

# 什么是Redis

Redis是一个使用C语言写的开源的高性能KV数据库,全称 Remote Dictionary Server(远程数据服务)。他支持丰富的数据类型,如 String, List, Set, ZSet(sorted set), Hash。

## Redis的数据存在哪里

Redis是把数据保存到内存中的,但是他也会定期的把数据持久化磁盘中以防止数据丢失。

## Redis怎么做持久化,有什么区别

为了避免数据丢失, Redis提供了两种持久化的方案,分别是 AOF(Redis DataBase) 和 RDB(Append only File),如果你没有数据持久化的需求,也完全可以关闭RDB和AOF方式,这样的话, redis将变成一个纯内存数据库,就像memcache一样。

**RDB**:在指定的时间间隔对数据进行快照存储,也是默认的持久化方式,这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。

**AOF**:记录每次对服务器写的操作,当服务器重启的时候就会重新执行这些命令来恢复原始的数据,AOF命令以Redis协议追加保存每次写的操作到文件末尾.Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大。

### RDB和AOF可以同时使用吗?

RDB和AOF两种方式可以同时使用,在这种情况下,如果redis重启的话,则会优先采用AOF方式来进行数据恢复,这是因为AOF方式的数据恢复完整度更高。

## 工作原理

### RDB

可以通过配置设置自动做快照持久化的方式。我们可以配置redis在n秒内如果超过m个key被修改就自动做快照,下面是默认的快照保存配置

```
save 900 1    #900秒内如果超过1个key被修改,则发起快照保存
save 300 10   #300秒内容如果超过10个key被修改,则发起快照保存
save 60 10000 #60秒内如果超果10000个key被修改,则发起快照保存
```

复制

### 优势

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。

### 劣势

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

Redis会将每一个收到的**写命令**都通过**write**函数追加到文件中(默认是 `appendonly.aof`)

当redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。当然由于os会在内核中缓存 write做的修改，所以可能不是立即写到磁盘上。这样aof方式的持久化也还是有可能丢失部分修改。不过我们可以通过配置文件告诉redis我们想要通过fsync函数强制os写入到磁盘的时机。有三种方式如下（默认是：每秒fsync一次）

```
appendonly yes           //启用aof持久化方式
# appendfsync always      //每次收到写命令就立即强制写入磁盘，最慢的，但是保证完全的持久化，不推荐使用
appendfsync everysec     //每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐
# appendfsync no         //完全依赖os，性能最好，持久化没保证
```

## 复制

aof 的方式也同时带来了另一个问题。持久化文件会变的越来越大。例如我们调用incr test命令100次，文件中必须保存全部的100条命令，其实有99条都是多余的。因为要恢复数据库的状态其实文件中保存一条set test 100就够了。

为了压缩aof的持久化文件。redis提供了 `bgrewriteaof` 命令。收到此命令redis将使用与快照类似的方式将内存中的数据以命令的方式保存到临时文件中，最后替换原来的文件。

## 优势

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种同步策略，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。

## 劣势

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。

## Redis是单线程的还是多线程的

Redis是单进程单线程的,Redis利用队列技术将并发访问变为串行访问,消除了传统数据库串行控制的开销。

## Redis为什么是单线程的?

多线程处理会涉及到锁，而且多线程处理会涉及到线程切换而消耗CPU。因为CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存或者网络带宽。单线程无法发挥多核CPU性能，不过可以通过在单机开多个Redis实例来解决。

## Redis是单线程的为什么还那么快

1. 完全基于内存,绝大部分请求都是纯粹的内存操作,非常快速.因为他的类型是KV,类似于HashMap,HashMap的优势就是查找和操作的时间复杂度都是O(1).
2. 数据结构简单,对数据的操作也简单,Redis中的数据结构是专门进行设计的.
3. 采用了单线程,避免了不需要的上下文切换和竞争条件,也不存在因为多进程和多线程导致的切换而消耗CPU的问题,不用去考虑各种锁的问题,不存在加锁释放锁操作,没有因为可能出现死锁而导致的性能消耗.
4. 使用多路I/O复用模型,非阻塞I/O.  
多路I/O复用模型是利用select,poll,epoll可以同时监察多个流的I/O事件的能力,在空闲的时候,会把当前线程阻塞掉,当有一个或多个流有I/O事件,就从阻塞中唤醒,于是程序就会轮训一遍所有的流(epoll只轮训那些真正发出了事件的流),并且只依次顺序的处理就绪的流,这种做法就避免了大量的无用操作.  
这里“多路”指的是多个网络连接,“复用”指的是复用同一个内核  
采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络 IO 的时间消耗），且 Redis 在内存中操作数据的速度非常快，也就是说内存内的操作不会成为影响Redis性能的瓶颈，主要由以上几点造就了 Redis 具有很高的吞吐量.
5. 使用底层模型不同,他们之间底层实现方式以及与客户端之间通信的应用协议不一样,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.

