1. redis数据结构有哪些
2. redis队列应用场景
3. redis和Memcached（支持数据持久化）
4. 分布式使用场景（储存session等）
5. 发布/订阅使用场景

redis集群（回答主从复制、分区就可以）

redis的持久化方式，redis3.0原生集群和redis读写分离+哨兵机制区别

Redis有几种数据结构，并且对于hash数据结构，如果要删除数据，那么redis的底层是怎么处理的?（数据结构的底层实现）

# 你对缓存有什么理解？缓存是解决什么问题？后端缓存有哪些，分别解决什么问题？

## 1.对缓存的理解

将程序或系统**经常要使用**的**对象存在内存**中，以便在使用时可以**快速调用**，也可以避免加载数据或者创建重复的实例，以达到**减少系统开销**，提高系统效率的目的。

## 2.缓存用来解决的问题

在日常开发有这样的一种场景：某些数据的**数据量不大**、**不经常变动**，但**访问却很频繁**。**受限于硬盘 IO** 性能或者**远程网络**等原因，每次都直接获取会消耗大量的资源。可能会导致我们的响应变慢甚至造成系统压力过大，这在一些业务上是不能忍的，而缓存正是解决这类问题的神器。

## 3.使用缓存的场景

对于缓存来说，数据**不常变更且查询比较频繁**是最好的场景，如果查询量不够大或者数据变动太频繁，缓存也就是失去了意义。

## 4.缓存的类别

**数据库型缓存**：数据库型缓存就是将**数据存储在非关系型数据库**中，非关系型数据库中用**key-value的形式存储数据**，这样就可以大大加快访问效率。

**文件型缓存**：数据库型缓存要远程访问数据库速度还是慢了点，文件型数据库就是**将数据临时存放到本地文件**，访问本地文件速度比远程数据库肯定要快，所以这个文件就是一个缓存。

**内存型缓存**：在进一步，访问文件还是有点慢，那么我们可以把**一些热数据直接放到内存中**，这样访问内存的速度比文件又更快了。

## 5. 内部缓存和外部缓存

日常工作使用的缓存可以分为内部缓存和外部缓存。

### 5.1内部缓存

内部缓存一般是指存放在运行实例内部并使用实例内存的缓存，这种缓存可以使用代码直接访问。

在系统中，有些数据量不大、不常变化，但是访问十分频繁，例如省、市、区数据。针对这种场景，可以将数据加载到应用的内存中，以提升系统的访问效率，减少无谓的数据库和网路的访问。内部缓存的限制就是存放的数据总量不能超出内存容量，毕竟还是在 JVM 里的。

### 最简单的数据缓存-Map

如果只是需要将一些数据缓存起来，避免不必要的数据库查询，那么 Map 就可以满足。

对于字典型的数据，在项目启动的时候加载到 Map 中，程序就可以使用了，也很容易更新。

### 功能强大的内部缓存 - Guava Cache / Caffeine

如果你需要缓存有强大的性能，或者对缓存有更多的控制，可以使用 Guava 里的 Cache 组件。

它是 Guava 中的缓存工具包，是非常简单易用且功能强大的 JVM 内缓存，支持多种缓存过期策略。

本地缓存的优点：

* 直接使用内存，速度快，通常存取的性能可以达到每秒千万级
* 可以直接使用 Java 对象存取

本地缓存的缺点：

* 数据保存在当前实例中，无法共享
* 重启应用会丢失

### 5.2外部缓存

**最著名的外部缓存 - Redis / Memcached**

Redis / Memcached 都是使用内存作为存储，所以性能上要比数据库要好很多，再加上Redis 还支持很多种数据结构，使用起来也挺方便，所以作为很多人的首选。

Redis 确实不错，不过即便是使用内存，也还是需要通过网络来访问，所以网络的性能决定了 Reids 的性能。

作为一般的系统来使用已经绰绰有余了，从目前来看，Redis 确实很适合来做系统中的缓存。

如果考虑多实例或者分布式，可以考虑下面的方式：

* Jedis 的 ShardedJedis（ 调用端自己实现分片 ）
* twemproxy / codis（ 第三方组件实现代理 ）
* Redis Cluster（ 3.0 之后官方提供的集群方案 ）

Redis有很多优点：

* 很容易做数据分片、分布式，可以做到很大的容量
* 使用基数比较大，库比较成熟

同时也有一些缺点：

* Java 对象需要序列化才能保存
* 如果服务器重启，在不做持久化的情况下会丢失数据，即使有持久化也容易出现各种各样的问题

# 我们来讲讲redis，你觉得redis是个怎样的东西，除了缓存还有什么知道它的过期策略吗？

一个高性能的(key/value)**分布式内存数据库**，**基于内存运行**并**支持持久化的NoSQL数据库**，是当前最热门的NoSql数据库之一,也被人们称为数据结构服务器。

特点：

1、数据结构丰富。Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。

2、支持数据的持久化。可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。

3、支持主从复制，主机会自动将数据同步到从机，可以进行读写分离。

4、读写性能好，但是读的效率高于写的效率，但是可以使用Redis+ssdb+Lua脚本联合使用，因为Redis和ssdb共用一套客户端，即./Redis-cli，同样应用于ssdb中，所以启动Redis后就可以拿到ssdb中的数据，ssdb写的效率大于读的效率，即Redis作为读的，ssdb作为写的。

# Redis 与 Memcached

两者都是**非关系型内存键值数据库**。主要有以下不同：

## ****1、数据类型支持不同****

与Memcached仅支持简单的key-value结构的数据记录不同，Redis支持的数据类型要丰富得多。最为常用的数据类型主要由五种：String、Hash、List、Set和Sorted Set。

## ****2、内存管理机制不同****

在Redis中，并不是所有的数据都一直存储在内存中的。这是和Memcached相比一个最大的区别。当物理内存用完时，Redis可以将一些很久没用到的value交换到磁盘。Redis只会缓存所有的key的信息，如果Redis发现内存的使用量超过了某一个阀值，将触发swap的操作，Redis根据“swappability = age\*log(size\_in\_memory)”计算出哪些key对应的value需要swap到磁盘。然后再将这些key对应的value持久化到磁盘中，同时在内存中清除。这种特性使得Redis可以保持超过其机器本身内存大小的数据。当然，机器本身的内存必须要能够保持所有的key，毕竟这些数据是不会进行swap操作的。

## ****3、数据持久化支持****

Redis虽然是基于内存的存储系统，但是它本身是支持内存数据的持久化的，而且提供两种主要的持久化策略：RDB快照和AOF日志。而memcached是不支持数据持久化操作的。

## ****4.集群管理的不同****

Memcached本身并不支持分布式，因此只能在客户端通过像一致性哈希这样的分布式算法来实现Memcached的分布式存储。当客户端向Memcached集群发送数据之前，首先会通过内置的分布式算法计算出该条数据的目标节点，然后数据会直接发送到该节点上存储。但客户端查询数据时，同样要计算出查询数据所在的节点，然后直接向该节点发送查询请求以获取数据。

相较于Memcached只能采用客户端实现分布式存储，Redis更偏向于在服务器端构建分布式存储。最新版本的Redis已经支持了分布式存储功能。Redis Cluster是一个实现了分布式且允许单点故障的Redis高级版本，它没有中心节点，具有线性可伸缩的功能。

为了保证单点故障下的数据可用性，Redis Cluster引入了Master节点和Slave节点。在Redis Cluster中，每个Master节点都会有对应的两个用于冗余的Slave节点。这样在整个集群中，任意两个节点的宕机都不会导致数据的不可用。当Master节点退出后，集群会自动选择一个Slave节点成为新的Master节点。

# redis的底层数据结构

Redis 数据库里面的每个**键值对（key-value）** 都是由**对象（object）**组成的：

数据库键总是一个字符串对象（string object）;

数据库的值则可以是字符串对象、列表对象（list）、哈希对象（hash）、集合对象（set）、有序集合（sort set）对象这五种对象中的其中一种。

Redis内部实现如下数据结构[2,3,4,10]：   
1. simple dynamic string：简单动态字符串  
2.字典（类似于hashtable）  
3 .Doubly Linked List ：链表  
4. Skip List ：跳跃表  
5 .Zip List ： 压缩列表  
6. Int Sets ：整数集合

下面分别来进行介绍：

### 1. simple dynamic string:动态字符串

redis中定义的动态字符串的数据结构为：

/\*

\* 保存字符串对象的结构

\*/

struct sdshdr {

// buf 中已占用空间的长度

int len;

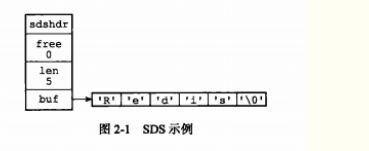
// buf 中剩余可用空间的长度

int free;

// 数据空间

char buf[];

};



　1、len 变量，用于记录buf 中已经使用的空间长度（这里指出Redis 的长度为5）

　2、free 变量，用于记录buf 中还空余的空间（初次分配空间，一般没有空余，在对字符串修改的时候，会有剩余空间出现）

3、buf 字符数组，用于记录我们的字符串（记录Redis）

与传统的字符串的区别：

1.我们可以通过获取len 属性的值，直接知道字符串长度。

2.杜绝缓冲区溢出。当我们需要对一个SDS 进行修改的时候，redis 会在执行拼接操作之前，预先检查给定SDS 空间是否足够，如果不够，会先拓展SDS 的空间，然后再执行拼接操作。

