

课前准备

安装了 Redis

课程主题

Redis数据类型的使用场景与 Redis内存模型

课程目标

1. 掌握Redis数据类型使用场景
2. 理解掌握Redis的特殊数据类型(重点)
3. 理解Redis的消息模式
4. 掌握 Redis Stream (重点)
5. 熟悉查看Redis内存统计
6. 理解Redis 内存划分
7. 掌握 Redis 数据存储的细节
8. 掌握理解Redis对象类型及其内部编码 (重点)

知识点

Redis基本数据类型使用场景

String

缓存功能

计数器

共享用户session。

hash

商品属性简单 并且是爆品

List

存储一些列表数据结构 粉丝列表，文章评价

利用Lrange命令 高性能分页

搞个简单的消息队列

Set

全局去重 举个栗子：共同好友

Zset

排行榜：有序集合经典使用场景

Zset来做带权重的队列，比如普通消息 score 为1，重要消息score 为2热搜。

Redis 的特殊数据类型（重点）

BitMap

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。Redis 从 2.2 版本之后新增了setbit, getbit, bitcount 等几个 bitmap 相关命令。虽然是新命令，但是本身都是对字符串的操作，我们先来看看语法：

```
1 | SETBIT key offset value
```

其中 offset 必须是数字，value 只能是 0 或者 1

bitop add

bitop not

存储一亿用户 12.5M

```
1 | 127.0.0.1:6379> setbit k1 5 1
2 | (integer) 0
3 | 127.0.0.1:6379> getbit k1 5
4 | (integer) 1
5 | 127.0.0.1:6379> getbit k1 4
6 | (integer) 0
7 | 127.0.0.1:6379> bitcount k1
8 | (integer) 1
9 | 127.0.0.1:6379> setbit k1 3 1
10 | (integer) 0
11 | 127.0.0.1:6379> bitcount k1
12 | (integer) 2
13 | 127.0.0.1:6379> setbit "200522:active" 67 1
14 | (integer) 0
15 | 127.0.0.1:6379> setbit "200522:active" 78 1
16 | (integer) 0
```

其中 offset 必须是数字，value 只能是 0 或者 1

通过 bitcount 可以很快速的统计，比传统的关系型数据库效率高很多

1、比如统计年活跃用户数量

用户的ID作为offset，当用户在一年内访问过网站，就将对应offset的bit值设置为“1”；

通过bitcount 来统计一年内访问过网站的用户数量

2、比如统计三天内活跃用户数量

时间字符串作为key，比如“200522:active”；

用户的ID就可以作为offset，当用户访问过网站，就将对应offset的bit值设置为“1”；

统计三天的活跃用户，通过bitop or 获取一周内访问过的用户数量

3、连续三天访问的用户数量 bitop and

4、三天内没有访问的用户数量 bitop not

5、统计在线人数 设置在线key：“online: active”，当用户登录时，通过setbit设置

bitmap的优势，以统计活跃用户为例

每个用户id占用空间为1bit，消耗内存非常少，存储1亿用户量只需要12.5M

HyperLogLog (2.8)

1. 基于bitmap 计数

2. 基于概率基数基数

这个数据结构的命令有三个：PFADD、PFCOUNT、PFMERGE

内部编码主要分稀疏型和密集型

用途：记录网站IP注册数，每日访问的IP数，页面实时UV、在线用户人数

局限性：只能统计数量，没有办法看具体信息

```
1 127.0.0.1:6379> pfadd h1 b
2 (integer) 1
3 127.0.0.1:6379> pfadd h1 a
4 (integer) 0
5 127.0.0.1:6379> pfcnt h1
6 (integer) 2
7 127.0.0.1:6379> pfadd h1 c
8 (integer) 1
9 127.0.0.1:6379> pfadd h2 a
10 (integer) 1
11 127.0.0.1:6379> pfadd h3 d
12 (integer) 1
13 127.0.0.1:6379> pfmerge h3 h1 h2
14 OK
15 127.0.0.1:6379> pfcnt h3
16 (integer) 4
17
```

Geospatial (3.2)

可以用来保存地理位置，并作位置距离计算或者根据半径计算位置等。有没有想过用Redis来实现附近的人？或者计算最优地图路径？Geo本身不是一种数据结构，它本质上还是借助于Sorted Set (ZSET)

GEOADD key 经度 纬度 名称

把某个具体的位置信息（经度，纬度，名称）添加到指定的key中，数据将会用一个sorted set存储，以便稍后能使用[GEORADIUS](#)和[GEORADIUSBYMEMBER](#)命令来根据半径来查询位置信息。

```

1 127.0.0.1:6379> GEOADD cities 116.404269 39.91582 "beijing" 121.478799
2 31.235456 "shanghai"
3 (integer) 2
4 127.0.0.1:6379> ZRANGE cities 0 -1
5 1) "shanghai"
6 2) "beijing"
7 127.0.0.1:6379> ZRANGE cities 0 -1 WITHSCORES
8 1) "shanghai"
9 2) "4054803475356102"
10 3) "beijing"
11 4) "4069885555377153"
12 127.0.0.1:6379> GEODIST cities beijing shanghai km
13 "1068.5677"
14 127.0.0.1:6379> GEOPOS cities beijing shanghai
15 1) 1) "116.40426903963088989"
16 2) "39.91581928642635546"
17 2) 1) "121.47879928350448608"
18 2) "31.23545629441388627"
19 127.0.0.1:6379> GEOADD cities 120.165036 30.278973 hangzhou
20 (integer) 1
21 127.0.0.1:6379> GEORADIUS cities 120 30 500 km
22 1) "hangzhou"
23 2) "shanghai"
24 127.0.0.1:6379> GEORADIUSBYMEMBER cities shanghai 200 km
25 1) "hangzhou"
26 2) "shanghai"
27 127.0.0.1:6379> ZRANGE cities 0 -1
28 1) "hangzhou"
29 2) "shanghai"
30 3) "beijing"
31 127.0.0.1:6379>

```

Redis 消息模式

队列模式

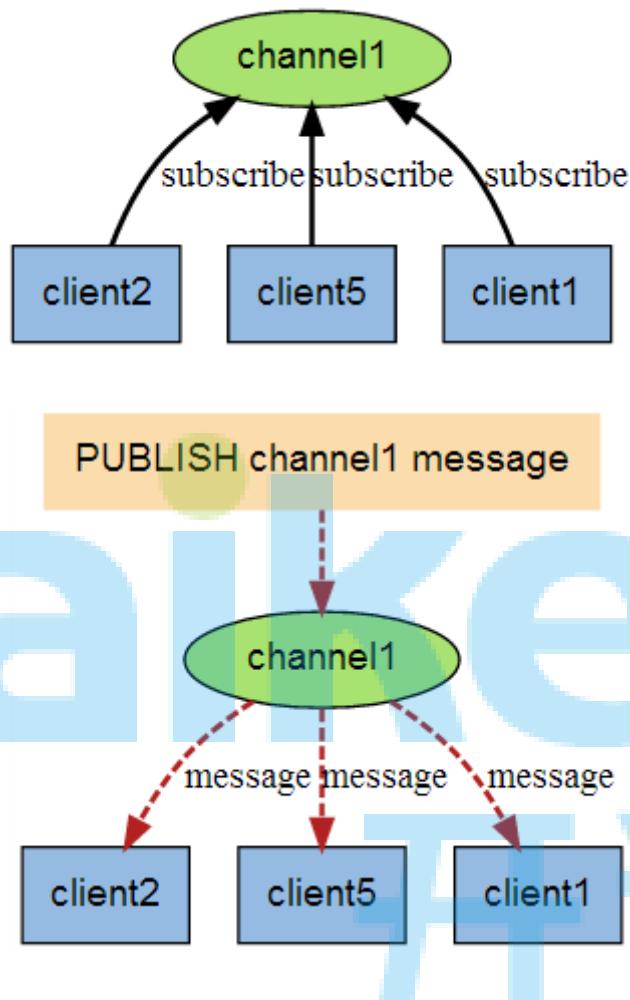
使用list类型的lpush和rpop实现消息队列



注意事项：

- 消息接收方如果不知道队列中是否有消息，会一直发送rpop命令，如果这样的话，会每一次都建立一次连接，这样显然不好。
- 可以使用brpop命令，它如果从队列中取不出来数据，会一直阻塞，在一定范围内没有取出则返回null

发布订阅模式



Redis Stream

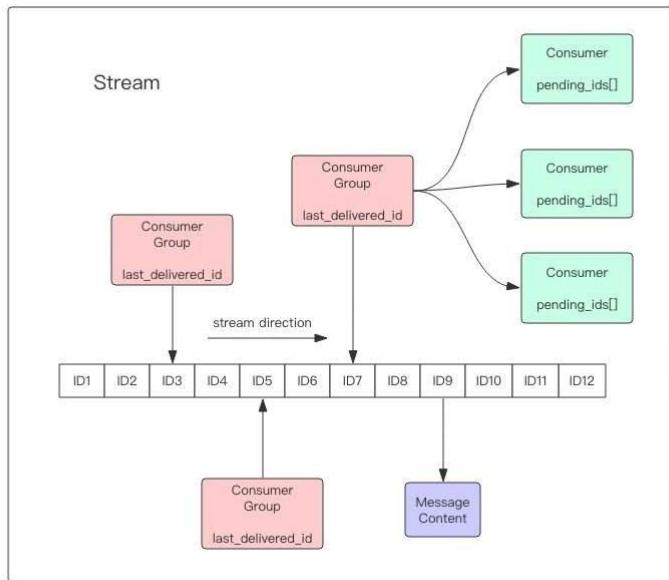
Redis 5.0 全新的数据类型：streams，官方把它定义为：以更抽象的方式建模日志的数据结构。Redis的streams主要是一个append only的数据结构，至少在概念上它是一种在内存中表示的抽象数据类型，只不过它们实现了更强大的操作，以克服日志文件本身的限制。

如果你了解MQ，那么可以把streams当做基于内存的MQ。如果你还了解kafka，那么甚至可以把streams当做基于内存的kafka。listpack存储信息，Rax组织listpack 消息链表

另外，这个功能有点类似于redis以前的Pub/Sub，但是也有基本的不同：

- streams支持多个客户端（消费者）等待数据（Linux环境开多个窗口执行XREAD即可模拟），并且每个客户端得到的是完全相同的数据。
- Pub/Sub是发送忘记的方式，并且不存储任何数据；而streams模式下，所有消息被无限期追加在streams中，除非用于显示执行删除（XDEL）。
- streams的Consumer Groups也是Pub/Sub无法实现的控制方式。

streams数据结构



它主要有消息、生产者、消费者、消费组4组成

streams数据结构本身非常简单，但是streams依然是Redis到目前为止最复杂的类型，其原因是实现的一些额外的功能：一系列的阻塞操作允许消费者等待生产者加入到streams的新数据。另外还有一个称为Consumer Groups的概念，Consumer Group概念最先由kafka提出，Redis有一个类似实现，和kafka的Consumer Groups的目的是一样的：允许一组客户端协调消费相同的信息流！