## 在项目中哪里使用到了Redis,解决了什么问题

### 缓存/并发

#### 缓存

1. 读取前，先去读Redis,如果没有数据,读取数据库,将数据拉入Redis。
2. 插入数据时，同时写入Redis。

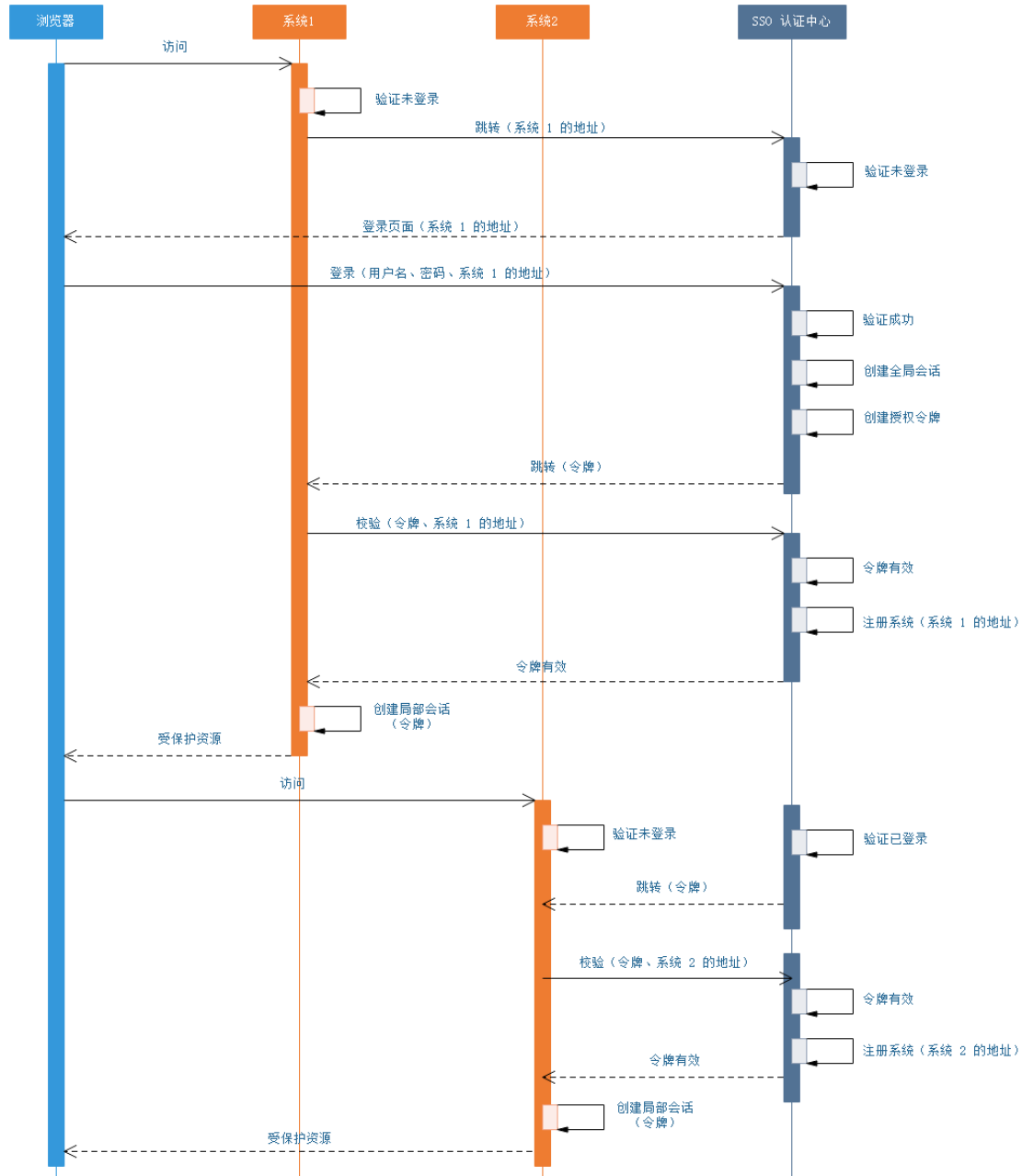
解决访问速度慢的问题,从原来访问页面需要几秒甚至几十秒提升到现在的毫秒级

#### SSO

分布式系统是由多个系统组成的应用群,面对这么多系统,难道要让用户一个一个登录,一个一个注销吗,这显然是不可能的.在分布式系统中,复杂性应该由系统内部承担,而不是用户,无论我系统内部有多么复杂,对用户而言,都是一个统一的整体,也就是说,用户在访问我这个应用群的时候,应该像访问单个系统一样,登录/注销只要一次就够了.

基于这个理念,就要使用到 **单点登录(sso)**,那么就要考虑到如何实现信息共享,因为http是无状态的,他根本不知道客户端的那个人是谁,那如何记录呢,我们想到了 **session**,后台设置sso服务,只要用户第一次成功,就把这个session存到sso服务中心来记录这个登录状态,用户只要访问服务端,会先去sso服务中心认证是否登录,也就是去Redis里查看有没有这个session,如果登录过了,直接跳转到访问的页面,如果没有登

录过,则踢回登录页面进行登录



[SSO流程](#)

## 验证码自动过期

Redis针对数据都可以设置过期时间，过期的数据清理无需使用方去关注，所以开发效率也比较高。妈妈再也不用担心验证码验证过后还需要再去手动删除过期数据啦

## 秒杀

问题:

并发太高导致程序阻塞

库存无法有效控制,出现超卖的情况

解决方案:

1. 数据尽量缓存,阻断用户和数据库的直接交互。
2. 通过锁来控制避免超卖现象

秒杀流程:

1. 提前预热数据，放入Redis
2. 商品列表放入Redis List
3. 商品的详情数据 Redis hash保存，设置过期时间
4. 商品的库存数据Redis sorted set保存
5. 用户的地址信息Redis set保存
6. 订单产生后发货的数据，产生Redis list，通过消息队列处理
7. 秒杀结束后，再把Redis数据和数据库进行同步

## Redis常用的数据类型有哪些

常用的类型有 String, Hash, List, Set, Sorted set

String是最常用的数据类型,普通的KV存储都可以用它

Hash呢我们一般用来存储用户的姓名,年龄等信息,我们把 用户ID 作为K,把 信息标签 作为值的field,把 信息值 作为值的value

## Redis高级数据类型有哪些

Bitmaps`, `Hyperloglogs`, `GEO

### Bitmaps

Bitmaps不是一个真实的数据结构。而是String类型上的一组面向bit操作的集合。由于strings是二进制安全的blob，并且它们的最大长度是512m，所以bitmaps能最大设置 $2^{32}$ 个不同的bit。

### HyperLogLogs

HyperLogLog用于计算唯一事物的概率数据结构（从技术上讲，这被称为估计集合的基数）。如果统计唯一项，项目越多，需要的内存就越多。因为需要记住过去已经看过的项，从而避免多次统计这些项。

### GEO

Redis的GEO特性在 Redis3.2版本中推出，这个功能可以将用户给定的地理位置（经度和纬度）信息储存起来，并对这些信息进行操作。

## Redis中String的实现原理

Redis底层是用C写的,对于String并没有直接使用C的字符数组(C的String原理和Java的原理一样),而是自己封装了一个 sds 的类型.

每个 sds.h/sdshdr 结构表示一个 SDS 值：

```
struct sdshdr {  
    // 记录 buf 数组中已使用字节的数量  
    // 等于 SDS 所保存字符串的长度  
    int len;  
  
    // 记录 buf 数组中未使用字节的数量  
    int free;  
  
    // 字节数组，用于保存字符串  
    char buf[];  
};
```

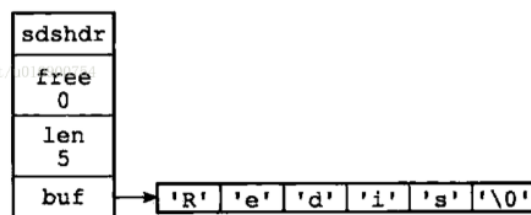



图 2-1 SDS 示例

[sds](#)

buf数组属于真正的字符. Redis新建数据类型,是为了抽象,使得编程更加简单.原有的C的字符串的api是不安全的,因为在使用字符数组以后,需要跟踪内存的分配.在使用之前,需要预分配空间,否则会有缓冲区溢出.用完一个字符串需要回收,否则会有内存泄漏.(因为C没有内存自动回收机制)

为了使编程时尽可能少的考虑这些问题,就要把分配内存的事情封起来,不让调用方得知,这就是的作用.每一次创建一个新的String时,不会按照实际大小分配,而是更多的内存,当字符串截断时,也不会立刻回收内存,而是,增加.字符串插入时,先看够不够,不够再分配.这样减少了内存分配回收的次数,效率提高.对应的策略叫做:和