3. SDS将连续增长N次字符串所需的内存重分配次数从必定N次降低为最多N次

4. 通过惰性空间释放，SDS 避免了缩短字符串时所需的内存重分配操作，并未将来可能有的增长操作提供了优化。

5. 在Redis中，不是靠空字符来判断字符串的结束的，而是通过len这个属性。

### 2.链表

每个链表节点使用一个 **listNode**结构表示（adlist.h/listNode）：

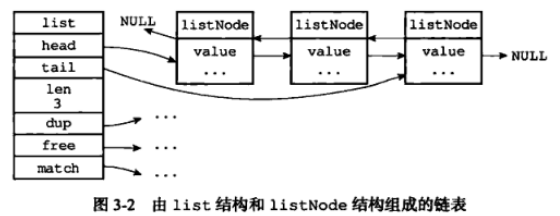
typedef struct listNode{

struct listNode \*prev;

struct listNode \* next;

void \* value;

}



链表的特性：

双端：链表节点带有prev 和next 指针，获取某个节点的前置节点和后置节点的时间复杂度都是O（N）；

无环：表头节点的 prev 指针和表尾节点的next 都指向NULL，对立案表的访问时以NULL为截止；

表头和表尾：因为链表带有head指针和tail 指针，程序获取链表头结点和尾节点的时间复杂度为O(1)；

长度计数器：链表中存有记录链表长度的属性 len；

多态：链表节点使用 void\* 指针来保存节点值，并且可以通过list 结构的dup 、 free、 match三个属性为节点值设置类型特定函数。

### 3.字典

typedef struct dict {

dictType \*type; // 类型特定函数

void \*privedata; // 私有数据

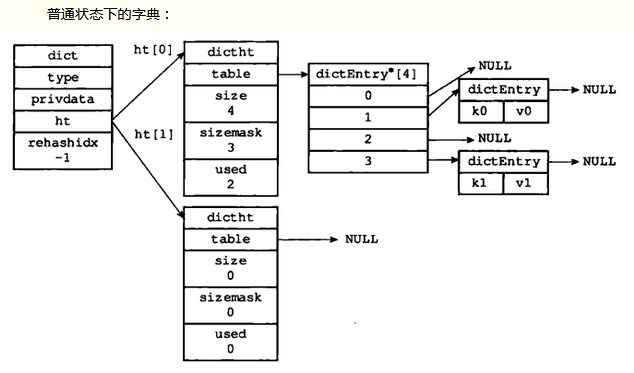
dictht ht[2]; // 哈希表

in trehashidx; // rehash 索引

}

type 属性 和privdata 属性是针对不同类型的键值对，为创建多态字典而设置的。

ht 属性是一个包含两个项（两个哈希表）的数组。



### ****3.1解决哈希冲突****

在插入一条新的数据时，会进行哈希值的计算，如果出现了hash值相同的情况，Redis 中采用了链地址法（separate chaining）来解决键冲突。采用头插法。

### ****3.2Rehash****

随着对哈希表的不断操作，哈希表保存的键值对会逐渐的发生改变，为了让哈希表的负载因子维持在一个合理的范围之内，我们需要对哈希表的大小进行相应的扩展或者压缩，这时候，我们可以通过 rehash（重新散列）操作来完成。

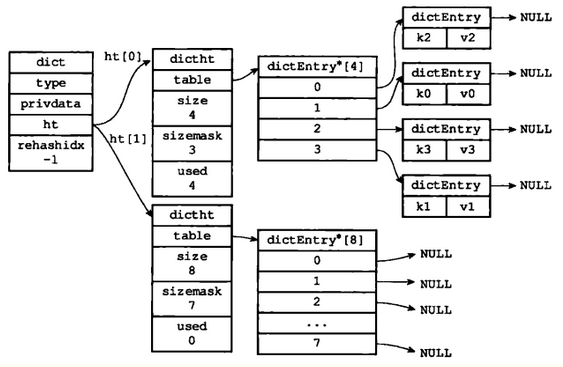
### ****为哈希表分配空间****

哈希表空间分配规则：

如果执行的是拓展操作，那么ht[1] 的大小为第一个大于等于ht[0] 的2的n次幂。

如果执行的是收缩操作，那么ht[1] 的大小为第一个小于等于ht[0] 的2的n次幂。

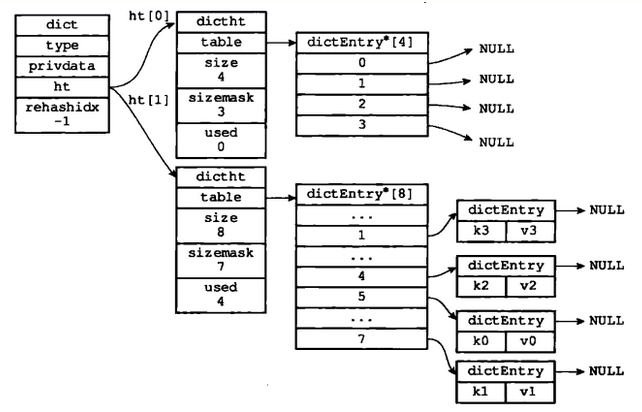
因此这里我们为ht[1] 分配 空间为8，



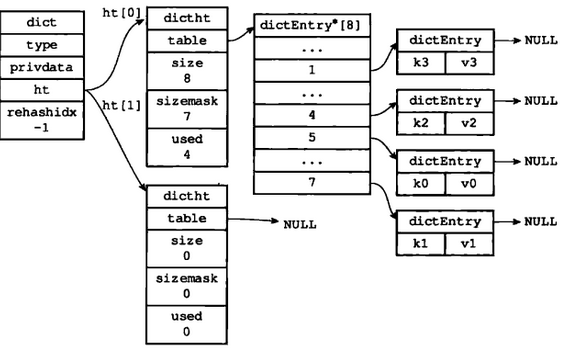
### 数据的迁移

将ht[0]中的数据转移到ht[1]中，在转移的过程中，需要对哈希表节点的数据重新进行哈希值计算。

数据转移后的结果：



### ****释放ht[0]****

将ht[0]释放，然后将ht[1]设置成ht[0]，最后为ht[1]分配一个空白哈希表：

### 渐进式 rehash

上面我们说到，在进行拓展或者压缩的时候，可以直接将所有的键值对rehash 到ht[1]中，这是因为数据量比较小。在实际开发过程中，这个rehash 操作并不是一次性、集中式完成的，而是分多次、渐进式地完成的。

　　渐进式rehash 的详细步骤：

　　　1、为ht[1] 分配空间，让字典同时持有ht[0]和ht[1]两个哈希表；

　　　2、维持一个索引计数器变量rehashidx，并将它的值设置为0，表示rehash 开始；

　　　3、在rehash 进行期间，每次对字典执行CRUD操作时，程序除了执行指定的操作以外，还会将ht[0]中的数据rehash 到ht[1]表中，并且将rehashidx加一；

　　　4、当ht[0]中所有数据转移到ht[1]中时，将rehashidx 设置成-1，表示rehash 结束。

采用渐进式rehash 的好处在于它采取分而治之的方式，避免了集中式rehash 带来的庞大计算量。

## 4.跳跃表

Redis 只在两个地方用到了跳跃表，一个是实现**有序集合键**，另外一个是在**集群节点**中用作**内部数据结构**。

## 5. 整数集合（Intset）

整数集合是**集合键**的底层实现之一，当一个集合中只包含整数，且这个集合中的元素数量不多时，redis就会使用整数集合intset作为集合的底层实现。  
typedef struct intset{

//编码方式

uint32\_t enconding;

// 集合包含的元素数量

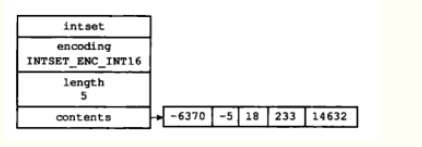
uint32\_t length;

//保存元素的数组

int8\_t contents[];

}

我们观察一下一个完成的整数集合结构图：



1、encoding：用于定义整数集合的编码方式；

2、length：用于记录整数集合中变量的数量；

 3、contents：用于保存元素的数组，虽然我们在数据结构图中看到，intset将数组定义为int8\_t，但实际上数组保存的元素类型取决于encoding。

### 整数集合的升级

在上述数据结构图中我们可以看到，intset 在默认情况下会帮我们设定整数集合中的编码方式，但是当我们存入的整数不符合整数集合中的编码格式时，就需要使用到Redis 中的升级策略来解决

Intset 中升级整数集合并添加新元素共分为三步进行：

1、根据新元素的类型，扩展整数集合底层数组的空间大小，并为新元素分配空间

2、将底层数组现有的所有元素都转换成新的编码格式，重新分配空间

3、将新元素加入到底层数组中

## 6.压缩列表

压缩列表是列表键和哈希键的底层实现之一。当一个列表键只包含少量列表项，并且每个列表项要么是小整数，要么就是长度比较短的字符串，那么Redis 就会使用压缩列表来做列表键的底层实现。

### 压缩列表的构成



1、zlbytes:用于记录整个压缩列表占用的内存字节数

2、zltail：记录列表尾节点距离压缩列表的起始地址有多少字节

3、zllen：记录了压缩列表包含的节点数量。

4、entryX：压缩列表包含的各个节点

5、zlend：用于标记压缩列表的末端

# redis支持的数据结构

redis支持五种数据类型

string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

## 1.String（字符串）

### 底层采用的数据结构是simple dynamic string（动态字符串）

**单值单value.**

string是redis最基本的类型，一个key对应一个value。

string类型是**二进制安全**的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

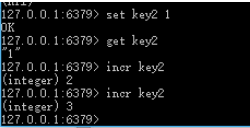
可以是**字符串，整数或者浮点数**，对整个字符串或者字符串中的一部分执行操作，对整个整数或者浮点执行自增(increment)或者自减(decrement)操作。

