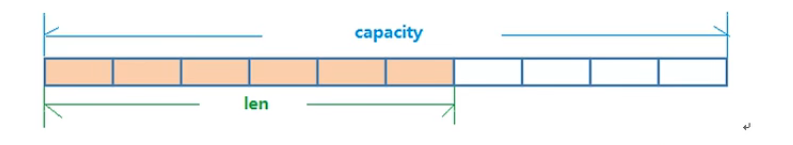
## 基本数据类型

### 1.1String

String 的**数据结构**为简单动态字符串(Simple Dynamic String,缩写SDS)。是可以修改的字符串，内部结构实现上类似于Ja的 ArrayList，采用预分配冗余空间的方式来减少内存的频繁分配.



如图中所示，内部为当前字符串实际分配的空间capacity一般要高于实际字符串长度len。当字符串长度小于1M时，扩容都是加倍现有的空间，如果超过1M，扩容时一次只会多扩1M的空间。需要注意的是字符串最大长度为512M。

### 1.2List

单键多值

Redis列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部(左边）或者尾部(右边)。

它的底层实际是个双向链表，对两端的操作性能很高，通过索引下标的操作中间的节点性能会较差。

**数据结构:**

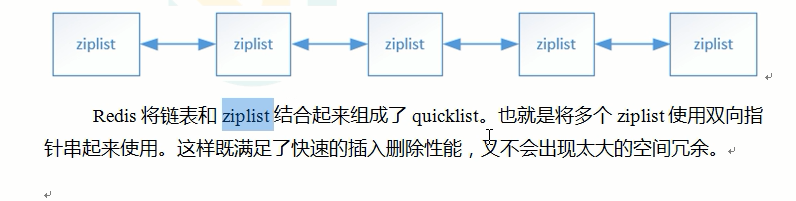
List的数据结构为快速链表quickList。

首先在列表元素较少的情况下会使用一块连续的内存存储，这个结构是ziplistI，也即是压缩列表。

它将所有的元素紧挨着一起存储，分配的是一块连续的内存。

当数据量比较多的时候才会改成quicklist。

因为普通的链表需要的附加指针空间太大，会比较浪费空间。比如这个列表里存的只是int类型的数据，结构上还需要两个额外的指针prev和next。



### 1.3Set

Redis的set是string类型的无序集合。它底层其实是一个value为null的hash表，所以添加，删除，查找的复杂度都是O(1)。"

一个算法，随着数据的增加，执行时间的长短，如果是O(1)，数据增加，查找数据的时间不变。

**数据结构:**

set数据结构是dict字典，字典是用哈希表实现的。

Java中HashSet的内部实现使用的是HashMap，只不过所有的value都指向同一个对象。Redis的set结构也是一样，它的内部也使用hash结构，所有的value都指向同一个内部值。

### 1.4 Hash

Hash类型对应的**数据结构**是两种: ziplist（压缩列表)，hashtable(哈希表)。当field-value长度较短且个数较少时，使用ziplist，否则使用hashtable。