1564013848901

使用了sds数据类型,带来了三点好处:

1. 获取长度变成了 $O(1)$ 的操作
2. 杜绝了缓冲区溢出
3. 减少了字符串改变造成的空间分配次数

## Redis常见的应用

[常见应用,不写了,大部分都差不多,直接上链接](#)

[上面也有自己写的一些](#)

<https://zhuanlan.zhihu.com/p/29665317>

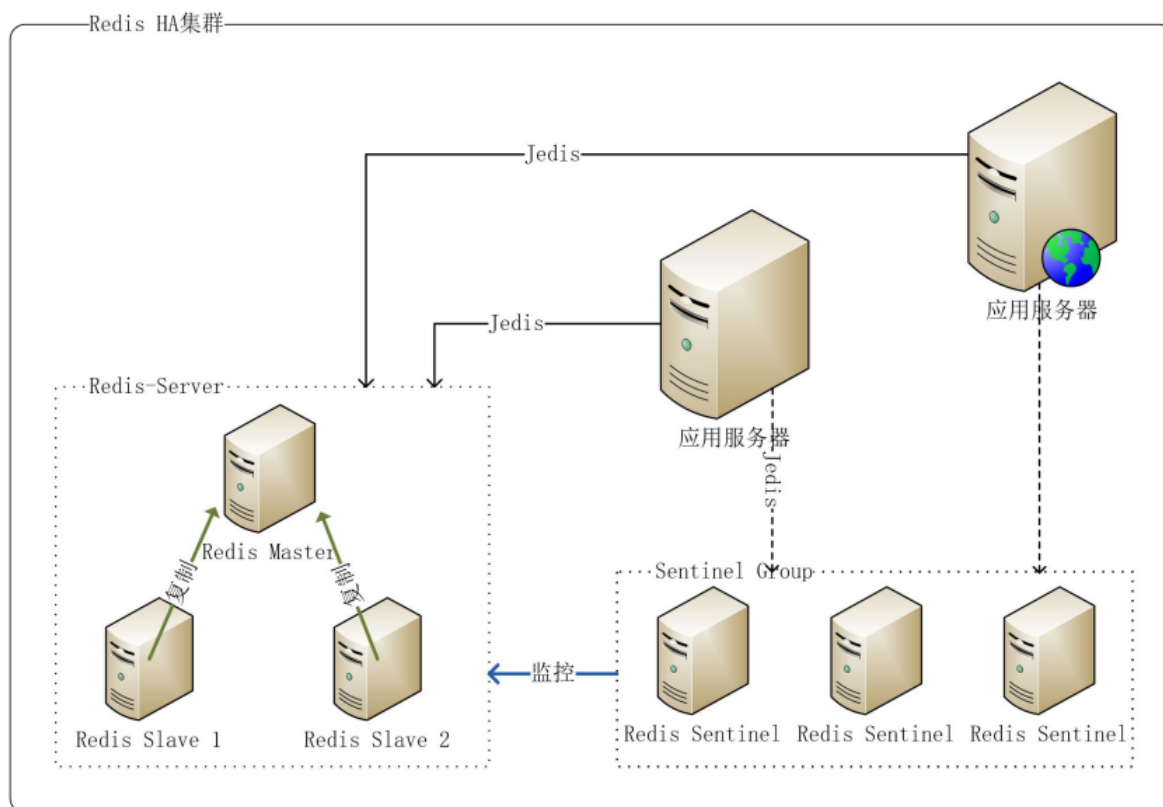
## Redis集群的搭建方式

首先要知道为什么要集群,其实很简单,就是为了实现基础设施的高可用HA(High Available),一般有两个或两个以上的节点,且分为活动节点及备用节点。通常把正在执行业务的称为活动节点,而作为活动节点的一个备份的则称为备用节点。当活动节点出现问题,导致正在运行的业务(任务)不能正常运行时,备用节点此时就会侦测到,并立即接续活动节点来执行业务。从而实现业务的不中断或短暂中断。

Redis 一般以主/从方式部署

- **Keepalived** 通过 keepalived 的虚拟 IP, 提供主从的统一访问, 在主出现问题时, 通过 keepalived 运行脚本将从提升为主, 待主恢复后先同步后自动变为主, 该方案的好处是主从切换后, 应用程序不需要知道(因为访问的虚拟 IP 不变), 坏处是引入 keepalived 增加部署复杂性, 在有些情况下会导致数据丢失
- **Zookeeper** 通过 zookeeper 来监控主从实例, 维护最新有效的 IP, 应用通过 zookeeper 取得 IP, 对 Redis 进行访问, 该方案需要编写大量的监控代码
- **Sentinel(Redis 哨兵模式)** 通过 Sentinel 监控主从实例, 自动进行故障恢复, 该方案有个缺陷: 因为主从实例地址(IP & PORT)是不同的, 当故障发生进行主从切换后, 应用程序无法知道新地址, 故在 Jedis2.2.2 中新增了对 Sentinel 的支持, 应用通过 `redis.clients.jedis.JedisSentinelPool.getResource()` 取得的 Jedis 实例会及时更新到新的主实例地址,同时Sentinel也是Redis官方推荐的HA解决方案.
- **Redis Cluster** 直接修改redis配置文件即可搭建集群,这也是官方的集群方案[点这里查看文档](#)





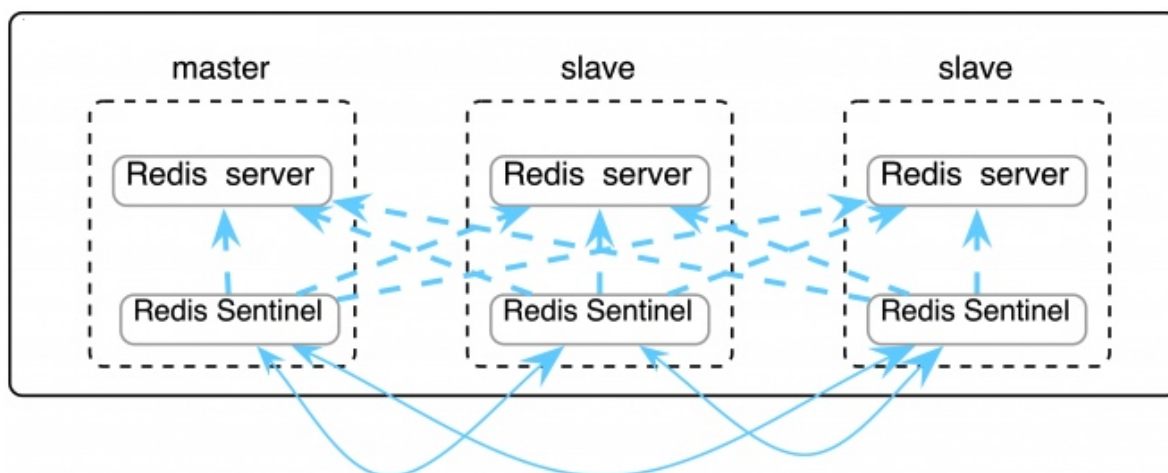
## Redis HA

Sentinel 是解决 HA 问题的，cluster 是解决主从复制问题的，不重复，并且经常一起用

## 基于Docker搭建

Redis 集群可以在一组 redis 节点之间实现高可用性和 sharding。在集群中会有 1 个 master 和多个 slave 节点。当 master 节点失效时，应选举出一个 slave 节点作为新的 master。然而 Redis 本身(包括它的很多客户端)没有实现自动故障发现并进行主备切换的能力，需要外部的监控方案来实现自动故障恢复。