字符串命令：

**get**、获取存储在指定键中的值

**set**、设置存储在指定键中的值

**del**、删除存储在指定键中的值（这个命令可以用于所有的类型）



## 2.Hash（哈希）

**新建的Hash类型也使用ziplist存储value，保存数据过多时，转而使用字典。**

单key多键值对。

hash类型是每个key对应一个**hashtable**。hash类型适用于存储对象。如存储用户信息对象。包含键值对无序散列表，添加，获取，移除当键值对，获取所有键值对。

这里介绍五个命令 ：  
hset key field value ：设置key对应的hash对象中指定域的值。   
hget key field ：返回key对应的hash对象中指定域的值。   
hkeys key ：返回key对应的hash对象中所有field名称。   
hvals key ： 返回key对应的hash对象中所有值。（没有key）   
hgetall key：返回key对应的hash对象中所有field和值

## 3.List

**List类型的key创建时使用zip list结构存储，当list内容较多时，转而使用双向链表。**

单值多value。

list数据类型指key对应的value是一个**双向链表结构**。所以list类型提供链表支持的所有操作。使用场景如微博中我的关注列表。

这里介绍四个命令   
lpush key string ：向key对应list头部添加一个字符串元素。   
rpush key string ：向key对应list尾部添加一个字符串元素。   
lrange key start end ：返回指定区间内（start~end）的元素。下标从0开始。支持负数。   
lpop key ：从list头部返回一个元素，并删除此元素。   
另外，list类型还可以实现队列功能。实现如，使用rpush命令把消息放进队列尾部，使用lpop命令把消息从队列头部取出。

## 4.Set

**创建Set类型的key-value时，如果value能够表示为整数，则使用intset类型保存value。数据量大时，切换为使用hash table保存各个value。**

单值多value。

set类型是一种 无序集合。set数据类型的优点在于快速查找元素是否存在。适用于，网站注册账号时，用户名不能重复。使用set记录注册用户，如果注册用户已经存在set中，则拒绝注册。

添加、获取、移除单个元素，检查一个元素是否存在于集合中，计算交集、并集、差集，从集合里面随机获取元素。

这里介绍三个命令   
sadd key member ：添加一个元素到key对于的set集合中。   
sismember key member ：判断member是否在set中。   
smembers key ：返回所以key对于的set元素。

srem key:检查指定元素是否存在于集合中，那么移除这个元素。  
另外，set还可以使用在投票系统中。当限制每个人只能投一票时。将已投过票的用户id存在key对应的set中。



## 5.Zset

**如果新建的zset包含value数大于配置或者value长度大于配置值，则直接使用hash table和skip list存储value，skip list实现对value的排序；否则直接使用skip list存储value。**

在set基础上，加一个score值。之前set是k1 v1 v2 v3，现在zset是k1 score1 v1 score2 v2。

**字符串成员(member)与浮点数分值(score)**之间的**有序映射**，元素的排列顺序由**分值的大小决定**。添加，获取，删除单个元素，根据**分值范围(ran**ge)或者成员来获取元素。

ZADD ： 将一个带有给定分值的成员添加到有序集合里面

ZRANGE ： 根据元素在有序排列中所处的位置，从有序集合里面获取多个元素

ZRANGEBYSCORE ：获取有序集合在给定分值范围内的所有元素

ZREM ：如果给定成员存在于有序集合，那么移除这个成员。

# Redis持久化

## RDB持久化

RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）。

### 1.优点

* RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。
* RDB 非常适用于**灾难恢复**（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心，或者亚马逊 S3 中。
* RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 **fork** 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。
* RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

### 2.fork

Fork的作用就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。

### 3.RDB持久化配置

在默认情况下， Redis 将数据库快照保存在名字为 dump.rdb 的二进制文件中。

你可以对 Redis 进行设置， 让它在“ N 秒内数据集至少有 M 个改动”这一条件被满足时， 自动保存一次数据集。我们可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

save 60 1000

以上设置会让 Redis 在满足“ 60 秒内有至少有 1000 个键被改动”这一条件时， 自动保存一次数据集。

你也可以通过调用 [SAVE](http://doc.redisfans.com/server/save.html#save) 或者 [BGSAVE](http://doc.redisfans.com/server/bgsave.html#bgsave) ， 手动让 Redis 进行数据集保存操作。

### 4.快照的运作方式

当 Redis 需要保存 dump.rdb 文件时， 服务器执行以下操作：

1. Redis 调用 fork() ，同时拥有父进程和子进程。
2. 子进程将数据集写入到一个临时 RDB 文件中。
3. 当子进程完成对新 RDB 文件的写入时，Redis 用新 RDB 文件替换原来的 RDB 文件，并删除旧的 RDB 文件。

这种工作方式使得 Redis 可以从写时复制（copy-on-write）机制中获益。

### 5.如何恢复

将备份文件(dump.rdb)移动到redis安装目录并启动服务即可。

### 6.RDB的缺点

 如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。

 每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

### 7.总结

RDB是一个非常紧凑的文件，在保存RDB文件时父进程唯一需要做的就是fork出一个子进程，接下来的工作全部由子进程来做，父进程不需要再做其他io操作，所以RDB持久化方式可以最大化redis的性能，并且与aof相比，在恢复大的数据集的时候，rdb方式会更快一些。但rdb数据丢失风险大，需要经常fork子进程来保存数据集到硬盘上，当数据集比较大的时候，fork的过程是非常耗时的，可能会导致Redis在一些毫秒级不能响应客户端请求 。

## AOF持久化

AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。 AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。 Redis 还可以在后台对 AOF 文件进行重写（rewrite），使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小。

### 1.优点

* 使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync ，每秒钟一次 fsync ，或者每次执行写入命令时 fsync 。 AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（ fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。
* AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 seek ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， redis-check-aof 工具也可以轻易地修复这种问题。
* Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。
* AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。

### 2.AOF的持久化配置

可以通过修改配置文件来打开 AOF 功能：

appendonly yes

在Redis的配置文件中存在三种同步方式，它们分别是：

appendfsync **always**   #每次有数据修改发生时都会写入AOF文件。

appendfsync **everysec** #每秒钟同步一次，该策略为AOF的缺省策略。

appendfsync **no**         #从不同步。高效但是数据不会被持久化。

### 3.AOF重写

因为 AOF 的运作方式是不断地将命令追加到文件的末尾， 所以随着写入命令的不断增加， AOF 文件的体积也会变得越来越大。为了处理这种情况， Redis 可以在不打断服务客户端的情况下， 对 AOF 文件进行重写（rebuild）。执行 [**BGREWRITEAOF**](http://doc.redisfans.com/server/bgrewriteaof.html#bgrewriteaof)命令， Redis 将生成一个新的 AOF 文件， 这个文件包含重建当前数据集所需的最少命令。

AOF 重写和 RDB 创建快照一样，都巧妙地利用了写时复制机制。

以下是 AOF 重写的执行步骤：

1. Redis 执行 fork() ，现在同时拥有父进程和子进程。
2. 子进程开始将新 AOF 文件的内容写入到**临时文件**。（先写到临时文件最后再rename）。**遍历新进程的内存中数据，**每条记录有一条**set语句，重写aof文件的操作**，并没有读取旧的aof文件而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件。
3. 对于所有新执行的写入命令，父进程一边将它们累积到一个**内存缓存中**，一边将这些改动追加到现有 AOF 文件的末尾： 这样即使在重写的中途发生停机，现有的 AOF 文件也还是安全的。
4. 当子进程完成重写工作时，它给父进程发送一个信号，父进程在接收到信号之后，将内存缓存中的所有数据追加到新 AOF 文件的末尾。
5. Redis 原子地用新文件替换旧文件，之后所有命令都会直接追加到新 AOF 文件的末尾。

触发机制：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。**

### 4.AOF文件修复

服务器可能在程序正在对 AOF 文件进行写入时停机， 如果停机造成了 AOF 文件出错（corrupt）， 那么 Redis 在重启时会拒绝载入这个 AOF 文件， 从而确保数据的一致性不会被破坏。

当发生这种情况时， 可以用以下方法来修复出错的 AOF 文件：

1. 为现有的 AOF 文件创建一个备份。
2. 使用 Redis 附带的 redis-check-aof 程序，对原来的 AOF 文件进行修复。

### 5.AOF缺点

* 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。
* 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB。
* AOF 在过去曾经发生过这样的 bug ： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。 阻塞命令 [BRPOPLPUSH](http://doc.redisfans.com/list/brpoplpush.html#brpoplpush) 就曾经引起过这样的 bug 。测试套件里为这种情况添加了测试： 它们会自动生成随机的、复杂的数据集， 并通过重新载入这些数据来确保一切正常。 虽然这种 bug 在 AOF 文件中并不常见， 但是对比来说， RDB 几乎是不可能出现这种 bug 的。

### 6.总结

aof文件是一个只进行追加的日志文件，redis可以在aof文件体积变得过大时自动地在后台对aof文件进行重写，aof文件有序的保存了对数据库执行的所有写入操作，这些写入操作以redis协议的格式保存，因此，aof文件的内容非常容易被人读懂，对文件进行分析也很轻松。对相同的数据集来说,aof文件的体积通常要大于RDB文件的体积，根据所使用的fsync策略，aof的速度可能会慢于rdb。

# RDB和AOF的优势和劣势

## RDB的优势

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的劣势

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的优势

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

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）。

# redis的过期策略

## 1.redis设置过期时间

    expire key time(以秒为单位)--这是最常用的方式

    setex(String key, int seconds, String value)--字符串独有的方式

注：

除了字符串自己独有设置过期时间的方法外，其他方法都需要依靠expire方法来设置时间；

如果没有设置时间，那缓存就是永不过期；

 如果设置了过期时间，之后又想让缓存永不过期，使用persist key。

## 2.三种过期策略

### a.定时删除

        含义：在设置key的过期时间的同时，为该key创建一个定时器，让定时器在key的过期时间来临时，对key进行删除。

        优点：保证内存被尽快释放。

        缺点：

* 若过期key很多，删除这些key会占用很多的CPU时间，在CPU时间紧张的情况下，CPU不能把所有的时间用来做要紧的事儿，还需要去花时间删除这些key；
* 定时器的创建耗时，若为每一个设置过期时间的key创建一个定时器（将会有大量的定时器产生），性能影响严重。

