**Redis**

# 缓存问题

## 缓存雪崩

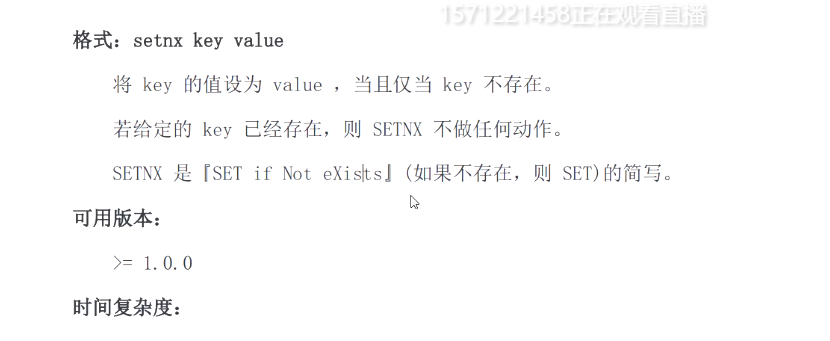
## 缓存击穿

## 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

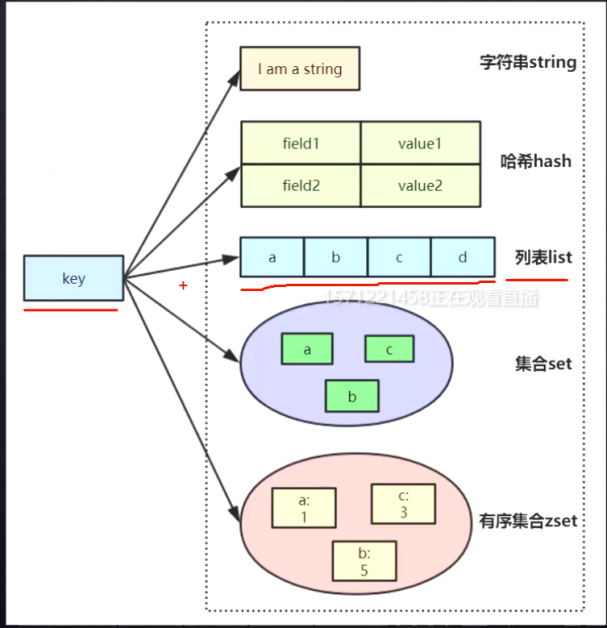
Strings类型：一个String类型的value最大可以存储512M

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

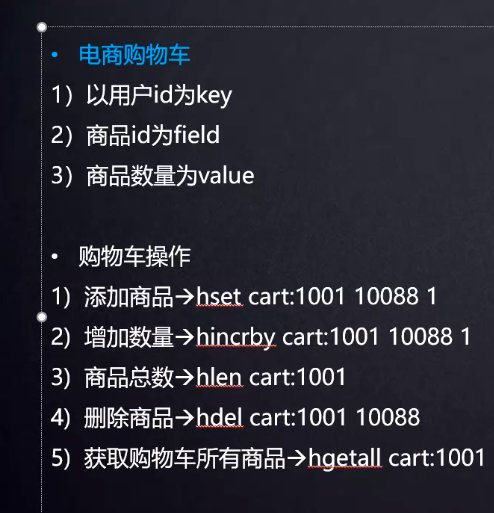
Sets类型：元素个数最多为2^32-1个，也就是4294967295个。

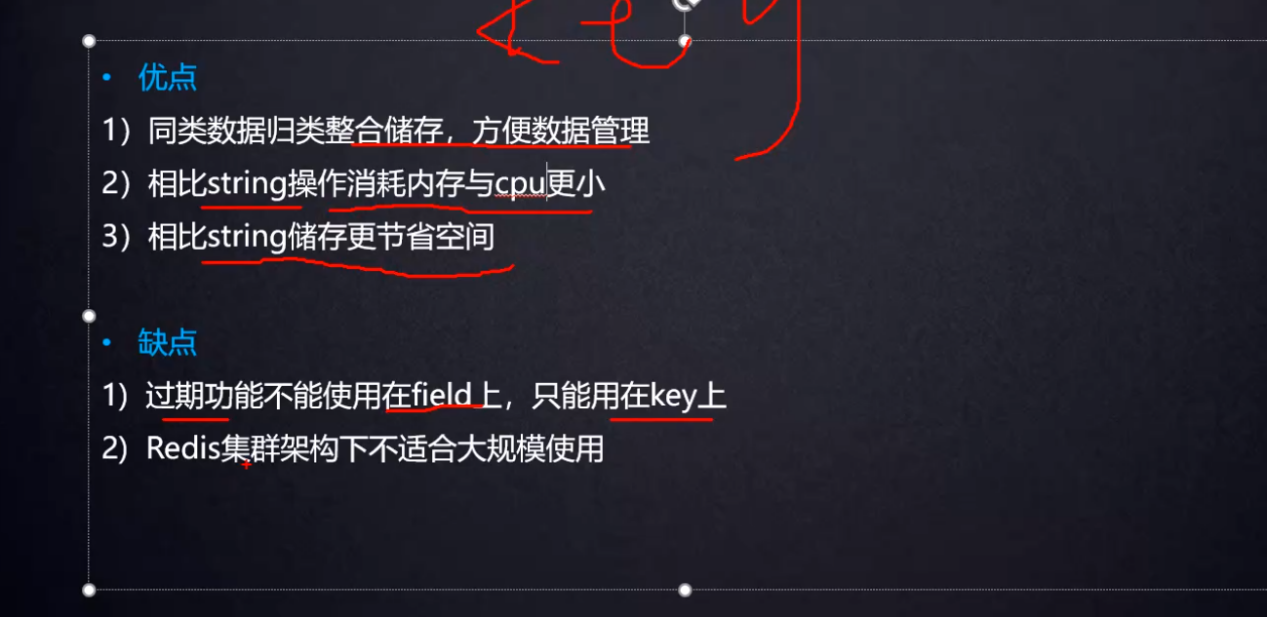
Hashes类型：键值对个数最多为2^32-1个，也就是4294967295个。

