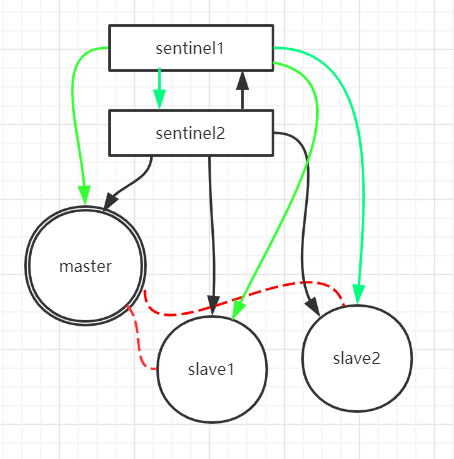
使用命令连接项相连的Sentinel可以通过向其他Sentinel发送命令请求来进行信息交换，Sentinel实现主观下线检测和客观下线检测都会使用Sentinel之间的命令连接。

##### Sentinel之间不会创建订阅连接

Sentinel在连接主服务器和从服务器时，会同时创建命令连接和订阅连接，但是在连向其他Sentinel时，却只会创建命令连接，而不创建订阅连接。这是因为Sentinel需要通过接收主服务器和从服务器发来的频道信息来发现未知的新的Sentinel，所以才需要建立订阅连接，而相互已知的Sentinel只要使用命令连接来进行通信就足够了。

### 检测主观下线状态

在默认情况下，Sentinel会以每秒一次的频率向所有与它创建了命令连接的实例（保护主服务器，从服务器，其他哨兵在内）发送PING命令，并通过实例返回的PONG命令来判断实例是否还在线。



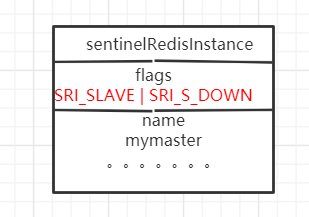
在上图中，展示了Sentinel1和Sentinel2如何向实例发送PING命令的：

* Sentinel1将向Sentinel2、master、slave1、slave2发送PING命令。
* Sentinel2将向Sentinel1、master、slave1、slave2发送PING命令。

实例对PING命令的回复可以分为以下两种情况：

* 有效回复：实例返回+PONG、-LOADING、-MASTERDOWN三种回复的其中一种。
* 无效回复：实例返回出+PONG、-LOADING、-MASTERDOWN三种回复之外的其他回复，或者在指定时限内没有任何回复。

以上图展示的情况为例子，如果配置文件指定的Sentinel1的down-after-milliseconds选项指定了Sentinel判断实例进入主观下线所需的时间长度：如果一个实例在down-after-milliseconds毫秒内，连续向Sentinel返回无效回复，那么Sentinel会修改这个实例所对应的实例结构，在结构的flags属性中打开SRI\_s\_DOWN标识，以此来表示这个实例已经进入主观下线状态。



sentinel.conf



|  |
| --- |
| **主观下线时长选项的作用**  **用户设置的**down-after-milliseconds选项的值，不仅会被Sentinel用来判断主服务器的主观下线状态，还会被用于判断主服务器属下的所有从服务器，以及所有同样监视这个主服务器的其他Sentinel的主观下线状态。 |

### 检查客观下线状态

当Sentinel将一个主服务器判断为主观下线之后，为了确认这个主服务器是否真的下线了，它会向同样监视这一主服务器的其他Sentinel进行询问，看它们是否也认为主服务器已经进行了下线状态(可以是主观下线或者客观下线)。当Sentinel从其他Sentinel那里接收到足够数量的已下线判断之后，Sentinel就会将从服务器判定为客观下线，并对主服务器执行故障转移操作。

#### 发送SENTINEL is-master-down-by-addr

Sentinel使用：

SENTINEL is-master-down-by-addr <ip> <port> <current\_epoch> <runid>

|  |  |
| --- | --- |
| 参数 | 意义 |
| Ip | 被Sentinel判断为主观下线的主服务器的ip地址 |
| Port | 被Sentinel判断为主观下线的主服务器的端口号 |
| current\_epoch | Sentinel当前的配置纪元，用于选举领头Sentinel |
| Runid | 可以是\*或者Sentinel的运行id：\*代表命令仅仅用于检测主服务器的客观下线状态，而Sentinel的运行id则用于选举领头Sentinel |

例如：SENTINEL is-master-down-by-addr 192.168.21.128 6381 0 \*

#### 接收SENTINEL is-master-down-by-addr

当一个Sentinel（目标Sentinel）接收到另一个Sentinel（源Sentinel）发来的19.2.7.2.SENTINEL is-master-down-by-addr命令时，目标Sentinel会分析并且取出命令请求中包含的各个参数，并根据其中的主服务器的ip和端口号，检查主服务器是否已经下线，然后向源Sentinel返回一条包含三个参数的Multi Bulk回复作为19.2.7.2.SENTINEL is-master-down-by-addr命令的回复

1. <down\_state>
2. <leader\_runid>
3. <leader\_epoch>

|  |  |
| --- | --- |
| 参数 | 意义 |
| down\_state | 返回目标Sentinel对主服务器的检查结果，1代表主服务器已经下线，0代表未下线。 |
| leader\_runid | 可以是\*或者目标Sentinel的局部领头Sentinel的运行ID：\*符号代表命令仅仅用于检测主服务器的下线状态，而局部领头Sentinel的运行ID则用于选举领头Sentinel |
| leader\_epoch | 目标Sentinel的局部领头Sentinel的配置纪元，用于选举领头Sentinel，仅在leader\_runid的值不为\*时有效，如果leader\_runid的值为\*，那么leader\_epoch的值宗伟0。 |

举个例子：

1. 1
2. \*
3. 0

#### 源Sentinel接收SENTINEL is-master-down-by-addr命令的回复

根据其他Sentinel发回的SENTINEL is-master-down-by-addr命令回复，Sentinel将统计其他Sentinel同意主服务器已下线的数量，当这一数量达到配置指定的判断客观下线所需的数量时，Sentinel会将主服务器实例结构flags属性的SRI\_O\_DOWN标识打开，表示主服务器已经进入客观下线状态

|  |
| --- |
| **客观下线的判断条件**  当认为主服务器已经进入下线状态的Sentinel的数量，超过Sentinel配置中设置的quorum参数的值，那么该Sentinel就会认为主服务器已经进入客观下线状态。 |