Redis Sentinel 是官方推荐的高可用性解决方案。它是 Redis 集群的监控管理工具，可以提供节点监控、通知、自动故障恢复和客户端配置发现服务。




## HA

这里我是基于[Docker-compose](#)搭建Redis Sentinel,不会 Docker 的同学还是先去了解一下Docker吧,这里是[官方链接](#)


## 搭建Redis Cluster

搭建一主两从环境, docker-compose.yml 内容如下:

1564013608770

## 搭建Redis Sentinel集群

我们至少需要创建三个 Sentinel 服务, docker-compose.yml 配置如下:

1564013637858

(肯定会有人问,你这看起来是在一台机器上部署的呀,不是应该分机部署吗?那还用说嘛,肯定是因为穷啦,但幸亏我们是docker,它有很好的隔离机制,这样就相当于在三台机器上部署啦,到时候拆分一下就OK)

## 修改Sentinel配置文件

需要三份 sentinel.conf 配置文件, 分别为 sentinel1.conf, sentinel2.conf, sentinel3.conf, 配置文件内容相同(还是因为穷,如果是分机直接每台机器上都是sentinel.conf了)

```
port 26379
dir /tmp
# 自定义集群名, 其中 127.0.0.1 为 redis-master 的 ip, 6379 为 redis-master 的端口, 2
为最小投票数 (因为有 3 台 sentinel 所以可以设置成 2)
sentinel monitor mymaster 127.0.0.1 6379 2
sentinel down-after-milliseconds mymaster 30000
sentinel parallel-syncs mymaster 1sentinel
failover-timeout mymaster 180000
sentinel deny-scripts-reconfig yes
```

执行以下命令进行复制操作

```
cp sentinel.conf sentinel1.conf && cp sentinel.conf sentinel2.conf && cp
sentinel.conf sentinel3.conf
```

启动容器

```
docker-compose up -d
```

## 查看集群是否生效

进入 Sentinel 容器, 使用 Sentinel API 查看监控情况:

```
docker exec -it redis-sentinel-1 /bin/bash
redis-cli -p 26379
sentinel master mymaster
sentinel slaves mymaster
```

复制



```
1) "name"
2) "mymaster"
3) "ip"
4) "192.168.75.140"
5) "port"
6) "6379"
7) "runid"
8) "43a4faba8280316dac2cd83a59c6f73b0c504b8d"
9) "flags"
10) "master"
11) "link-pending-commands"
12) "0"
13) "link-refcount"
14) "1"
15) "last-ping-sent"
16) "0"
17) "last-ok-ping-reply"
18) "59"
19) "last-ping-reply"
20) "59"
21) "down-after-milliseconds"
22) "30000"
23) "info-refresh"
24) "8838"
25) "role-reported"
26) "master"
27) "role-reported-time"
28) "1805626"
29) "config-epoch"
30) "0"
31) "num-slaves"
32) "2"
33) "num-other-sentinels"
34) "2"
35) "quorum"
36) "2"
37) "failover-timeout"
38) "180000"
39) "parallel-syncs"
40) "1"
```

[sentinel](#)

至此 sentinel HA 方案搭建成功.

## Redis 为什么用跳表而不用平衡树?

(O\_O)?这个咱也不会,直接上链接吧,可能后期会懂个皮毛

<https://juejin.im/post/57fa935b0e3dd90057c50fbc>

## Redis常用命令有哪些

常用命令?下面这么多,随便选几个吧🤔🤔

### 连接操作相关的命令

- ping: 测试连接是否存活如果正常会返回 pong
- echo: 打印

- select: 切换到指定的数据库, 数据库索引号 index 用数字值指定, 以 0 作为起始索引值
- quit: 关闭连接 (connection)
- auth: 简单密码认证

## 服务端相关命令

- time: 返回当前服务器时间
- client list: 返回所有连接到服务器的客户端信息和统计数据
- client kill ip:port
- save: 将数据同步保存到磁盘
- bgsave: 将数据异步保存到磁盘
- lastsave: 返回上次成功将数据保存到磁盘的Unix时戳
- shutdown: 将数据同步保存到磁盘, 然后关闭服务
- info: 提供服务器的信息和统计
- config resetstat: 重置 info 命令中的某些统计数据
- config get: 获取配置文件信息
- config set: 动态地调整 Redis 服务器的配置(configuration)而无须重启, 可以修改的配置参数可以使用命令 CONFIG GET \* 来列出
- config rewrite: Redis 服务器时所指定的 redis.conf 文件进行改写
- monitor: 实时转储收到的请求
- slaveof: 改变复制策略设置

## 发布订阅相关命令

- psubscribe: 订阅一个或多个符合给定模式的频道 例如 psubscribe news.\* tweet.\*
- publish: 将信息 message 发送到指定的频道 channel 例如 publish msg "good morning"
- pubsub channels: 列出当前的活跃频道 例如 PUBSUB CHANNELS news.i\*
- pubsub numsub: 返回给定频道的订阅者数量 例如 PUBSUB NUMSUB news.it news.internet news.sport news.music
- pubsub numpat: 返回客户端订阅的所有模式的数量总和
- punsubscribe: 指示客户端退订所有给定模式。
- subscribe: 订阅给定的一个或多个频道的信息。例如 subscribe msg chat\_room
- unsubscribe: 指示客户端退订给定的频道。

## 对KEY操作的命令

- exists(key): 确认一个 key 是否存在
- del(key): 删除一个 key
- type(key): 返回值的类型
- keys(pattern): 返回满足给定 pattern 的所有 key
- randomkey: 随机返回 key 空间的一个
- rename(oldname, newname): 重命名 key
- dbsize: 返回当前数据库中 key 的数目
- expire: 设定一个 key 的活动时间 (s)
- ttl: 获得一个 key 的活动时间
- move(key, dbindex): 移动当前数据库中的 key 到 dbindex 数据库
- flushdb: 删除当前选择数据库中的所有 key
- flushall: 删除所有数据库中的所有 key