发布消息

```

1 127.0.0.1:6379> xadd mystream * message apple
2 "1589994652300-0"
3 127.0.0.1:6379> xadd mystream * message orange
4 "1589994679942-0"
5

```

读取消息

```

1 127.0.0.1:6379> xrange mystream - +
2 1) "1589994652300-0"
3   2) "message"
4     2) "apple"
5 2) "1589994679942-0"
6   2) "message"
7     2) "orange"
8

```

阻塞读取

```
1 | xread block 0 streams mystream $
```

发布新消息

```
1 | 127.0.0.1:6379> xadd mystream * message strawberry
```

创建消费组

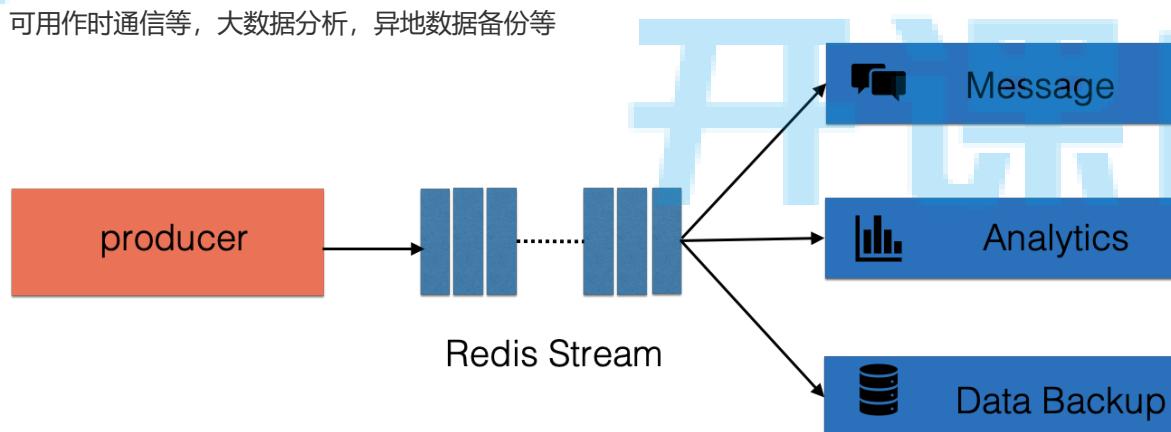
```
1 127.0.0.1:6379> xgroup create mystream mygroup1 0
2 OK
3 127.0.0.1:6379> xgroup create mystream mygroup2 0
4 OK
```

通过消费组读取消息

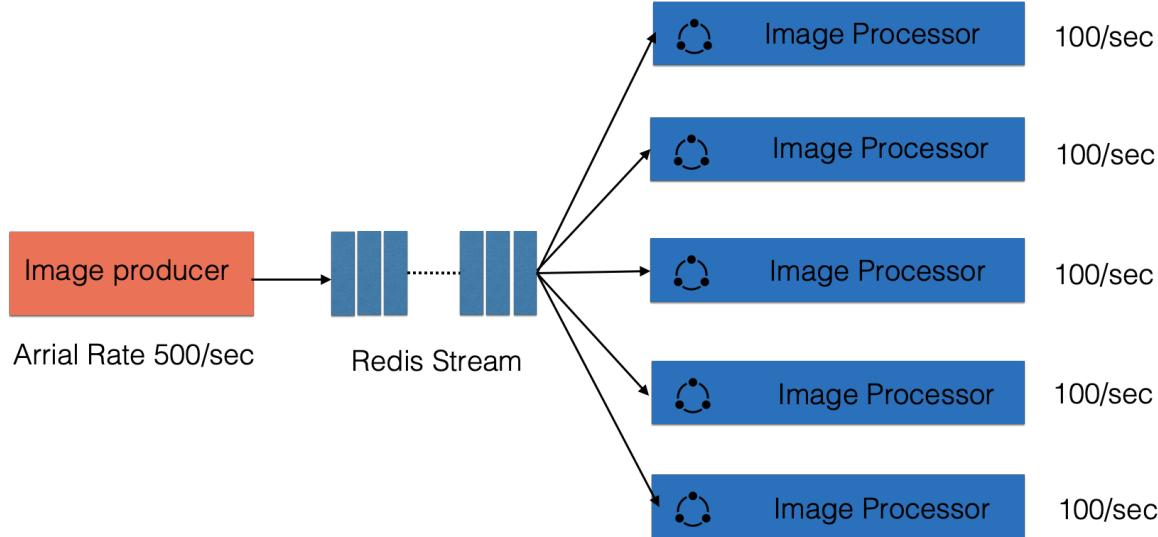
```
1 127.0.0.1:6379> xreadgroup group mygroup1 zange count 2 streams mystream >
2 1) 1) "mystream"
3 2) 1) "1589994652300-0"
4      2) 1) "message"
5          2) "apple"
6      2) 1) "1589994679942-0"
7          2) 1) "message"
8              2) "orange"
9 127.0.0.1:6379> xreadgroup group mugroup1 tuge count 2 streams mystream >
10 1) 1) "mystream"
11     2) 1) "1589995171242-0"
12         2) 1) "message"
13             2) "strawberry"
14
15 127.0.0.1:6379> xreadgroup group mugroup2 tuge count 1 streams mystream >
16 1) 1) "mystream"
17     2) 1) "1589995171242-0"
18         2) 1) "message"
19             2) "apple"
20
```

Redis Stream使用场景

可用作时通信等，大数据分析，异地数据备份等



客户端可以平滑扩展，提高处理能力



Redis 内存模型

查看Redis内存统计

```
1 127.0.0.1:6379> info memory
2 # Memory
3 #Redis分配的内存总量,包括虚拟内存(字节)
4 used_memory:853464
5 #占操作系统的内存,不包括虚拟内存(字节)
6 used_memory_rss:12247040
7 #内存碎片比例 如果小于0说明使用了虚拟内存
8 mem_fragmentation_ratio:15.07
9 #内存碎片字节数
10 mem_fragmentation_bytes
11 #Redis使用的内存分配器
12 mem_allocator:jemalloc-5.1.0
```

used_memory

由Redis内存分配器分配的数据内存和缓冲内存的内存总量(单位是字节), 包括使用的虚拟内存(即swap) ; used_memory_human只是显示更加人性化。

used_memory_rss

记录的是由操作系统分配的Redis进程内存和Redis内存中无法再被jemalloc分配的内存碎片(单位是字节)。

used_memory和used_memory_rss的区别:

前者是从Redis角度得到的量, 后者是从操作系统角度得到的量。二者之所以有所不同, 一方面是因为内存碎片和Redis进程运行需要占用内存, 使得前者可能比后者小, 另一方面虚拟内存的存在, 使得前者可能比后者大。

由于在实际应用中, Redis的数据量会比较大, 此时进程运行占用的内存与Redis数据量和内存碎片相比, 都会小得多; 因此used_memory_rss和used_memory的比例, 便成了衡量Redis内存碎片率的参数; 这个参数就是mem_fragmentation_ratio。

mem_fragmentation_ratio

内存碎片比率，该值是`used_memory_rss / used_memory`的比值。

`mem_fragmentation_ratio`一般大于1，且该值越大，内存碎片比例越大。

`mem_fragmentation_ratio < 1`，说明Redis使用了虚拟内存，由于虚拟内存的媒介是磁盘，比内存速度要慢很多，当这种情况出现时，应该及时排查，如果内存不足应该及时处理，如增加Redis节点、增加Redis服务器的内存、优化应用等。

一般来说，`mem_fragmentation_ratio`在1.03左右是比较健康的状态（对于jemalloc来说）；刚开始的`mem_fragmentation_ratio`值很大，是因为还没有向Redis中存入数据，Redis进程本身运行的内存使得`used_memory_rss`比`used_memory`大得多。

mem_allocator

Redis使用的内存分配器，在编译时指定；可以是 libc、jemalloc或者tcmalloc，默认是jemalloc；

redis 内存划分

数据

作为数据库，数据是最主要的部分；这部分占用的内存会统计在 `used_memory` 中。

Redis 使用键值对存储数据，其中的值（对象）包括 5 种类型，即字符串、哈希、列表、集合、有序集合。

这 5 种类型是 Redis 对外提供的，实际上，在 Redis 内部，每种类型可能有 2 种或更多的内部编码实现。

进程

Redis 主进程本身运行肯定需要占用内存，如代码、常量池等等；这部分内存大约几M，在大多数生产环境中与 Redis 数据占用的内存相比可以忽略。

这部分内存不是由 jemalloc 分配，因此不会统计在 `used_memory` 中。

缓冲内存

缓冲内存包括客户端缓冲区、复制积压缓冲区、AOF 缓冲区等；其中，客户端缓冲区存储客户端连接的输入输出缓冲；复制积压缓冲区用于部分复制功能；AOF 缓冲区用于在进行 AOF 重写时，保存最近的写入命令。

在了解相应功能之前，不需要知道这些缓冲的细节；这部分内存由 jemalloc 分配，因此会统计在 `used_memory` 中。

内存碎片

内存碎片是 Redis 在分配、回收物理内存过程中产生的。例如，如果对数据的更改频繁，而且数据之间的大小相差很大，可能导致 Redis 释放的空间在物理内存中并没有释放。

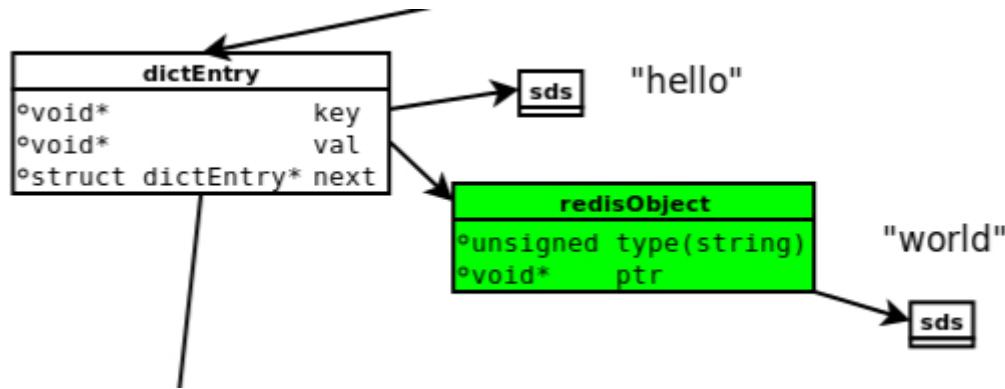
但 Redis 又无法有效利用，这就形成了内存碎片，内存碎片不会统计在 `used_memory` 中。