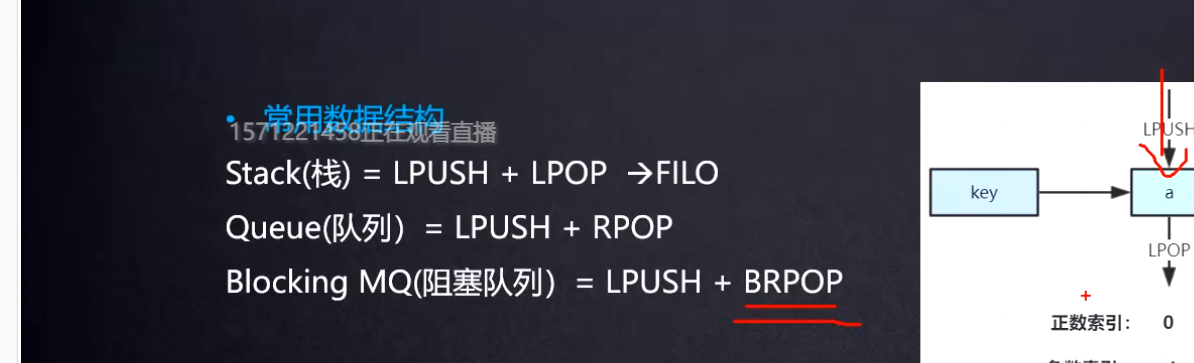
Sorted sets类型：跟Sets类型相似。

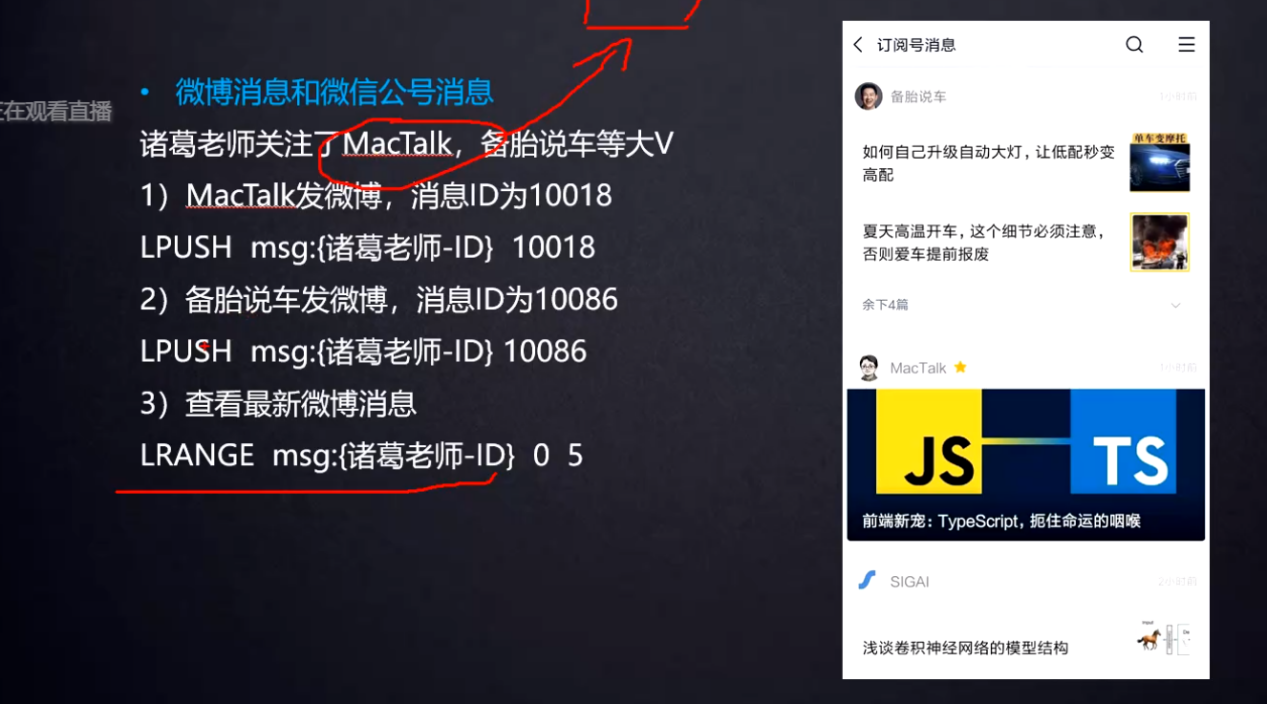


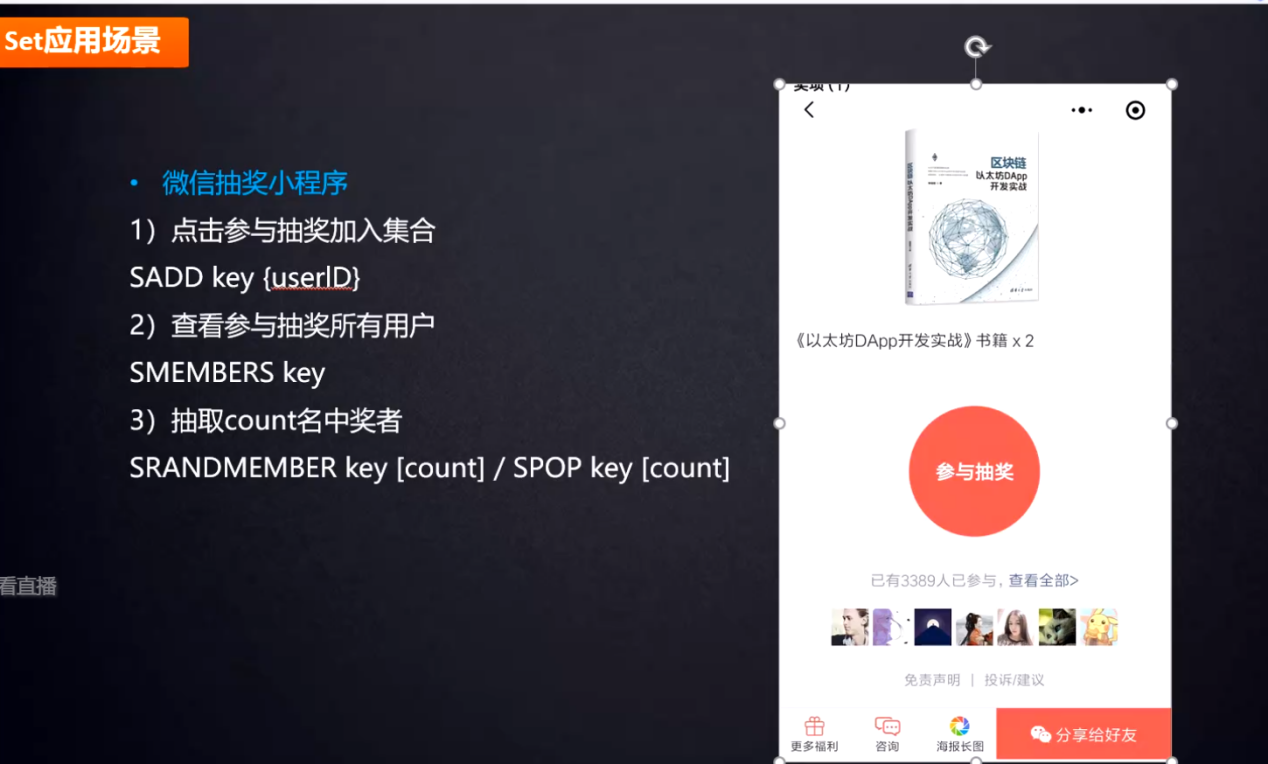
## Hash

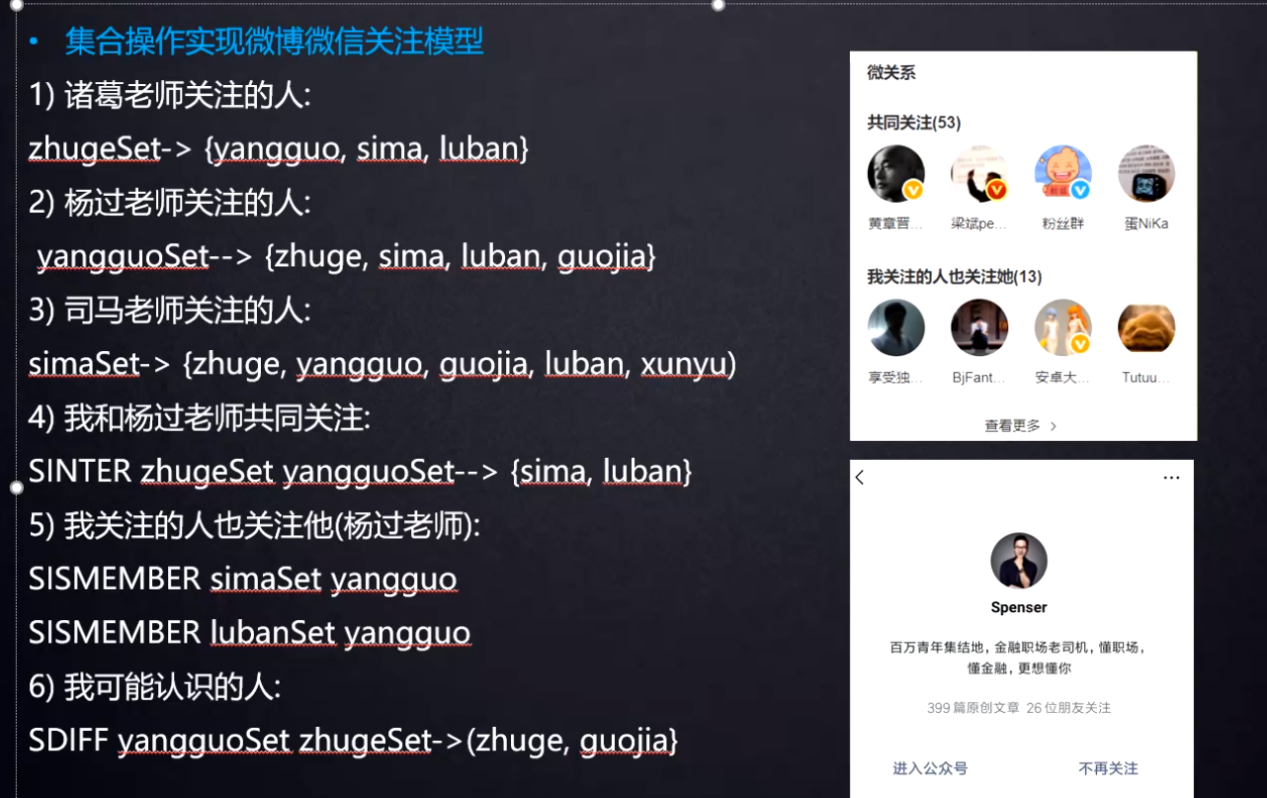


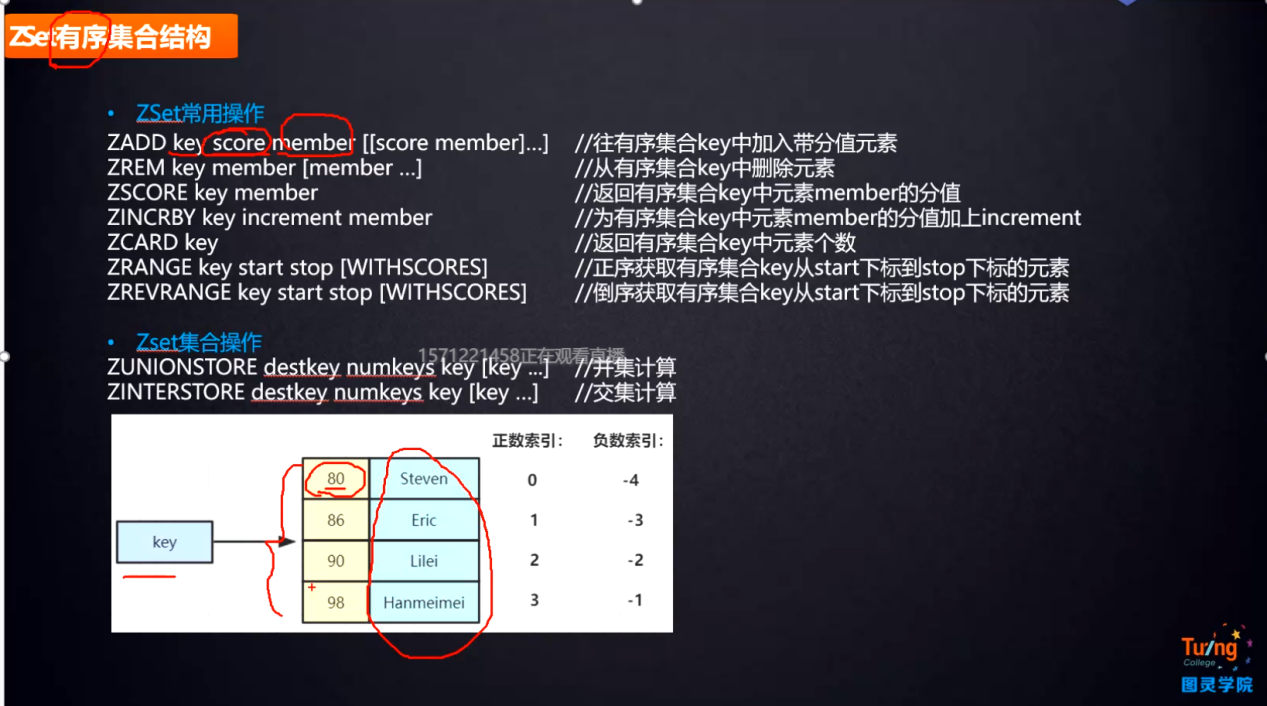






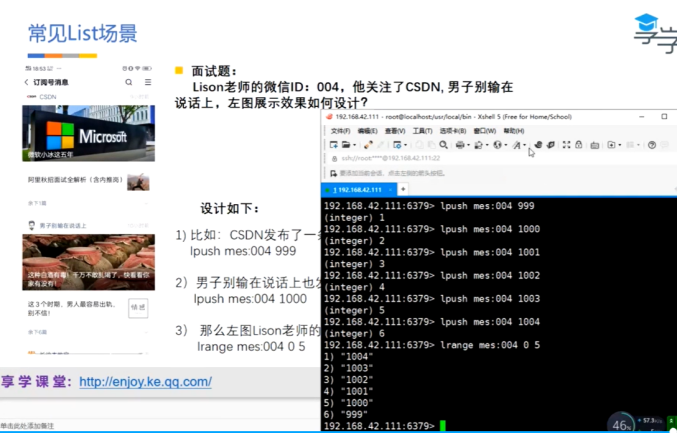






## LIst

使用场景：消息推送



## Set

抽奖:**srandmember**

## Zset

可以做热榜

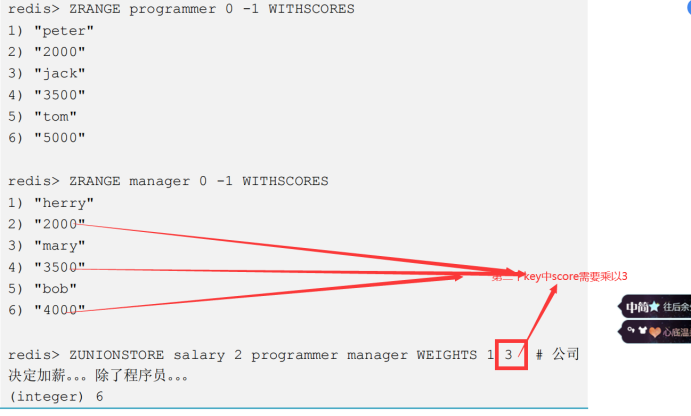
### ZUNIONSTORE

可以用来将两个key中相同的元素的score相加