## 对String操作的命令

- set(key, value): 给数据库中名称为 key 的 string 赋予值 value
- get(key): 返回数据库中名称为 key 的 string 的 value
- getset(key, value): 给名称为 key 的 string 赋予上一次的 value

- mget(key1, key2,..., key N): 返回库中多个 string 的 value
- setnx(key, value): 添加 string, 名称为 key, 值为 value
- setex(key, time, value): 向库中添加 string, 设定过期时间 time
- mset(key N, value N): 批量设置多个 string 的值
- msetnx(key N, value N): 如果所有名称为 key i 的 string 都不存在
- incr(key): 名称为 key 的 string 增 1 操作
- incrby(key, integer): 名称为 key 的 string 增加 integer
- decr(key): 名称为 key 的 string 减 1 操作
- decrby(key, integer): 名称为 key 的 string 减少 integer
- append(key, value): 名称为 key 的 string 的值附加 value
- substr(key, start, end): 返回名称为 key 的 string 的 value 的子串

## 对List操作的命令

- rpush(key, value): 在名称为 key 的 list 尾添加一个值为 value 的元素
- lpush(key, value): 在名称为 key 的 list 头添加一个值为 value 的元素
- llen(key): 返回名称为 key 的 list 的长度
- lrange(key, start, end): 返回名称为 key 的 list 中 start 至 end 之间的元素
- ltrim(key, start, end): 截取名称为 key 的 list
- lindex(key, index): 返回名称为 key 的 list 中 index 位置的元素
- lset(key, index, value): 给名称为 key 的 list 中 index 位置的元素赋值
- lrem(key, count, value): 删除 count 个 key 的 list 中值为 value 的元素
- lpop(key): 返回并删除名称为 key 的 list 中的首元素
- rpop(key): 返回并删除名称为 key 的 list 中的尾元素
- blpop(key1, key2,... key N, timeout): lpop 命令的 block 版本。
- brpop(key1, key2,... key N, timeout): rpop 的 block 版本。
- rpoplpush(srckey, dstkey): 返回并删除名称为 srckey 的 list 的尾元素, 并将该元素添加到名称为 dstkey 的 list 的头部

## 对Set操作的命令

- sadd(key, member): 向名称为 key 的 set 中添加元素 member
- srem(key, member): 删除名称为 key 的 set 中的元素 member
- spop(key): 随机返回并删除名称为 key 的 set 中一个元素
- smove(srckey, dstkey, member): 移到集合元素
- scard(key): 返回名称为 key 的 set 的基数
- sismember(key, member): member 是否是名称为 key 的 set 的元素
- sinter(key1, key2,...key N): 求交集
- sinterstore(dstkey, (keys)): 求交集并将交集保存到 dstkey 的集合
- sunion(key1, (keys)): 求并集
- sunionstore(dstkey, (keys)): 求并集并将并集保存到 dstkey 的集合
- sdiff(key1, (keys)): 求差集
- sdiffstore(dstkey, (keys)): 求差集并将差集保存到 dstkey 的集合
- smembers(key): 返回名称为 key 的 set 的所有元素
- srandmember(key): 随机返回名称为 key 的 set 的一个元素

## 对Hash操作的命令

- hset(key, field, value): 向名称为 key 的 hash 中添加元素 field
- hget(key, field): 返回名称为 key 的 hash 中 field 对应的 value
- hmget(key, (fields)): 返回名称为 key 的 hash 中 field i 对应的 value
- hmset(key, (fields)): 向名称为 key 的 hash 中添加元素 field
- hincrby(key, field, integer): 将名称为 key 的 hash 中 field 的 value 增加 integer
- hexists(key, field): 名称为 key 的 hash 中是否存在键为 field 的域

- hdel(key, field): 删除名称为 key 的 hash 中键为 field 的域
- hlen(key): 返回名称为 key 的 hash 中元素个数
- hkeys(key): 返回名称为 key 的 hash 中所有键
- hvals(key): 返回名称为 key 的 hash 中所有键对应的 value
- hgetall(key): 返回名称为 key 的 hash 中所有的键 (field) 及其对应的 value

## Redis Sentinel

- ping: 返回 pong
- sentinel masters: 列出所有被监视的主服务器，以及这些主服务器的当前状态。
- sentinel slaves: 列出给定主服务器的所有从服务器，以及这些从服务器的当前状态。
- sentinel get-master-addr-by-name: 返回给定名字的主服务器的 IP 地址和端口号。如果这个主服务器正在执行故障转移操作，或者针对这个主服务器的故障转移操作已经完成，那么这个命令返回新的主服务器的 IP 地址和端口号。
- sentinel reset: 重置所有名字和给定模式 pattern 相匹配的主服务器。pattern 参数是一个 Glob 风格的模式 重置操作清楚主服务器目前的所有状态，包括正在执行中的故障转移，并移除目前已经发现和关联的，主服务器的所有从服务器和 Sentinel。
- sentinel failover: 当主服务器失效时，在不询问其他 Sentinel 意见的情况下，强制开始一次自动故障迁移（不过发起故障转移的 Sentinel 会向其他 Sentinel 发送一个新的配置，其他 Sentinel 会根据这个配置进行相应的更新）。

## Redis优化

### 数据持久化

- 选择恰当的持久化方式。Redis提供 RDB 和 AOF 两种持久化方式。用户需要根据实际场景对两种持久化方式进行考量和选择。

RDB 会在一定时间间隔内一次性将内存中的所有数据刷到磁盘上，非增量。它的一个主要缺点是如果Redis宕机了，那么可能会造成部分数据丢失，丢失的数据量和RDB持久化的时间间隔有关。此外，如果数据集非常大，Redis在创建子进程时可能会消耗相对较长的时间，这期间客户端请求都会被阻塞。这种情况下我们可以关闭自动保存，通过手动发送 SAVE 或者 BGSAVE 命令来控制停顿出现的时间。由于 SAVE 命令不需要创建子进程，从而避免了与子进程的资源竞争，因此它比 BGSAVE 速度更快。手动生成快照可以选择在线用户很少的情况下执行，比如使用脚本在凌晨三点执行 SAVE 命令。

从上述内容可以看出，RDB 适用于即使丢失部分数据也不会造成问题的场景。同时我们需要注意快照是否生成得过于频繁或者稀少。

AOF 持久化会将执行的命令追加到 AOF 文件末尾。在 redis.conf 中该功能默认是关闭的，设置 appendonly yes 以开启该功能。这种方式会对磁盘进行大量写入，因此Redis处理命令的速度会受到硬盘性能的限制。并且 AOF 文件通常比 RDB 文件更大，数据恢复速度比 RDB 慢，性能消耗也比 RDB 高。由于它记录的是实际执行的命令，所以也易读。为了兼顾写入性能和数据安全，可以在配置文件设置 appendfsync everysec。并不推荐 appendfsync no 选项，因为这种方式是由操作系统决定何时对AOF文件进行写入。在缓冲区被待写入硬盘的数据填满时，可能造成Redis的写入操作被阻塞，严重影响性能。

