**Redis概述**

**Redis的定义**

Redis（Remote Dictionary Server）即远程字典服务。

是一个开源的使用ANSI [C语言](https://baike.baidu.com/item/C%E8%AF%AD%E8%A8%80?fromModule=lemma_inlink)编写、支持网络、可基于内存亦可持久化的日志型、Key-Value[数据库](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/103728?fromModule=lemma_inlink)，并提供多种语言的API。

**Redis的特性和优势**

1.内存存储，读取速度快。

2.效率高，可以用于高速缓存。

3.多样的数据类型（五大基本数据类型String、List、Set、Hashe、Zset，三种特殊数据类型geo、hyperioglog、bitmap）

4.支持数据的持久化

5.事务

6.发布订阅

7.集群

.......

**Redis是单线程**

Redis是基于内存的操作，cpu不是redis的瓶颈，redis的瓶颈是机器内存大小或者网络带宽。既然单线程容易实现而且cpu不会成为瓶颈，那么就顺理成章的采用单线程的方案了。redis是C语言编写的，官方数据可以达到100000+QPS(每秒查询次数)。

多线程（CPU上下文切换耗时），而redis单线程对内存中的数据进行读写时，速度比较快。

**Redis的安装**

1.Docker运行Redis：

docker run -d -p 6379:6379 --name myredis redis:latest #official

2.启用redis-cli来链接Redis(ping-PONG)

**基础知识**

Redis默认有16个数据库，默认使用第0个数据库

可以使用select 进行切换数据库

|  |
| --- |
| 127.0.0.1:6379> select 3 #切换数据库  OK  127.0.0.1:6379[3]> dbsize #查看DB大小  (integer) 0  127.0.0.1:6379[3]> |

不同数据库可以存不同数据

|  |
| --- |
| 127.0.0.1:6379[3]> set name yuanlong  OK  127.0.0.1:6379[3]> get name  "yuanlong"  127.0.0.1:6379[3]> dbsize  (integer) 1  127.0.0.1:6379[3]> select 7  OK  127.0.0.1:6379[7]> dbsize  (integer) 0  127.0.0.1:6379[7]> get name  (nil)  127.0.0.1:6379[7]> |

查看所有key

|  |
| --- |
| 127.0.0.1:6379[7]> select 3  OK  127.0.0.1:6379[3]> keys \*  1) "name" |

查看 key是否存在

|  |
| --- |
| 127.0.0.1:6379> set name yuanlong  OK  127.0.0.1:6379> get name  "yuanlong"  127.0.0.1:6379> exists name  (integer) 1  127.0.0.1:6379> exists names  (integer) 0 |

设置key生存时间当key过期会被自动删除

|  |
| --- |
| 127.0.0.1:6379[3]> expire name 3  (integer) 1  127.0.0.1:6379[3]> get name  "yuanlong"  127.0.0.1:6379[3]> get name  (nil)  127.0.0.1:6379[3]> ttl name  (integer) -2 |

查看还有多少秒过期： -1表示永不过期（不设expire），-2表示已经过期。

|  |
| --- |
| 127.0.0.1:6379[3]> ttl name  (integer) -2 |

查看key的类型：

|  |
| --- |
| 127.0.0.1:6379[3]> set name yuanlong  OK  127.0.0.1:6379[3]> type name  string |

删除key:

|  |
| --- |
| 127.0.0.1:6379[3]> ttl name  (integer) -1  127.0.0.1:6379[3]> del name  (integer) 1  127.0.0.1:6379[3]> get name  (nil) |

清除当前数据库flushdb ，清除全部数据库的内容flushall

|  |
| --- |
| 127.0.0.1:6379[3]> keys \*  1) "name"  127.0.0.1:6379[3]> flushdb  OK  127.0.0.1:6379[3]> keys \*  (empty array)  127.0.0.1:6379[3]> |

Redis命令文档

<https://redis.io/commands/>

**字符串String**

string类型redis最基本的数据类型，一个key对应一个value，一个redis中字符串value最多可以是512M。

**string类型是二进制安全的，即redis的string可以包含任何数据，比如jpg图片或者序列化的对象**

append name .Li: 对已经存在的key, append就是在后面追加。

append age 30: 对不存在的key进行append,等同于set age 30

strlen name: 获取字符串长度

|  |
| --- |
| 127.0.0.1:6379[3]> flushdb  OK  127.0.0.1:6379[3]> keys \*  (empty array)  127.0.0.1:6379[3]> set name yuanlong  OK  127.0.0.1:6379[3]> append name .Li  (integer) 11  127.0.0.1:6379[3]> get name  "yuanlong.Li"  127.0.0.1:6379[3]> strlen name  (integer) 11  127.0.0.1:6379[3]> append age 30  (integer) 2  127.0.0.1:6379[3]> keys \*  1) "age"  2) "name"  127.0.0.1:6379[3]> |

set views 0: 要设置为数字才能用incr,decr命令 ，本质上这是一个字符串操作，因为Redis没有专门的整数类型。存储在 key 中的字符串被转换为十进制有符号整数

incr views: 执行一次将key中存储的数字值增一(i++)

decr views: 执行一次将key中存储的数字值减一(i--)

incrby views 10: 执行一次key中存储的值+10(i+=10)

decrby views 10: 执行一次key中存储的值-10 (i-=10)

|  |
| --- |
| 127.0.0.1:6379[3]> set views 0  OK  127.0.0.1:6379[3]> incr views  (integer) 1  127.0.0.1:6379[3]> incr views  (integer) 2  127.0.0.1:6379[3]> get views  "2"  127.0.0.1:6379[3]> decr views  (integer) 1  127.0.0.1:6379[3]> decr views  (integer) 0  127.0.0.1:6379[3]> decr views  (integer) -1  127.0.0.1:6379[3]> get views  "-1"  127.0.0.1:6379[3]> incrby views 11  (integer) 10  127.0.0.1:6379[3]> get views  "10"  127.0.0.1:6379[3]> decrby views 5  (integer) 5  127.0.0.1:6379[3]> get views  "5"  127.0.0.1:6379[3]> |

getrange name 2 4: 获取[m,n]区间范围内的字符串。n可以为负数，表示偏移，-1 表示最后一个字符， -2 表示倒数第二个字符，以此类推。（substring）