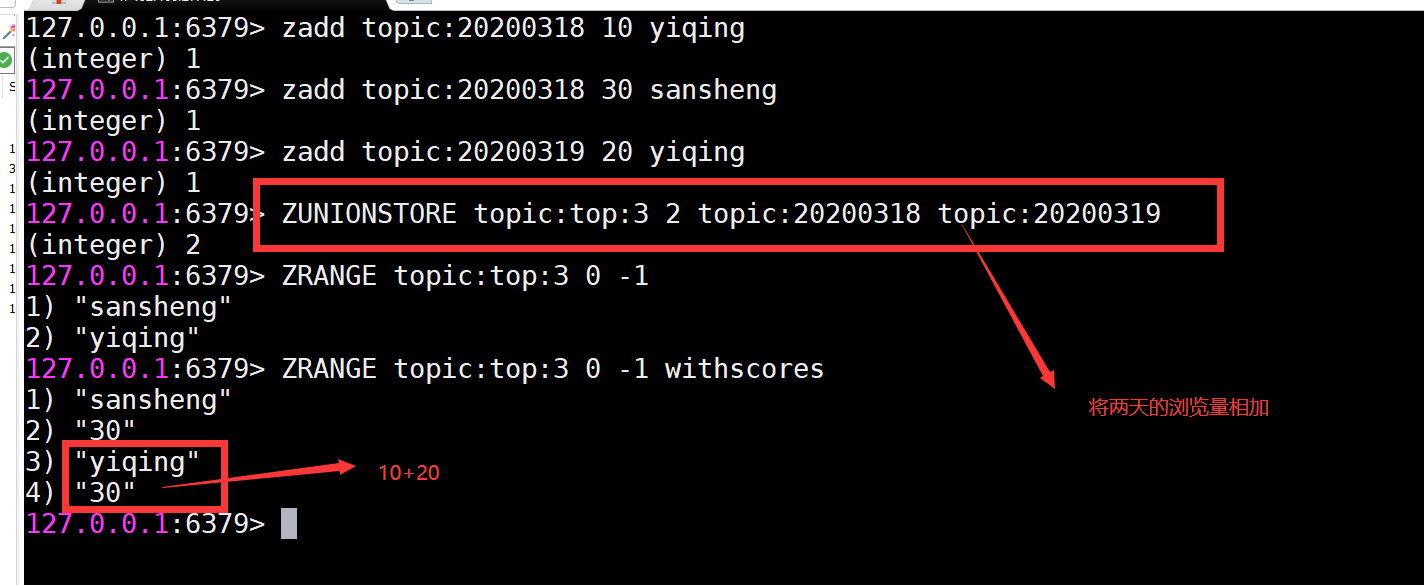
比如keya：1 one

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

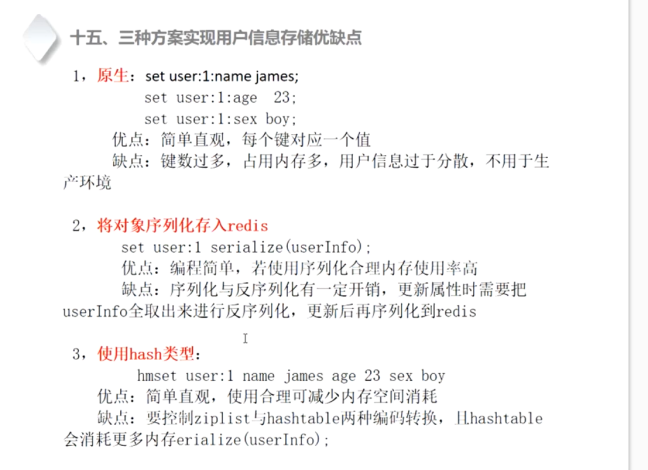


# 特性

速度快:数据存储到内存中，多路复用



# 数据存储的优缺点



# Redis持久化

## rdb

## AOF

# Lua

## 安装教程

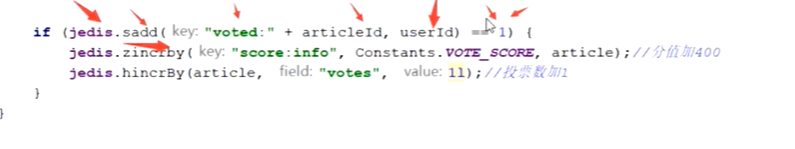
1. 到http://www.lua.org/ftp/下载包
2. 解压
3. yum install libtermcap-devel ncurses-devel libevent-devel readline-devel
4. Make linux test
5. Make install