### b.懒汉式删除

含义：key过期的时候不删除，每次通过key获取值的时候去检查是否过期，若过期，则删除，返回null。

优点：删除操作只发生在通过key取值的时候发生，而且只删除当前key，所以对CPU时间的占用是比较少的，而且此时的删除是已经到了非做不可的地步（如果此时还不删除的话，我们就会获取到了已经过期的key了）。

缺点：若大量的key在超出超时时间后，很久一段时间内，都没有被获取过，那么可能发生内存泄露（无用的垃圾占用了大量的内存）。

### c.定期删除

含义：每隔一段时间执行一次删除过期key操作

 优点：

* 通过限制删除操作的时长和频率，来减少删除操作对CPU时间的占用--处理"定时删除"的缺点
* 定期删除过期key--处理"懒汉式删除"的缺点

缺点：

* 在内存友好方面，不如"定时删除"（会造成一定的内存占用，但是没有懒汉式那么占用内存）
* 在CPU时间友好方面，不如"懒汉式删除"（会定期的去进行比较和删除操作，cpu方面不如懒汉式，但是比定时好）

**定期删除的难点**

合理设置删除操作的执行时长（每次删除执行多长时间）和执行频率（每隔多长时间做一次删除）（这个要根据服务器运行情况来定了），每次执行时间太长，或者执行频率太高对cpu都是一种压力。

        每次进行定期删除操作执行之后，需要记录遍历循环到了哪个标志位，以便下一次定期时间来时，从上次位置开始进行循环遍历。

## 3.Redis采用的过期策略

懒汉式删除+定期删除

对于定期删除，在程序中有一个全局变量current\_db来记录下一个将要遍历的库，假设有16个库，我们这一次定期删除遍历了10个，那此时的current\_db就是11，下一次定期删除就从第11个库开始遍历，假设current\_db等于15了，那么之后遍历就再从0号库开始（此时current\_db==0）。

**RDB对过期key的处理**

过期key对RDB没有任何影响

从内存数据库持久化数据到RDB文件:

持久化key之前，会检查是否过期，过期的key不进入RDB文件

从RDB文件恢复数据到内存数据库数据载入数据库之前，会对key先进行过期检查，如果过期，不导入数据库（主库情况）

**AOF对过期key的处理**

过期key对AOF没有任何影响

从内存数据库持久化数据到AOF文件：

当key过期后，还没有被删除，此时进行执行持久化操作（该key是不会进入aof文件的，因为没有发生修改命令）

当key过期后，在发生删除操作时，程序会向aof文件追加一条del命令（在将来的以aof文件恢复数据的时候该过期的键就会被删掉）

**AOF重写**

重写时，会先判断key是否过期，已过期的key不会重写到aof文件 。

# redis事务

## 什么是redis事务？

本质是一组命令的集合,可以一次执行多个命令，所有命令都会序列化，按顺序地串行化执行而不会被其它命令插入，不许插队。将一组需要一起执行的命令放到**multi**和**exec**两个命令之间。**multi**命令代表**事务开始**，**exec**命令代表**事务结束**，它们之间的命令是原子顺序执行的。

即一个队列中，一次性、顺序性、排他性的执行一系列命令。

## redis事务的相关命令？

**1. MULTI**

用于标记事务块的开始。Redis会将后续的命令逐个放入队列中，然后才能使用EXEC命令原子化地执行这个命令序列。

这个命令的运行格式如下所示：

MULTI

这个命令的返回值是一个简单的字符串，总是OK。

**2. EXEC**

在一个事务中执行所有先前放入队列的命令，然后恢复正常的连接状态。

当使用WATCH命令时，只有当受监控的键没有被修改时，EXEC命令才会执行事务中的命令，这种方式利用了检查再设置（CAS）的机制。

这个命令的运行格式如下所示：

EXEC

这个命令的返回值是一个数组，其中的每个元素分别是原子化事务中的每个命令的返回值。 当使用WATCH命令时，如果事务执行中止，那么EXEC命令就会返回一个Null值。

**3. DISCARD**

清除所有先前在一个事务中放入队列的命令，然后恢复正常的连接状态。

如果使用了WATCH命令，那么DISCARD命令就会将当前连接监控的所有键取消监控。

这个命令的运行格式如下所示：

DISCARD

这个命令的返回值是一个简单的字符串，总是OK。

**4. WATCH**

当某个事务需要按条件执行时，就要使用这个命令将给定的键设置为受监控的。

这个命令的运行格式如下所示：

WATCH key [key ...]

这个命令的返回值是一个简单的字符串，总是OK。

对于每个键来说，时间复杂度总是O(1)。

**5. UNWATCH**

清除所有先前为一个事务监控的键。

如果你调用了EXEC或DISCARD命令，那么就不需要手动调用UNWATCH命令。

这个命令的运行格式如下所示：

UNWATCH

这个命令的返回值是一个简单的字符串，总是OK。

时间复杂度总是O(1)。

## redis事务执行的三个阶段？

1. **开启**：以MULTI开始一个事务
2. **入队**：将多个命令入队到事务中，接到这些命令并不会立即执行，而是放到等待执行的事务队列里面
3. **执行**：由EXEC命令触发事务

## 为什么不支持回滚？

只有当被调用的Redis命令有语法错误时，这条命令才会执行失败（在将这个命令放入事务队列期间，Redis能够发现此类问题），或者对某个键执行不符合其数据类型的操作：实际上，这就意味着只有程序错误才会导致Redis命令执行失败，这种错误很有可能在程序开发期间发现，一般很少在生产环境发现。   
Redis已经在系统内部进行功能简化，这样可以确保更快的运行速度，因为Redis不需要事务回滚的能力。

## redis事务的三个特性

1. 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**。
2. 没有隔离级别的概念：队列中的命令没有提交之前都不会实际的被执行，因为事务提交前任何指令都不会被实际执行，也就不存在”事务内的查询要看到事务里的更新，在事务外查询不能看到”这个让人万分头痛的问题。
3. 不保证原子性：redis同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚。

## watch监控

**悲观锁(Pessimistic Lock)：**就是很悲观，每次去拿数据的时候都认为别人会修改，所以**每次在拿数据的时候都会上锁**，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

**乐观锁(Optimistic Lock)：**就是很乐观，每次去拿数据的时候都认为别人不会修改，所以**不会上锁**，但是**在更新的时候会判断一下在此期间别人有没有去更新这个数据**，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量。

**乐观锁策略**:**提交版本必须大于记录当前版本才能执行更新。**

**watch指令，类似于乐观锁。**

Redis使用WATCH命令实现事务的“检查再设置”（CAS）行为。

作为WATCH命令的参数的键会受到Redis的监控，Redis能够检测到它们的变化。在执行EXEC命令之前，如果Redis检测到至少有一个键被修改了，那么整个事务便会中止运行，然后EXEC命令会返回一个Null值，提醒用户事务运行失败。

# redis的主从复制

## 概念

我们所说的主从复制，即主机数据更新后根据配置和策略，自动同步到备机的master/slaver机制，Master以写为主，Slave以读为主。实现了**读写分离**。

## 主从复制的作用（好处）

1.读写分离。读写分离，不仅可以提高服务器的负载能力，并且可以根据读请求的规模自由增加或减少从库的数量；

2.容灾恢复。数据被赋值了好几份，就算有一台机器出现故障，也可以使用其他的机器的数据快速恢复。

## 配置

配从不配主。在redis中，要实现主从配置非常简单，只需要在从服务器配置中把 【 salveof 主库地址 主库端口】 就行，主服务器不需要修改。

## 主从复制的原理

https://blog.csdn.net/qq\_34021712/article/details/72026313

**1.全量同步**

　　Redis全量复制一般发生在Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份。具体步骤如下：

　　1）从服务器连接主服务器，发送SYNC命令；

　　2）主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；

　　3）主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；

　　4）从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；

　　5）主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；

6）从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令。

完成上面几个步骤后就完成了从服务器数据初始化的所有操作，从服务器此时可以接收来自用户的读请求。

**2.增量同步**

　　Redis增量复制是指Slave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。

增量复制的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令。

**3 Redis主从同步策略**

主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。当然，如果有需要，slave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步。

**4 注意点**

如果多个Slave断线了，需要重启的时候，因为只要Slave启动，就会发送sync请求和主机全量同步，当多个同时出现的时候，可能会导致Master IO剧增宕机。

### 乐观复制

Redis采用了乐观复制(optimistic replication)的复制策略，容忍在一定时间内主从库的内容是不同的，但是两者的数据会最终同步。具体来说，Redis在主从库之间复制数据的过程本身是异步的，这意味着，主库执行完客户端请求的命令后会立即将命令在主库的执行结果返回给客户端，并异步地将命令同步给从库，而不会等待从库接收到该命令后再返回给客户端。

        这一特性保证了启用复制后主库的性能不会受到影响，但另一方面也会产生一个主从库数据不一致的时间窗口，当主库执行了一条写命令后，主库的数据已经发生的变动，然而在主库将该命令传送给从库之前，如果两个数据库之间的网络连接断开了，此时二者之间的数据就会是不一致的。