* 如果负数值大于字符串长度，取的总是第一个字符串。

getrange name 0 -1 就是获取全部字符串 和 get key 是一样的。

|  |
| --- |
| 127.0.0.1:6379[3]> set name hello,Word  OK  127.0.0.1:6379[3]> get name  "hello,Word"  127.0.0.1:6379[3]> getrange name 2 4  "llo"  127.0.0.1:6379[3]> getrange name 0 -1  "hello,Word" |

setrange name 6 yuanlong：将key的值从第n个开始替换为指定的value（replace）

|  |
| --- |
| 127.0.0.1:6379[3]> get name  "hello,Word"  127.0.0.1:6379[3]> setrange name 6 yuanlong  (integer) 14  127.0.0.1:6379[3]> get name  "hello,yuanlong" |

setex name 10 li: 将key的值设置为li,并且key的生存时间设置为10s。这个命令等价于这个命令等价于set name li, expire name 10

**区别是setex 命令是一个原子操作，在同一时间内完成设置值和设置过期时间这两个操作。**

|  |
| --- |
| 127.0.0.1:6379> setex name 10 li  OK  127.0.0.1:6379> get name  "li"  127.0.0.1:6379> ttl name  (integer) -2 |

setnx name yuanlong：在指定的key不存在时，为key设置指定的值，这种情况下等同于set，当key存在时,什么也不做。返回值：0是失败， 1 是成功。

|  |
| --- |
| 127.0.0.1:6379> flushdb  OK  127.0.0.1:6379> setnx name yuanlong  (integer) 1  127.0.0.1:6379> setnx name li  (integer) 0  127.0.0.1:6379> get name  "yuanlong" |

mset k1 v1 k2 v2 k3 v3: 设置多个key和value，如果这其中已经存在key，则会覆盖掉。

mget k1 k2 k3: 返回所有给定的key的值

|  |
| --- |
| 127.0.0.1:6379> mset k1 v1 k2 v2 k3 v3  OK  127.0.0.1:6379> get k1  "v1"  127.0.0.1:6379> mget k1 k2 k3 k4  1) "v1"  2) "v2"  3) "v3"  4) (nil)  127.0.0.1:6379> |

（mset user:1:name zhangsan user:1:age 28 可以用来缓存对象）

|  |
| --- |
| 127.0.0.1:6379> mset user:1:name yuanlong user:1:age 18  OK  127.0.0.1:6379> get user:1:name  "yuanlong"  127.0.0.1:6379> mget user:1:name user:1:age  1) "yuanlong"  2) "18" |

msetnx k1 v1 k4 v4: 设置多个 key value，当且仅当所有给定的健都不存在时，即使有一个健存在值，msetnx也会拒绝执行操作。

这是一个原子操作 ，要么一起成功，要么一起失败。

|  |
| --- |
| 127.0.0.1:6379> keys \*  1) "name"  2) "k1"  3) "k3"  4) "k2"  127.0.0.1:6379> msetnx k1 v1 k4 v4  (integer) 0  127.0.0.1:6379> get k1  "v1"  127.0.0.1:6379> get k4  (nil) |

getset name yuanlong：将key的值修改为value,并返回在修改之前的值。如果key在执行这个命令前不存在，则返回nil。

先get原来值 再修改成新值。

|  |
| --- |
| 127.0.0.1:6379> getset name "hello yuanlong"  "yuanlong"  127.0.0.1:6379> get name  "hello yuanlong"  127.0.0.1:6379> |

**应用**

1.计数器

2.统计多单位的数量

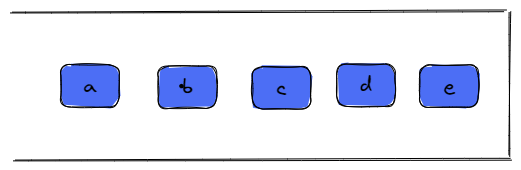
3.粉丝数

4.对象缓存存储

**列表List**

List是一个单值多value类型，可以用来当作栈、队列、阻塞队列。

它的底层就是一个链表



lpush list a : 将一个值或者多个值插入到列表的头部（左边）

lrange list n m: 返回指定区间的元素 [n,m] m为-1表示返回n后面所有的元素。

|  |
| --- |
| 127.0.0.1:6379> keys \*  (empty array)  127.0.0.1:6379> lpush list c  (integer) 1  127.0.0.1:6379> lpush list b  (integer) 2  127.0.0.1:6379> lpush list a  (integer) 3  127.0.0.1:6379> lrange list 0 -1  1) "a"  2) "b"  3) "c"  127.0.0.1:6379> lrange list 0 1  1) "a"  2) "b"  127.0.0.1:6379> |

rpush list d：:将一个值或者多个值插入到列表的尾部(右边)