# 应用

## 投票

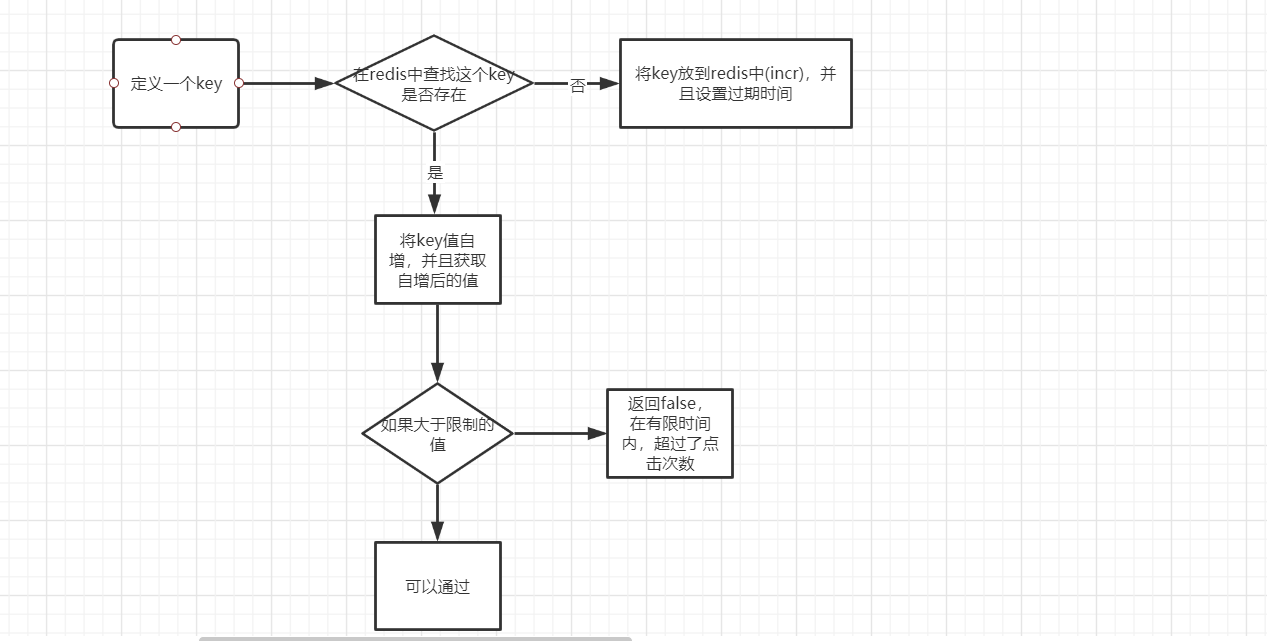
给文章投票

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

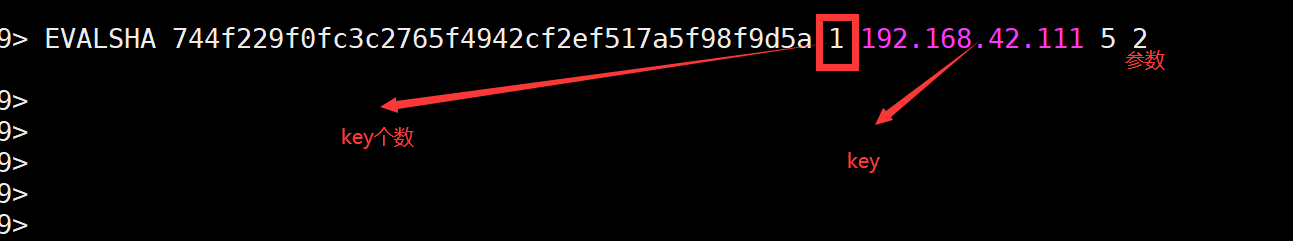
## 流程图



## 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 |

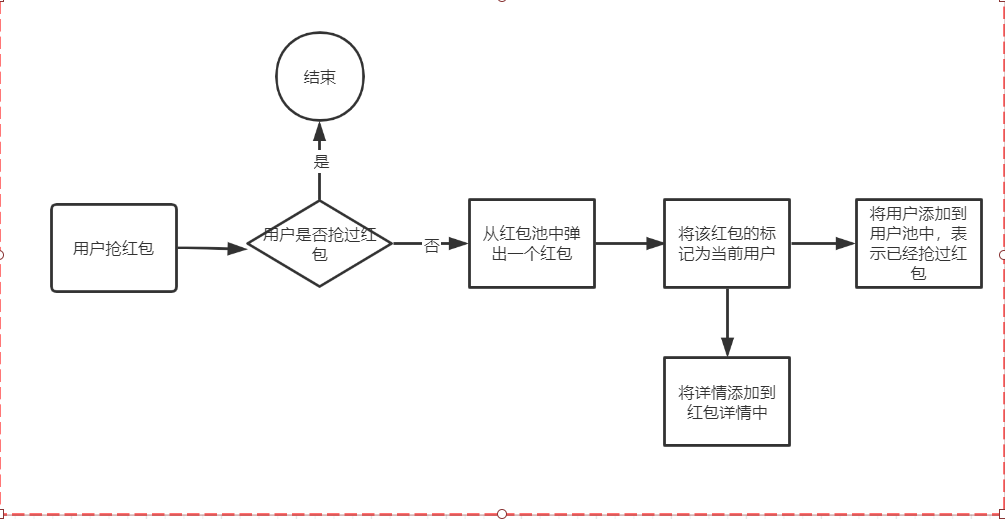
执行:EVALSHA 744f229f0fc3c2765f4942cf2ef517a5f98f9d5a 1 192.168.42.111 5 2