        从这个角度来看，主库是无法得知某个命令最终同步给了多少个从库的，不过 Redis 提供了两个配置选项，来限制只有当数据至少同步给指定数量的从库时，主库才是可写的：

min-slaves-to-write 3

min-slaves-max-lag 10

min-slaves-to-write”表示只有当3个（或以上）的从库连接到主库时，主库才是可写的，否则会返回错误；

min-slaves-max-lag”表示允许从库最长失去连接的时间，如果从库最后与主库联系（即发送“replconf  ack”命令）的时间小于这个值，则认为从库还在保持与主库的连接。

//一主二仆

   一个Master，两个Slave，Slave只能读不能写；当Slave与Master断开后需要重新slave of连接才可建立之前的主从关系；Master挂掉后，Master关系依然存在，Master重启即可恢复。

 //薪火相传

上一个Slave可以是下一个Slave的Master，Slave同样可以接收其他slaves的连接和同步请求，那么该slave作为链条中下一个slave的Master，如此可以有效减轻Master的写压力。如果slave中途变更转向，会清除之前的数据，重新建立最新的。

//反客为主

当Master挂掉后，Slave可键入命令 slaveof no one使当前redis停止与其他Master redis数据同步，转成Master redis。

### 哨兵模式（sentinel）

反客为主的自动版，能够后台监控Master库是否故障，如果故障了根据投票数自动将slave库转换为主库。一组sentinel能同时监控多个Master。

使用步骤：

1、在Master对应redis.conf同目录下新建sentinel.conf文件，名字绝对不能错；

2、配置哨兵，在sentinel.conf文件中填入内容：

     sentinel monitor 被监控数据库名字（自己起名字） ip port 1

      说明：上面最后一个数字1，表示主机挂掉后slave投票看让谁接替成为主机，得票数多少后成为主机。

 3、启动哨兵模式：

       命令键入：redis-sentinel  /myredis/sentinel.conf

       注：上述sentinel.conf路径按各自实际情况配置

启动哨兵的过程：

redis-sentinel /myredis/sentinel.conf 正常主从演示 ---->原有的master挂了---->投票新选------>重新主从继续开工，info replication查查。

//问题：如果之前的master重启回来，会不会双master冲突？ 不会。

### 主从复制的缺点

延时，由于所有的写操作都是在Master上操作，然后同步更新到Slave上，所以从Master同步到Slave机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，Slave机器数量的增加也会使得这个问题更加严重。

//

1 .切入点问题？slave1、slave2是从头开始复制还是从切入点开始复制?比如从k4进来，那之前的123是否也可以复制

从头开始

2 .从机是否可以写？set可否？

不可以

3 .主机shutdown后情况如何？从机是上位还是原地待命

待命，等主机重启之后继续

4. 主机又回来了后，主机新增记录，从机还能否顺利复制？

能

5 .其中一台从机down后情况如何？依照原有它能跟上大部队吗？

不能，得重新监听主机，或者是直接配置进入了conf之中

//

redis常见性能问题和解决方案：

(1) 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

(Master写内存快照，save命令调度rdbSave函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以Master最好不要写内存快照;AOF文件过大会影响Master重启的恢 复速度)

(2) 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

(3) 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

(4) 尽量避免在压力很大的主库上增加从库

(5) 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...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

# redis的淘汰策略

**Redis内存淘汰指的是用户存储的一些键被可以被Redis主动地从实例中删除，从而产生读miss的情况**。那么Redis为什么要有这种功能？这就是我们需要探究的设计初衷。Redis最常见的两种应用场景为缓存和持久存储，首先要明确的一个问题是内存淘汰策略更适合于那种场景？是持久存储还是缓存？

内存的淘汰机制的初衷是为了更好地使用内存，用一定的缓存miss来换取内存的使用效率。

作为Redis用户，我如何使用Redis提供的这个特性呢？看看下面配置：

maxmemory配置指令

# maxmemory <bytes>

 maxmemory 用于指定 Redis 能使用的最大内存。既可以在 redis.conf 文件中设置, 也可以在运行过程中通过 [CONFIG SET](https://redis.io/commands/config-set) 命令动态修改。

将 maxmemory 设置为 0, 则表示不进行内存限制。当然, 对32位系统来说有一个隐性的限制条件: 最多 3GB 内存。

当内存使用达到最大限制时, 如果需要存储新数据, 根据配置的策略(**policies**)的不同, Redis可能直接返回错误信息, 或者删除部分老的数据。驱逐过程如下：

1.      客户端发起了需要申请更多内存的命令（如set）。

2.      Redis检查内存使用情况，如果已使用的内存大于maxmemory则开始根据用户配置的不同淘汰策略来淘汰内存（key），从而换取一定的内存。

3.      如果上面都没问题，则这个命令执行成功。

当前版本，Redis 3.0 支持的策略包括:

* **noeviction**: 不删除策略, 达到最大内存限制时, 如果需要更多内存, 直接返回错误信息。
* **allkeys-lru**: 所有key通用; 优先删除最近最少使用(less recently used ,LRU) 的 key。
* **volatile-lru**: 只限于设置了 **expire** 的部分; 优先删除最近最少使用(less recently used ,LRU) 的 key。
* **allkeys-random**: 所有key通用; 随机删除一部分 key。
* **volatile-random**: 只限于设置了 **expire** 的部分; 随机删除一部分 key。
* **volatile-ttl**: 只限于设置了 **expire** 的部分; 优先删除剩余时间(time to live,TTL) 短的key。

需要根据系统的特征, 来选择合适的驱逐策略。 当然, 在运行过程中也可以通过命令动态设置驱逐策略, 并通过 [INFO](https://redis.io/commands/info) 命令监控缓存的 miss 和 hit, 来进行调优。

* allkeys-lru：如果我们的应用对缓存的访问符合幂律分布（也就是存在相对热点数据），或者我们不太清楚我们应用的缓存访问分布状况，我们可以选择allkeys-lru策略。
* allkeys-random：如果我们的应用对于缓存key的访问概率相等，则可以使用这个策略。
* volatile-ttl：这种策略使得我们可以向Redis提示哪些key更适合被eviction。

另外，volatile-lru策略和volatile-random策略适合我们将一个Redis实例既应用于缓存和又应用于持久化存储的时候，然而我们也可以通过使用两个Redis实例来达到相同的效果，值得一提的是将key设置过期时间实际上会消耗更多的内存，因此我们建议使用allkeys-lru策略从而更有效率的使用内存。

Redis 使用的并不是完全LRU算法。自动驱逐的 key , 并不一定是最满足LRU特征的那个. 而是通过近似LRU算法, 抽取少量的 key 样本, 然后删除其中访问时间最古老的那个key。

//过期策略和淘汰策略的区别？？

Redis的过期策略就是指当Redis中缓存的key过期了，Redis如何处理。

Redis的内存淘汰策略是指在Redis的用于缓存的内存不足时，怎么处理需要新写入且需要申请额外空间的数据。

# redis为什么是单线程

因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了。

Redis操作的是内存中的数据结构。如果在多线程中操作，那就需要为这些**对象加锁**。所以使用多线程可以提高性能，但是每个线程的**效率严重下降了**，而且程序的逻辑严重**复杂化。**

//CPU成为Redis瓶颈解决方法

多起几个Redis进程就好了。Redis是key-value数据库，又不是关系数据库，数据之间没有约束。只要客户端分清哪些key放在哪个Redis进程上就可以了。redis-cluster可以帮你做的更好。

**单进程模型来处理客户端的请求**。对读写等事件的响应是通过**对epoll函数的包装来做到的**。Redis的实际处理速度完全依靠主进程的执行效率。

**epoll**是**Linux内核为处理大批量文件描述符而作了改进的epoll**，是Linux下**多路复用IO接口select/poll的增强版本**，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率。

# redis为什么这么快？

1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，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，对数据操作也简单，Redis中的数据结构是专门进行设计的；

3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，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IO；

5、使用底层模型不同，它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。

以上几点都比较好理解，下边我们针对多路 I/O 复用模型进行简单的探讨：

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 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll 是只轮询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”指的是复用同一个线程。**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。

# redis 分区

面试问题：如何将数据分布在不同的redis？

## 1.概念

Redis Partitioning即Redis分区，简单的说就是将数据分布到不同的redis实例中，因此对于每个redis实例所存储的内容仅仅是所有内容的一个子集。

## 2.为什么要分区

我们为什么要分区？分区的动机是什么？通常来说，Redis分区的好处大致有如下两个方面：

1. 性能的提升。单机Redis的网络I/O能力和计算资源是有限的，将请求分散到多台机器，充分利用多台机器的计算能力和网络带宽，有助于提高Redis总体的服务能力。
2. 存储的横向扩展。即使Redis的服务能力能够满足应用需求，但是随着存储数据的增加，单台机器受限于机器本身的存储容量，将数据分散到多台机器上存储使得Redis服务可以横向扩展。

## 3.如何分区？

实际应用中有很多分区的具体策略，举个例子，假设我们已经有了一组四个Redis实例分别为R0、R1、R2、R3，另外我们有一批代表用户的键，如：user:1，user:2，……等等，其中“user:”后面的数字代表的是用户的ID，我们要做的事情是把这些键分散存储在这四个不同的Redis实例上。怎么做呢？

### 范围分区

所谓范围分区，就是将一个范围内的key都映射到同一个Redis实例中。具体做法如下：

我们可以将用户ID从0到10000的用户数据映射到R0实例，而将用户ID从10001到20000的对象映射到R1实例，依次类推。

这种方法虽然简单，但是在实际应用中是很有效的，不过还是有问题：

* 我们需要一张表，这张表用来存储用户ID范围到Redis实例的映射关系，比如用户ID0-10000的是映射到R0实例……。
* 我们不仅需要对这张表进行维护，而且对于每种对象类型我们都需要一个这样的表，比如我们当前存储的是用户信息，如果存储的是订单信息，我们就需要再建一张映射关系表。
* 如果我们想要存储的数据的key并不能按照范围划分怎么办，比如我们的key是一组uuid，这个时候就不好用范围分区了。