|  |
| --- |
| 127.0.0.1:6379> rpush list d  (integer) 4  127.0.0.1:6379> lrange list 0 -1  1) "a"  2) "b"  3) "c"  4) "d"  127.0.0.1:6379> |

lpop list :移除并返回列表的第一个元素，左边的第一个

rpop list :移除并返回列表的最后一个元素，右边第一个。

|  |
| --- |
| 127.0.0.1:6379> lpop list  "a"  127.0.0.1:6379> lrange list 0 -1  1) "b"  2) "c"  3) "d"  127.0.0.1:6379> rpop list  "d"  127.0.0.1:6379> lrange list 0 -1  1) "b"  2) "c"  127.0.0.1:6379> |

lrem list n value: 移除列表中与value相等的元素，移除n个，顺序是从前往后(精确匹配)。**n>0 从头到尾删除值为value的元素， n<0，从尾到头删除值为value的元素， n=0,移除所有值为value的元素**

|  |
| --- |
| 127.0.0.1:6379> lrange list 0 -1  1) "c"  2) "c"  3) "b"  4) "a"  127.0.0.1:6379> lrem list 1 a  (integer) 1  127.0.0.1:6379> lrange list 0 -1  1) "c"  2) "c"  3) "b"  127.0.0.1:6379> lrem list 2 c  (integer) 2  127.0.0.1:6379> lrange list 0 -1  1) "b"  127.0.0.1:6379> |

lindex list n :按照索引下标n获得元素(-1代表最后一个，0表示第一个，以此类推)

|  |
| --- |
| 27.0.0.1:6379> lindex list 0  "b"  127.0.0.1:6379> lindex list 1  "c"  127.0.0.1:6379> lindex list -1  "c"  127.0.0.1:6379> |

llen list :返回列表的长度

|  |
| --- |
| 127.0.0.1:6379> flushdb  OK  127.0.0.1:6379> lpush list a  (integer) 1  127.0.0.1:6379> lpush list b  (integer) 2  127.0.0.1:6379> lpush list c  (integer) 3  127.0.0.1:6379> llen list  (integer) 3  127.0.0.1:6379> |

ltrim lsit n m :对一个列表进行修剪，只保留区间[n,m]内的元素, 通过下标截取固定长度，其他全部删除。

|  |
| --- |
| 127.0.0.1:6379> flushdb  OK  127.0.0.1:6379> lpush list hello1  (integer) 1  127.0.0.1:6379> lpush list hello2  (integer) 2  127.0.0.1:6379> lpush list hello3  (integer) 3  127.0.0.1:6379> lpush list hello4  (integer) 4  127.0.0.1:6379> lrange list 0 -1  1) "hello4"  2) "hello3"  3) "hello2"  4) "hello1"  127.0.0.1:6379> ltrim list 1 2  OK  127.0.0.1:6379> lrange list 0 -1  1) "hello3"  2) "hello2"  127.0.0.1:6379> |

lset list n value ：将list中固定下标n元素的值更新为value，更新操纵。如果这个列表不存在更新报错，如果下标不存在也更新报错。

|  |
| --- |
| 127.0.0.1:6379> flushdb  OK  127.0.0.1:6379> exists list  (integer) 0  127.0.0.1:6379> lset list 0 a  (error) ERR no such key  127.0.0.1:6379> lpush list a  (integer) 1  127.0.0.1:6379> lpush list b  (integer) 2  127.0.0.1:6379> lpush list c  (integer) 3  127.0.0.1:6379> lrange list 0 -1  1) "c"  2) "b"  3) "a"  127.0.0.1:6379> lset list 0 d  OK  127.0.0.1:6379> lrange list 0 -1  1) "d"  2) "b"  3) "a"  127.0.0.1:6379> lset list 4 e  (error) ERR index out of range  127.0.0.1:6379> |

insert list before|after value value1：将value1插入到value的前边或者后边。

|  |
| --- |
| 127.0.0.1:6379> flushdb  OK  127.0.0.1:6379> lpush list a  (integer) 1  127.0.0.1:6379> lpush list b  (integer) 2  127.0.0.1:6379> lrange list 0 -1  1) "b"  2) "a"  127.0.0.1:6379> linsert list before a c  (integer) 3  127.0.0.1:6379> linsert list after c e  (integer) 4  127.0.0.1:6379> lrange list 0 -1  1) "b"  2) "c"  3) "e"  4) "a"  127.0.0.1:6379> |

**性能总结**

* 它是一个字符串列表，left,right都可以插入添加
* 如果键不存在，创建新的链表。
* 如果已经存在，新增内容
* 如果值全部移除，空链表，对应的键也就消失了
* 链表的操作无论是头和尾效率都极高，但假如是对中间元素进行操作，效率就比较低。

应用

消息队列（lpush rpop）

栈（lpush lpop）

**集合Set**

redis的集合set是string类型的无序集合，值是不能重复的。(无序不重复集合)

sadd key member[...]:将一个或者多个成员元素加入到集合中。

smembers key :返回存储在key中的所有成员

sismember key member:判断元素member是否是集合key中的成员。返回1表示是，0表示不是。

|  |
| --- |
| 127.0.0.1:6379> sadd myset member1 member2  (integer) 2  127.0.0.1:6379> smembers myset  1) "member1"  2) "member2"  127.0.0.1:6379> sismember myset member1  (integer) 1  127.0.0.1:6379> sismember myset member3  (integer) 0  127.0.0.1:6379> |

scard myset: 返回集合中元素的数量。

|  |
| --- |
| 127.0.0.1:6379> scard myset  (integer) 2  127.0.0.1:6379> smembers myset  1) "member1"  2) "member2"  127.0.0.1:6379> |