内存碎片的产生与对数据进行的操作、数据的特点等都有关；此外，与使用的内存分配器也有关系：如果内存分配器设计合理，可以尽可能的减少内存碎片的产生。如果 Redis 服务器中的内存碎片已经很大，可以通过安全重启的方式减小内存碎片：因为重启之后，Redis 重新从备份文件中读取数据，在内存中进行重排，为每个数据重新选择合适的内存单元，减小内存碎片。

Redis数据存储的细节

- 1 Redis 是一个K-V NOSQL
- 2 五种类型 都是针对K-V中的V的

下图是执行`set hello world`时，所涉及到的数据模型。



(1) [dictEntry](#): Redis是Key-Value数据库，因此对每个键值对都会有一个dictEntry，里面存储了指向Key和Value的指针；next指向下一个dictEntry，与本Key-Value无关。

(2) [Key](#): 图中右上角可见，[Key \("hello"\)](#) 并不是直接以字符串存储，而是[存储在SDS结构中](#)。

(3) [redisObject](#): [Value\("world"\)](#)既不是直接以字符串存储，也不是像Key一样直接存储在SDS中，而是[存储在redisObject中](#)。实际上，不论Value是5种类型的哪一种，都是通过redisObject来存储的；而redisObject中的type字段指明了Value对象的类型，ptr字段则指向对象所在的地址。不过可以看出，字符串对象虽然经过了redisObject的包装，但仍然需要通过SDS存储。

实际上，redisObject除了type和ptr字段以外，还有其他字段图中没有给出，如用于指定对象内部编码的字段；后面会详细介绍。

(4) [jemalloc](#): 无论是DictEntry对象，还是redisObject、SDS对象，都需要内存分配器（如jemalloc）分配内存进行存储。

jemalloc

Redis在编译时便会指定内存分配器；内存分配器可以是 libc、jemalloc或者tcmalloc，默认是[jemalloc](#)。

jemalloc作为Redis的默认内存分配器，在减小内存碎片方面做的相对比较好。jemalloc在64位系统中，将内存空间划分为小、大、巨大三个范围；每个范围内又划分了许多小的内存块单位；当Redis存储数据时，会选择大小最合适的内存块进行存储。

jemalloc划分的内存单元如下图所示：

Category	Spacing	Size
Small	8	[8]
	16	[16, 32, 48, ..., 128]
	32	[160, 192, 224, 256]
	64	[320, 384, 448, 512]
	128	[640, 768, 896, 1024]
	256	[1280, 1536, 1792, 2048]
Large	512	[2560, 3072, 3584]
	4 KiB	[4 KiB, 8 KiB, 12 KiB, ..., 4072 KiB]
Huge	4 MiB	[4 MiB, 8 MiB, 12 MiB, ...]

例如，如果需要存储大小为130字节的对象，jemalloc会将其放入160字节的内存单元中。

redisObject

Redis对象有5种类型；无论是哪种类型，Redis都不会直接存储，而是通过redisObject对象进行存储。

redisObject对象非常重要，Redis对象的类型、内部编码、内存回收、共享对象等功能，都需要redisObject支持，下面将通过redisObject的结构来说明它是如何起作用的。

Redis中的每个对象都是由如下结构表示（列出了与保存数据有关的三个属性）

```

1 {
2     unsigned type:4;//类型 五种对象类型
3     unsigned encoding:4;//编码
4     void *ptr;//指向底层实现数据结构的指针
5     //...
6     int refcount;//引用计数
7     //...
8     unsigned lru:24;//记录最后一次被命令程序访问的时间
9     //...
10 }robj;

```

(1) type

type字段表示对象的类型，占4个比特；目前包括REDIS STRING(字符串)、REDIS LIST(列表)、REDIS HASH(哈希)、REDIS SET(集合)、REDIS ZSET(有序集合)。

当我们执行type命令时，便是通过读取RedisObject的type字段获得对象的类型；如下图所示：

```
127.0.0.1:6379> set mystring helloredis
OK
127.0.0.1:6379> type mystring
string
127.0.0.1:6379> sadd myset member1 member2 member3
<integer> 3
127.0.0.1:6379> type myset
set
```

(2) encoding

[encoding](#)表示对象的内部编码，占4个比特。

对于Redis支持的每种类型，都有至少两种内部编码，例如对于字符串，有int、embstr、raw三种编码。[通过encoding属性，Redis可以根据不同的使用场景来为对象设置不同的编码，大大提高了Redis的灵活性和效率](#)。以列表对象为例，有压缩列表和双端链表两种编码方式；如果列表中的元素较少，Redis倾向于使用压缩列表进行存储，因为压缩列表占用内存更少，而且比双端链表可以更快载入；当列表对象元素较多时，压缩列表就会转化为更适合存储大量元素的双端链表。

通过[object encoding](#)命令，可以查看对象采用的编码方式，如下图所示：

```
127.0.0.1:6379> set key1 33
OK
127.0.0.1:6379> object encoding key1
"int"
127.0.0.1:6379> set key2 helloworld
OK
127.0.0.1:6379> object encoding key2
"embstr"
```

5种对象类型对应的编码方式以及使用条件，将在后面介绍。

(3) lru

[lru](#)记录的是对象最后一次被命令程序访问的时间，占据的比特数不同的版本有所不同（2.6版本占22比特，4.0版本占24比特）。

通过对比lru时间与当前时间，可以计算某个对象的闲置时间；[object idletime](#)命令可以显示该闲置时间（单位是秒）。[object idletime](#)命令的一个特殊之处在于它不改变对象的lru值。

```
127.0.0.1:6379> set mystring helloredis
OK
127.0.0.1:6379> object idletime mystring
<integer> 9
127.0.0.1:6379> object idletime mystring
<integer> 12
127.0.0.1:6379> object idletime mystring
<integer> 82
127.0.0.1:6379> get mystring
"helloredis"
127.0.0.1:6379> object idletime mystring
<integer> 4
```

lru值除了通过object idletime命令打印之外，还与Redis的内存回收有关系：如果Redis打开了[maxmemory](#)选项，且内存回收算法选择的是[volatile-lru](#)或[allkeys-lru](#)，那么当Redis内存占用超过maxmemory指定的值时，Redis会优先选择空转时间最长的对象进行释放。

(4) refcount

[refcount](#)与共享对象

refcount记录的是该对象被引用的次数，类型为整型。 refcount的作用，主要在于对象的引用计数和内存回收。

当创建新对象时，refcount初始化为1；当有新程序使用该对象时，refcount加1；当对象不再被一个新程序使用时，refcount减1；当refcount变为0时，对象占用的内存会被释放。

Redis中被多次使用的对象(refcount>1)，称为共享对象。 Redis为了节省内存，当有一些对象重复出现时，新的程序不会创建新的对象，而是仍然使用原来的对象。这个被重复使用的对象，就是共享对象。目前共享对象仅支持整数值的字符串对象。

共享对象的具体实现

Redis的共享对象目前只支持整数值的字符串对象。之所以如此，实际上是对内存和CPU（时间）的平衡：共享对象虽然会降低内存消耗，但是判断两个对象是否相等却需要消耗额外的时间。对于整数值，判断操作复杂度为O(1)；对于普通字符串，判断复杂度为O(n)；而对于哈希、列表、集合和有序集合，判断的复杂度为O(n^2)。

虽然共享对象只能是整数值的字符串对象，但是5种类型都可能使用共享对象（如哈希、列表等的元素可以使用）。

就目前的实现来说，Redis服务器在初始化时，会创建10000个字符串对象，值分别是0~9999的整数值：当Redis需要使用值为0~9999的字符串对象时，可以直接使用这些共享对象。10000这个数字可以通过调整参数REDIS_SHARED_INTEGERS (4.0中是OBJ_SHARED_INTEGERS) 的值进行改变。

共享对象的引用次数可以通过object refcount命令查看，如下图所示。命令执行的结果页佐证了只有0~9999之间的整数会作为共享对象。

```
127.0.0.1:6379> set k1 9999
OK
127.0.0.1:6379> set k2 9999
OK
127.0.0.1:6379> set k3 9999
OK
127.0.0.1:6379> object refcount k1
<integer> 4
127.0.0.1:6379> set k4 10000
OK
127.0.0.1:6379> set k5 10000
OK
127.0.0.1:6379> set k6 10000
OK
127.0.0.1:6379> object refcount k4
<integer> 1
127.0.0.1:6379> set k7 hello
OK
127.0.0.1:6379> set k8 hello
OK
127.0.0.1:6379> set k9 hello
OK
127.0.0.1:6379> object refcount k7
<integer> 1
127.0.0.1:6379>
```

(5) ptr

ptr指针指向具体的数据，如前面的例子中，set hello world，ptr指向包含字符串world的SDS。

(6) 总结

综上所述，redisObject的结构与对象类型、编码、内存回收、共享对象都有关系；一个redisObject对象的大小为16字节：

4bit+4bit+24bit+4Byte+8Byte=16Byte

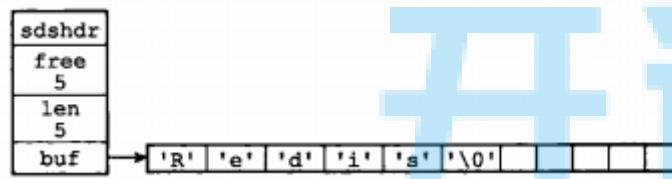
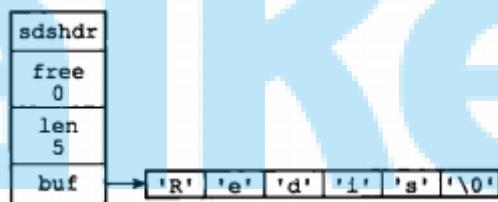
SDS

Redis没有直接使用C字符串(即以空字符'\0'结尾的字符数组)作为默认的字符串表示，而是使用了SDS。SDS是简单动态字符串(Simple Dynamic String)的缩写。

3.2 之前

```
1 struct sdshdr{  
2     //记录buf数组中已使用字节的数量  
3     //等于 SDS 保存字符串的长度  
4     int len;  
5     //记录 buf 数组中未使用字节的数量  
6     int free;  
7     //字节数组，用于保存字符串  
8     char buf[];  
9 }
```