### 哈希分区

哈希分区跟范围分区相比一个明显的优点是哈希分区适合任何形式的key，而不像范围分区一样需要key的形式为object\_name:<id>，而且分区方法也很简单，一个公式就可以表达：

id=hash(key)%N

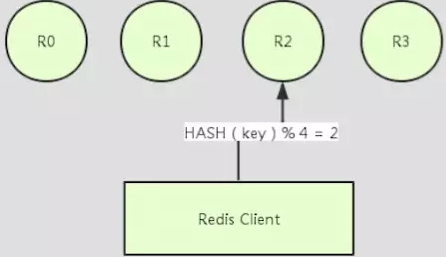
其中id代表Redis实例的编号，公式描述的是首先根据key和一个hash函数（如crc32函数）计算出一个数值型的值。然后哈希结果进行取模，取模的目的是计算出一个介于0到3之间的值，因此这个值才可以被映射到我们的一台Redis实例上面。比如93024922%4结果是2，我们就会知道foobar将要被存储在R2上面。

## 4.分区的实现

分区可以在redis软件栈的不同部分被实现，我们来看看下面几种：

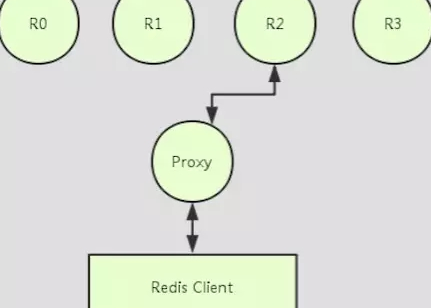
### 客户端实现

客户端实现即key在redis客户端就决定了要被存储在那台Redis实例中，见下图：



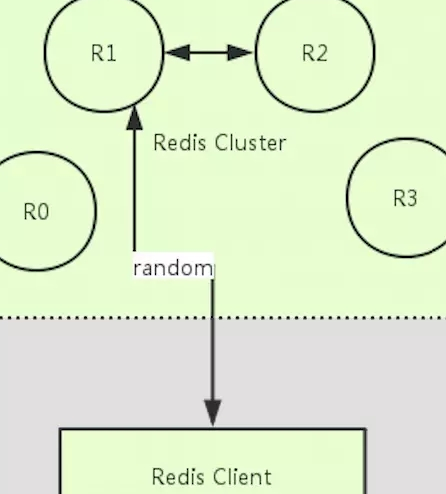
### 代理实现

代理实现即客户端将请求发往代理服务器，代理服务器实现了Redis协议，因此代理服务器可以代理客户端和Redis服务器通信。代理服务器通过配置的分区schema来将客户端的请求转发到正确的Redis实例中，同时将反馈消息返回给客户端。代理实现Redis分区示意图如下：



### 查询路由

查询路由是Redis Cluster实现的一种Redis分区方式：



查询路由的过程中，我们可以将查询请求随机的发送到任意一个Redis实例，这个Redis实例负责将请求转发至正确的Redis实例中。Redis集群实现了一个通过和客户端协作的hybrid来做查询路由。

## 5.Redis分区的缺点

* 多键操作是不被支持的，比如我们将要批量操作的键被映射到了不同的Redis实例中。
* 多键的Redis事务是不被支持的。
* 分区的最小粒度是键，因此我们不能将关联到一个键的很大的数据集映射到不同的实例。
* 当应用分区的时候，数据的处理是非常复杂的，比如我们需要处理多个rdb/aof文件，将分布在不同实例的文件聚集到一起备份。
* 添加和删除机器是很复杂的，例如Redis集群支持几乎运行时透明的因为增加或减少机器而需要做的rebalancing，然而像客户端和代理分区这种方式是不支持这种功能的。

## 6.Pre-Sharding

通过上面的介绍，我们知道Redis分区应用起来是有问题的，除非我们只是使用Redis当做缓存，否则对于增加机器或删除机器是非常麻烦的。

通常我们Redis容量变动在实际应用中是非常常见的，比如今天我需要10台Redis机器，明天可能就需要50台机器了。鉴于Redis是很轻量级的服务（每个实例仅仅占用1M），对于上面的问题一种简单的解决办法是：

我们可以开启多个Redis实例，尽管是一台物理机器，我们在刚开始的时候也可以开启多个实例。我们可以从中选择一些实例，比如32或64个实例来作为我们的工作集群。当一台物理机器存储不够的时候，我们可以将一般的实例移动到我们的第二台物理机上，依次类对，我们可以保证集群中Redis的实例数不变，又可以达到扩充机器的目的。

### 实例移动的步骤

 在新的物理机上启动一个新的Redis实例。

 将新的物理机作为要移动的那台的slave机器。

 停止客户端。

 更新将要被移动的那台Redis实例的IP地址。（旧的ip地址）

 对于slave机器发送SLAVEOF ON ONE命令。

 使用新的IP启动Redis客户端。

 关闭不再使用的那个Redis实例

# redis cluster分区（深入）

Redis Cluster本身提供了自动将数据分散到Redis Cluster不同节点的能力，分区实现的关键点问题包括：如何将数据自动地打散到不同的节点，使得不同节点的存储数据相对均匀；如何保证客户端能够访问到正确的节点和数据；如何保证重新分片的过程中不影响正常服务。

## ****认识Redis Cluster****

Redis Cluster是由多个同时服务于一个数据集合的Redis实例组成的整体，对于用户来说，用户只关注这个数据集合，而整个数据集合的某个数据子集存储在哪个节点对于用户来说是透明的。Redis Cluster具有分布式系统的特点，也具有分布式系统如何实现高可用性与数据一致性的难点。

### Redis Cluster特点如下：

* 所有的节点相互连接；
* 集群消息通信通过集群总线通信，集群总线端口大小为客户端服务端口+10000，这个10000是固定值；
* 节点与节点之间通过二进制协议进行通信；
* 客户端和集群节点之间通信和通常一样，通过文本协议进行；
* 集群节点不会代理查询；

## ****Redis Cluster分区实现原理****

### ****槽（slot）概念****

Redis Cluster中有一个16384长度的槽的概念，他们的编号为0、1、2、3……16382、16383。这个槽是一个虚拟的槽，并不是真正存在的。正常工作的时候，Redis Cluster中的每个Master节点都会负责一部分的槽，当有某个key被映射到某个Master负责的槽，那么这个Master负责为这个key提供服务，至于哪个Master节点负责哪个槽，这是可以由用户指定的，也可以在初始化的时候自动生成（redis-trib.rb脚本）。这里值得一提的是，在Redis Cluster中，只有Master才拥有槽的所有权，如果是某个Master的slave，这个slave只负责槽的使用，但是没有所有权。Redis Cluster怎么知道哪些槽是由哪些节点负责的呢？某个Master又怎么知道某个槽自己是不是拥有呢？

### ****位序列结构****

Master节点维护着一个16384/8字节的位序列，Master节点用bit来标识对于某个槽自己是否拥有。比如对于编号为1的槽，Master只要判断序列的第二位（索引从0开始）是不是为1即可。

集群同时还维护着槽到集群节点的映射，是由长度为16384类型为节点的数组实现的，槽编号为数组的下标，数组内容为集群节点，这样就可以很快地通过槽编号找到负责这个槽的节点。位序列这个结构很精巧，即不浪费存储空间，操作起来又很便捷。

**键空间分布基本算法**

这里讲的是Redis Cluster如何将键空间分布在不同的节点的，键空间意为Redis Cluster所拥有用户所有数据集合的键的取值范围，这个范围叫做键空间。提到空间分布，必然会想到哈希算法，没错，通过哈希算法再加上取模运算可以将一个值固定地映射到某个区间，在这里，这个区间叫做slots，区间由连续的slot组成。在Redis Cluster中，我们拥有16384个slot，这个数是固定的，我们存储在Redis Cluster中的所有的键都会被映射到这些slot中，下面讲讲Redis Cluster是如何做映射的。

键到slot的基本映射算法如下：

HASH\_SLOT = CRC16(key) mod 16384

经过简单的计算就得到了当前key应该是存储在哪个slot里面，值得注意的是，指定的key会被存储在哪个slot，这个关系是铁打不变的。如果我提交了一批命令，往Redis中存储一批键，那么这些键一般会被映射到不同的slot，而不同的slot又可能由Redis Cluster中不同的节点服务，这样就和的预期有点不同，有没有办法将这批键映射到同一个slot呢？答案是可以。

**键哈希标签原理**

键哈希标签是一种可以让用户指定将一批键都能够被存放在同一个槽中的实现方法，用户唯一要做的就是按照既定规则生成key即可，这个规则是这样的，如果我有对于同一个用户有两种不同含义的两份数据，我只要将他们的键设置为下面即可:

abc{userId}def和ghi{userId}jkl

redis在计算槽编号的时候只会获取{}之间的字符串进行槽编号计算，这样由于上面两个不同的键，{}里面的字符串是相同的，因此他们可以被计算出相同的槽，相关代码如下：

unsigned int keyHashSlot(char \*key, int keylen) {

    int s, e;

    for (s = 0; s < keylen; s++)

        if (key[s] == '{') break;

    if (s == keylen) return crc16(key,keylen) & 0x3FFF;

    for (e = s+1; e < keylen; e++)

        if (key[e] == '}') break;

    if (e == keylen || e == s+1) return crc16(key,keylen) & 0x3FFF;

    return crc16(key+s+1,e-s-1) & 0x3FFF;