srem key value:移除集合key中的一个或者多个元素value

|  |
| --- |
| 127.0.0.1:6379> sadd myset member3  (integer) 1  127.0.0.1:6379> smembers myset  1) "member3"  2) "member1"  3) "member2"  127.0.0.1:6379> srem myset member2  (integer) 1  127.0.0.1:6379> smembers myset  1) "member3"  2) "member1"  127.0.0.1:6379> |

srandmember key [count]: 返回集合key中的随机元素，count表示返回几个元素。

|  |
| --- |
| 127.0.0.1:6379> smembers myset  1) "member3"  2) "member1"  3) "member2"  127.0.0.1:6379> srandmember myset  "member2"  127.0.0.1:6379> srandmember myset  "member3"  127.0.0.1:6379> srandmember myset 2  1) "member1"  2) "member2"  127.0.0.1:6379> |

**抽奖**

spop key [count]: 从集合key中删除一个或者多个元素，并返回删除的元素。(删除是随机的)

|  |
| --- |
| 127.0.0.1:6379> smembers myset  1) "member3"  2) "member1"  3) "member2"  127.0.0.1:6379> spop myset  "member1"  127.0.0.1:6379> spop myset  "member2"  127.0.0.1:6379> smembers myset  1) "member3"  127.0.0.1:6379> |

smove key1 key2 member: 指定的元素从集合key1移动到key2 。这是一个原子操作。

|  |
| --- |
| 127.0.0.1:6379> smembers myset  1) "member3"  127.0.0.1:6379> smove myset myset2 member3  (integer) 1  127.0.0.1:6379> smembers myset  (empty array)  127.0.0.1:6379> smembers myset2  1) "member3"  127.0.0.1:6379> |

sdiff key1 key2 :返回第一个集合key与其他key之间的差异，即**第一个集合key独有的元素。**

sinter key1 key2: 返回所有给定集合key的成员**交集**

union key1 key2: 返回所有给定集合key的**并集**

|  |
| --- |
| 127.0.0.1:6379> sadd myset1 a b c d  (integer) 4  127.0.0.1:6379> sadd myset2 a c d e  (integer) 4  127.0.0.1:6379> sdiff myset1 myset2  1) "b"  127.0.0.1:6379> sinter myset1 myset2  1) "a"  2) "d"  3) "c"  127.0.0.1:6379> sunion myset1 myset2  1) "a"  2) "b"  3) "e"  4) "d"  5) "c"  127.0.0.1:6379> |

应用

* 微博 ， A用户将所有关注的人放在一个set集合中。将他的粉丝也放在一个set集合中。
* 实现共同关注，共同好友等功能。

**哈希Hash**

Hash是一个键值对集合，相当于map集合(key-map集合)，是一个String类型的field和value的映射表，特别适合存储对象。

hset key field value[field value...] :为哈希表key的field字段赋值

hget key field :返回哈希表key中的field字段的值

hgetall key: 返回哈希表key中所有的域和值

|  |
| --- |
| 127.0.0.1:6379> hset myhash field1 a field2 b  (integer) 2  127.0.0.1:6379> hget myhash field1  "a"  127.0.0.1:6379> hmset myhash field3 c field4 d  OK  127.0.0.1:6379> hgetall myhash  1) "field1"  2) "a"  3) "field2"  4) "b"  5) "field3"  6) "c"  7) "field4"  8) "d"  127.0.0.1:6379> |

hdel key field[field...]:删除哈希表key中的一个或者多个指定字段,对应的value值也没有了。

|  |
| --- |
| 127.0.0.1:6379> hdel myhash field1  (integer) 1  127.0.0.1:6379> hgetall myhash  1) "field2"  2) "b"  3) "field3"  4) "c"  5) "field4"  6) "d" |

hlen key: 获取哈希表key中的字段fields的数量

|  |
| --- |
| 127.0.0.1:6379> hlen myhash  (integer) 3  127.0.0.1:6379> |

hexists key field: 查看哈希表的指定字段field是否存在。0表示不存在，1表示存在

|  |
| --- |
| 127.0.0.1:6379> hgetall myhash  1) "field2"  2) "b"  3) "field3"  4) "c"  5) "field4"  6) "d"  127.0.0.1:6379> hexists myhash field2  (integer) 1  127.0.0.1:6379> hexists myhash field1  (integer) 0  127.0.0.1:6379> |

hkeys key: 返回哈希表key中所有的域field

hvals key: 返回哈希表中所有域的值

|  |
| --- |
| 127.0.0.1:6379> hgetall myhash  1) "field2"  2) "b"  3) "field3"  4) "c"  5) "field4"  6) "d"  127.0.0.1:6379> hkeys myhash  1) "field2"  2) "field3"  3) "field4"  127.0.0.1:6379> hvals myhash  1) "b"  2) "c"  3) "d"  127.0.0.1:6379> |

hincrby key field increment: 为哈希表key中的域field的**值**加上增量increment.增量为负数表示进行减法操作

|  |
| --- |
| 127.0.0.1:6379> hincrby myhash field1 3 # 如果field不存在默认是0  (integer) 3  127.0.0.1:6379> hincrby myhash field1 3  (integer) 6  127.0.0.1:6379> hget myhash field1  "6"  127.0.0.1:6379> hincrby myhash field1 -2  (integer) 4  127.0.0.1:6379> hget myhash field1  "4"  127.0.0.1:6379> |

hsetnx key field value:为哈希表中不存在的字段field赋值为value。0表示设置失败，1表示设置成功。

|  |
| --- |
| 127.0.0.1:6379> hkeys myhash  1) "field2"  2) "field3"  3) "field4"  4) "field1"  127.0.0.1:6379> hsetnx myhash field3 f  (integer) 0  127.0.0.1:6379> hsetnx myhash field5 g  (integer) 1 |