### 选举领头Sentinel

当一个主服务器被判断为主观下线时，监视这个下线主服务器的各个Sentinel会进行协商，选举出一个领导Sentinel，并由领头Sentinel对下线主服务器执行故障转移操作。

以下是Redis选举领头Sentinel的规则和方法：

* 所有在线的Sentinel都由被选为领头Sentinel的资格，换句话说，监视同一个主服务器的多个在线Sentinel中的任意一个都由可能成为领头Sentinel。
* 每次进行领头Sentinel选举只有，不论选举是否成功，所有Sentinel的配置纪元的值都会自增一次。配置纪元实际上就是一个计数器，并没有什么特别的。
* 在一个配置纪元里面，所有Sentinel都由一次将某个Sentinel设置为Sentinel为局部领头Sentinel的机会，并且局部领头一旦设置，在这个配置纪元里面就不能更改。
* 当一个Sentinel（源Sentinel）向另一个Sentinel（目标Sentinel）发送SENTINEL is-master-down-by-addr命令，并且命令中的runid参数不是\*而是源Sentinel的运行ID时，这表示源Sentinel要求目标Sentinel将前者设置为后者的局部领头Sentinel。
* Sentinel设置局部领头Sentinel的规则是先到先得：最先向目标Sentinel发送设置要求的源Sentinel将成为目标的局部领头Sentinel，而之后接收到的所有设置要求都会目标Sentinel拒绝。
* 目标Sentinel在接收到SENTINEL is-master-down-by-addr命令之后，将向源Sentinel返回一条命令回复，回复中的leader\_runid参数和leader\_epoch参数分别记录了目标Sentinel的局部领头Sentinel的运行ID和配置纪元。
* 源Sentinel在接收到目标Sentinel返回的命令回复之后，会检查回复中leader\_epoch参数的值和自己的配置纪元是否相同，如果相同的话，那么元Sentinel继续取出回复中的runid参数，如果leader\_runid的值和源Sentinel的runid的值一致，那么表示目标Sentinel将源Sentinel设置成了局部领头Sentinel。
* 如果有某个Sentinel被半数以上的Sentinel设置成了局部领头Sentinel，那么这个Sentinel就成为了领头Sentinel。举个例子，在一个由10个Sentinel组成的Sentinel系统里面，只要有大于等于10/2+1=6个Sentinel将某个Sentinel设置成了局部领头Sentinel，那么被设置的那个Sentinel就会成为领头Sentinel。
* 因为领头Sentinel的产生需要半数以上的Sentinel的支持，并且每个Sentinel在每个配置纪元里面只能设置一次局部领头Sentinel，所以在一个配置纪元里面，只会出现一个领头Sentinel。
* 如果在给定时限内，没有一个Sentinel被选举成为领头Sentinel，那么各个Sentinel将在一段时间之后再次进行选举，直到选出领头Sentinel为止。

### 故障转移

在选举产生出领头Sentinel之后，领头Sentinel将对已下线的主服务器执行故障转移操作，该操作主要包括以下三个步骤：

1. 在已下线主服务器属下的所有从服务器里面，挑选出一个从服务器，并将其设置成为主服务器。
2. 让已下线的主服务器属下的所有从服务器改为复制新的主服务器。
3. 将已下线主服务器设置为新的主服务器的从服务器，当这个旧的主服务器重新上线时，就成为新的主服务器的从服务器。

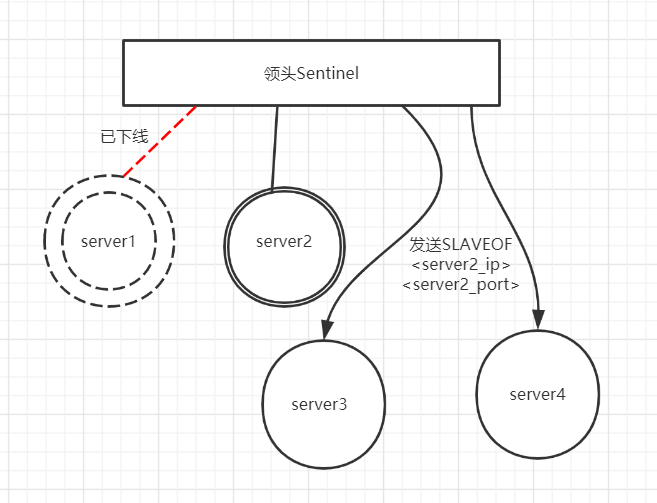
#### 选出新的主服务器

故障转移操作的第一步就是在已下线的属下的所有从服务器中，挑选出一个状态良好、数据完整的从服务器，然后向这个从服务器发送SLAVEOF no one命令，将这个从服务器转换为主服务器。

|  |
| --- |
| **新的主服务器是怎样挑选出来的**  **领头Sentinel会将已下线主服务器的所有从服务器保存到一个列表里面，然后按照以下规则，一项一项地对列表进行过滤：**   1. **剔除列表中所有处于下线或者断线状态的从服务器，这可以保证列表中剩余的从服务器都是正常在线的。** 2. **剔除列表中所有最近5秒内都没有回复过领头Sentinel的INFO命令的从服务器，这可以保证列表中剩余的从服务器都是租金成功进行通过信的。** 3. **剔除所有与已下线主服务器连接断开超过**down-after-milliseconds\*10毫秒的从服务器，可以保证剩余的从服务器没有过早的与主服务器断开连接，换句话说，列表中剩余的从服务器保存的数据都是比较新的。 4. 之后，领头Sentinel将根据从服务器的优先级，对列表中剩余的从服务器进行排序，并选出优先级最高的从服务器。 5. 如果有多个具有相同最高优先级的从服务器，那么领头Sentinel将按照从服务器的复制偏移量，对具有相同优先级的所有从服务器进行排序，并选出其中偏移量最大的从服务器（复制偏移量最大的从服务器就是保存着最新数据的从服务器）。 6. 如果有多个优先级、复制偏移量相同的从服务器，那么领头Sentinel将按照运行id金雄排序，并选出运行ID最小的从服务器。。 |