}

客户端是怎么在Redis Cluster中找到正确的节点的呢？下面看看。

**重定向客户端**

Redis Cluster并不会代理查询，那么如果客户端访问了一个key并不存在的节点，这个节点是怎么处理的呢？比如我想获取key为msg的值，msg计算出来的槽编号为254，当前节点正好不负责编号为254的槽，那么就会返回客户端下面信息：

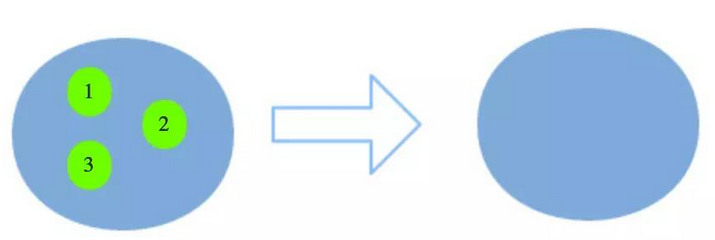
GET msg

-MOVED 254 127.0.0.1:6381

表示客户端想要的254槽由运行在IP为127.0.0.1，端口为6381的Master实例服务。如果根据key计算得出的槽恰好由当前节点负责，则当期节点会立即返回结果。这里明确一下，没有代理的Redis Cluster可能会导致客户端两次连接急群中的节点才能找到正确的服务，推荐客户端缓存连接，这样最坏的情况是两次往返通信。

## ****重新分片（Resharding）****

重新分片意为槽到集群节点的映射关系要改变，不变的是键到槽的映射关系，因此当重新分片的时候，如果槽中有键，那么键也是要被移动到新的节点的。下面看看重新分片是怎么做的，假如我们有一批槽需要从一个Master节点移动到另一个Master节点：



这里简化模型，假设这批待迁移的槽编号为1、2、3，并假设左边的节点为MasterA节点，右边的节点为MasterB节点。

### 槽迁移的过程

槽迁移的过程中有一个不稳定状态，这个不稳定状态会有一些规则，这些规则定义客户端的行为，从而使得Redis Cluster不必宕机的情况下可以执行槽的迁移。

**MIGRATING状态**

本例中MIGRATING状态是发生在MasterA节点中的一种槽的状态，预备迁移槽的时候槽的状态首先会变为MIGRATING状态，这种状态的槽会实际产生什么影响呢?当客户端请求的某个Key所属的槽处于MIGRATING状态的时候，影响有下面几条：

1. 如果Key存在则成功处理 。
2. 如果Key不存在，则返回客户端ASK，仅当这次请求会转向另一个节点，并不会刷新客户端中node的映射关系，也就是说下次该客户端请求该Key的时候，还会选择MasterA节点 。
3. 如果Key包含多个命令，如果都存在则成功处理，如果都不存在，则返回客户端ASK，如果一部分存在，则返回客户端TRYAGAIN，通知客户端稍后重试，这样当所有的Key都迁移完毕的时候客户端重试请求的时候回得到ASK，然后经过一次重定向就可以获取这批键。

**IMPORTING状态**

本例中的IMPORTING状态是发生在MasterB节点中的一种槽的状态，预备将槽从MasterA节点迁移到MasterB节点的时候，槽的状态会首先变为IMPORTING。IMPORTING状态的槽对客户端的行为有下面一些影响：

1. 正常命令会被MOVED重定向，如果是ASKING命令则命令会被执行，从而Key没有在老的节点已经被迁移到新的节点的情况可以被顺利处理；
2. 如果Key不存在则新建；
3. 没有ASKING的请求和正常请求一样被MOVED，这保证客户端node映射关系出错的情况下不会发生写错；

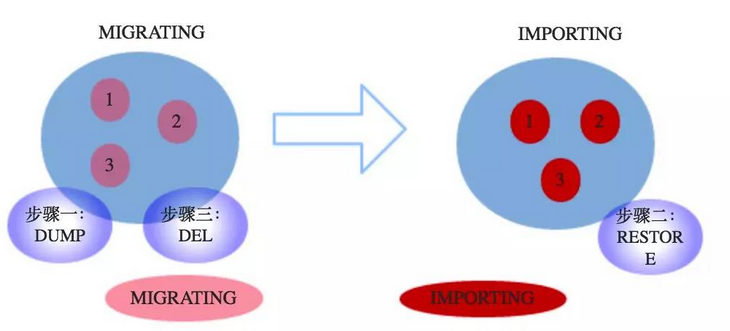
**键空间迁移**

键空间迁移是指当满足了槽迁移前提的情况下，我们就可以通过相关命令将槽1、2、3中的键空间从MasterA节点转移到MasterB节点，这个过程真正实现数据转移。相关命令：

MIGRATE

1. DUMP
2. RESTORE
3. DEL

MIGRATE命令通过三步将数据转移，示意图如下：



经过上面三步可以将键空间数据迁移，然后再将处于MIGRATING和IMPORTING状态的槽变为常态即可，完成整个重新分片的过程。然而MIGRATE并不是原子的，如果在MIGRATE出现错误的情况可能会导致下面问题：

* 键空间在两个节点都存在；
* 键空间只存在第一个节点；

# redis的跳跃表

跳跃表(skiplist)是一种有序数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。

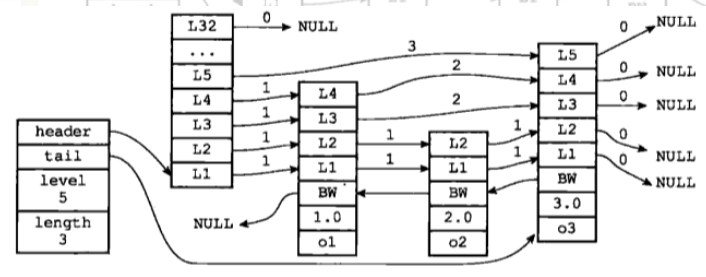
跳跃表支持平均O(logN)、最坏O(N)复杂度的节点查找，还可以通过顺序性操作来批量处理节点。

在大部分情况下，跳跃表的效率可以和平衡树相媲美，并且因为跳跃表的实现比平衡树要来得更为简单，所以有不少程序都使用跳跃表来代替平衡树。

Redis只在两个地方用到了跳跃表，一个是实现有序集合键，另一个是在集群节点中用作内部数据结构，除此之外，跳跃表在Redis里面没有其他用途。

## 跳跃表的实现

Redis的跳跃表由redis.h/zskiplistNode和redis.h/zskiplist两个结构定义，其中zskiplistNode结构用于表示跳跃表节点，而zskiplist结构则用于保存跳跃表节点的相关信息。



### zskiplist结构

该结构包含以下属性：

* header：指向跳跃表的表头节点
* tail：指向跳跃表的表尾节点
* level：记录目前跳跃表内，层数最大的那个节点的层数(表头节点的层数不计算在内)
* length：记录跳跃表的长度，也即是，跳跃表目前包含节点的数量(表头节点不计算在内)

### zskiplist结构的定义如下：

typedef struct zskiplist {

struct zskiplistNode \*header, \*tail; //header指向跳跃表的表头节点，tail指向跳跃表的表尾节点

unsigned long length; //记录跳跃表的长度，也即是，跳跃表目前包含节点的数量(表头节点不计算在内)

int level; //记录目前跳跃表内，层数最大的那个节点的层数(表头节点的层数不计算在内)

} zskiplist;

### zskiplistNode结构

该结构包含以下属性：

* 层(level)：节点中用L1、L2、L3等字样标记节点的各个层，L1代表第一层，L2代表第二层，依次类推。每个层都带有两个属性：前进指针和跨度。前进指针用于访问位于表尾方向的其他节点，而跨度则记录了前进指针所指向节点和当前节点的距离。在上面的图片中，连线上带有数字的箭头就代表前进指针，而那个数字就是跨度。当程序从表头向表尾进行遍历时，访问会沿着层的前进指针进行。
* 后退(backward)指针：节点中用BW字样标记节点的后退指针，它指向位于当前节点的前一个节点。后退指针在程序从表尾向表头遍历时使用。
* 分值(score)：各个节点中的1.0、2.0和3.0是节点所保存的分值。在跳跃表中，节点按各自所保存的分值从小到大排列。
* 成员对象(obj)：各个节点中的o1、o2和o3是节点所保存的成员对象。

### 跳跃表节点的结构定义：

/\* ZSETs use a specialized version of Skiplists \*/

typedef struct zskiplistNode {

robj \*obj; /\*成员对象\*/

double score; /\*分值\*/

struct zskiplistNode \*backward; /\*后退指针\*/

struct zskiplistLevel { /\*层\*/

struct zskiplistNode \*forward; /\*前进指针\*/

unsigned int span; /\*跨度\*/

} level[];

} zskiplistNode;

### 1、分值和成员

  节点的分值(score属性)是一个double类型的浮点数，跳跃表中的所有节点都按分值从小到大来排序。

  节点的成员对象(obj属性)是一个指针，它指向一个字符串对象，而字符串对象则保存着一个SDS值。

在同一个跳跃表中，各个节点保存的成员对象必须是唯一的，但是多个节点保存的分值却可以是相同的：分至相同的节点将按照成员对象在字典中的大小来进行排序，成员对象较小的节点会排在前面(靠近表头的方向)，而成员对象较大的节点则会排在后面(靠近表尾的方向)。

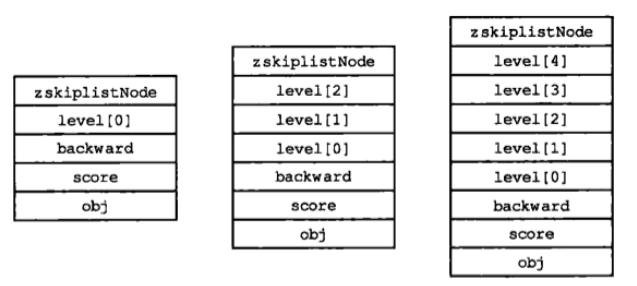
### 2、后退指针

  节点的后退指针(backward属性)用于从表尾向表头方向访问节点：跟可以一次跳过多个节点的前进指针不同，因为每个节点只有一个后退指针，所以每次只能后退至前一个节点。