应用

Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。 存储部分变更的数据，如用户信息等。

**有序集合Zset**

在set基础上，加上一个score。 set key value zset key score value

zadd key score member：将一个或者多个member元素及其score值加入到有序集合key中。**其中score可以是整数值或者双精度浮点数。可以为正或者负数**

zrange key n m ：返回有序集合key中的[n,m]区间内的元素。下标从0开始。如果m超出范围也不会报错，只会返回包含的。

zrevrange myzset n m: 返回有序集合key中的[n,m]区间内的元素,从大到小排序

zrangebyscore key n m：返回有序集合key中的成员，按照**score**的从小到大排序，范围是[n,m]，**n,m可以为-inf,+inf，这样在不知道有序集合数量的时候获取所有的成员**。**注意[n,m]这个范围是score的范围，不是成员的下标。**

|  |
| --- |
| 127.0.0.1:6379> zadd myzset 1 one 2 two 3 three  (integer) 3  127.0.0.1:6379> zrange myzset 0 -1  1) "one"  2) "two"  3) "three"  127.0.0.1:6379> zrevrange myzset 0 -1  1) "three"  2) "two"  3) "one"  127.0.0.1:6379> zrangebyscore myzset -inf +inf  1) "one"  2) "two"  3) "three"  127.0.0.1:6379>  127.0.0.1:6379> zrangebyscore myzset -inf +inf withscores  1) "one"  2) "1"  3) "two"  4) "2"  5) "three"  6) "3"  127.0.0.1:6379> zrangebyscore myzset -inf 2 withscores  1) "one"  2) "1"  3) "two"  4) "2"  127.0.0.1:6379> |

zrem key member ：从有序集合key中删除指定的成员member

|  |
| --- |
| 127.0.0.1:6379> zrem myzset two  (integer) 1  127.0.0.1:6379> zrange myzset 0 -1  1) "one"  2) "three"  127.0.0.1:6379> |

zcard key：返回有序集合key中的成员个数。

|  |
| --- |
| 127.0.0.1:6379> zrange myzset 0 -1  1) "one"  2) "three"  127.0.0.1:6379> zcard myzset  (integer) 2  127.0.0.1:6379> |

zcount key n m：返回有序集key中，**score**在[n,m]之间的成员数量。

|  |
| --- |
| 127.0.0.1:6379> zcount myzset 2 3  (integer) 2  127.0.0.1:6379> |

zrank key member ：返回有序集key中成员的排名,按照score**从低到高**排名，**从0开始**

zrevrank key member：返回有序集key中成员的排名，按照score**从高到低**排名，**从0开始**

|  |
| --- |
| 127.0.0.1:6379> zrank myzset one  (integer) 0  127.0.0.1:6379> zrank myzset two  (integer) 1  127.0.0.1:6379> zrank myzset three  (integer) 2  127.0.0.1:6379> zrevrank myzset three  (integer) 0  127.0.0.1:6379> |

**应用**

* set 排序 存储班级成绩表，工资表排序等
* 带权重判断，例如1普通消息 2 重要消息
* 排行榜应用，取top n操作。

**地理位置Geospatial**

可以将用户给定的地理位置信息存储起来，并对这些信息进行操作。

查询测试数据：<http://www.jsons.cn/lngcode/>

GEO的数据结构总共有六个常用命令：**geoadd ,geopos ,geodist , georadius, georadiusbymember , gethash**

Select 8

geoadd：将指定的地理空间位置(经度，纬度，名称)添加到指定的key中。（一般是下载城市数据，直接通过程序一次性导入）

**这些数据会存储到有序集合Zset中**，目的是为了方便使用georadius,或者georadiusbymember命令对数据进行半径查询等操作。

注意：该命令采用标准格式的参数xy,**所以经度必须在纬度之前**。有效的经度 **-180~180**度，有效的纬度 **-85.05~85.05**

|  |
| --- |
| 127.0.0.1:6379[8]> geoadd china:city 116.40 39.90 Beijing 121.47 31.23 Shanghai  (integer) 2  127.0.0.1:6379[8]> geoadd china:city 106.50 29.53 Chongqin 114.05 22.52 shenzhen  (integer) 2  127.0.0.1:6379[8]> geoadd china:city 120.16 30.24 Hangzhou 108.96 34.26 Xian  (integer) 2  127.0.0.1:6379[8]> |

geopos: 从key里面返回所有给定位置元素的位置(**经度和纬度**)

|  |
| --- |
| 127.0.0.1:6379[8]> geopos china:city Beijing  1) 1) "116.39999896287918091"     2) "39.90000009167092543"  127.0.0.1:6379[8]> geopos china:city Shanghai Xian  1) 1) "121.47000163793563843"     2) "31.22999903975783553"  2) 1) "108.96000176668167114"     2) "34.25999964418929977"  127.0.0.1:6379[8]> |

geodist: 返回两个给定位置之间的距离，指定的参数**unit**必须是以下中的一种。

单位：m 米 ，km 千米，mi 英里，ft 英尺，默认是m。

|  |
| --- |
| 127.0.0.1:6379[8]> geodist china:city Beijing Shanghai  "1067378.7564"  127.0.0.1:6379[8]> geodist china:city Beijing Shanghai km  "1067.3788"  127.0.0.1:6379[8]> |