其中，buf表示字节数组，用来存储字符串；len表示buf已使用的长度，free表示buf未使用的长度。下面是两个例子。



通过SDS的结构可以看出，buf数组的长度=free+len+1（其中1表示字符串结尾的空字符）；所以，一个SDS结构占据的空间为：free所占长度+len所占长度+buf数组的长度
=4+4+free+len+1=free+len+9。

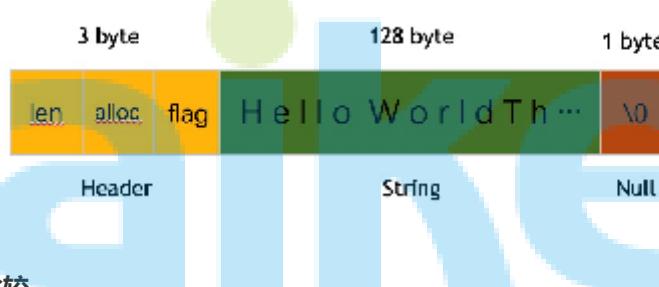
3.2 之后

```
1 typedef char *sds;  
2  
3 struct __attribute__((__packed__)) sdshdr5 {      // 对应的字符串长度小于 1<<5  
4     unsigned char flags; /* 3 lsb of type, and 5 msb of string length */  
5     char buf[];  
6 };  
7 struct __attribute__((__packed__)) sdshdr8 {      // 对应的字符串长度小于 1<<8  
8     uint8_t len; /* used */  
9     uint8_t alloc;  
10    unsigned char flags;  
后续解释，其余5bit目前没有使用
```

```

11     char buf[];                                // 柔性数组, 以'\0'结尾
12 };
13 struct __attribute__ ((__packed__)) sdshdr16 {    // 对应的字符串长度小于 1<<16
14     uint16_t len; /*已使用长度, 用2字节存储*/
15     uint16_t alloc; /* 总长度, 用2字节存储*/
16     unsigned char flags; /* 3 lsb of type, 5 unused bits */
17     char buf[];
18 };
19 struct __attribute__ ((__packed__)) sdshdr32 {    // 对应的字符串长度小于 1<<32
20     uint32_t len; /*已使用长度, 用4字节存储*/
21     uint32_t alloc; /* 总长度, 用4字节存储*/
22     unsigned char flags; /* 低3位存储类型, 高5位预留 */
23     char buf[]; /*柔性数组, 存放实际内容*/
24 };
25 struct __attribute__ ((__packed__)) sdshdr64 {    // 对应的字符串长度小于 1<<64
26     uint64_t len; /*已使用长度, 用8字节存储*/
27     uint64_t alloc; /* 总长度, 用8字节存储*/
28     unsigned char flags; /* 低3位存储类型, 高5位预留 */
29     char buf[]; /*柔性数组, 存放实际内容*/
30 };
31

```



SDS与C字符串的比较

- 获取字符串长度**: SDS是O(1), C字符串是O(n)
- 缓冲区溢出**: 使用C字符串的API时, 如果字符串长度增加 (如strcat操作) 而忘记重新分配内存, 很容易造成缓冲区的溢出; 而SDS由于记录了长度, 相应的API在可能造成缓冲区溢出时会自动重新分配内存, 杜绝了缓冲区溢出。
- 修改字符串时内存的重分配**: 对于C字符串, 如果要修改字符串, 必须要重新分配内存 (先释放再申请), 因为如果没有重新分配, 字符串长度增大时会造成内存缓冲区溢出, 字符串长度减小时会造成内存泄露。而对于SDS, 由于可以记录len和free, 因此解除了字符串长度和空间数组长度之间的关联, 可以在此基础上进行优化: 空间预分配策略 (即分配内存时比实际需要的多) 使得字符串长度增大时重新分配内存的概率大大减小; 惰性空间释放策略使得字符串长度减小时重新分配内存的概率大大减小。
- 存取二进制数据**: SDS可以, C字符串不可以。因为C字符串以空字符作为字符串结束的标识, 而对于一些二进制文件 (如图片等), 内容可能包括空字符串, 因此C字符串无法正确存取; 而SDS以字符串长度len来作为字符串结束标识, 因此没有这个问题。

此外, 由于SDS中的buf仍然使用了C字符串 (即以'\0'结尾), 因此SDS可以使用C字符串库中的部分函数; 但是需要注意的是, 只有当SDS用来存储文本数据时才可以这样使用, 在存储二进制数据时则不行 ('\0'不一定是结尾)。

Redis的对象类型与内存编码

Redis支持5种对象类型, 而每种结构都有至少两种编码;

这样做的好处在于:

一方面接口与实现分离，当需要增加或改变内部编码时，用户使用不受影响；

另一方面可以根据不同的应用场景切换内部编码，提高效率。

Redis各种对象类型支持的内部编码如下图所示(只列出重点的)：

Redis各种对象类型支持的内部编码如下图所示(只列出重点的)：

类型	编码	OBJECT ENCODING命令输出	对象
REDIS_STRING	REDIS_ENCODING_INT	“int”	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	“embstr”	使用embstr编码的简单动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	“raw”	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	“ziplist”	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	“linkedlist”	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	“ziplist”	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	“hashtable”	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	“intset”	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	“hashtable”	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	“ziplist”	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	“skiplist”	使用跳跃表和字典实现的有序集合对象

1 字符串 (SDS)

(1) 概况

字符串是最基础的类型，因为所有的键都是字符串类型，且字符串之外的其他几种复杂类型的元素也是字符串。

[字符串长度不能超过512MB。](#)

(2) 内部编码

字符串类型的内部编码有3种，它们的应用场景如下：

- [int](#): [8个字节的长整型](#)。字符串值是整型时，这个值使用long整型表示。
- [embstr](#): [<=44字节的字符串](#)。embstr与raw都使用redisObject和sds保存数据，[区别在于](#)，[embstr的使用只分配一次内存空间（因此redisObject和sds是连续的）](#)，而raw需要分配两次内存空间（分别为redisObject和sds分配空间）。因此与raw相比，embstr的好处在于创建时少分配一次空间，删除时少释放一次空间，以及对象的所有数据连在一起，寻找方便。[而embstr的坏处也很明显](#)，[如果字符串的长度增加需要重新分配内存时，整个redisObject和sds都需要重新分配空间](#)，因此redis中的embstr实现为只读。

- raw: 大于44个字节的字符串

3.2之后 embstr和raw进行区分的长度，是44；是因为redisObject的长度是16字节，sds的长度是4+字符串长度；因此当字符串长度是44时，embstr的长度正好是16+4+44 =64，jemalloc正好可以分配64字节的内存单元。

3.2之前embstr和raw进行区分的长度，是39，因为redisObject的长度是16字节，sds的长度是9+字符串长度；因此当字符串长度是44时，embstr的长度正好是16+9+39 =64，jemalloc正好可以分配64字节的内存单元。

2 列表

(1) 概况

列表 (list) 用来存储多个有序的字符串，每个字符串称为元素；

一个列表可以存储 $2^{32}-1$ 个元素。

[Redis中的列表](#)支持两端插入和弹出，并可以获得指定位置（或范围）的元素，[可以充当数组、队列、栈等](#)。

(2) 内部编码

Redis3.0之前列表的内部编码可以是[压缩列表 \(ziplist\)](#) 或[双端链表 \(linkedlist\)](#)。选择的折中方案是两种数据类型的转换，但是在3.2版本之后 因为转换也是个费时且复杂的操作，引入了一种新的数据格式，结合了双向列表linkedlist和ziplist的特点，称之为quicklist。所有的节点都用quicklist存储，省去了到临界条件是的格式转换。

(3) 压缩列表

压缩列表 (ziplist) 是列表键和哈希键的底层实现之一。当一个列表只包含少量列表项时，并且每个列表项时小整数值或短字符串，那么Redis会使用压缩列表来做该列表的底层实现。

压缩列表 (ziplist) 是Redis为了节省内存而开发的，是由一系列特殊编码的连续内存块组成的顺序型数据结构，一个压缩列表可以包含任意多个节点 (entry)，每个节点可以保存一个字节数组或者一个整数值，放到一个连续内存区。

压缩列表的每个节点构成如下：

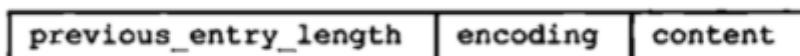


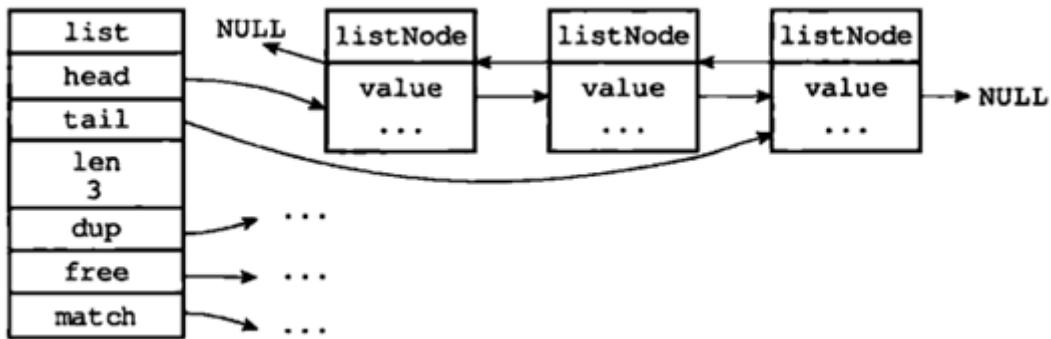
图 7-4 压缩列表节点的各个组成部分

- previous_entry_length: 记录压缩列表前一个字节的长度。
- encoding: 节点的encoding保存的是节点的content的内容类型
- content: content区域用于保存节点的内容，节点内容类型和长度由encoding决定。

(4) 双向链表

双向链表：由一个list结构和多个listNode结构组成；

典型结构如下图所示：

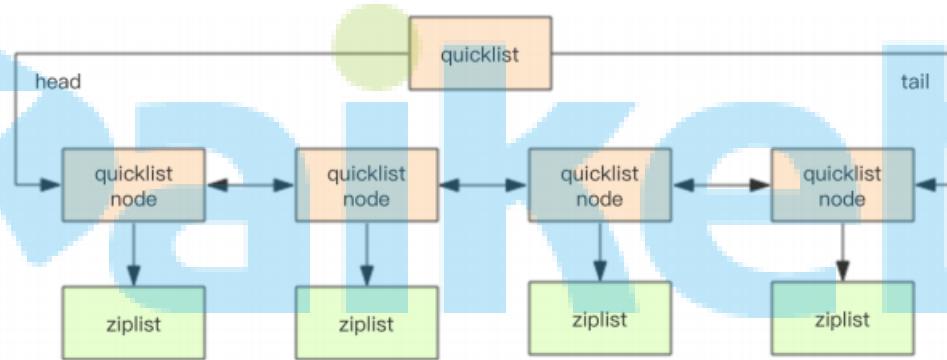


通过图中可以看出，双端链表同时保存了表头指针和表尾指针，并且每个节点都有指向前和指向后的指针；链表中保存了列表的长度；dup、free和match为节点值设置类型特定函数，所以链表可以用于保存各种不同类型的值。而链表中每个节点指向的是type为字符串的redisObject。