# 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

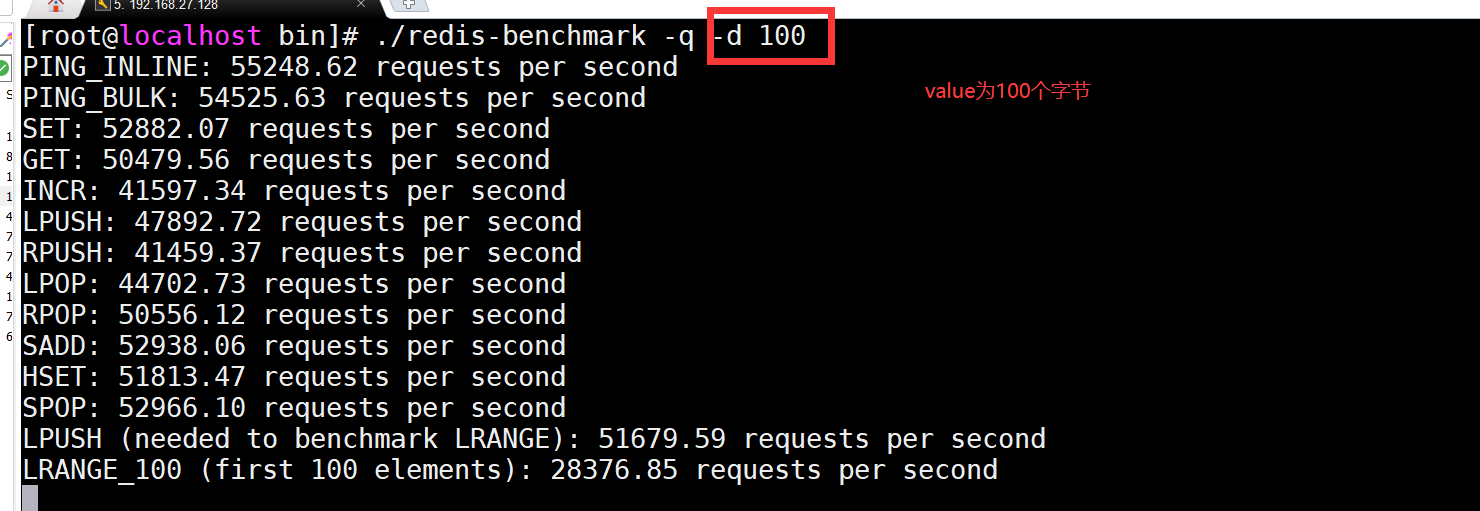
Jedis连接池默认的最大连接数为8个，用完之后记得关闭

# 测试性能

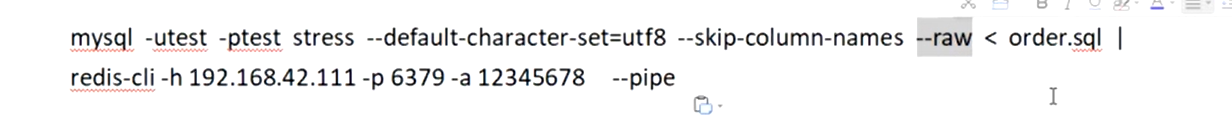
./redis-benchmark -c 100 -n 10000

100个并发连接 10000个请求

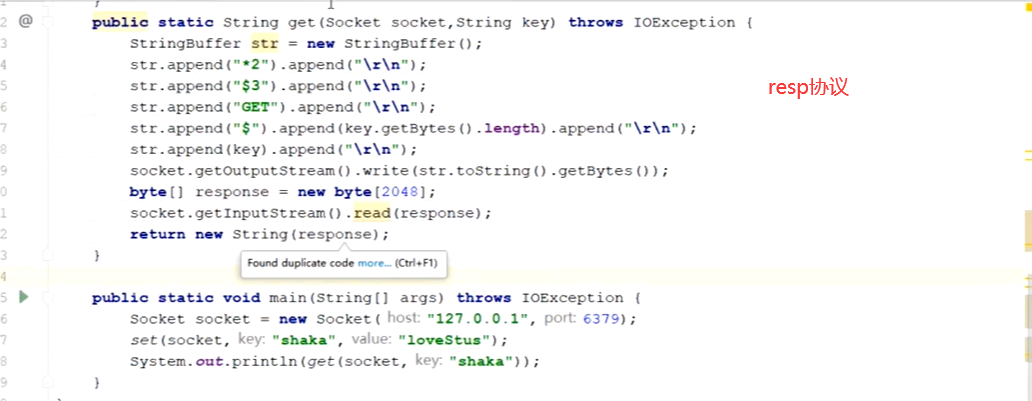
Value为100个字节的测试



# 从mysql往redis导入数据

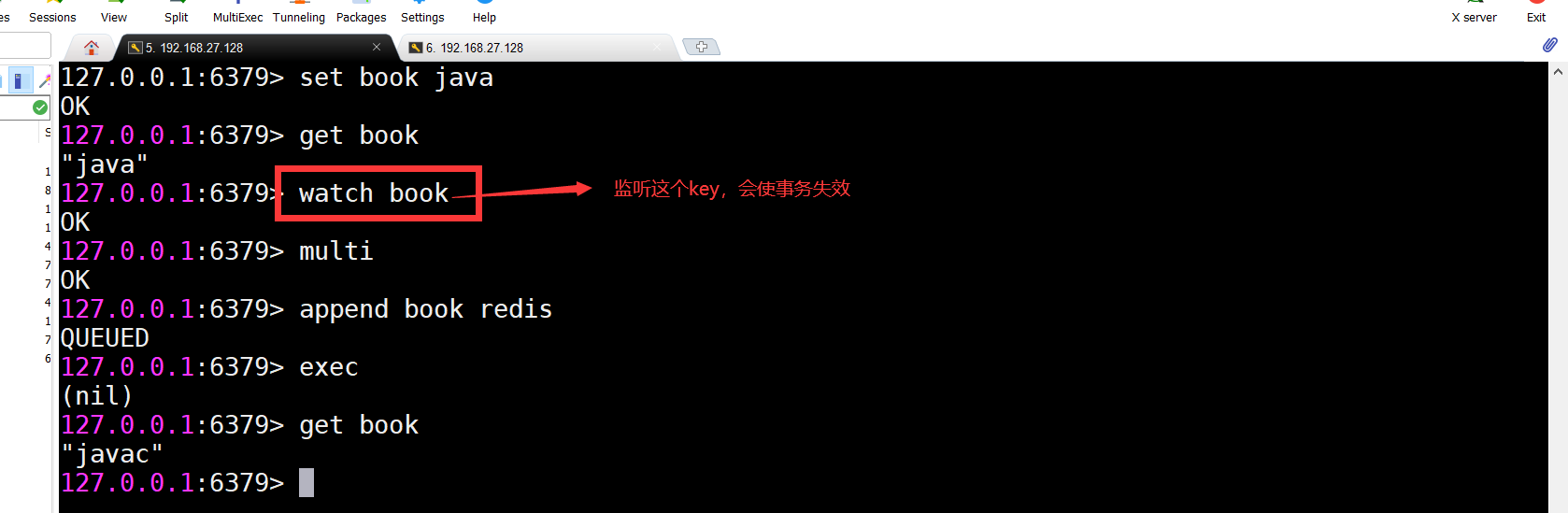


# 手写jedis

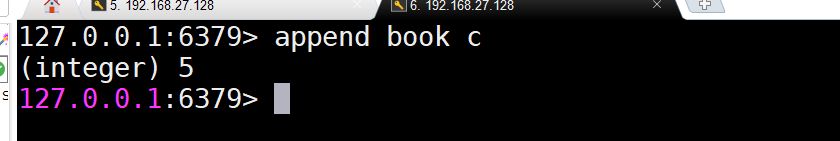


# Watch机制

会话一：监听完key之后，就去第二个会话追加



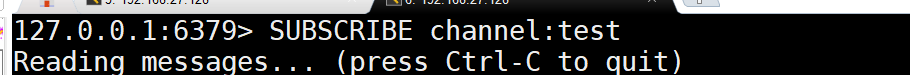
会话二



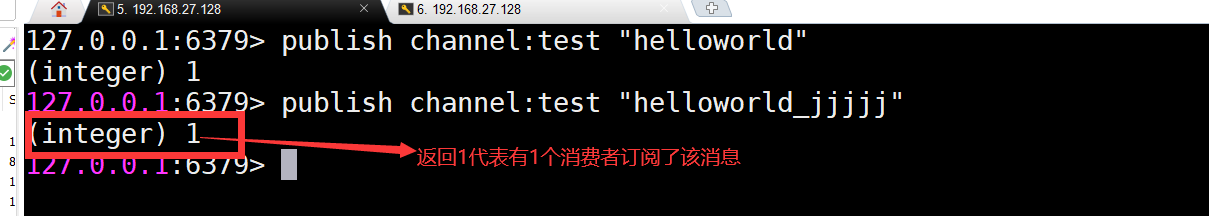
在第一个会话中查询book，发现是javac，在事务中追加的redis并没有追加进去，这是因为监听机制会使事务失效，所以没有追加进去