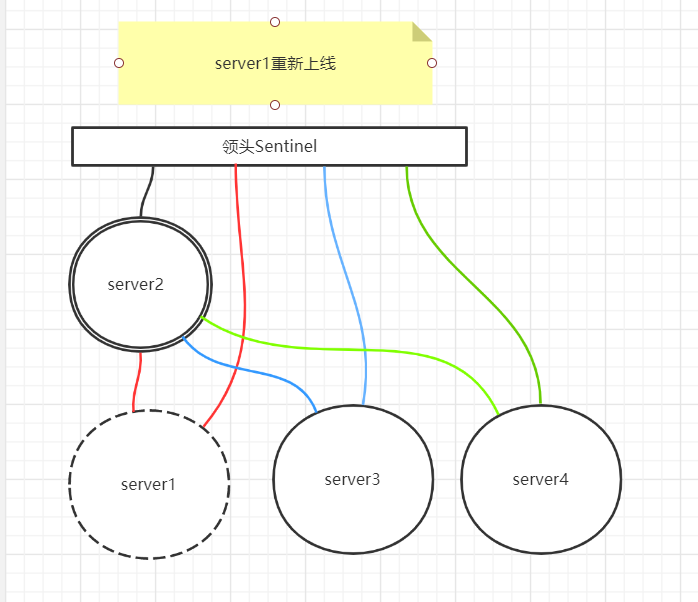
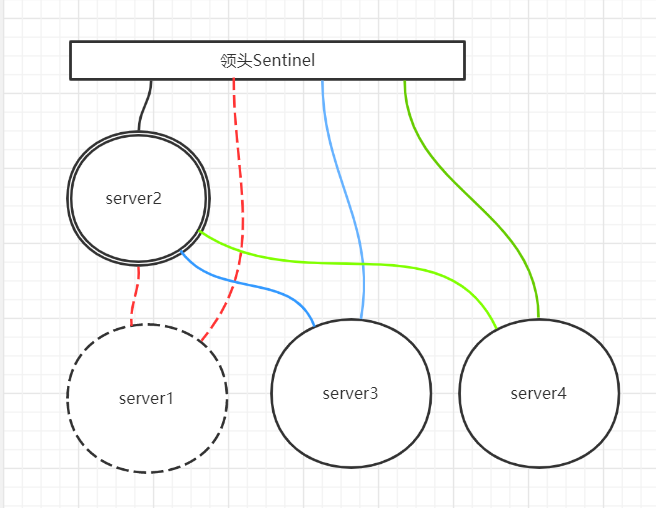
#### 修改从服务器的复制目标

当新的主服务器出现之后，领头Sentinel下一步要做的就是，让已下线主服务器属下的所有从服务器去复制新的主服务器。这一动作通过向从服务器发送SLAVEOF命令来实现