(5) 快速列表

简单的说，我们仍旧可以将其看作一个双向列表，但是列表的每个节点都是一个ziplist，其实质是linkedlist和ziplist的结合。quicklist中的每个节点ziplist都能够存储多个数据元素。

Redis3.2开始，列表采用quicklist进行编码。



```

1 //32byte 的空间
2 typedef struct quicklist {
3     // 指向quicklist的头部
4     quicklistNode *head;
5     // 指向quicklist的尾部
6     quicklistNode *tail;
7     // 列表中所有数据项的个数总和
8     unsigned long count;
9     // quicklist节点的个数, 即ziplist的个数
10    unsigned int len;
11    // ziplist大小限定, 由list-max-ziplist-size给定
12    // 表示不用整个int存储fill, 而是只用了其中的16位来存储
13    int fill : 16;
14    // 节点压缩深度设置, 由list-compress-depth给定
15    unsigned int compress : 16;
16 } quicklist;
17
18 typedef struct quicklistNode {
19     struct quicklistNode *prev; // 指向上一个ziplist节点
20     struct quicklistNode *next; // 指向下一个ziplist节点
21     unsigned char *z1; // 数据指针, 如果没有被压缩, 就指向ziplist结构,
// 反之指向quicklistLZF结构
22     unsigned int sz; // 表示指向ziplist结构的总长度(内存占用长度)
23     unsigned int count : 16; // 表示ziplist中的数据项个数
}

```

```

24     unsigned int encoding : 2; // 编码方式, 1--ziplist, 2--quicklistLZF
25     unsigned int container : 2; // 预留字段, 存放数据的方式, 1--NONE, 2--ziplist
26     unsigned int recompress : 1; // 解压标记, 当查看一个被压缩的数据时, 需要暂时解
压, 标记此参数为1, 之后再重新进行压缩
27     unsigned int attempted_compress : 1; // 测试相关
28     unsigned int extra : 10; // 扩展字段, 暂时没用
29 } quicklistNode;

```

3 哈希 (压缩列表和哈希表)

(1) 概况

哈希（作为一种数据结构），不仅是redis对外提供的5种对象类型的一种（与字符串、列表、集合、有序结合并列），也是Redis作为Key-Value数据库所使用的数据结构。为了说明的方便，后面当使用“内层的哈希”时，代表的是redis对外提供的5种对象类型的一种；使用“外层的哈希”代指Redis作为Key-Value数据库所使用的数据结构。

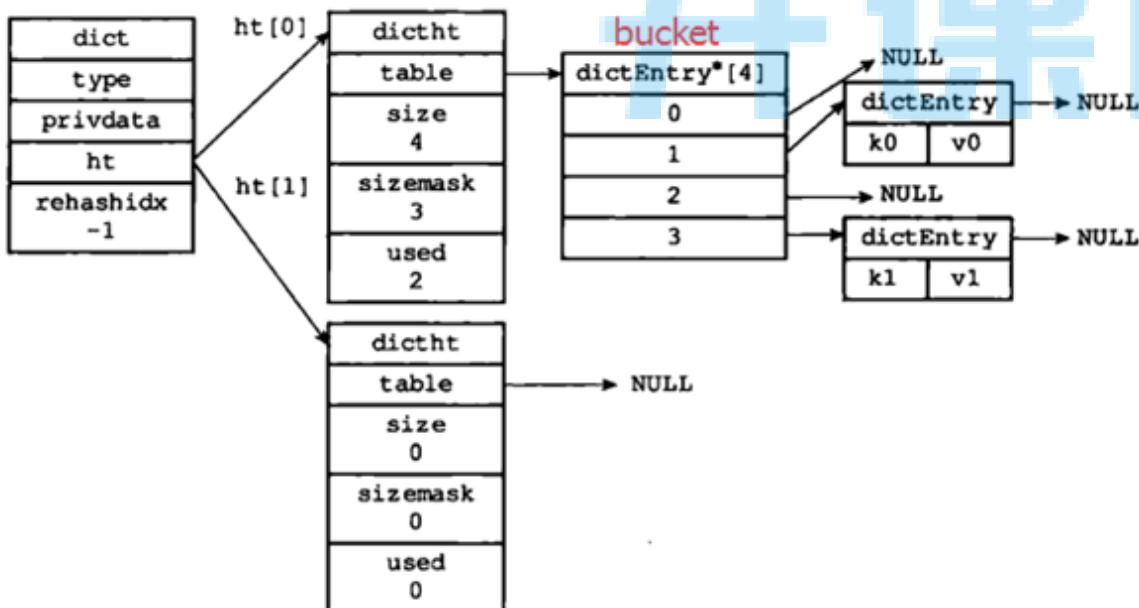
(2) 内部编码

内层的哈希使用的内部编码可以是压缩列表 (ziplist) 和哈希表 (hashtable) 两种；Redis的外层的哈希则只使用了hashtable。

压缩列表前面已介绍。与哈希表相比，压缩列表用于元素个数少、元素长度小的场景；其优势在于集中存储，节省空间；同时，虽然对于元素的操作复杂度也由O(1)变为了O(n)，但由于哈希中元素数量较少，因此操作的时间并没有明显劣势。

hashtable：一个hashtable由1个dict结构、2个dictht结构、1个dictEntry指针数组（称为bucket）和多个dictEntry结构组成。

正常情况下（即hashtable没有进行rehash时）各部分关系如下图所示：



dict

一般来说，通过使用dictht和dictEntry结构，便可以实现普通哈希表的功能；但是Redis的实现中，在dictht结构的上层，还有一个dict结构。下面说明dict结构的定义及作用。

dict结构如下：

```
1 | typedef struct dict{
2 |     dictType *type; // type里面主要记录了一系列的函数，可以说是规定了一系列的接口
3 |     void *privdata; // privdata保存了需要传递给那些类型特定函数的可选参数
4 |     //两张哈希表
5 |     dictht ht[2]; //便于渐进式rehash
6 |     int trehashidx; //rehash 索引，并没有rehash时，值为 -1
7 |     //目前正在运行的安全迭代器的数量
8 |     int iterators;
9 | } dict;
```

```
1 | typedef struct dict{
2 |     dictType *type;
3 |     void *privdata;
4 |     dictht ht[2];
5 |     int trehashidx; //rehash
6 |     int iterators;
7 | } dict;
```

其中，type属性和privdata属性是为了适应不同类型的键值对，用于创建多态字典。

ht属性和trehashidx属性则用于rehash，即当哈希表需要扩展或收缩时使用。ht是一个包含两个项的数据组，每项都指向一个dictht结构，这也是Redis的哈希会有1个dict、2个dictht结构的原因。通常情况下，所有的数据都是存在放dict的ht[0]中，ht[1]只在rehash的时候使用。dict进行rehash操作的时候，将ht[0]中的所有数据rehash到ht[1]中。然后将ht[1]赋值给ht[0]，并清空ht[1]。

因此，Redis中的哈希之所以在dictht和dictEntry结构之外还有一个dict结构，一方面是为了适应不同类型的键值对，另一方面是为了rehash。

dictht

dictht结构如下：

```
1 | typedef struct dictht{
2 |     //哈希表数组，每个元素都是一条链表
3 |     dictEntry **table;
4 |     //哈希表大小
5 |     unsigned long size;
6 |     // 哈希表大小掩码，用于计算索引值
7 |     // 总是等于 size - 1
8 |     unsigned long sizemask;
9 |     // 该哈希表已有节点的数量
10 |    unsigned long used;
11 | }dictht;
```

其中，各个属性的功能说明如下：

- table属性是一个指针，指向bucket；
- size属性记录了哈希表的大小，即bucket的大小；
- used记录了已使用的dictEntry的数量；
- sizemask属性的值总是为size-1，这个属性和哈希值一起决定一个键在table中存储的位置。

bucket

bucket是一个数组，数组的每个元素都是指向dictEntry结构的指针。redis中bucket数组的大小计算规则如下：大于dictEntry的、最小的 2^n ；

例如，如果有1000个dictEntry，那么bucket大小为1024；如果有1500个dictEntry，则bucket大小为2048。

dictEntry

dictEntry结构用于保存键值对，结构定义如下：

```
// 键
1 typedef struct dictEntry{
2     void *key;
3     union{ //值v的类型可以是以下三种类型
4         void *val;
5         uint64_t u64;
6         int64_t s64;
7     }v;
8     // 指向下一个哈希表节点，形成链表
9     struct dictEntry *next;
10 }dictEntry;
```

其中，各个属性的功能如下：

- key: 键值对中的键
 - val: 键值对中的值, 使用union(即共用体)实现, 存储的内容既可能是一个指向值的指针, 也可能
是64位整型, 或无符号64位整型;
 - next: 指向下一个dictEntry, 用于解决哈希冲突问题

在64位系统中，一个dictEntry对象占24字节（key/val/next各占8字节）。

(3) 编码转换

如前所述，Redis中内层的哈希既可能使用哈希表，也可能使用压缩列表。

只有同时满足下面两个条件时，才会使用压缩列表：

- 哈希中元素数量小于512个；
 - 哈希中所有键值对的键和值字符串长度都小于64字节。

下图展示了Redis内层的哈希编码转换的特点：

4、集合（整数集合和哈希表）

(1) 概况

[集合 \(set\)](#) 与列表类似，都是用来保存多个字符串，但集合与列表有两点不同：集合中的元素是无序的，因此不能通过索引来操作元素；集合中的元素不能有重复。

一个集合中最多可以存储 $2^{32}-1$ 个元素；除了支持常规的增删改查，Redis还支持多个集合取交集、并集、差集。

(2) 内部编码

集合的内部编码可以是[整数集合 \(intset\)](#) 或[哈希表 \(hashtable\)](#)。

哈希表前面已经讲过，这里略过不提；需要注意的是，集合在使用哈希表时，值全部被置为null。

整数集合的结构定义如下：

```
1 typedef struct intset{  
2     uint32_t encoding;      // 编码方式  
3     uint32_t length;        // 集合包含的元素数量  
4     int8_t contents[];      // 保存元素的数组  
5 } intset;
```

其中，encoding代表contents中存储内容的类型，虽然contents（存储集合中的元素）是int8_t类型，但实际上其存储的值是int16_t、int32_t或int64_t，具体的类型便是由encoding决定的；length表示元素个数。