georadius：以给定的**经纬度**为中心，找出某一**半径**内的元素。

withdist: 返回位置元素的同时，将位置元素与中心之间的距离也一并返回。距离的单位与用户给定范围单位保持一致。

withcoord: 将位置元素的经纬度也返回。

count:获取前n个匹配的元素，可以减少带宽当数据量大时。

withhash:以 52 位有符号整数的形式， 返回位置元素经过原始 geohash 编码的有序集合分值。 这个选项主要用于底层应用或者调试， 实际中的作用并不大

asc: 根据中心的位置， 按照从近到远的方式返回位置元素。

desc：根据中心的位置， 按照从远到近的方式返回位置元素。

|  |
| --- |
| 127.0.0.1:6379[8]> georadius china:city 110 30 500 km  1) "Chongqin"  2) "Xian"  127.0.0.1:6379[8]> georadius china:city 110 30 500 km withcoord  1) 1) "Chongqin"     2) 1) "106.49999767541885376"        2) "29.52999957900659211"  2) 1) "Xian"     2) 1) "108.96000176668167114"        2) "34.25999964418929977"  127.0.0.1:6379[8]> georadius china:city 110 30 500 km withcoord withdist  1) 1) "Chongqin"     2) "341.9374"     3) 1) "106.49999767541885376"        2) "29.52999957900659211"  2) 1) "Xian"     2) "483.8340"     3) 1) "108.96000176668167114"        2) "34.25999964418929977"  127.0.0.1:6379[8]> georadius china:city 110 30 500 km withcoord withdist count 1  1) 1) "Chongqin"     2) "341.9374"     3) 1) "106.49999767541885376"        2) "29.52999957900659211"  127.0.0.1:6379[8]> |

Georadiusbymember: 找出位于指定范围内的元素，**中心点是由给定的位置元素决定的**。

|  |
| --- |
| 127.0.0.1:6379[8]> georadiusbymember china:city beijing 1000 km  (error) ERR could not decode requested zset member  127.0.0.1:6379[8]> georadiusbymember china:city Beijing 1000 km  1) "Beijing"  2) "Xian"  127.0.0.1:6379[8]> georadiusbymember china:city Shanghai 4000 km  1) "Chongqin"  2) "Xian"  3) "shenzhen"  4) "Hangzhou"  5) "Shanghai"  6) "Beijing"  127.0.0.1:6379[8]> |

geohash: 返回一个或者多个位置元素的geohash表示。

redis使用geohash将二维经纬度转换为一维11位字符串，字符串越长表示位置更精确，两个字符串越相似表示距离越近。

127.0.0.1:6379[8]> geohash china:city Beijing  
1) "wx4fbxxfke0"  
127.0.0.1:6379[8]>

geo底层的实现原理其实就是zset，所以有些可以通过zset命令来操作geo

例如zrem zrang。

|  |
| --- |
| 127.0.0.1:6379[8]> zrange china:city 0 -1  1) "Chongqin"  2) "Xian"  3) "shenzhen"  4) "Hangzhou"  5) "Shanghai"  6) "Beijing"  127.0.0.1:6379[8]> |

**应用**

* 朋友的定位，附件的人，打车距离计算等。
* 两点之间距离。

**基数统计Hyperloglog**

HyperLogLog是用来做基数统计的算法。

**[\*](https://redis.com.cn/redis-hyperloglog.html" \l "%E4%BB%80%E4%B9%88%E6%98%AF%E5%9F%BA%E6%95%B0)什么是基数?**

比如数据集 {1, 3, 5, 7, 5, 7, 8}， 那么这个数据集的基数集为 {1, 3, 5 ,7, 8}, 基数(不重复元素)为5。 可以接受误差。

假设算一个网页浏览量（一个人访问一个网站多次，但是还算作一个人）

传统方式：set保存用户id，然后统计set中元素数量。

但是如果保存大量的用户id，就会比较麻烦，目的是计数而不是保存用户id。

使用hyperloglog优点：是在输入元素的数量和体积非常大时，**计算基数所需的空间总是固定的，每个HyperLogLog键只需要花费12kb内存，并且很小**。

缺点：0.81%的错误率。

[PFADD](https://redis.com.cn/commands/pfadd.html)添加指定元素到 HyperLogLog 中。

[PFCOUNT](https://redis.com.cn/commands/pfcount.html)返回给定 HyperLogLog 的基数估算值。

[PFMERGE](https://redis.com.cn/commands/pfmerge.html)将多个 HyperLogLog 合并为一个 HyperLogLog，并集计算。

|  |
| --- |
| 127.0.0.1:6379[7]> pfadd myset a b c d e f g h i j  (integer) 1  127.0.0.1:6379[7]> pfcount myset  (integer) 10  127.0.0.1:6379[7]> pfadd myset2 i j kz x c v b n m  (integer) 1  127.0.0.1:6379[7]> pfcount myset2  (integer) 9  127.0.0.1:6379[7]> pfmerge myset3 myset myset2  OK  127.0.0.1:6379[7]> pfcount myset3  (integer) 15  127.0.0.1:6379[7]> zrange myset3 0 -1  (error) WRONGTYPE Operation against a key holding the wrong kind of value  127.0.0.1:6379[7]> |