#### 将旧的主服务器变为从服务器

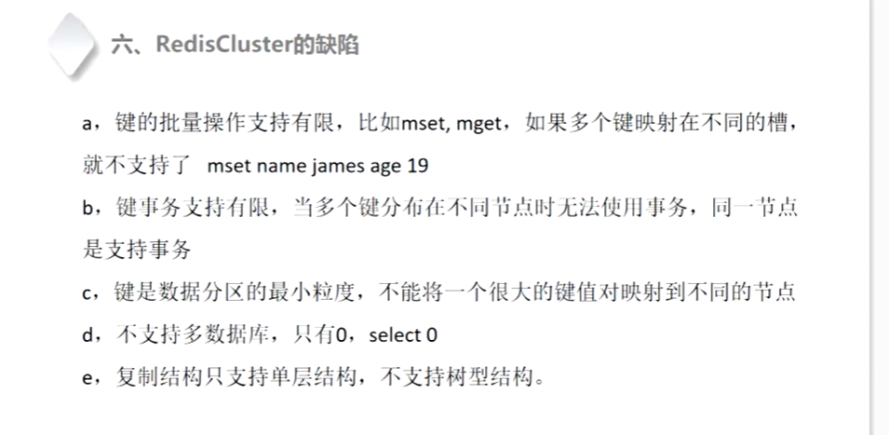
故障转移操作最后要做的就是，将已下线的主服务器设置为新的主服务器的从服务器。



# 分布式的解决方案

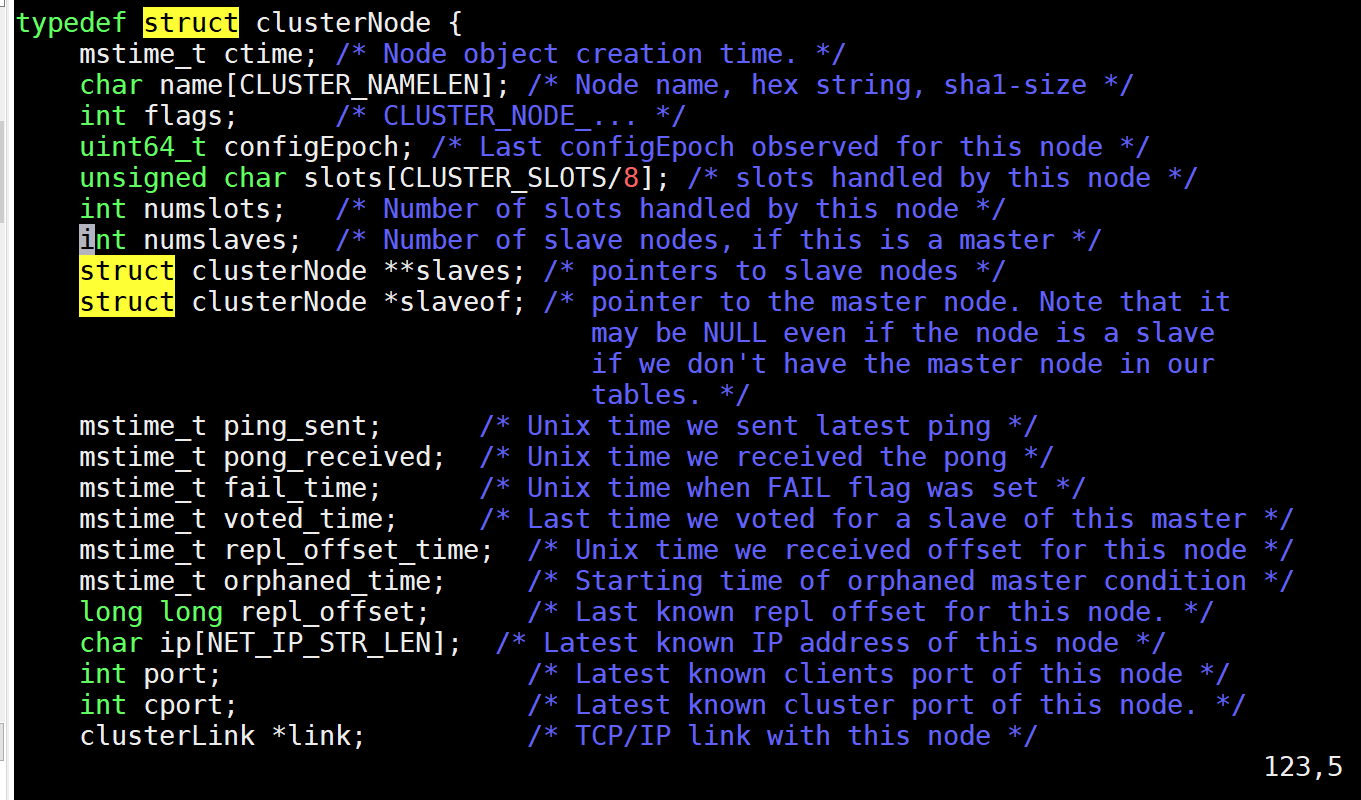
## 集群

### 缺陷

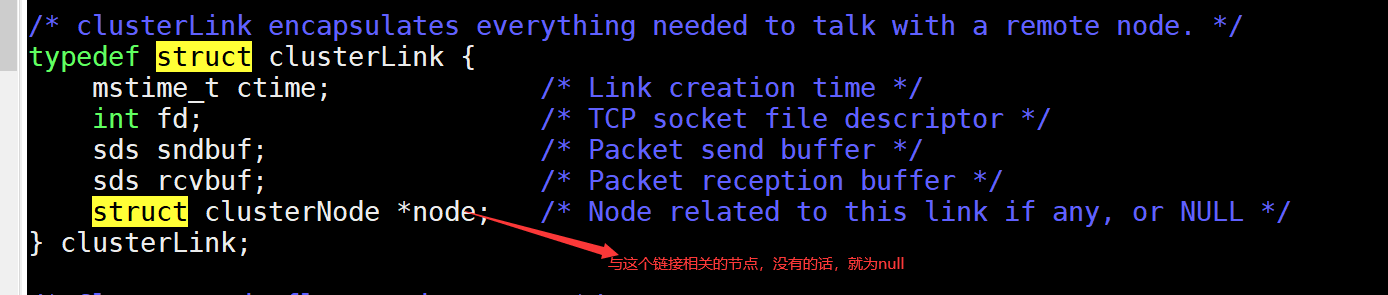


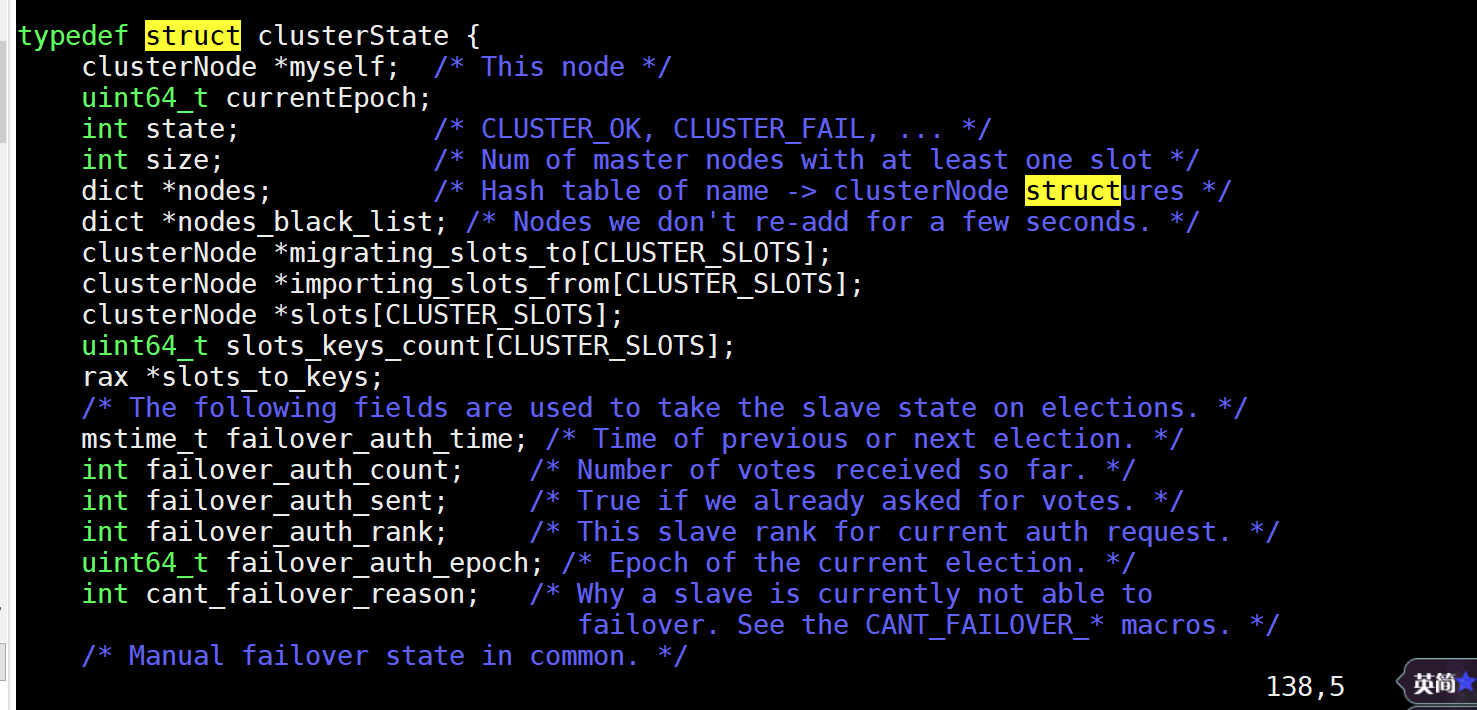
### 集群数据结构

clusterNode:



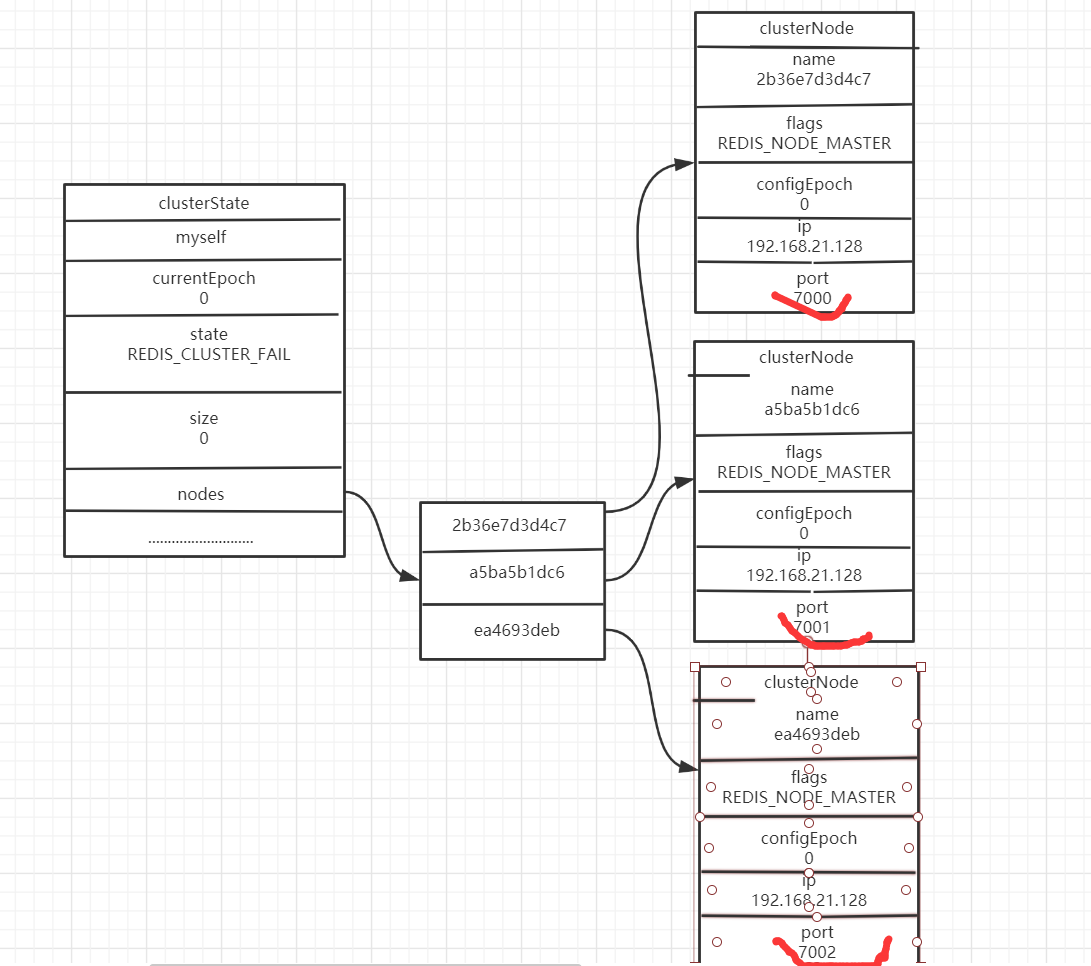
clusterLink:





#### **ClusterState**结构

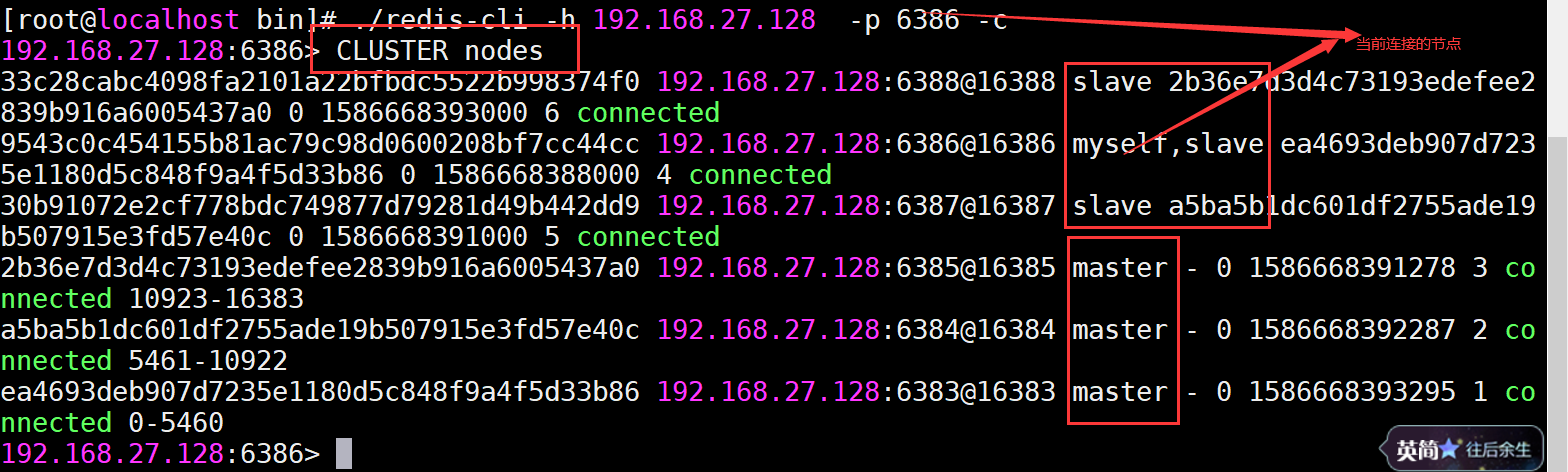
假设当前有三个节点7000,7001,7002，节点7000的clusterState结构如下图所示：