整数集合适用于集合所有元素都是整数且集合元素数量较小的时候，与哈希表相比，整数集合的优势在于集中存储，节省空间；同时，虽然对于元素的操作复杂度也由O(1)变为了O(n)，但由于集合数量较少，因此操作的时间并没有明显劣势。

(3) 编码转换

只有同时满足下面两个条件时，集合才会使用整数集合：

- 集合中[元素数量小于512个](#)；
- 集合中[所有元素都是整数值](#)。

如果有一个条件不满足，则使用哈希表；且编码只可能[由整数集合转化为哈希表，反方向则不可能](#)。

下图展示了集合编码转换的特点：

```
127.0.0.1:6379> sadd myset 111 222 333  
<integer> 3  
127.0.0.1:6379> object encoding myset  
"intset"  
127.0.0.1:6379> sadd myset helloworld  
<integer> 1  
127.0.0.1:6379> object encoding myset  
"hashtable"  
127.0.0.1:6379> srem myset helloworld  
<integer> 1  
127.0.0.1:6379> object encoding myset  
"hashtable"
```

5 有序集合 (压缩列表和跳跃表)

(1) 概况

有序集合与集合一样，元素都不能重复；但与集合不同的是，有序集合中的元素是有顺序的。与列表使用索引下标作为排序依据不同，有序集合为每个元素设置一个分数 (score) 作为排序依据。

(2) 内部编码

有序集合的内部编码可以是[压缩列表（ziplist）](#) 或[跳跃表（skiplist）](#)。ziplist在列表和哈希中都有使用，前面已经讲过，这里略过不提。

跳跃表是一种有序数据结构，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。除了跳跃表，实现有序数据结构的另一种典型实现是平衡树；大多数情况下，跳跃表的效率可以和平衡树媲美，且跳跃表实现比平衡树简单很多，因此redis中选用跳跃表代替平衡树。跳跃表支持平均 $O(\log N)$ 、最坏 $O(N)$ 的复杂度进行节点查找，并支持顺序操作。

[Redis的跳跃表实现由`zskiplist`和`zskiplistNode`两个结构组成：前者用于保存跳跃表信息（如头结点、尾节点、长度等），后者用于表示跳跃表节点。

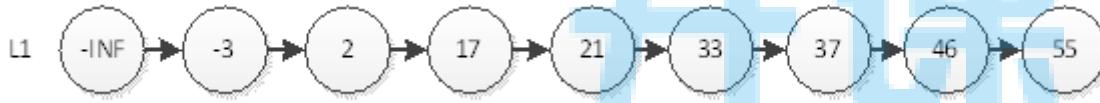
只有同时满足下面两个条件时，才会使用压缩列表：有序集合中元素数量小于128个；有序集合中所有成员长度都不足64字节。如果有一个条件不满足，则使用跳跃表；且编码只可能由压缩列表转化为跳跃表，反方向则不可能。

下图展示了有序集合编码转换的特点：

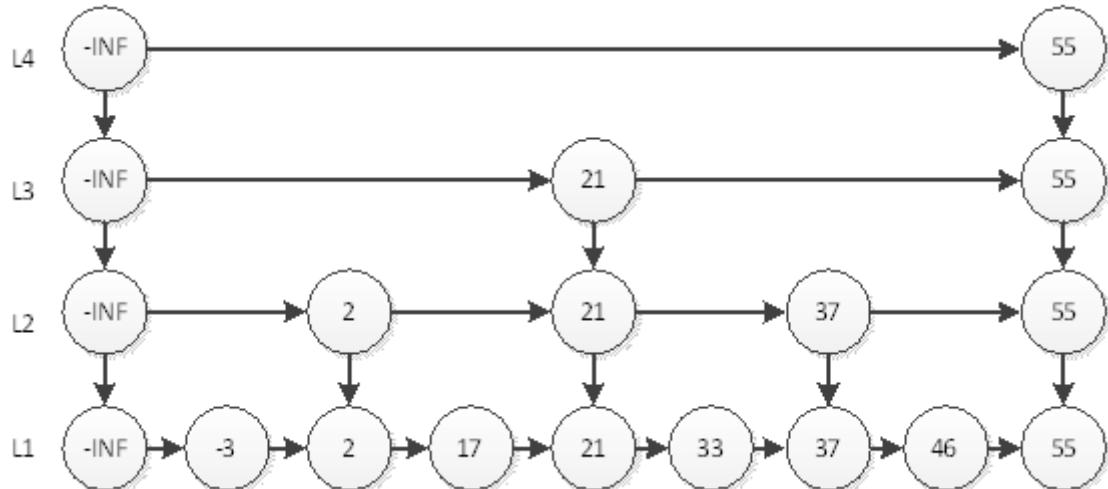
(3) 跳跃表

图示

普通单向链表图示：



跳跃表图示：



查询

查找一个节点时，我们只需从高层到低层，一个个链表查找，每次找到该层链表中小于等于目标节点的最大节点，直到找到为止。由于高层的链表迭代时会“跳过”低层的部分节点，所以跳跃表会比正常的链表查找少查部分节点，这也是skiplist名字的由来。

例如：

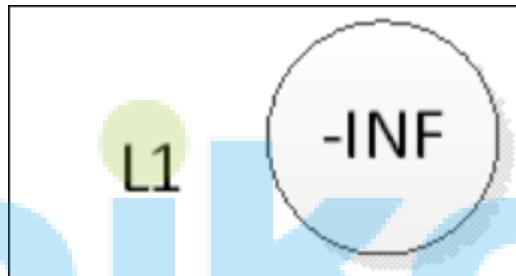
查找46： 55---21---55--37--55--46

插入

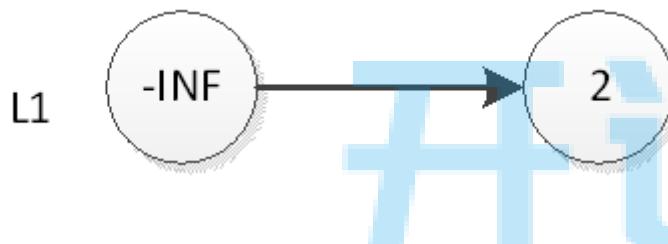
L1 层

概率算法

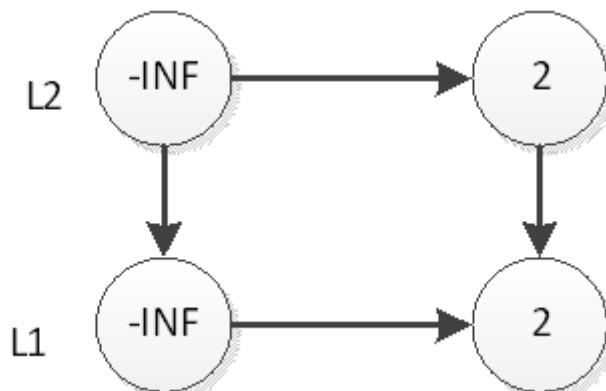
在此还是以上图为例：跳跃表的初试状态如下图，表中没有一个元素：



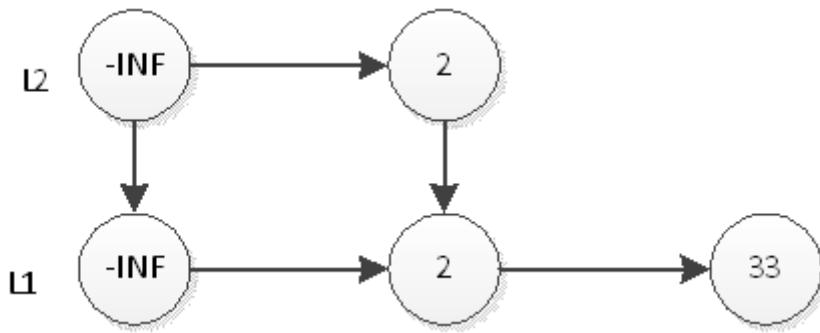
如果我们要插入元素2，首先是在底部插入元素2，如下图：



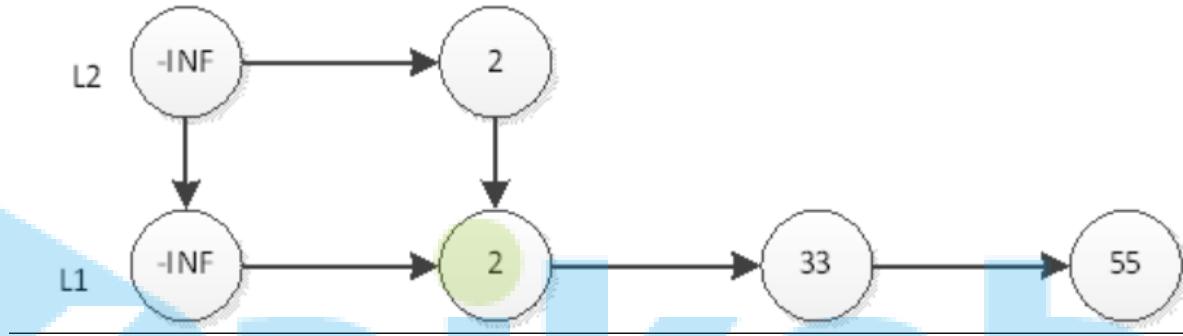
然后我们抛硬币，结果是正面，那么我们要将2插入到L2层，如下图



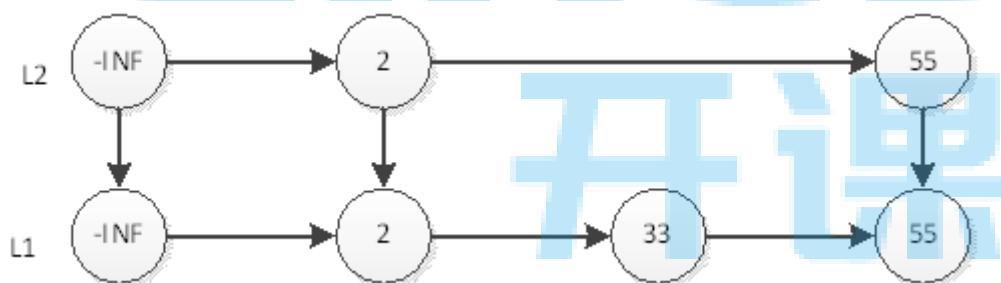
继续抛硬币，结果是反面，那么元素2的插入操作就停止了，插入后的表结构就是上图所示。接下来，我们插入元素33，跟元素2的插入一样，现在L1层插入33，如下图：