### 1.5 Zset

**数据结构:**

( 1 ) hash , hash的作用就是关联元素value和权重score，保障元素value的唯一性，可以通过元素value找到相应的score值。

(2)跳跃表，跳跃表的目的在于给元素value排序，根据score的范围获取元素列表。

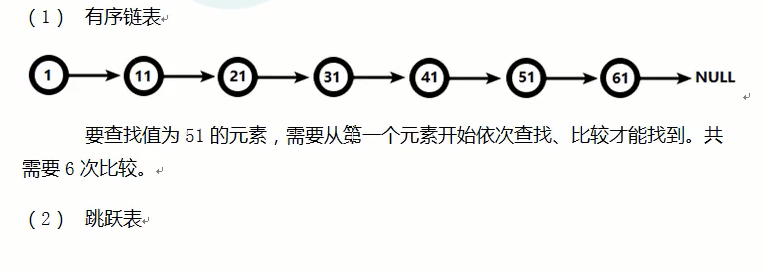
**跳跃表:**

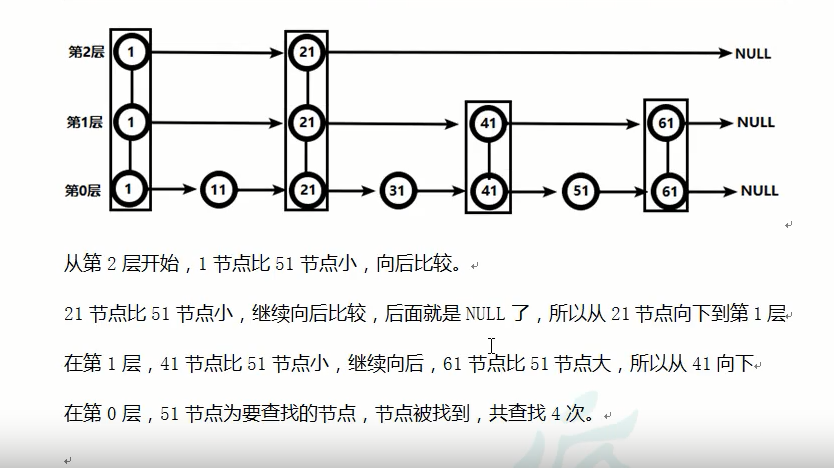
1、简介

有序集合在生活中比较常见，例如根据成绩对学生排名，根据得分对玩家排名等。对于有序集合的底层实现，可以用数组、平衡树、链表等。数组不便元素的插入、删除;平衡树或红黑树虽然效率高但结构复杂;链表查询需要遍历所有效率低。Redis采用的是跳跃表。跳跃表效率堪比红黑树，实现远比红黑树简单。v

2、实例

对比有序链表和跳跃表，从链表中查询出51

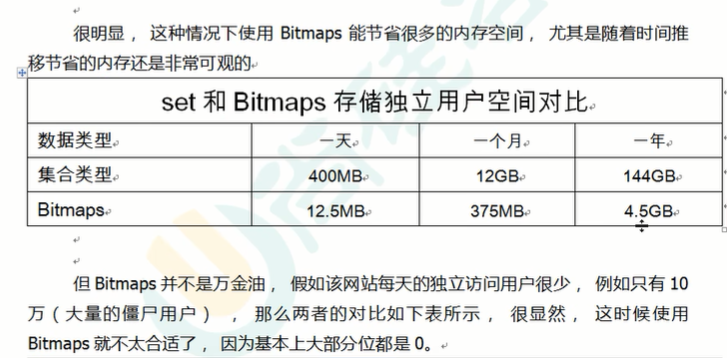




### 1.6 Bitmaps

(1)Bitmaps 本身不是一种数据类型，实际上它就是字符串( key-value ) ,但是它可以对字符串的位进行操作。

(2)Bitmaps 单独提供了一套命令，所以在 Redis 中使用Bitmaps和使用字符串的方法不太相同。可以把 Bitmaps想象成一个以位为单位的数组，数组的每个单元只能存储0和1，数组的下标在Bitmaps 中叫做偏移量。



### 1.7 HyperLogLog

在工作当中，我们经常会遇到与统计相关的功能需求，比如统计网站PV( PageView页面访问量) ,可以使用Redis的incr、incrby轻松实现。

但像UV ( UniqueVisitor，独立访客)、独立IP数、搜索记录数等需要去重和计数的问题如何解决?这种求集合中不重复元素个数的问题称为基数问题。

解决基数问题有很多种方案:

(1)数据存储在MySQL表中，使用distinct count 计算不重复个数

(2)使用Redis提供的 hash、set、bitmaps等数据结构来处理,

以上的方案结果精确，但随着数据不断增加，导致占用空间越来越大，对于非常大的数据集是不切实际的。

**能否能够降低一定的精度来平衡存储空间?**

Redis推出了HyperLogLog

Redis HyperLogLog是用来做基数统计的算法，HyperLogLog的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。

在Redis 里面，每个HyperLogLog键只需要花费12 KB内存，就可以计算接近2^64个不同元素的基数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。

但是，因为 HyperLogLog只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog不能像集合那样，返回输入的各个元素。

命令:pfcount,pfadd,pfmerge

### 1.8Geospatial

Redis 3.2中增加了对GEO类型的支持。GEO，Geographic，地理信息的缩写。该类型，就是元素的⒉维坐标，在地图上就是经纬度。redis 基于该类型，提供了经纬度设置，查询，范围查询，距离查询，经纬度Hash等常见操作。