### 查看节点信息

**连接集群节点的命令：** ./redis-cli -h 192.168.27.128 -p 6386 -c

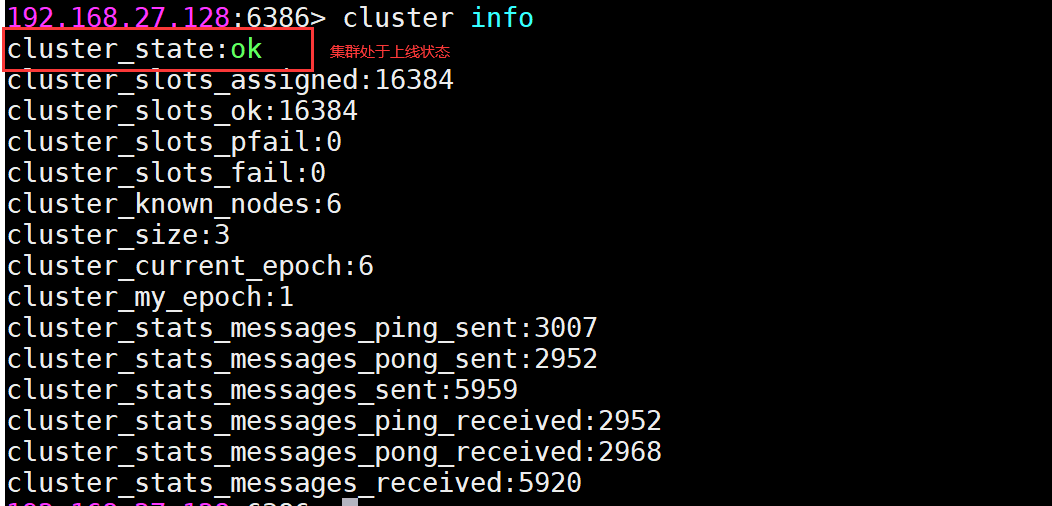
**查看节点信息的命令：**CLUSTER nodes

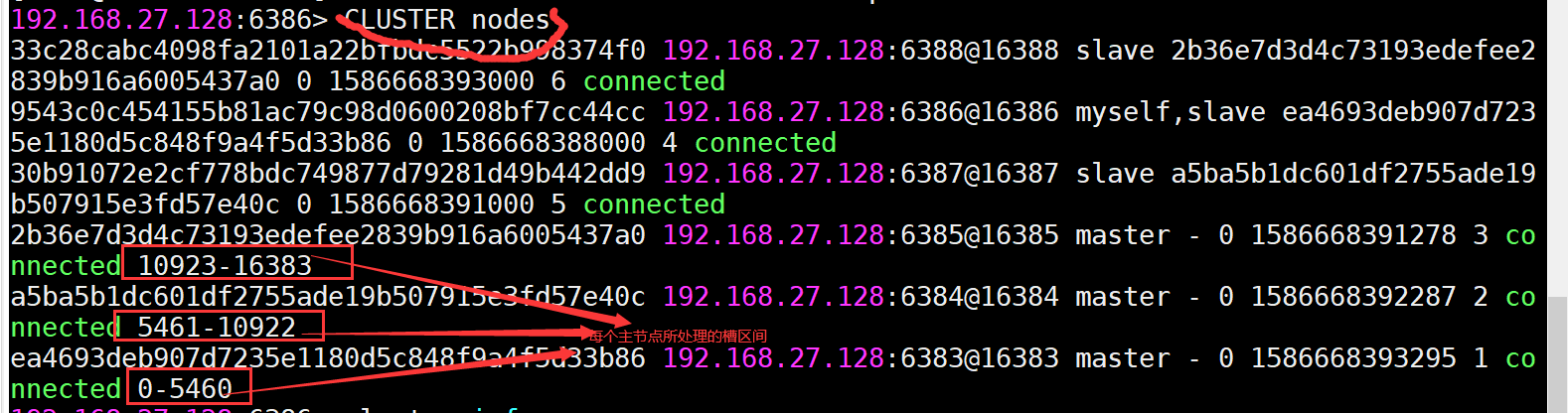


### 槽指派

Redis集群通过分片的方式来保存数据库中的键值对。集群的整个数据库被分为16384个槽，数据库中的每个键都属于这16384个槽中的其中一个，集群中的每个节点可以处理0个或者最多16384个槽。

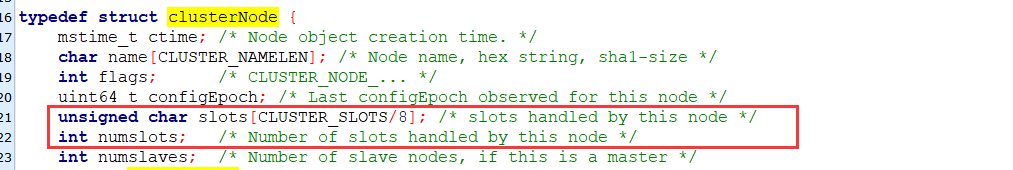
当数据库中的16384个槽都有节点在处理时，集群处于上线状态，否则集群处于下线状态。