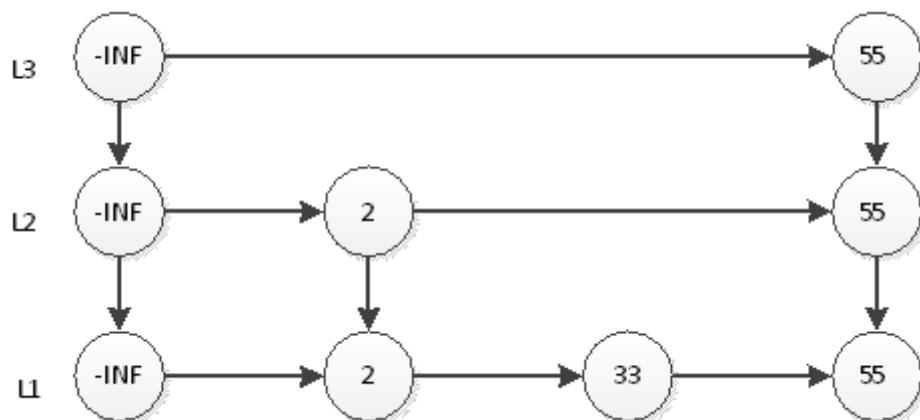
然后抛硬币，结果是反面，那么元素33的插入操作就结束了，插入后的表结构就是上图所示。接下来，我们插入元素55，首先在L1插入55，插入后如下图：



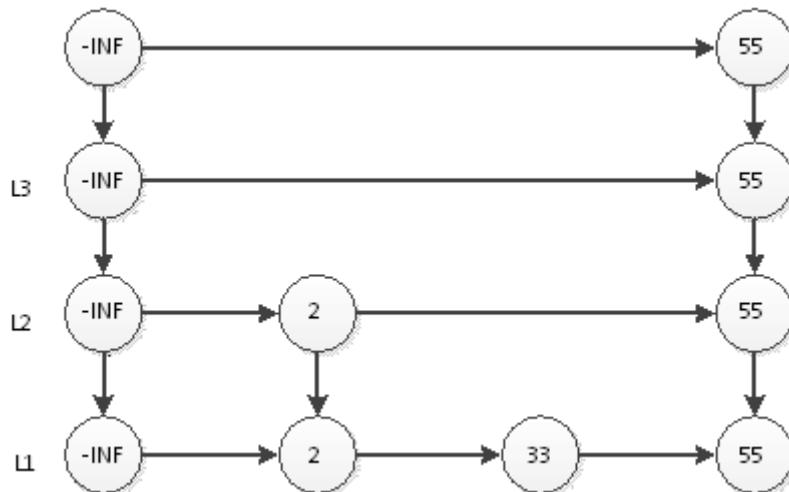
然后抛硬币，结果是正面，那么L2层需要插入55，如下图：



继续抛硬币，结果又是正面，那么L3层需要插入55，如下图：



继续抛硬币，结果又是正面，那么要在L4插入55，结果如下图：

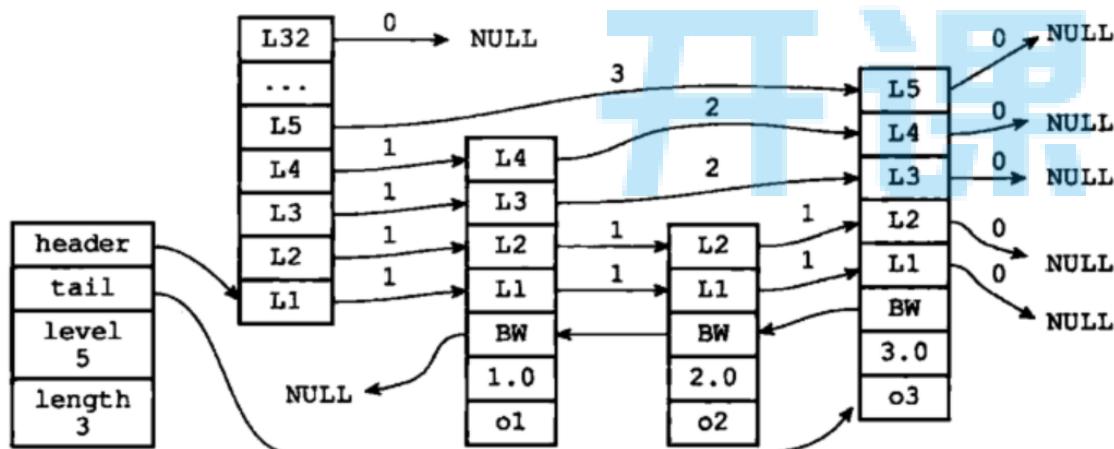


继续抛硬币，结果是反面，那么55的插入结束，表结构就如上图所示。

以此类推，我们插入剩余的元素。当然因为规模小，结果很可能不是一个理想的跳跃表。但是如果元素个数n的规模很大，学过概率论的同学都知道，最终的表结构肯定非常接近于理想跳跃表（隔一个一跳）。

删除

直接删除元素，然后调整一下删除元素后的指针即可。跟普通的链表删除操作完全一样。



总结

①、搜索：从最高层的链表节点开始，如果比当前节点要大和比当前层的下一个节点要小，那么则往下找，也就是和当前层的下一层的节点的下一个节点进行比较，以此类推，一直找到最底层的最后一个节点，如果找到则返回，反之则返回空。

②、插入：首先确定插入的层数，有一种方法是假设抛一枚硬币，如果是正面就累加，直到遇见反面为止，最后记录正面的次数作为插入的层数。当确定插入的层数k后，则需要将新元素插入到从底层到k层。

③、删除：在各个层中找到包含指定值的节点，然后将节点从链表中删除即可，如果删除以后只剩下头尾两个节点，则删除这一层。

跳跃表的完整实现

```
1 typedef struct zskiplistNode {
2     //层
3     struct zskiplistLevel{
4         //前进指针 后边的节点
5         struct zskiplistNode *forward;
6         //跨度
7         unsigned int span;
8     }level[];
9
10    //后退指针
11    struct zskiplistNode *backward;
12    //分值
13    double score;
14    //成员对象
15    robj *obj;
16
17 } zskiplistNode
18
19 --链表
20 typedef struct zskiplist{
21     //表头节点和表尾节点
22     struct zskiplistNode *header, *tail;
23     //表中节点的数量
24     unsigned long length;
25     //表中层数最大的节点的层数
26     int level;
27
28 }zskiplist;
```

附录

Redis版本历史

1. Redis2.6 Redis2.6在2012年正式发布
2. Redis2.8
Redis2.8在2013年11月22日正式发布
Redis Sentinel第二版，相比于Redis2.6的Redis Sentinel，此版本已经变成生产可用。
3. Redis3.0
Redis3.0在2015年4月1日正式发布
4. Redis3.2
Redis3.2在2016年5月6日正式发布，集群高可用
5. Redis4.0
Redis 4.0在2017年7月发布为GA，主要是增加了混合持久化和LFU淘汰策略
6. Redis5.0
Redis5.0 2018年10月18日正式发布，Stream 是重要新增特性

Redis 5.0 源码清单

1. 基本数据结构
动态字符串sds.c
整数集合intset.c
压缩列表ziplist.c
快速链表quicklist.c
字典dict.c

2. Redis数据类型的底层实现
Redis对象object.c
字符串t_string.c
列表t_list.c
字典t_hash.c
集合及有序集合t_set.c和t_zset.c

3. Redis数据库的实现
数据库的底层实现db.c
持久化rdb.c和aof.c

4. Redis服务端和客户端实现
事件驱动ae.c和ae_epoll.c
网络连接anet.c和networking.c
服务端程序server.c
客户端程序redis-cli.c

5. 集群相关
主从复制replication.c
哨兵sentinel.c
集群cluster.c

6. 特殊数据类型
其他数据结构，如hyperloglog.c、geo.c
数据流t_stream.c
Streams的底层实现结构listpack.c和rax.c

开课吧

课程主题

Redis内存优化与 架构原理

课程目标

1. 熟悉缓存类型
2. 掌握Memcache和Redis的对比
3. 估算Redis内存使用量及设计优化
4. 掌握Redis的内存淘汰机制么
5. 理解LRU并掌握Redis缓存淘汰策略
6. 掌握 redis内存优化方案
7. 掌握Redis持久化机制
8. 掌握选择持久化方案
9. 掌握Redis的主从同步机制

知识要点

课程主题

课程目标

知识要点

缓存通识

缓存类型

淘汰策略

Memcache特点

Reids 特点

Redis6.0 引入了多线程

Redis 设计优化

1 估算Redis内存使用量

2 优化内存占用

(1) 利用jemalloc特性进行优化

(2) 使用整型/长整型

(3) 共享对象

(4) 缩短键值对的存储长度

基于Redis内存淘汰机制的优化

设置键值的过期时间

使用 lazy free 特性

限制 Redis 内存大小 (重点)

最大缓存

LRU原理

案例分析

LFU原理

Redis缓存淘汰策略

Redis 持久化

RDB

触发RDB快照的时机

设置快照规则

RDB快照的实现原理

AOF

AOF介绍

同步磁盘数据

开课吧

AOF重写原理（优化AOF文件）

如何选择RDB和AOF（4.0之前的还需要考虑）

混合持久化方式

① 通过命令行开启

② 通过修改 Redis 配置文件开启

Redis 主从复制

什么是主从复制

实现原理

全量同步

增量同步

主从配置

主Redis配置

从Redis配置

缓存通识

缓存类型

缓存的类型分为：本地缓存、分布式缓存和多级缓存。

本地缓存：

本地缓存就是在进程的内存中进行缓存，比如我们的 **JVM** 堆中，可以用 **LRUMap** 来实现，也可以使用 **Ehcache** 这样的工具来实现。

本地缓存是内存访问，没有远程交互开销，性能最好，但是受限于单机容量，一般缓存较小且无法扩展。

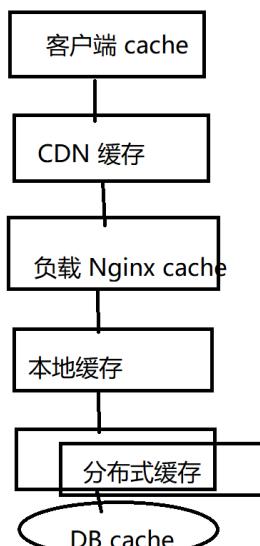
分布式缓存：

分布式缓存可以很好得解决这个问题。

分布式缓存一般都具有良好的水平扩展能力，对较大数据量的场景也能应付自如。缺点就是需要进行远程请求，性能不如本地缓存。

多级缓存：

为了平衡这种情况，实际业务中一般采用多级缓存，本地缓存只保存访问频率最高的部分热点数据，其他的热点数据放在分布式缓存中。



淘汰策略

不管是本地缓存还是分布式缓存，为了保证较高性能，都是使用内存来保存数据，由于成本和内存限制，当存储的数据超过缓存容量时，需要对缓存的数据进行剔除。

一般的剔除策略有 **FIFO** 淘汰最早数据、**LRU** 剔除最近最少使用、和 **LFU** 剔除最近使用频率最低的数据几种策略。

Memcache特点

- MC 处理请求时使用多线程异步 IO 的方式，可以合理利用 CPU 多核的优势，性能非常优秀；
- MC 功能简单，使用内存存储数据
- MC 对缓存的数据可以设置失效期，过期后的数据会被清除；
- 失效的策略采用延迟失效，就是当再次使用数据时检查是否失效；
- 当容量存满时，会对缓存中的数据进行剔除，剔除时，除了会对过期 key 进行清理，还会按 LRU 策略对数据进行剔除。