命令:geoadd,geopos,geodist,georadius

## 发布订阅

subscribe <通道> #订阅通道

publish <通道> {消息}

## Redis事务

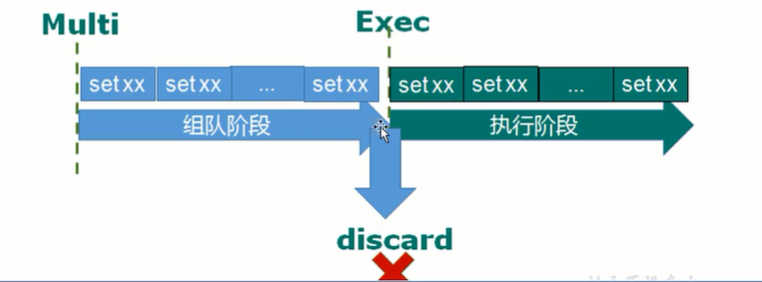
Redis事务是一个单独的隔离操作∶事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。·

Redis事务的主要作用就是串联多个命令防止别的命令插队。

### 3.1 Multi、Exec、 discard

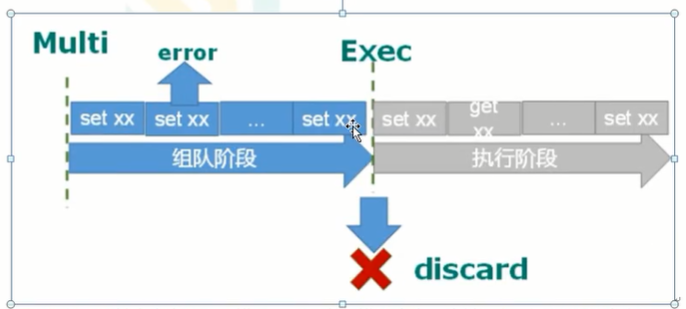
从输入Multi命令开始，输入的命令都会依次进入命令队列中，但不会执行，直到输入Exec后，Redis会将之前的命令队列中的命令依次执行。”

组队的过程中可以通过discard来放弃组队。“



#### 3.1.1事务的错误处理.

组队中某个命令出现了报告错误，执行时整个的所有队列都会被取消。



如果执行阶段某个命令报出了错误，则只有报错的命令不会被执行，而其他的命令都会执行，不会回滚。”

## Redis秒杀

### Watch+multi+exec

伪代码:

Jedis.watch()

Jedis.getKey(kcKey);

//使用事务

Transaction multi = jedis.multi();

//队列操作

multi.decr(kcKey);

//执行

multi.exec();

### Lua

Lua是一个小巧的脚本语言，Lua脚本可以很容易的被C/C++ 代码调用，也可以反过来调用C/C++的函数，Lua并没有提供强大的库，一个完整的Lua解释器不过200k，所以Lua不适合作为开发独立应用程序的语言，而是作为嵌入式脚本语言。

很多应用程序、游戏使用LUA作为自己的嵌入式脚本语言，以此来实现可配置性、可扩展性。

#### LUA 脚本在Redis中的优势

将复杂的或者多步的redis操作，写为一个脚本，一次提交给redis执行，减少反复连接redis的次数。提升性能。

LUA脚本是类似redis事务，有一定的原子性，不会被其他命令插队，可以完成—些redis事务性的操作。

但是注意redis的lua脚本功能，只有在Redis 2.6以上的版本才可以使用。

利用lua脚本淘汰用户，解决超卖问题。

redis 2.6版本以后，通过lua脚本解决争抢问题，实际上是redis利用其单线程的特性，用任务队列的方式解决多任务并发问题。

## RDB

### 是什么

在指定的时间间隔内将内存中的数据集快照写入磁盘，也就是行话讲的 Snapshot快照，它恢复时是将快照文件直接卖到内存里

### 备份如何执行

Redis,会单独创建( fork )一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。RDB的缺点是最后一次持久化后的数据可能丢失。

### Fork

Fork 的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程

在Linux程序中，fork(会产生一个和父进程完全相同的子进程，但子进程在此后多会exec系统调用，出于效率考虑，Linux中引入了“写时复制技术”

一般情况父进程和子进程会共用同一段物理内存，只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程。

### 优点

适合大规模的数据恢复

对数据完整性和一致性要求不高更适合使用

节省磁盘空间

恢复速度快

### 缺点

Fork的时候，内存中的数据被克隆了一份，大致2倍的膨胀性需要考虑

虽然 Redis.在fork时使用了写时拷贝技术,但是如果数据庞大时还是比较消耗性能。

在备份周期在一定间隔时间做一次备份，所以如果Redis.,意外down掉的话，就会丢失最后一次快照后的所有修改。

### 动态停止

动态停止RDB : redis-cli config set save ""#save后给空值，表示禁用保存策略

## AOF

以日志的形式来记录每个写操作（增量保存），将Redis执行过的所有写指令记录下来(读操作不记录)，只许追加文件但不可以改写文件，redis启动之初会读取该文件重新构建数据，换言之，redis重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作(默认不开启,,和rdb同时开启,默认读取aof)