- 重写 AOF 文件。如果用户开启了 AOF 功能，Redis运行时间越长，AOF 文件也会越来越大。用户可以发送 BGREWRITEAOF 重写 AOF 文件，它会移除AOF文件中的冗余命令以此来减小AOF文件的体积。由于AOF文件重写会用到子进程，因此也存在 BGSAVE 命令持久化快照时因为创建子进程而导致的性能问题和内存占用问题。除了使用命令重写AOF文件，也可以在配置文件中配置，以让Redis自动执行重写命令。

```
# 当aof文件体积大于64mb且比上次重写之后的体积增大了至少一倍
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

## 内存优化

- 设置 `maxmemory`。设置Redis使用的最大物理内存，即Redis在占用 `maxmemory` 大小的内存之后就拒绝后续的写入请求，该参数可以确保Redis因为使用了大量内存严重影响速度或者发生 **OOM(Out of Memory)**。此外，可以使用 `info` 命令查看Redis占用的内存及其它信息。
- 让键名保持简短。键的长度越长，Redis需要存储的数据也就越多
- 使用短结构
- 对数据进行分片。比如当单个散列比较大的时候，可以按一定规则(`key+id%shard_num`)对数据进行分片，然后 `ziplist` 便更不容易退化为 `hashtable`，且不会出现编码解码引起的性能问题。

## 读写优化

- 扩展读性能。在 `redis.conf` 中添加 `slaveof host port` 即可将其配置为另一台Redis服务器的从服务器。注意，在从服务器连接主服务器的时候，从服务器之前的数据会被清空。可以用这种方式建立从服务器树，扩展其读能力。但这种方式并未做故障转移，高可用Redis部署方案可以参考 [Redis Sentinel](#)
- 扩展写性能。(1)使用集群分片技术，比如Redis Cluster;(2)单机上运行多个Redis实例。由于Redis是单线程设计，在涉及到cpu bound的操作的时候，可能速度会大大降低。如果服务器的cpu、io资源充足，可以在同一台机器上运行多个Redis服务器。
- Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件
- 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次
- 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内
- 尽量避免在压力很大的主库上增加从库
- 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

## 应用程序优化

应用程序优化部分主要是客户端和Redis交互的一些建议。主要思想是尽可能减少操作Redis往返的通信次数。

- 使用流水线操作 Redis支持 `流水线(pipeline)` 操作，其中包括了事务流水线和非事务流水线。Redis提供了 `WATCH` 命令与事务搭配使用，实现 [CAS乐观锁](#) 的机制。`WATCH` 的机制是：在事务 `EXEC` 命令执行时，Redis会检查被 `WATCH` 的key，只有被 `WATCH` 的key从 `WATCH` 起始时至今没有发生过变更，`EXEC` 才会被执行。如果 `WATCH` 的key在 `WATCH` 命令到 `EXEC` 命令之间发生过变化，则 `EXEC` 命令会返回失败。使用事务的一个好处是被 `MULTI` 和 `EXEC` 包裹的命令在执行时不会被其它客户端打断。但是事务会消耗资源，随着负载不断增加，由 `WATCH`、`MULTI`、`EXEC` 组成的事务 (CAS) 可能会进行大量重试，严重影响程序性能。  
如果用户需要向Redis发送多个命令，且一个命令的执行结果不会影响另一个命令的输入，那么我们可以使用非事务流水线来代替事务性流水线。非事务流水线主要作用是将待执行的命令一次性全部发送给Redis，减少来回通信的次数，以此来提升性能。
- 使用 `mset`、`lpush`、`zadd` 等批量操作数据。它的原理同非事务性流水线操作。
- 尽可能使用时间复杂度为  $O(1)$  的操作，避免使用复杂度为  $O(N)$  的操作。避免使用这些  $O(N)$  命令主要有几个办法：(1)不要把List当做列表使用，仅当做队列来使用;(2)通过机制严格控制Hash、Set、Sorted Set的大小;(3)可能的话，将排序、并集、交集等操作放在客户端执行;(4)绝对禁止使用 `KEYS` 命令;(5)避免一次性遍历集合类型的所有成员，而应使用 `SCAN` 类的命令进行分批的，游标式的遍历

Redis提供了Slow Log功能，可以自动记录耗时较长的命令，`redis.conf` 中的配置如下

```
#执行时间慢于10000毫秒的命令计入slow logs
slowlog-log-slower-than 10000
#最大纪录多少条slow logs
slowlog-max-len 128
```

使用SLOWLOG GET n命令，可以输出最近n条慢查询日志。使用SLOWLOG RESET命令，可以重置Slow Log。

## Redis内存回收策略

Redis会因为内存不足而产生错误，也会因为回收过久而导致系统长期的停顿，因此了解掌握Redis的回收策略十分重要。当Redis的内存达到规定的最大值时，可以进行配置进行淘汰键值，并且将一些键值对进行回收。

## 内存溢出控制策略

打开Redis安装目录下的 `redis.conf` 文件,有这么一段话

```
# 设置内存使用限制为指定的字节数。
# 达到内存限制时，Redis将尝试删除密钥
# 根据所选择的驱逐政策（参见maxmemory-policy）。
# Set a memory usage limit to the specified amount of bytes.
# When the memory limit is reached Redis will try to remove keys
# according to the eviction policy selected (see maxmemory-policy).
# 如果Redis无法根据策略删除密钥，或者策略设置为“noeviction”，则Redis将开始对使用更多内存
# 的命令
# （如SET，LPUSH等）进行错误回复，并将继续 回复像GET这样的只读命令。
# If Redis can't remove keys according to the policy, or if the policy i
s# set to 'noeviction', Redis will start to reply with errors to commands
# that would use more memory, like SET, LPUSH, and so on, and will continue
# to reply to read-only commands like GET.
### 当将Redis用作LRU或LFU缓存或为实例设置硬内存限制时（使用“noeviction”策略），此选项通常
# 很有用。
# This option is usually useful when using Redis as an LRU or LFU cache, or to#
# set a hard memory limit for an instance (using the 'noeviction' policy).
### 警告：如果在启用了maxmemory的实例上连接了从属设备，则从使用的内存计数中减去为从属设备提供
# 的输出缓冲区的大小，
# 以便网络问题/重新同步不会触发关键被驱逐的循环，并且 反过来，
# 从属的输出缓冲区已满，删除的键被删除触发删除更多键，依此类推，直到数据库完全被清空为止。
# WARNING: If you have slaves attached to an instance with maxmemory on,
# the size of the output buffers needed to feed the slaves are subtracted
# from the used memory count, so that network problems / resyncs will
# not trigger a loop where keys are evicted, and in turn the output
# buffer of slaves is full with DELs of keys evicted triggering the deletion
# of more keys, and so forth until the database is completely emptied.
### 简而言之.....如果你有附加的slaves，建议你设置maxmemory的下限，
# 以便系统上有一些空闲的RAM用于从输出缓冲区
# （但如果策略是'noeviction'则不需要这个）。
### In short... if you have slaves attached it is suggested that you set a lower
# limit for maxmemory so that there is some free RAM on the system for slave#
# output buffers (but this is not needed if the policy is 'noeviction').
# 最大内存 <字节单位>
# maxmemory <bytes> Redis将如何选择maxmemory时删除的内容
# MAXMEMORY POLICY: how Redis will select what to remove when maxmemory
# is reached. You can select among five behaviors:
### volatile-lru -> Evict using approximated LRU among the keys with an expire
# set.
# allkeys-lru -> Evict any key using approximated LRU.
# volatile-lfu -> Evict using approximated LFU among the keys with an expire
# set.
# allkeys-lfu -> Evict any key using approximated LFU.
# volatile-random -> Remove a random key among the ones with an expire set.
```