# 发布订阅

订阅者订阅消息

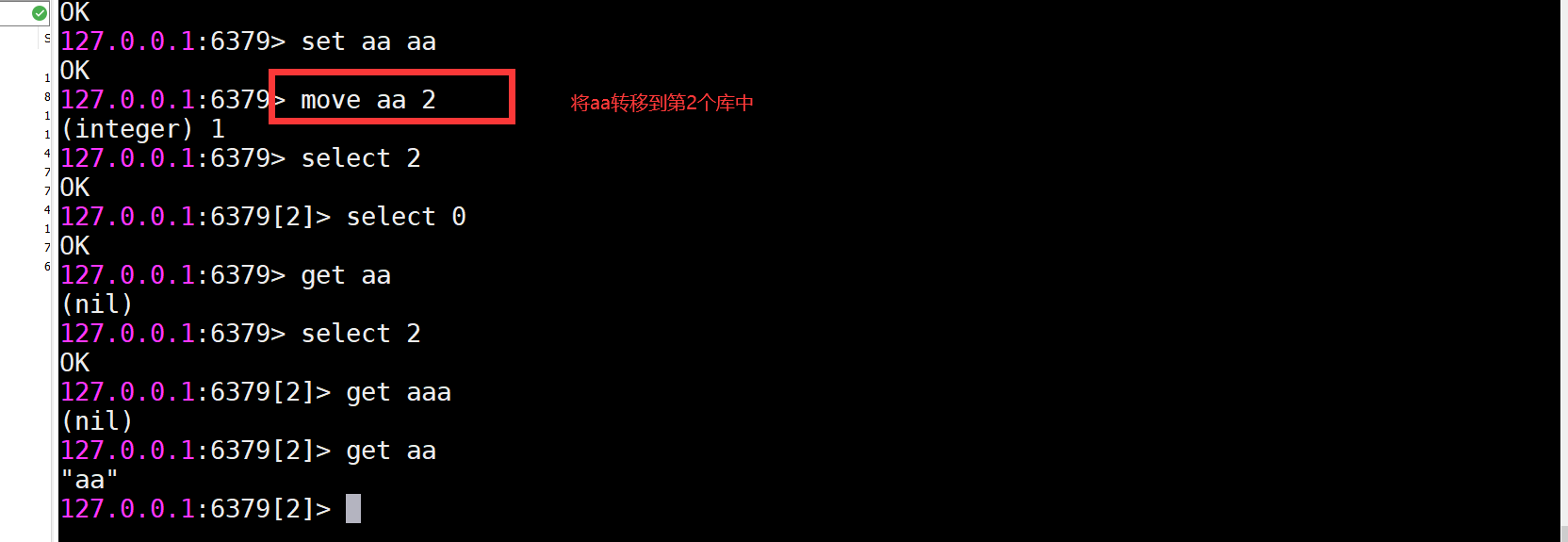


发布者发布消息

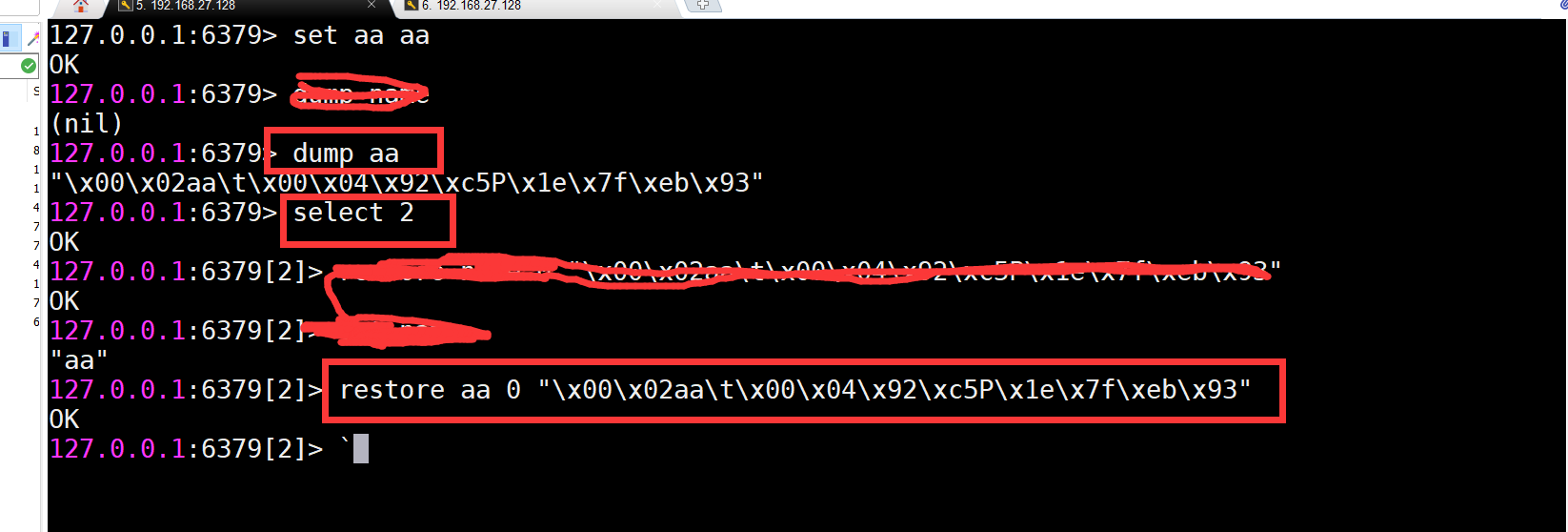


# 键的转移

1、Move key db



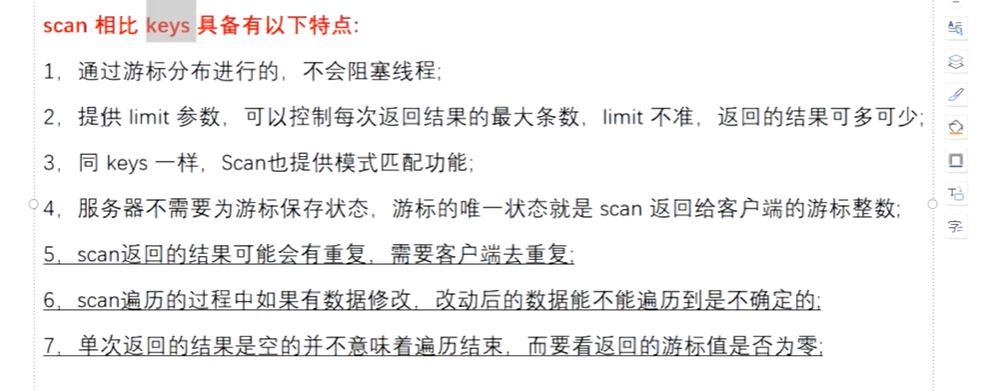
1. dump key



1. migrate



# 遍历key



# Redis高可用

## 主从拓扑

1. 一主多从

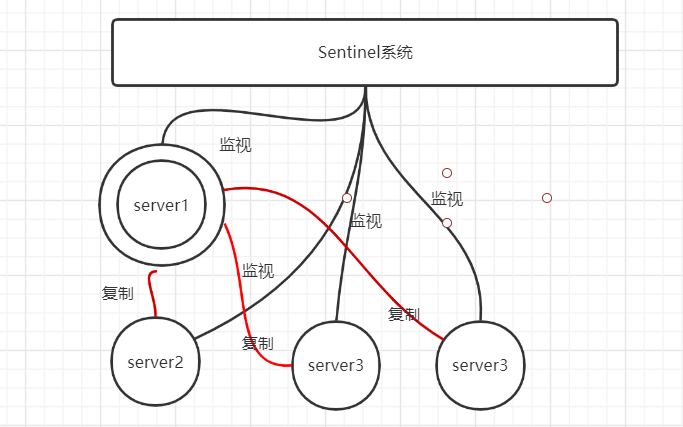
针对”读”较多的情景，”读”由较多个节点来分担，但节点越多，主节点同步到从节点的次数也越多，影响带宽，也影响主节点的稳定性。

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

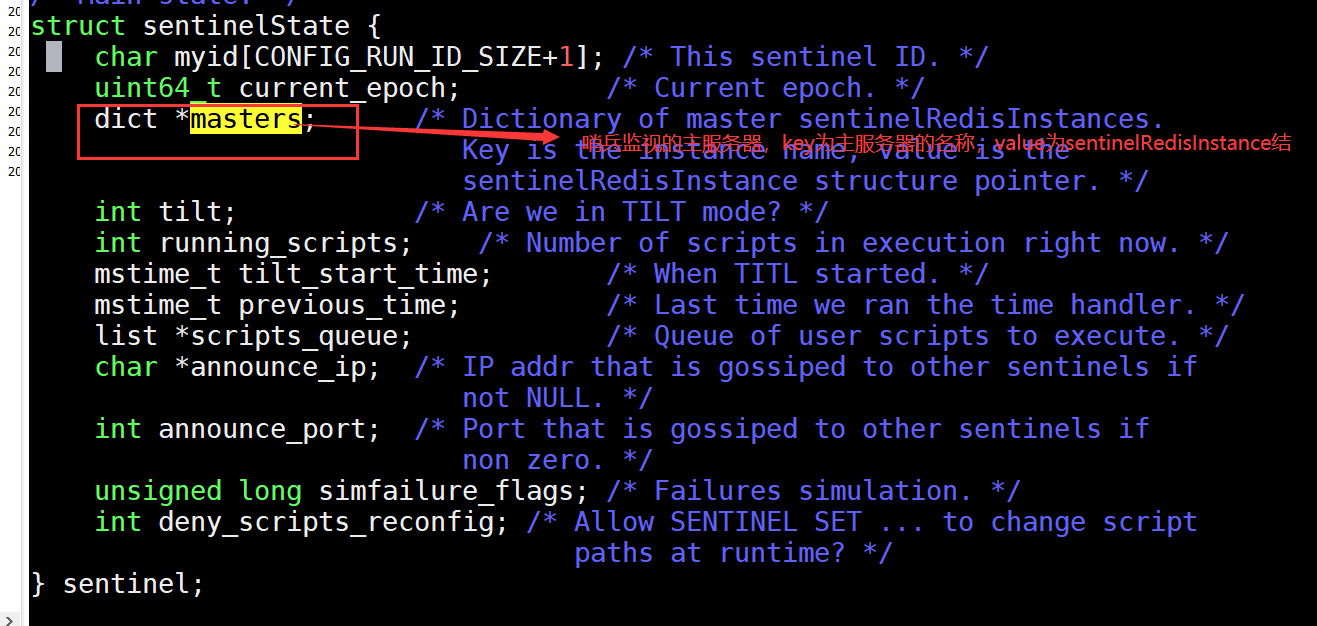
1. 可以监视任意多个主服务器，以及这些主服务器下的从服务器
2. 自动故障转移: 在主服务器下线时，自动将主服务器下的从服务升级为新的主服务器，然后由新的主服务器代替已下线的主服务器继续处理命令请求。