### AOF 启动/修复

修改默认的appendonly no，改为yes

如遇到AOF文件损坏，通过/usr/local/bin/redis-check-aof—fix appendonly.aof 进行恢复

备份被写坏的AOF文件

恢复:重启redis，然后重新加载

### AOF同步频率设置

**appendfsync always**

始终同步，每次 Redis的写入都会立刻记入日志;性能较差但数据完整性比较好**appendfsync everysec**

每秒同步，每秒记入日志一次，如果宕机，本秒的数据可能丢失。

appendfsync no

redis不主动进行同步，把同步时机交给操作系统。

### Rewrite压缩

**1是什么**

AOF采用文件追加方式，文件会越来越大为避免出现此种情况，新增了重写机制,当AOF文件的大小超过所设定的阈值时，Redis就会启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集.可以使用命令bgrewriteaof

**2重写原理，如何实现重写**

AOF文件持续增长而过大时，会fork出一条新进程来将文件重写(也是先写临时文件最后再rename) , redis4.0版本后的重写，是指上就是把rdb 的快照，以二级制的形式附在新的 aof头部，作为已有的历史数据，替换掉原来的流水账操作。

### no-appendfsync-on-rewrite

如果no-appendfsync-on-rewrite=yes ,不写入aof文件只写入缓存，用户请求不会阻塞，但是在这段时间如果宕机会丢失这段时间的缓存数据。（降低数据安全性，提高性能)·

如果no-appendfsync-on-rewrite=no，还是会把数据往磁盘里刷，但是遇到重写操作，可能会发生阻塞。（数据安全，但是性能降低)

### 触发机制，何时重写

Redis会记录上次重写时的AOF大小，默认配置是当AOF文件大小是上次rewrite后大小的一倍且文件大于64M时触发

重写虽然可以节约大量磁盘空间，减少恢复时间。但是每次重写还是有一定的负担的，因此设定Redis要满足一定条件才会进行重写。

auto-aof-rewrite-percentage:设置重写的基准值，文件达到100%时开始重写（文件

是原来重写后文件的2倍时触发)

auto-aof-rewrite-min-size:设置重写的基准值，最小文件64MB。达到这个值开始重写

重写过程与RDB类似,临时文件覆盖旧文件

### AOF持久化流程

( 1)客户端的请求写命令会被append追加到AOF缓冲区内;

( 2 )AOF缓冲区根据AOF持久化策略[always,everysec,no]将操作sync同步到磁盘的AOF文件中;

( 3 )AOF文件大小超过重写策略或手动重写时，会对AOF文件rewrite重写，压缩AOF文件容量;

( 4 ) Redis服务重启时，会重新load 加载 AOF 文件中的写操作达到数据恢复的目的;

### 优点

备份机制更稳健，丢失数据概率更低。

可读的日志文本，通过操作AOF稳健，可以处理误操作。

### 缺点

比起RDB占用更多的磁盘空间。

恢复备份速度要慢。

每次读写都同步的话，有一定的性能压力。

存在个别 Bug，造成恢复不能。

## 主从复制

一主二仆

薪火相传:大从管小从

反客为主:从变主

### 原理

1.Slave启动成功连接到master后会发送一个sync命令

2.Master接到命令启动后台的存盘进程，同时收集所有接收到的用于修改数据集命令，在后台进程执行完毕之后，master将传送整个数据文件到slave,以完成一次完全同步

3.全量复制:而slave服务在接收到数据库文件数据后，将其存盘并加载到内存中。

4.增量复制:Master继续将新的所有收集到的修改命令依次传给slave,完成同步

5.但是只要是重新连接master,一次完全同步（全量复制)将被自动执行

## 哨兵模式

添加自定义sentinel.conf 文件

文件中添加

sentinel monitor mymaster 127.0.0.1 6379 1

其中mymaster为监控对象起的服务器名称，1为至少有多少个哨兵同意迁移的数量。

### 故障恢复

从变主优先级:

1、选择优先级靠前的

2、选择偏移量最大的

3、选择runid最小的从服务

优先级在redis.conf中默认: replica-priority 100，值越小优先级越高

偏移量是指获得原主机数据最全的

每个redis实例启动后都会随机生成一个40位的runid

Java中使用JedisSentinelPool根据sentinel.conf文件中的服务器名找到主服务器

## 集群

去中心化集群配置

### 什么是集群?

Redis集群实现了对Redis 的水平扩容，即启动N个redis节点，将整个数据库分布存储在这N个节点中，每个节点存储总数据的1/N。

Redis集群通过分区( partition )来提供一定程度的可用性( availability ) :即使集群中有一部分节点失效或者无法进行通讯，集群也可以继续处理命令请求。

### 什么事slots?

All 16384 slots covered.