### 3、层

跳跃表节点的level数组可以包含多个元素，每个元素都包含一个指向其他节点的指针，程序可以通过这些层来加快访问其他节点的速度，一般来说，层的数量越多，访问其他节点的速度就越快。

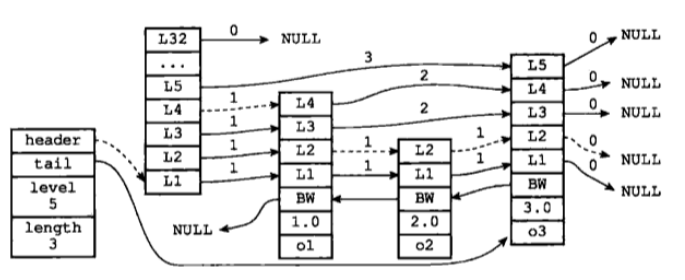
下图分别展示了三个高度为1层、3层和5层的节点，因为C语言的数组索引总是从0开始的，所以节点的第一层是level[0]，而第二层是level[1]，依次类推。



### 4、前进指针

每个层都有一个指向表尾方向的前进指针(level[i].forward属性)，用于从表头向表尾方向访问节点。下图用虚线表示出了程序从表头向表尾方向。

下图用虚线表示出了程序从表头向表尾方向，遍历跳跃表中所有节点的路径：



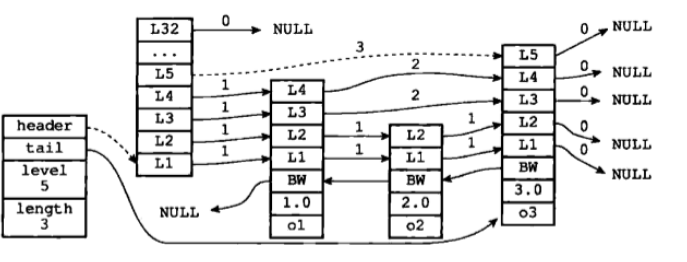
### 5、跨度

  层的跨度(level[i].span属性)用于记录两个节点之间的距离：

* 两个节点之间的跨度越大，它们相距得就越远。
* 指向NULL的所有前进指针的跨度都为0，因为它们没有连向任何节点。

跨度实际上是用来计算排位(rank)的：在查找某个节点的过程中，将沿途访问过的所有层的跨度累计起来，得到的结果就是目标节点在跳跃表中的排位。

举个例子，下图用虚线标记了在跳跃表中查找分值为3.0、成员对象为o3的节点时，沿途经历的层：查找的过程只经过了一个层，并且层的跨度为3，所以目标节点在跳跃表中的排位为3。



# redis场景实现点赞

采用的是redis bitmap来实现。

## bitmap简介

**bitmap**

bitmap是一连串的二进制数字(0,1)，每一位所在的位置为偏移(offset)，在bitmap上可以执行AND,OR,XOR以及其他操作。

**位图计数**

位图计数的意思是统计bitmap中值为1的位的个数，位图计数的效率是很高的。

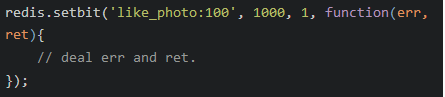
## redis bitmap

redis中允许使用二进制的Key和二进制的Value，bitmap就是二进制的Value。

**点赞/取消点赞**

假设用户的数字id为1000，对照片id为100的照片点赞。首先根据照片id生成赞数据存储的redis key，比如生成策略为like\_photo:{photo\_id}，id为1000的用户点赞，只需要将like\_photo:100的第1000位置为1即可（取消赞则置为0）。

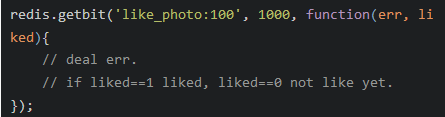
[redis setbit](http://redis.io/commands/setbit)操作的时间复杂度为O(1)，所以这种点赞方式十分高效。



**当前是否点赞**

用户打开图片的时候需要查询当前是否点赞过该照片，查询是否点赞可以通过redis getbit操作来实现。比如查询用户id为1000的用户是否点赞过照片id为100的照片，只需要对like\_photo:100bitmap的第1000位取值即可。

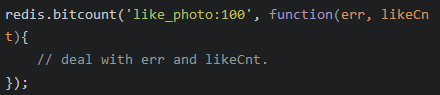
[redis getbit](http://redis.io/commands/getbit)操作的时间复杂度同样是O(1)。



**查询点赞总次数**

比如需要显示照片id为100的照片的获赞次数，只需要对like\_photo:100bitmap进行位图计数操作即可。

[redis bitcount](http://redis.io/commands/bitcount)操作的时间复杂度虽然是O(N)的，但是大部分数据量的情况下是不需要担心bitcount效率问题的。



# redis事务跟mysql事务的区别，redis的qps能达到多少？



### ****事务的ACID 性质****

**1.原子性：**对于一个存在问题的命令，如果在入队的时候就已经知道其出错，整个事务内的命令将都不会被执行（其后续的命令依然可以入队）（**此处可以实现原子性**）；如果这个错误命令在入队的时候并没有报错，而是在执行的时候出错了，那么redis默认跳过这个命令执行后续命令。也就是说，redis**只实现了部分事务**（并不能保证事务的原子性）。

**2.隔离性：**单独的隔离操作：事务中的所有命令会被序列化、按顺序执行，在执行的过程中不会被其他客户端发送来的命令打断。

**3.一致性：**事务的一致性指数据库中的数据全部符合数据库的规范，不会出现不符合规范的数据存在。Redis通过严格的错误检测保证事务的一致性。   
　　第一、入队时的错误会直接导致整个事务失败，保证一致性。   
　　第二、执行时错误会跳过错误命令，继续执行其他命令。   
　　第三、服务器重启，如果有持久化则通过RDB、AOF文件恢复数据，否则数据库为空。

**4.持久性：**当redis 服务器使用AOF 持久化模式并appendfsync 设置为always 时，程序执行sql 后会调用sync 函数将数据保存到硬盘里，因此redis 事务也可以具有持久性（Durability）。

qps：Query Per Second

每秒查询率QPS是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。

# 进行redis数据库写时加锁了吗？（redis加锁）

### redis加锁的几种实现

redis能用的加锁命令分别是INCR、SETNX、SET。

### 1. 第一种锁命令INCR

这种加锁的思路是， key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCR 操作进行加一。   
然后其它用户在执行 INCR 操作进行加一时，如果返回的数大于 1 ，说明这个锁正在被使用当中。

1、 客户端A请求服务器获取key的值为1表示获取了锁

2、 客户端B也去请求服务器获取key的值为2表示获取锁失败

3、 客户端A执行代码完成，删除锁

4、 客户端B在等待一段时间后在去请求的时候获取key的值为1表示获取锁成功

5、 客户端B执行代码完成，删除锁

$redis->incr($key);

$redis->expire($key, $ttl); //设置生成时间为1秒

**2.第二种锁SETNX**

这种加锁的思路是，如果 key 不存在，将 key 设置为 value   
如果 key 已存在，则 SETNX 不做任何动作

1、 客户端A请求服务器设置key的值，如果设置成功就表示加锁成功

2、 客户端B也去请求服务器设置key的值，如果返回失败，那么就代表加锁失败

3、 客户端A执行代码完成，删除锁

4、 客户端B在等待一段时间后在去请求设置key的值，设置成功

5、 客户端B执行代码完成，删除锁

$redis->setNX($key, $value);

$redis->expire($key, $ttl);

### 3. 第三种锁SET

上面两种方法都有一个问题，会发现，都需要设置 key 过期。那么为什么要设置key过期呢？如果请求执行因为某些原因意外退出了，导致创建了锁但是没有删除锁，那么这个锁将一直存在，以至于以后缓存再也得不到更新。于是乎我们需要给锁加一个过期时间以防不测。   
但是借助 Expire 来设置就不是原子性操作了。所以还可以通过事务来确保原子性，但是还是有些问题，所以官方就引用了另外一个，使用 SET 命令本身已经从版本 2.6.12 开始包含了设置过期时间的功能。

1、 客户端A请求服务器设置key的值，如果设置成功就表示加锁成功

2、 客户端B也去请求服务器设置key的值，如果返回失败，那么就代表加锁失败

3、 客户端A执行代码完成，删除锁

4、 客户端B在等待一段时间后在去请求设置key的值，设置成功

5、 客户端B执行代码完成，删除锁

$redis->set($key, $value, array('nx', 'ex' => $ttl)); //ex表示秒

### 4.其它问题

虽然上面一步已经满足了我们的需求，但是还是要考虑其它问题：  
1、 redis发现锁失败了要怎么办？中断请求还是循环请求？   
2、 循环请求的话，如果有一个获取了锁，其它的在去获取锁的时候，是不是容易发生抢锁的可能？   
3、 锁提前过期后，客户端A还没执行完，然后客户端B获取到了锁，这时候客户端A执行完了，会不会在删锁的时候把B的锁给删掉？

### 5.解决办法

针对问题1：使用循环请求，循环请求去获取锁 。  
针对问题2：针对第二个问题，在循环请求获取锁的时候，加入睡眠功能，等待几毫秒在执行循环 。  
针对问题3：在加锁的时候存入的key是随机的。这样的话，每次在删除key的时候判断下存入的key里的value和自己存的是否一样 。