### 重要的属性

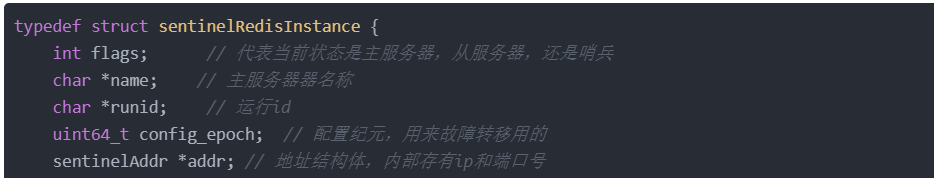
#### quorm

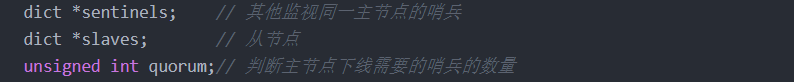
判断主服务器客观下线所需的支持投票数量

### 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。**

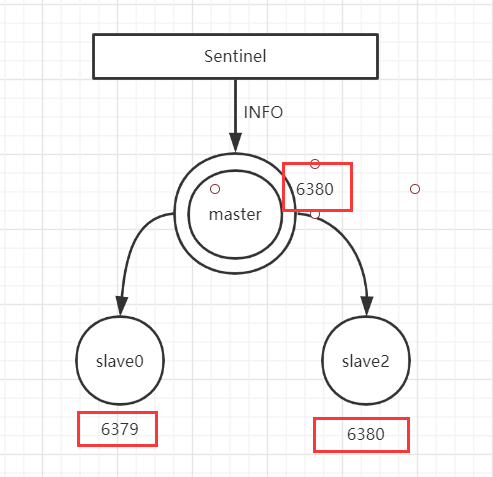
**另一方面，除了订阅频道之外，Sentinel还必须向主服务器发送命令，以此来与主服务器进行通信，所以Sentinel还必须向主服务器发送命令，依次来与主服务器进行通信，所以Sentinel还必须向主服务器创建命令连接。**

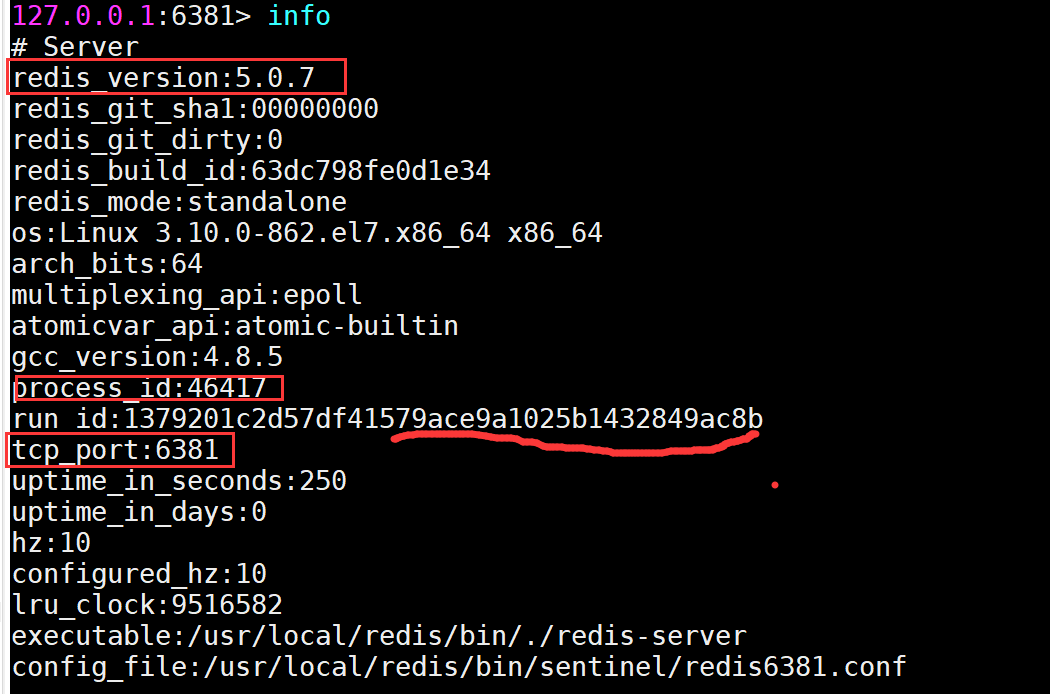
**因为Sentinel需要与多个实例创建多个网络连接，所以Sentinel使用的是异步连接。**

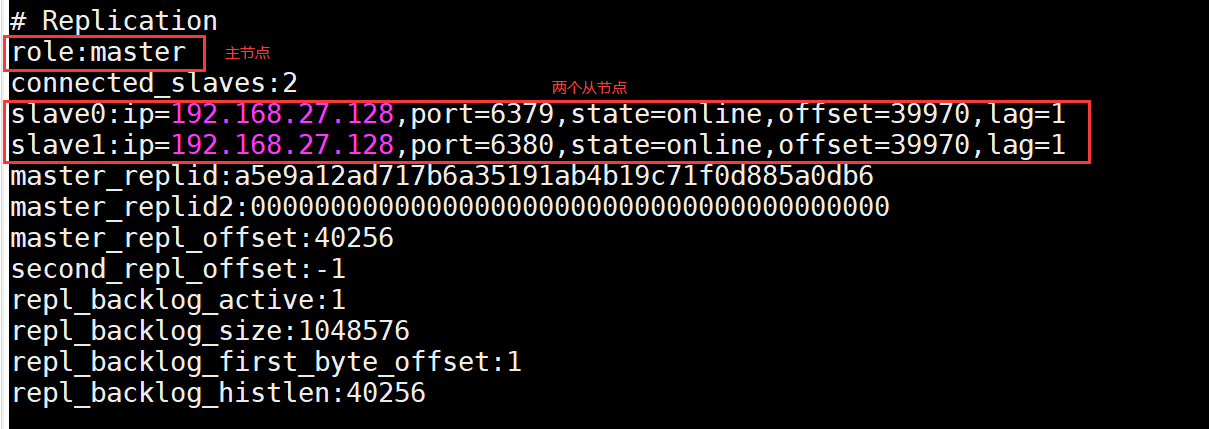
#### 获取主服务器信息

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:

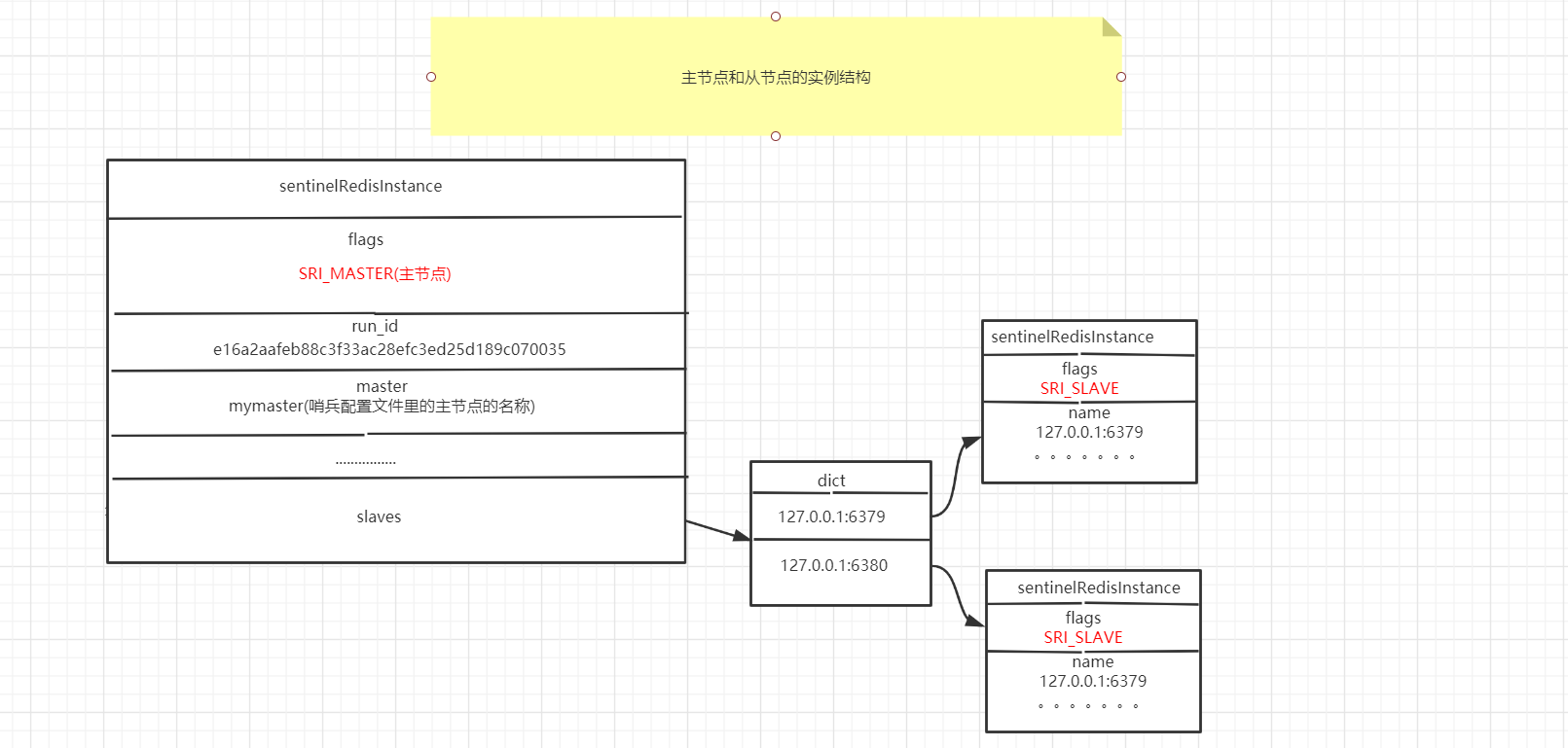
* 一方面是关于主服务器的信息，包括run\_id域记录的服务器运行id，以及role域记录的服务器角色。
* 另一方面是关于主服务器下的从服务器的信息，每个从服务器都由一个“slave”开头的行记录，每行的ip=域记录了从服务器的IP地址，而port=域则记录了从服务器的端口号。根据这些IP地址和端口号，Sentinel无需用户提供从服务器的地址信息，就可以自动发现从服务器。