#### 记录节点的槽指派信息

ClusterNode结构的slots属性和numslots属性记录了节点负责处理哪些槽。



Slots属性是一个二进制位数组，这个数组的长度为16384/8=2048个字节，共包含16384个二进制位。

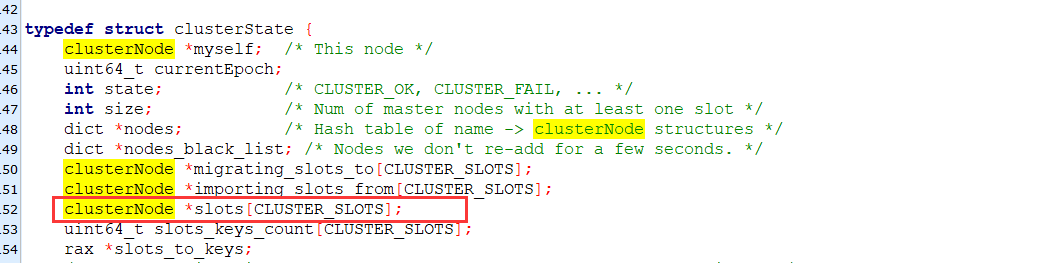
Redis以0位起始索引，16383位终止索引，对slots数组中的16384个二进制位进行编号，并根据索引i上的二进制位的值来判断节点是否负责处理槽i：

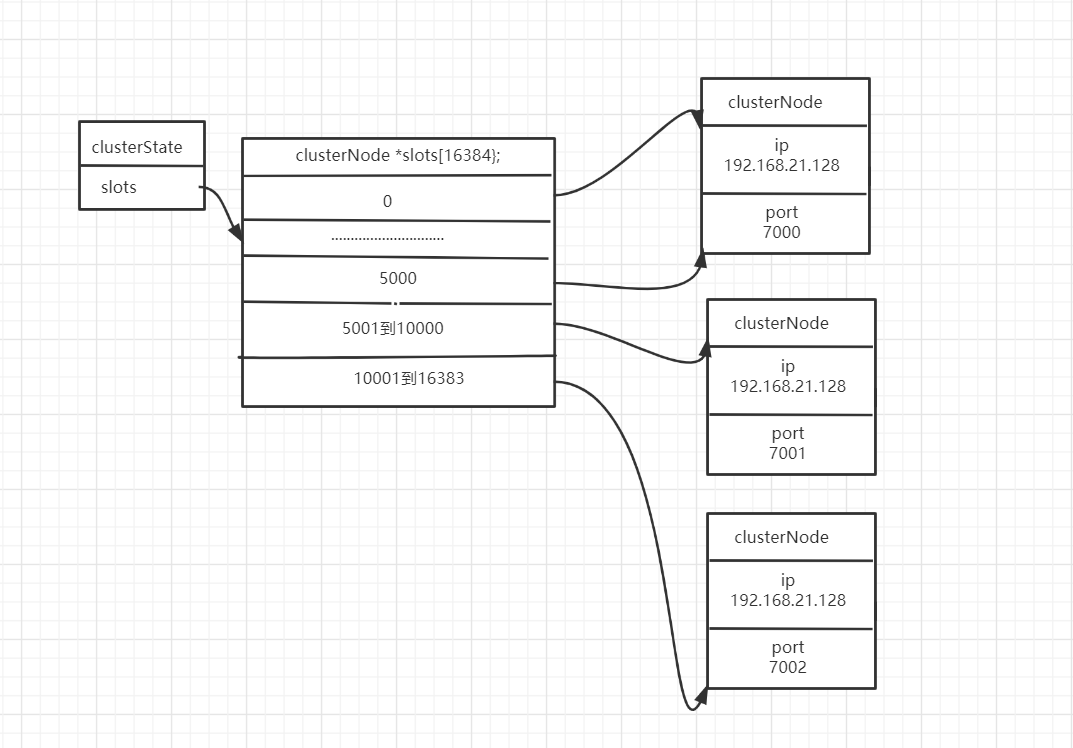
* 如果slots数组在索引i上的二进制位的值为1，表示节点负责处理槽i。
* 如果slots数组在索引i上的二进制位位0，表示节点不负责处理槽i。

Numslots表示节点负责处理槽的数量。

#### 记录集群所有槽的指派信息

clusterState中的slots属性记录了集群中所有槽的指派信息：



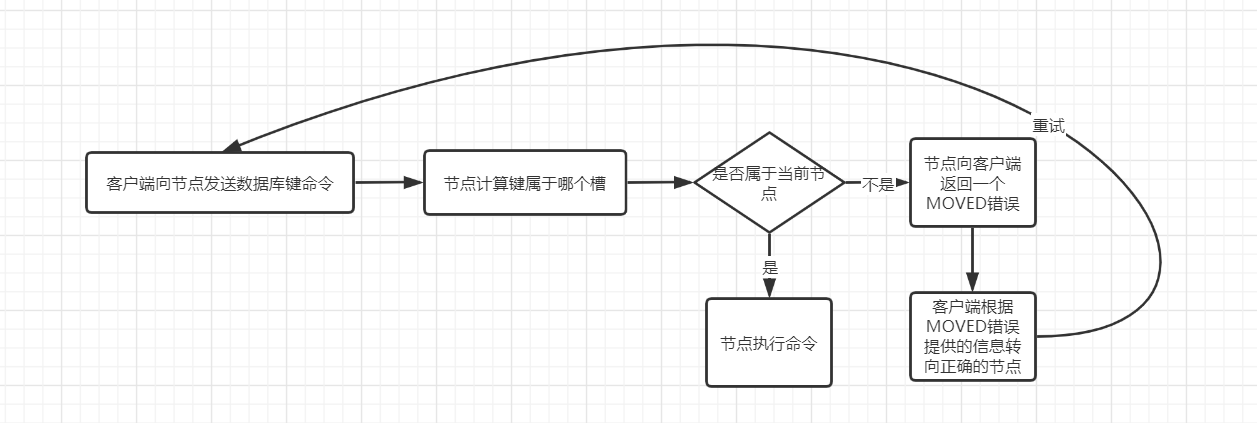


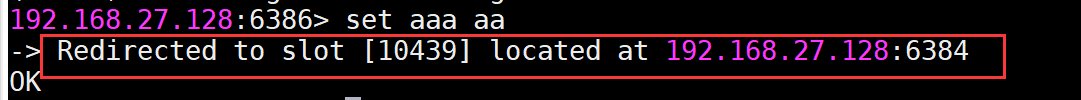
虽然clusterState.slots数组记录了集群中所有槽的指派信息，但使用clusterNode结构slots数组来记录单个节点的槽指派信息仍然是必要的。