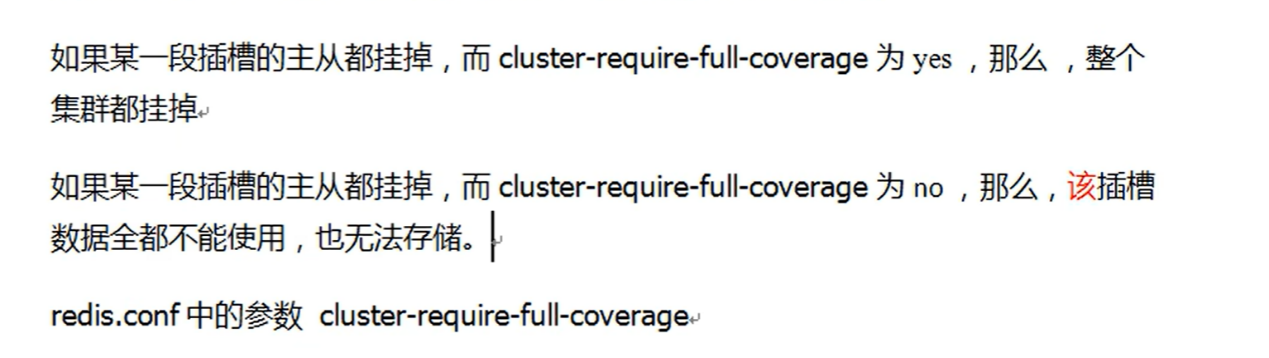
一个Redis集群包含16384个插槽( hash slot)，数据库中的每个键都属于这16384个插槽的其中一个，

集群使用公式 CRC16(key)% 16384来计算键key属于哪个槽，其中 CRC16(key)语句用于计算键key的CRC16校验和。

集群中的每个节点负责处理一部分插槽。

### 故障恢复?





## 缓存穿透

### 描述

key对应的数据在数据源并不存在，每次针对此 key的请求从缓存获取不到，请求都会压到数据源，从而可能压垮数据源。比如用一个不存在的用户id获取用户信息，不论缓存还是数据库都没有，若黑客利用此漏洞进行攻击可能压垮数据库。

### 解决方案

一个一定不存在缓存及查询不到的数据，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。

( 1)**对空值缓存**∶如果一个查询返回的数据为空(不管是数据是否不存在），我们仍然把这个空结果( null )进行缓存，设置空结果的过期时间会很短，最长不超过五分钟,

( 2)**设置可访问的名单(白名单)**:使用bitmaps类型定义一个可以访问的名单，名单id作为 bitmaps的偏移量，每次访问和 bitmap里面的 id进行比较，如果访问id不在bitmaps里面，进行拦截，不允许访问。

( 3）**采用布隆过滤器**∶(布隆过滤器(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(位图)和一系列随机映射函数（哈希函数)。将所有可能存在的数据哈希到一个足够大的bitmaps 中，一个一定不存在的数据会被这个bitmaps拦截掉，从而避免了对底层存储系统的查询压力。

(4)**进行实时监控**:当发现 Redis 的命中率开始急速降低，需要排查访问对象和访问的数据，和运维人员配合，可以设置黑名单限制服务

## 缓存击穿

### 描述

key 对应的数据存在，但在redis 中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

### 解决方案

key可能会在某些时间点被超高并发地访问，是一种非常“热点的数据。这个时候，需要考虑一个问题:缓存被“击穿”的问题。

解决问题:

(1）预先设置热门数据︰在redis高峰访问之前，把一些热门数据提前存入到redis里面，加大这些热门数据key的时长

(2)实时调整:现场监控哪些数据热门，实时调整key 的过期时长,

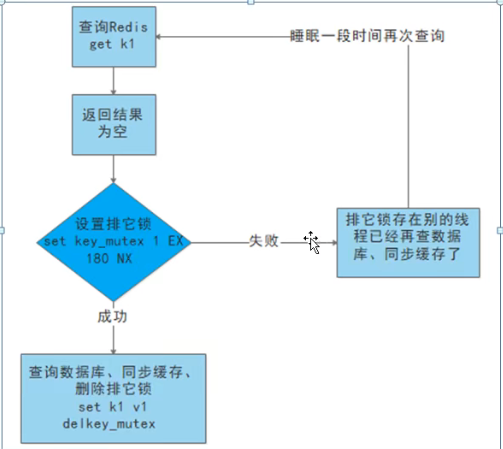
(3)使用锁:

(1)就是在缓存失效的时候（判断拿出来的值为空），不是立即去 load db。

(2)先使用缓存工具的某些带成功操作返回值的操作（比如 Redis 的 SETNX ) 去set—个mutex key

(3)当操作返回成功时，再进行load db的操作，并回设缓存,最后删除 mutex key ;

(4）当操作返回失败，证明有线程在 load db，当前线程睡眠一段时间再重试整个get缓存的方法。



## 缓存雪崩

### 描述

key对应的数据存在，但在redis 中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

缓存雪崩与缓存击穿的区别在于这里针对很多key缓存，前者则是某一个key正常访问

### 解决方案

缓存失效时的雪崩效应对底层系统的冲击非常可怕!

解决方案:

(1)构建多级缓存架构:nginx缓存+redis缓存+其他缓存( ehcache等)

(2)使用锁或队列:

用加锁或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。不适用高并发情况

(3)设置过期标志更新缓存:

记录缓存数据是否过期（设置提前量），如果过期会触发通知另外的线程在后台去更新实际key的缓存。

(4)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。

## 分布式锁

### 描述

随着业务发展的需要，原单体单机部署的系统被演化成分布式集群系统后，由于分布式系统多线程、多进程并且分布在不同机器上，这将使原单机部署情况下的并发控制锁策略失效，单纯的Java API并不能提供分布式锁的能力。为了解决这个问题就需要一种跨JVM的互斥机制来控制共享资源的访问，这就是分布式锁要解决的问题!

时

分布式锁主流的实现方案:,

1.基于数据库实现分布式锁。

⒉基于缓存( Redis等) ,

3.基于Zookeeper

每一种分布式锁解决方案都有各自的优缺点:

### 解决方案

redis:命令

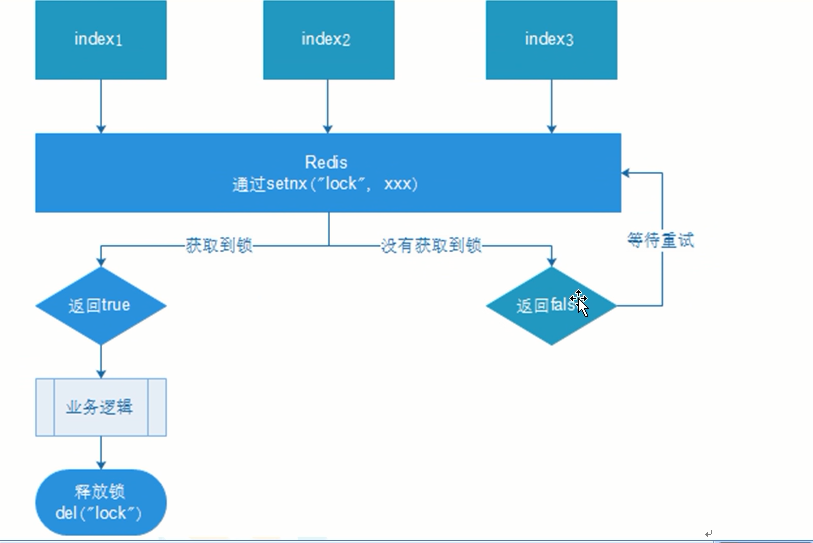
# set sku:1:info "OK”NX PX 10000.

EX second :设置键的过期时间为second秒。SET key value EX second效果等同于SETEX key second value 。

PX millisecond :设置键的过期时间为millisecond毫秒。SET key value PX milisecond效果等同于PSETEX key millisecond value 。

NX︰只在键不存在时，才对键进行设置操作。SET key value NX效果等同于SETNXkey value 。

XX:只在键已经存在时，才对键进行设置操作。



### 上锁之后断电了怎么办?

上锁的同时设置过期时间:set user 10 nx ex 10(原子性操作,setnx 非原子性操作

)

(nx:上锁 ex:过期时间)