```
# allkeys-random -> Remove a random key, any key.
# volatile-ttl -> Remove the key with the nearest expire time (minor TTL)
# noeviction -> Don't evict anything, just return an error on write operations.
#LRU means Least Recently Used
#LFU means Least Frequently Used
#Both LRU, LFU and volatile-ttl are implemented using approximated
# randomized algorithms.
#Note: with any of the above policies, Redis will return an error on write
#operations, when there are no suitable keys for eviction.
# At the date of writing these commands are: set setnx setex append
# incr decr rpush lpush rpushx lpushx linsert lset rpoplpush sadd
#sinter sinterstore sunion sunionstore sdiff sdiffstore zadd zincrby
#zunionstore zinterstore hset hsetnx hincrby incrby decrby
#getset mset msetnx exec sort
#The default is:
#maxmemory-policy noeviction
#LRU, LFU and minimal TTL algorithms are not precise algorithms but approximated
# algorithms (in order to save memory), so you can tune it for speed or
#accuracy. For default Redis will check five keys and pick the one that was
#used less recently, you can change the sample size using the following
#configuration directive.
#The default of 5 produces good enough results. 10 Approximates very closely
#true LRU but costs more CPU. 3 is faster but not very accurate.
#maxmemory-samples 5
```

- **volatile-lru**: 采用最近使用最少的淘汰策略，Redis将回收那些超时的（仅仅是超时的）键值对，也就是它只淘汰那些超时的键值对。
- **allkeys-lru**: 采用最近最少使用的淘汰策略，Redis将对所有（不仅仅是超时的）的键值对采用最近最少使用的淘汰策略。
- **volatile-lfu**: 采用最近最不常用的淘汰策略，所谓最近最不常用，也就是一定时期内被访问次数最少的。Redis将回收超时的键值对。
- **allkeys-lfu**: 采用最近最不常用的淘汰策略，Redis将对所有的键值对采用最近最不常用的淘汰策略。
- **volatile-random**: 采用随机淘汰策略删除超时的键值对。
- **allkeys-random**: 采用随机淘汰策略删除所有的键值对，这个策略不常用。
- **volatile-ttl**: 采用删除存活时间最短的键值对策略。
- **noeviction**: 不淘汰任何键值对，当内存满时，如果进行读操作，例如get命令，它将正常工作，而做写操作，它将返回错误，也就是说，当Redis采用这个策略内存达到最大的时候，它就只能读不能写了。

Redis默认采用noeviction策略。

**LRU算法或者TTL算法都是不精确的算法，而是一个近似算法。**

Redis不会通过对全部的键值对进行比较来确定最精确的时间值，因为这太消耗时间，导致回收垃圾占用的时间太多造成服务器卡顿。在配置文件中，有一个参数maxmemory-samples，它的默认值是5,如果采用volatile-ttl算法，我们可以看看下面这个过程，假设有7个即将超时的键值对

**当设置maxmemory-samples越大，则Redis删除的就越精确，但消耗CPU。**

## key超时策略

redis所有键都可以设置过期时间，因为内存中大量键维护消耗大量CPU，对单线程的redis来说成本过高，因此redis采用惰性删除和定时任务删除机制实现过期键的内存回收。

- 惰性删除，如果已经超过过期时间限制，执行删除并返回空。优点是节省CPU成本，不需要单独维护TTL链表来处理过期键删除。缺点造成内存泄漏，过期键没访问也无法删除，但这个问题可以通过另一种方法解决
- 定时任务删除：Redis内部维护一个定时任务，默认每秒运行10次。定时任务中删除过期键逻辑采用了自适应算法，根据键的过期比例，使用快慢两种速率回收键，流程如下：

定时任务在每个数据库空间随机检查20个键，当发现过期时删除对应的键  
如果超过检查数的25%的键过期，循环执行回收逻辑知道不足25%或运行超时为止，慢模式下超时时间为25毫秒  
如果之前回收键逻辑超时，则在Redis触发内部事件之前再次以快模式运行回收过期键任务，快模式下超时时间为1毫秒且2秒内只能运行1次  
快慢两种模式内部删除逻辑相同，只是执行的超时时间不同。

回收超时策略的缺点是必须指明超时的键值对，这会给程序开发带来一些设置超时的代码，增加开发者的工作量。对所有的键值对进行回收，有可能把正在使用的键值对删掉，增加了存储的不稳定性。对于垃圾回收的策略，还需要控制回收的时间。

## 集群怎么监控

使用Redis集群监控客户端

RedisClusterManager, Redis monitor, RedisLive 等等

## 怎么做数据同步

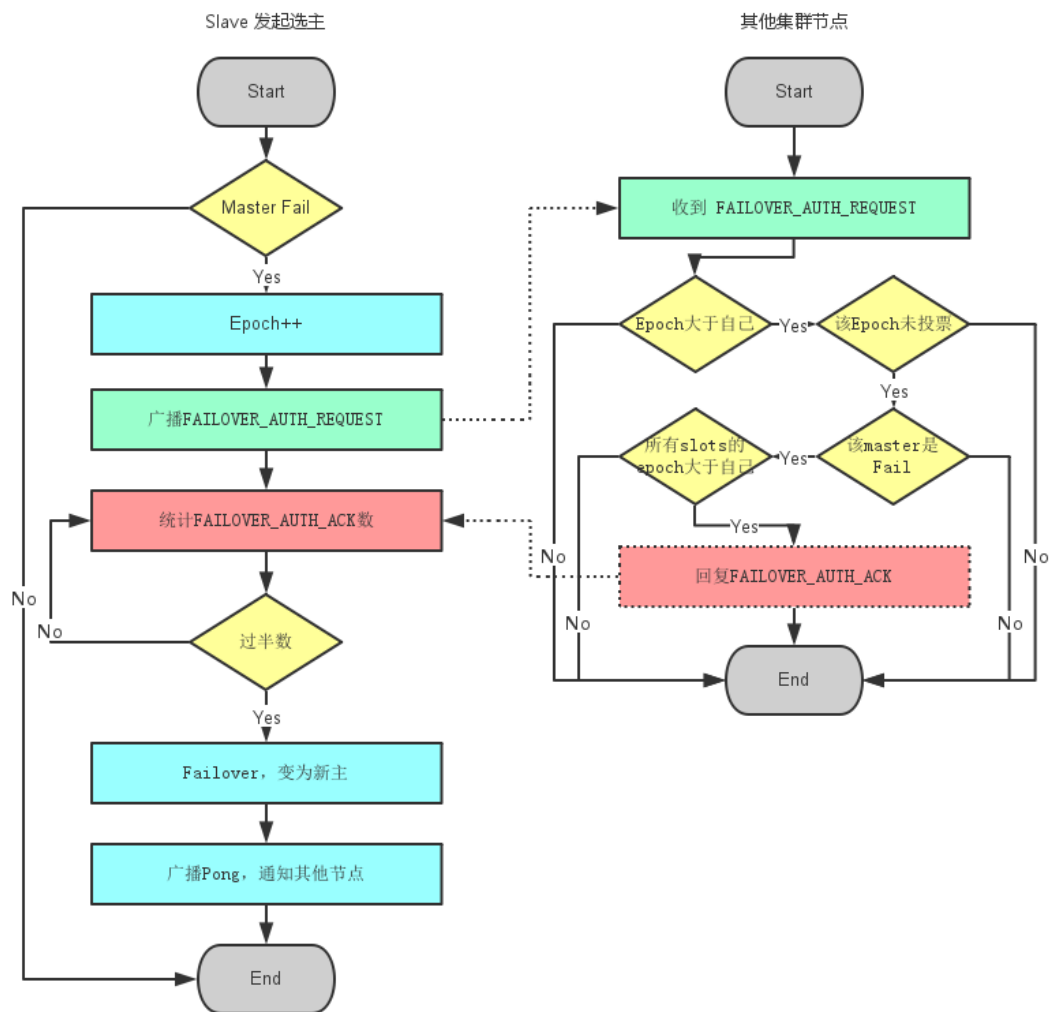
## 怎么把新的Redis加入到集群中

\*增加新的节点：首先还是要启动一个新的redis实例，然后启动一个redis客户端链接该数据库实例，执行 `CLUSTER MEET IP PORT`（其中ip与port是集群节点中任意一个数据库实例的ip地址与端口号）

## Redis集群选主原理

当slave发现自己的master变为FAIL状态时，便尝试进行Failover，以期成为新的master。由于挂掉的master可能会有多个slave，从而存在多个slave竞争成为master节点的过程，其过程如下：

1. slave发现自己的master变为FAIL
2. 将自己记录的集群currentEpoch加1，并广播FAILOVER\_AUTH\_REQUEST信息
3. 其他节点收到该信息，只有master响应，判断请求者的合法性，并发送FAILOVER\_AUTH\_ACK，对每一个epoch只发送一次ack
4. 尝试failover的slave收集FAILOVER\_AUTH\_ACK
5. 超过半数后变成新Master
6. 广播Pong通知其他集群节点。



原理图

项目中使用Redis是不是服务器内存越大会越好,为什么

在开发中,使用的Redis服务器内存是多少?

Redis集群中插槽有多少个,怎么计算

Redis集群有16384个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，集群的每个节点负责一部分hash槽。

缓存雪崩是什么?怎么解决

缓存雪崩是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。

解决:

- 用加锁或者队列的方式保证缓存的单线程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。
- 将缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

缓存击穿是什么?怎么解决

在平常高并发的系统中，大量的请求同时查询一个 `key`（热点数据）时，此时这个key正好失效了，就会导致大量的请求都打到数据库上面去。这种现象我们称为缓存击穿,会造成某一时刻数据库请求量过大,压力剧增,甚至有DB崩溃的可能.

## 解决:

- **互斥锁** 在根据key获得的value值为空时，先锁上，再从数据库加载，加载完毕，释放锁。若其他线程发现获取锁失败，则睡眠50ms后重试。