如果允许容错，可以用hyperloglog

**位图BitMap**

统计用户的某些信息，如活跃或不活跃，登录或者不登录；又如需要记录用户一年的打卡情况，打卡了是1， 没有打卡是0. 两个状态的，都可以使用Bitmaps。

如果使用普通的 key/value存储，则要记录 365条记录，如果用户量很大，需要的空间也会很大，所以 Redis 提供了 Bitmap 位图这种数据结构， Bitmap 就是通过操作二进制位来进行记录，即为 0 和 1；如果要记录 365 天的打卡情况，使用 Bitmap 表示的形式大概如下：0101000111000111...........................，节约内存 了，365 天相当于 365 bit，又 1 字节 = 8 bit , 所以相当于使用 46 个字节即可。

​BitMap 就是通过一个 bit 位来表示某个元素对应的值或者状态, 其中的 key 就是对应元素本身，实际上 底层也是通过对字符串的操作来实现。

setbit key n value：设置key的第n位的值value是0或1，**n是从0开始的。**

|  |
| --- |
| 127.0.0.1:6379> select 6  OK  127.0.0.1:6379[6]> setbit sign 0 0  (integer) 0  127.0.0.1:6379[6]> setbit sign 1 1  (integer) 0  127.0.0.1:6379[6]> setbit sign 2 1  (integer) 0  127.0.0.1:6379[6]> setbit sign 3 1  (integer) 0  127.0.0.1:6379[6]> setbit sign 4 1  (integer) 0  127.0.0.1:6379[6]> setbit sign 5 0  (integer) 0 |

getbit key n :获取第n位的值。如果没有设置返回0

|  |
| --- |
| 127.0.0.1:6379[6]> getbit sign 0  (integer) 0  127.0.0.1:6379[6]> getbit sign 3  (integer) 1  127.0.0.1:6379[6]> |

bitcount key [n m]: 统计key上值为1的个数

|  |
| --- |
| 127.0.0.1:6379[6]> bitcount sign  (integer) 4  127.0.0.1:6379[6]> |

**Demos**

**demo01 : 分布式Id自增器**

在构建应用程序的时候，经常会用到各式各样的ID（identif ier，标识符）。比如，存储用户信息的程序在每次出现一个新用户的时候就需要创建一个新的用户ID，而博客程序在作者每次发表一篇新文章的时候也需要创建一个新的文章ID。

# 全局唯一id特点

全局唯一性：不能出现重复的ID号

趋势递增：在MySQL InnoDB引擎中使用的是聚合索引，由于多数RDBMS使用B-tree的数据结构来存储索引数据，在主键的选择上面应该尽量使用有序的主键保证写入性能；

单调递增：保证下一个ID一定大于上一个ID，例如事务版本号、IM增量消息、排序等特殊需求；

信息安全：如果ID是连续的，恶意用户的扒取工作就非常容易做了，直接按照顺序下载指定URL即可；如果是订单号就更危险了，竞对可以直接知道一天的单量。所以在一些应用场景下，会需要ID无规则、不规则；

高可用性：同时除了对ID号码自身的要求，业务还对ID号生成系统的可用性要求极高，想象一下，如果ID生成系统瘫痪，这就会带来一场灾难。所以不能有单点故障；

分片支持：可以控制ShardingId。比如某一个用户的文章要放在同一个分片内，这样查询效率高，修改也容易；

长度适中。

# 常用的创建分布式id的几种方案

## 数据库自增长序列或字段生成id

最常见的一种生成id方式。利用数据库本身来进行设置，在全数据库内保持唯一。

【优点】

非常简单。利用现有数据库系统的功能实现，成本小，代码简单，性能可以接受。ID号单调递增。可以实现一些对ID有特殊要求的业务，比如对分页或者排序结果这类需求有帮助。

【缺点】

强依赖DB。不同数据库语法和实现不同，数据库迁移的时候、多数据库版本支持的时候、或分表分库的时候需要处理，会比较麻烦。当DB异常时整个系统不可用，属于致命问题。

单点故障。在单个数据库或读写分离或一主多从的情况下，只有一个主库可以生成。有单点故障的风险。

数据一致性问题。配置主从复制可以尽可能的增加可用性，但是数据一致性在特殊情况下难以保证。主从切换时的不一致可能会导致重复发号。

难于扩展。在性能达不到要求的情况下，比较难于扩展。ID发号性能瓶颈限制在单台MySQL的读写性能。

【部分优化方案】

针对主库单点， 如果有多个Master库，则每个Master库设置的起始数字不一样，步长一样，可以是Master的个数。比如：Master1 生成的是 1,4,7,10，Master2生成的是2,5,8,11 Master3生成的是 3,6,9,12。这样就可以有效生成集群中的唯一ID，也可以大大降低ID生成数据库操作的负载。

## UUID

常见的生成id方式，利用程序生成。

UUID (Universally Unique Identifier) 的目的，是让分布式系统中的所有元素，都能有唯一的辨识资讯，而不需要透过中央控制端来做辨识资讯的指定。如此一来，每个人都可以建立不与其它人冲突的 UUID。在这样的情况下，就不需考虑数据库建立时的名称重复问题。

UUID的标准形式包含32个16进制数字，以连字号分为五段，形式为8-4-4-4-12的36个字符，示例：550e8400-e29b-41d4-a716-446655440000，到目前为止业界一共有5种方式生成UUID。

在Java中我们可以直接使用下面的API生成UUID:

UUID uuid = UUID.randomUUID(); String s = UUID.randomUUID().toString();

【优点】

1. 非常简单，本地生成，代码方便，API调用方便。
2. 性能非高。生成的id性能非常好，没有网络消耗，基本不会有性能问题。
3. 全球唯一。在数据库迁移、系统数据合并、或者数据库变更的情况下，可以 从容应对。