* 因为当程序需要将某个节点的槽指派信息通过消息发送给其他节点时，程序只需要将相应节点的clsuterNode.slots数组整个发送出去就行了。
* 另一方面，如果Redis如果不适用clusterNode.slots数组，而单独使用clusterState.slots数组的话，那么每次要将节点A的槽指派信息传播给其他节点时，程序必须买遍历整个clusterState.slots数组，记录节点A负责处理哪些槽，然后才能发送节点A的槽指派信息，这比直接发送clusterNode.slots数组要麻烦和低效的多。

### 在集群中执行命令

执行命令的流程：





#### 计算键属于哪个槽

节点使用以下算法来计算键属于哪个槽：

Def slot\_number(key):

Return CRC16(key) & 16383;

其中CRC16(key)语句用于计算键的CRC-16校验和。

### 复制与故障转移

#### 故障检测

集群中的每个节点都会定期的向集群中的其他节点发送PING消息，以此来检测对方是否在线，如果接受PING消息的节点没有在规定的时间内日，向发送PING消息的节点返回PONG消息，那么发送PING消息的节点就会将接收PING消息的节点标记为疑似下线。

如果在一个集群里面，半数以上负责处理槽的主节点都将某个主节点x报告为疑似下线，那么这个主节点x将被标记为已下线，将主节点x标记为已下线的节点会向集群广播一条关于主节点x的FAIL消息，所有收到这条FAIL消息的节点都会立即将主节点x标记为已下线。

#### 故障转移

当一个从节点发现自己正在复制的主节点进入了已下线状态时，从节点将开始对下线主节点进行故障转移，以下是故障转移的步骤：

1. 复制下线主节点的所有从节点里面，会有一个从节点被选中。
2. 被选中的从节点执行SLAVE no one命令，成为新的主节点。
3. 新的主节点会撤销所有对已下线主节点的槽指派，并将这些槽全部指派给自己。
4. 新的主节点向集群广播一条PONG消息，这条PONG消息可以让集群中的其他节点立即知道这个节点已经由从节点变成了主节点，并且这个主节点已经接管了原本由已下线节点负责处理的槽。
5. 新的主节点开始接受和自己负责处理的槽有关的命令请求，故障转移完成。

#### 选举新的主节点

新的主节点是通过选举产生的。

1. 集群的配置纪元是一个自增计数器，它的初始值为0.
2. 当集群里的某个节点开始一次故障转移操作时，集群配置纪元的值会被增1.
3. 对于每个配置纪元，集群里每个负责处理槽的主节点都有一次投票的机会，而第一个向主节点要求投票的从节点将获得主节点的投票。
4. 当从节点发现自己正在复制的主节点进入已下线状态时，从节点会向集群广播一条CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_REQUEST消息，要求所有收到这条消息、并且具有投票权的主节点向这个从节点投票。
5. 如果一个主节点具有投票权（它正在负责处理槽），并且这个主节点尚未投票给其他从节点，那么主节点将向要求投票的从节点返回一条CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK消息，表示这个主节点支持从节点成为新的主节点。
6. 每个参与选举的从节点都会接收CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK消息，并根据自己收到了多少条这种消息来统计自己获得了多少主节点的支持。
7. 因为在一个配置纪元里面，每个具有投票权的主节点只能头一次票，所以如果有N个主节点进行投票，那么具有大于等于N/2+1张支持票的从节点只会有一个，这确保了新的主节点只会有一个。
8. 如果在一个配置纪元里面没有从节点能收集到足够多的投票，那么集群计入有四个新的配置纪元，并再次进行选举，知道选举出新的主节点为止。

和选举领头Sentinel的方法非常相似，都是基于raft算法的领头选举。

# 主从复制怎么实现

# 和memcache的区别

1、 Redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可用于缓存其他东西，例如图片、视频等等。   
2、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储。   
3、虚拟内存–Redis当物理内存用完时，可以将一些很久没用到的value 交换到磁盘   
4、过期策略–memcache在set时就指定，例如set key1 0 0 8,即永不过期。Redis可以通过例如expire 设定，例如expire name 10   
5、分布式–设定memcache集群，利用magent做一主多从;redis可以做一主多从。都可以一主一从   
6、存储数据安全–memcache挂掉后，数据没了；redis可以定期保存到磁盘（持久化）   
7、灾难恢复–memcache挂掉后，数据不可恢复; redis数据丢失后可以通过aof恢复   
8、Redis支持数据的备份，即master-slave模式的数据备份。

关于redis和memcache的不同，下面罗列了一些相关说法，供记录：

redis和memecache的不同在于[2]：   
1、存储方式：   
memecache 把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小   
redis有部份存在硬盘上，这样能保证数据的持久性，支持数据的持久化（笔者注：有快照和AOF日志两种持久化方式，在实际应用的时候，要特别注意配置文件快照参数，要不就很有可能服务器频繁满载做dump）。   
2、数据支持类型：   
redis在数据支持上要比memecache多的多。   
3、使用底层模型不同：   
新版本的redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。   
4、运行环境不同：   
redis目前官方只支持LINUX 上去行，从而省去了对于其它系统的支持，这样的话可以更好的把精力用于本系统 环境上的优化，虽然后来微软有一个小组为其写了补丁。但是没有放到主干上

个人总结一下，有持久化需求或者对数据结构和处理有高级要求的应用，选择redis，其他简单的key/value存储，选择memcache。

# 假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如何将它们全部找出来？

使用keys指令可以扫出指定模式的key列表。  
对方接着追问：如果这个redis正在给线上的业务提供服务，那使用keys指令会有什么问题？  
这个时候你要回答redis关键的一个特性：redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。