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



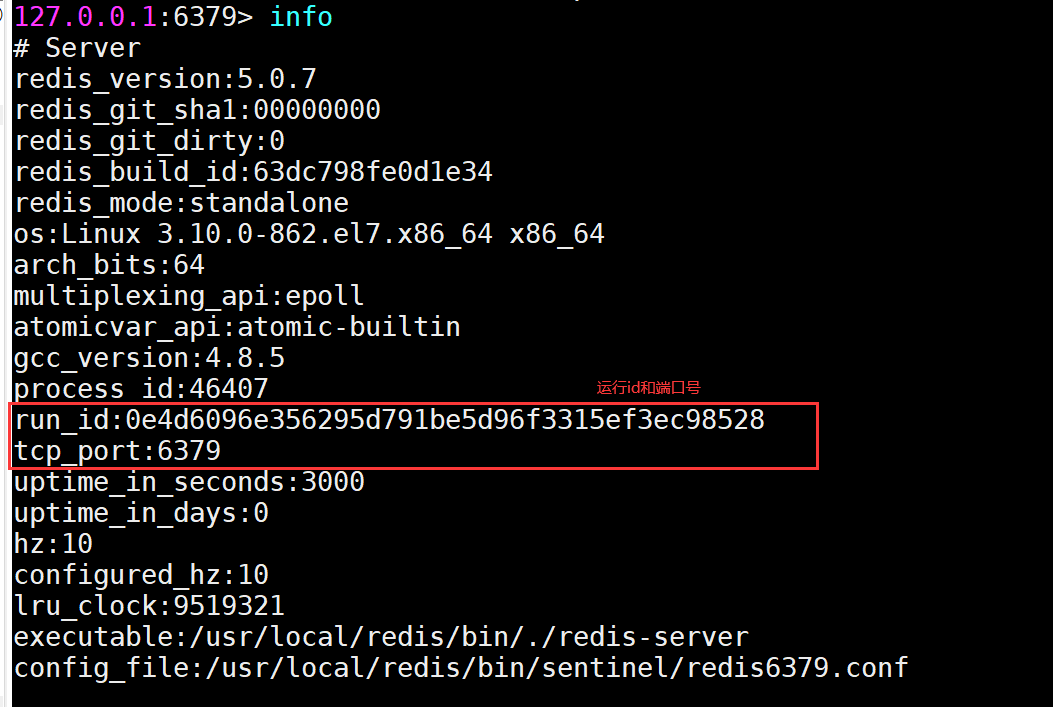
主节点和从节点的实例结构：

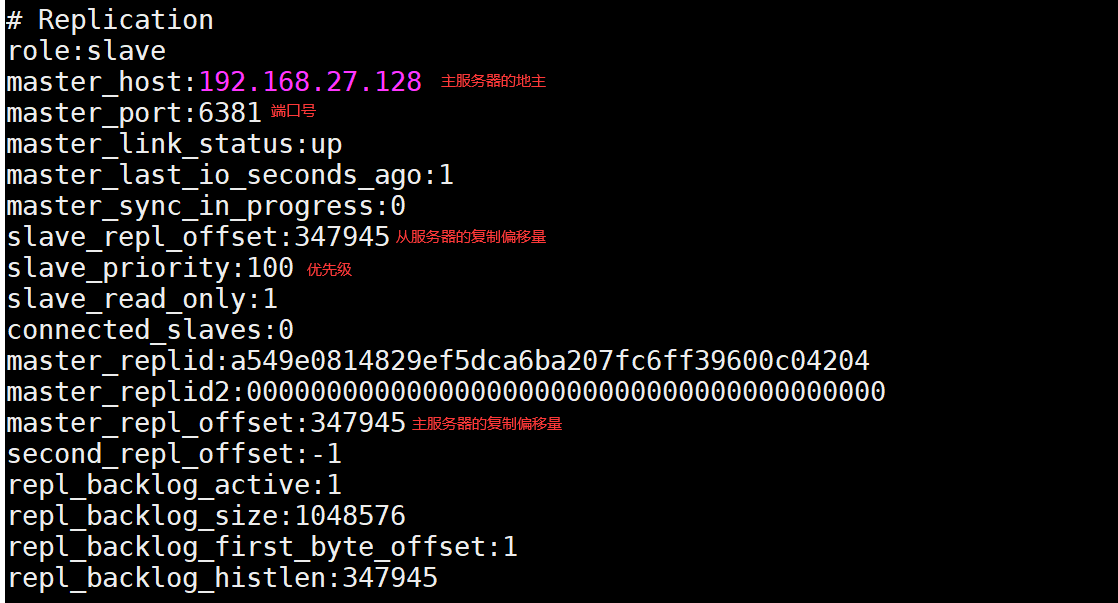


#### 获取从服务器信息

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:

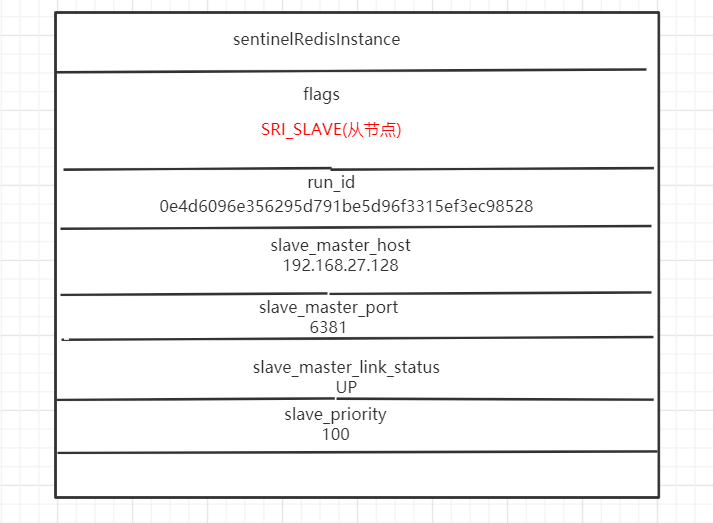




根据INFO命令的回复，Sentinel会提取出以下信息：

* 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。

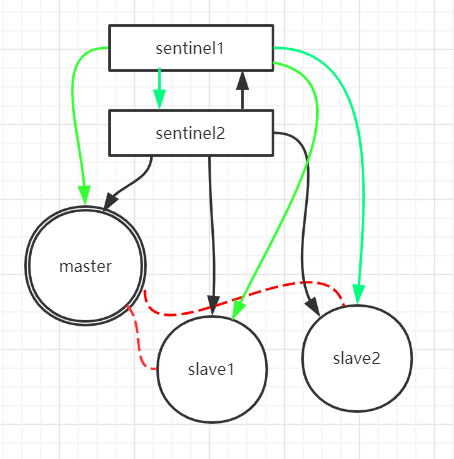
使用命令连接项相连的Sentinel可以通过向其他Sentinel发送命令请求来进行信息交换，Sentinel实现主观下线检测和客观下线检测都会使用Sentinel之间的命令连接。

##### 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。



# 分布式的解决方案

## 集群

### 缺陷