```
`private Object get(final String key, final ProceedingJoinPoint pjp, final
Cacheable cache) throws Throwable {    @SuppressWarnings("unchecked")
    valueOperations<String, Object> valueOper = redisTemplate.opsForValue();
    Object value = valueOper.get(key); // 从缓存获取数据    if (value ==
null) { // 代表缓存值过期        // 设置2min的超时，防止del操作失败的时候，下次缓
存过期一直不能load db        String keynx = key.concat(":nx");        if
(CacheUtils.setnx(keynx, "1", ExpireTime.TWO_MIN)) { // 代表设置成功
        value = pjp.proceed();        if (cache.expire().getTime() <= 0)
{ // 如果没有设置过期时间,则无限期缓存        valueOper.set(key,
value);        } else { // 否则设置缓存时间
valueOper.set(key, value, cache.expire().getTime(), TimeUnit.SECONDS);
        }
        CacheUtils.del(keynx);        return value;
    } else { // 这个时候代表同时时候的其他线程已经load db并回设到缓存了，这时候重试
获取缓存值即可        Thread.sleep(50);        return get(key,
pjp, cache); // 重试    }    } else {        return value;
    }    }`
```

- **缓存预热** 提前把 `key`(热点数据) 放到Redis直到这个事件结束,此期间尽量不要访问数据库

## Redis BloomFilter实现原理

### 概念

布隆过滤器（英语：Bloom Filter）是1970年由布隆提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难 摘自[维基百科](#),相比于传统的 List、Set、Map 等数据结构，它更高效、占用空间更少，但是缺点是其返回的结果是概率性的，而不是确切的。

如果想判断一个元素是不是在一个集合里，一般想到的是将集合中所有元素保存起来，然后通过比较确定。链表、树、散列表（又叫哈希表，Hash table）等等数据结构都是这种思路。但是随着集合中元素的增加，我们需要的存储空间越来越大。同时检索速度也越来越慢，上述三种结构的检索时间复杂度分别为： $O(n)$ ,  $O(\log n)$ ,  $O(n/k)$ 。

### 原理

当一个元素被加入集合时,通过K个[散列函数](#)将这个元素映射成一个位数组中的K个点,把它们置为1.检索时，我们只要看看这些点是不是都是1就（大约）知道集合中有没有它了：如果这些点有任何一个0，则被检元素一定不在；如果都是1，则被检元素很可能在。这就是布隆过滤器的基本思想。

本人对 BloomFilter 理解尚浅,还请各位自行[Google](#)

## 缓存穿透是什么?怎么解决

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。在流量大时，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。

### 解决:

- 采用布隆过滤器(BloomFilter)，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。
- 如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。