【缺点】

1. 存储成本高。UUID太长，16字节128位，通常以36长度的字符串表示，很多场景不适用。如果是海量数据库，就需要考虑存储量的问题。
2. 信息不安全。基于MAC地址生成UUID的算法可能会造成MAC地址泄露，这个漏洞曾被用于寻找梅丽莎病毒的制作者位置。
3. 不适用作为主键，ID作为主键时在特定的环境会存在一些问题，比如做DB主键的场景下，UUID就非常不适用。UUID往往是使用字符串存储，查询的效率比较低。
4. UUID是无序的。不是单调递增的，而现阶段主流的数据库主键索引都是选用的B+树索引，对于无序长度过长的主键插入效率比较低。
5. 传输数据量大。
6. 不可读。

【部分优化方案】

为了解决UUID不可读， 可以使用UUID to Int64的方法 。

为了解决UUID无序的问题， NHibernate在其主键生成方式中提供了Comb算法（combined guid/timestamp）。保留GUID的10个字节，用另6个字节表示GUID生成的时间（DateTime）。

## Redis生成ID

当使用数据库来生成ID性能不够要求的时候，我们可以尝试使用Redis来生成ID。这主要依赖于Redis是单线程的，所以也可以用生成全局唯一的ID。可以用Redis的原子操作 INCR和INCRBY来实现。

可以使用Redis集群来获取更高的吞吐量。假如一个集群中有5台Redis。可以初始化每台Redis的值分别是1,2,3,4,5，然后步长都是5。各个Redis生成的ID为：

A：1,6,11,16,21

B：2,7,12,17,22

C：3,8,13,18,23

D：4,9,14,19,24

E：5,10,15,20,25

这个负载到哪台机器上需要提前设定好，未来很难做修改。但是3-5台服务器基本能够满足，都可以获得不同的ID。步长和初始值一定需要事先设定好。使用Redis集群也可以防止单点故障的问题。

比较适合使用Redis来生成日切流水号。比如订单号=日期+当日自增长号。可以每天在Redis中生成一个Key，使用INCR进行累加。

【优点】

1. 不依赖于数据库，灵活方便，且性能优于数据库。
2. 数字ID天然排序，对分页或者需要排序的结果很有帮助。

【缺点】

1. 如果系统中没有Redis，还需要引入新的组件，增加系统复杂度。。
2. 需要编码和配置的工作量比较大。
3. Redis单点故障，影响序列服务的可用性。

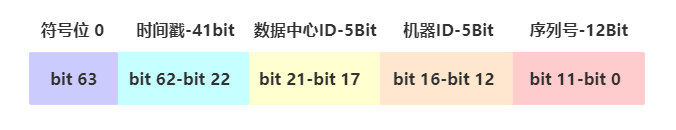
## zookeeper生成ID

zookeeper主要通过其znode数据版本来生成序列号，可以生成32位和64位的数据版本号，客户端可以使用这个版本号来作为唯一的序列号。

很少会使用zookeeper来生成唯一ID。主要是由于需要依赖zookeeper，并且是多步调用API，如果在竞争较大的情况下，需要考虑使用分布式锁。因此，性能在高并发的分布式环境下，也不理想。

## Twitter的snowflake算法

snowflake(雪花算法)是Twitter开源的分布式ID生成算法，结果是一个long型的ID。这种方案把64-bit分别划分成多段，分开来标示机器、时间等。如图：



其核心思想是：使用41bit作为毫秒数，10bit作为机器的ID（5个bit是数据中心，5个bit的机器ID），12bit作为毫秒内的流水号（意味着每个节点在每毫秒可以产生 4096 个 ID），最后还有一个符号位，永远是0。具体实现的代码可以参看github。

snowflake算法可以根据自身项目的需要进行一定的修改。比如估算未来的数据中心个数，每个数据中心的机器数以及统一毫秒可以能的并发数来调整在算法中所需要的bit数。

【优点】

1. 稳定性高，不依赖于数据库等第三方系统，以服务的方式部署，稳定性更高，生成ID的性能也是非常高的。

2. 灵活方便，可以根据自身业务特性分配bit位。

3. 单机上ID单调自增，毫秒数在高位，自增序列在低位，整个ID都是趋势递增的。

【缺点】

1. 强依赖机器时钟，如果机器上时钟回拨，会导致发号重复或者服务会处于不可用状态。
2. ID可能不是全局递增。在单机上是递增的，但是由于涉及到分布式环境，每台机器上的时钟不可能完全同步，也许有时候也会出现不是全局递增的情况。

**demo02: 实现简单分布式锁**

锁是一种同步机制，用于保证一项资源在任何时候只能被一个进程使用，如果有其他进程想要使用相同的资源，那么就必须等待，直到正在使用资源的进程放弃使用权为止。

一个锁的实现通常会有获取（acquire）和释放（release）这两种操作：

**获取操作：**取得资源的独占使用权。在任何时候，最多只能有一个进程取得锁。把成功取得锁的这个进程称为锁的持有者。在锁已经被持有的情况下，所有尝试再次获取锁的操作都会失败。

**释放操作**：放弃资源的独占使用权，一般由锁的持有者调用。在锁被释放之后，其他进程就可以再次尝试获取这个锁了。

demo 存在的问题

Q:

1.加锁之后宕机，锁一直无法释放；

2.A线程添加了锁，被B线程释放了；

3.加锁之后，执行任务时间太长，超过了超时时间，锁自动释放了。

A:

1. 加上超时时间

2.value中加上线程标识，自己添加的锁，只能自己释放

3.超时续命