MC 有一些限制，这些限制在现在的互联网场景下很致命，成为大家选择**Redis**、**MongoDB**的重要原因：

- key 不能超过 250 个字节；
- value 不能超过 1M 字节；
- key 的最大失效时间是 30 天；
- 只支持 K-V 结构，不提供持久化和主从同步功能。

Reids 特点

- 与 MC 不同的是，Redis 采用单线程模式处理请求。这样做的原因有 2 个：一个是因为采用了非阻塞的异步事件处理机制；另一个是缓存数据都是内存操作 IO 时间不会太长，单线程可以避免线程上下文切换产生的代价。
- Redis 支持持久化，所以 Redis 不仅仅可以用作缓存，也可以用作 NoSQL 数据库。
- 相比 MC，Redis 还有一个非常大的优势，就是除了 K-V 之外，还支持多种数据格式，例如 list、set、sorted set、hash 等。
- Redis 提供主从同步机制，以及 Cluster 集群部署能力，能够提供高可用服务。

Redis6.0 引入了多线程

主流使用 2.8 3.2 4.0 5.0

1. 为什么 Redis 一开始选择单线程模型（单线程的好处）？

(1) IO多路复用

Redis顶层设计

REDIS IO MULTIPLEXING

FD

FD

FD

IO Multiplexing

Thread

FD是一个文件描述符，意思是表示当前文件处于可读、可写还是异常状态。使用 I/O 多路复用机制同时监听多个文件描述符的可读和可写状态。**你可以理解为具有了多线程的特点。**

一旦受到网络请求就会在内存中快速处理，由于绝大多数的操作都是纯内存的，所以处理的速度会非常地快。**也就是说在单线程模式下，即使连接的网络处理很多，因为有IO多路复用，依然可以在高速的内存处理中得到忽略。**

(2) 可维护性高

多线程模型虽然在某些方面表现优异，但是它却引入了程序执行顺序的不确定性，带来了并发读写的一系列问题。单线程模式下，可以方便地进行调试和测试。

(3) 基于内存，单线程状态下效率依然高

多线程能够充分利用CPU的资源，但对于Redis来说，由于基于内存速度那是相当的高，能达到在一秒内处理10万个用户请求，如果一秒十万还不能满足，那我们就可以使用Redis分片的技术来交给不同的Redis服务器。这样的做法避免了在同一个 Redis 服务中引入大量的多线程操作。

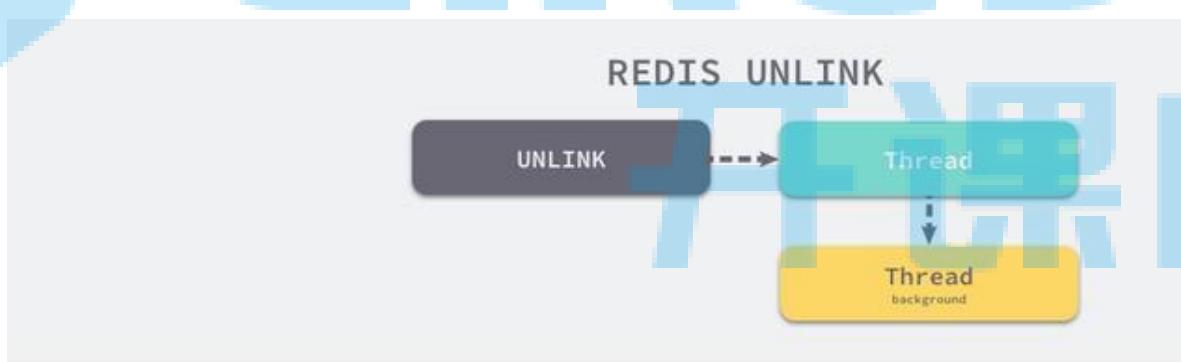
而且基于内存，除非是要进行AOF备份，否则基本上不会涉及任何的 I/O 操作。这些数据的读写由于只发生在内存中，所以处理速度是非常快的；用多线程模型处理全部的外部请求可能不是一个好的方案。

总结成两句话，基于内存而且使用多路复用技术，单线程速度很快，又保证了多线程的特点。因为没有必要使用多线程。

2. 为什么 Redis 在 6.0 之后加入了多线程（在某些情况下，单线程出现了缺点，多线程可以解决）

因为读写网络的read/write系统调用在Redis执行期间占用了大部分CPU时间，如果把网络读写做成多线程的方式对性能会有很大提升。

Redis可以使用del命令删除一个元素，如果这个元素非常大，可能占据了几十兆或者是几百兆，那么在短时间内是不能完成的，这样一来就需要多线程的异步支持。



使用多线程删除工作可以在后台

总结：Redis 选择使用单线程模型处理客户端的请求主要还是因为 CPU 不是 Redis 服务器的瓶颈，所以使用多线程模型带来的性能提升并不能抵消它带来的开发成本和维护成本，系统的性能瓶颈也主要在网络 I/O 操作上；而 Redis 引入多线程操作也是出于性能上的考虑，对于一些大键值对的删除操作，通过多线程非阻塞地释放内存空间也能减少对 Redis 主线程阻塞的时间，提高执行的效率

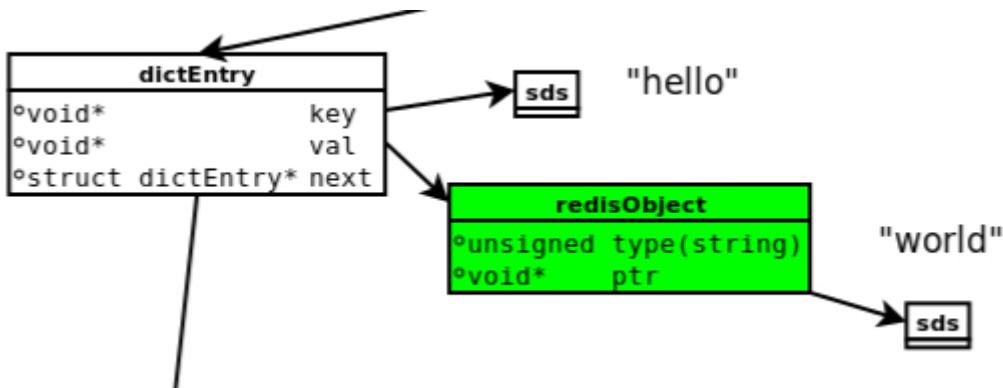
Redis 设计优化

1 估算Redis内存使用量

要估算redis中的数据占据的内存大小，需要对redis的内存模型有比较全面的了解，包括第一节课介绍的hashtable、sds、redisobject、各种对象类型的编码方式等。

下面以最简单的字符串类型来进行说明。

假设有90000个键值对，每个key的长度是10个字节，每个value的长度也是10个字节（且key和value都不是整数）；



下面来估算这90000个键值对所占用的空间。在估算占据空间之前，首先可以判定字符串类型使用的编码方式：embstr。

90000个键值对占据的内存空间主要可以分为两部分：[一部分是90000个dictEntry占据的空间](#)；[一部分是键值对所需要的bucket空间](#)。

每个dictEntry占据的空间包括：

- 1) 一个dictEntry结构，24字节，jemalloc会分配32字节的内存块([64位操作系统下，一个指针8字节，一个dictEntry由三个指针组成](#))
- 2) 一个key，10字节，所以SDS(key)需要10+6=16个字节 ([\[SDS的长度=6+字符串长度\]](#))，jemalloc会分配16字节的内存块
- 3) 一个redisObject，16字节，jemalloc会分配16字节的内存块 ([\(4bit+4bit+24bit+4Byte+8Byte=16Byte\)](#))
- 4) 一个value，10字节，所以SDS(value)需要10+4=16个字节 ([\[SDS的长度=4+字符串长度\]](#))，jemalloc会分配16字节的内存块
- 5) 综上，一个dictEntry所占据的空间需要[32+16+16+16=80](#)个字节。

bucket空间：

bucket数组的大小为大于90000的最小的 2^n ，是131072；每个bucket元素（[bucket中存储的都是指针元素](#)）为8字节（[因为64位系统中指针大小为8字节](#)）。

因此，可以估算出这90000个键值对占据的内存大小为： $[90000 * 80 + 131072 * 8 = 8248576]$

作为对比将key和value的长度由10字节增加到11字节，则对应的SDS变为17个字节，jemalloc会分配32个字节，因此每个dictEntry占用的字节数也由80字节变为112字节。此时估算这90000个键值对占据内存大小为：[90000*112 + 131072*8 = 11128576](#)。

2 优化内存占用

了解redis的内存模型，对优化redis内存占用有很大帮助。下面介绍几种优化场景和方式（4）缩短键值对的存储长度

(1) 利用jemalloc特性进行优化

上一小节所讲述的90000个键值便是一个例子。由于jemalloc分配内存时数值是不连续的，因此key/value字符串变化一个字节，可能会引起占用内存很大的变动；在设计时可以利用这一点。

例如，如果key的长度如果是11个字节，则SDS为17字节，jemalloc分配32字节；此时将key长度缩减为11个字节，则SDS为16字节，jemalloc分配16字节；则每个key所占用的空间都可以缩小一半。

(2) 使用整型/长整型

如果是整型/长整型，Redis会使用int类型（8字节）存储来代替字符串，可以节省更多空间。因此在可以使用长整型/整型代替字符串的场景下，尽量使用长整型/整型。

(3) 共享对象

利用共享对象，可以减少对象的创建（同时减少了redisObject的创建），节省内存空间。目前redis中的共享对象只包括10000个整数（0-9999）；可以通过调整REDIS_SHARED_INTEGERS参数提高共享对象的个数；

例如将REDIS_SHARED_INTEGERS调整到20000，则0-19999之间的对象都可以共享

论坛网站在redis中存储了每个帖子的浏览数，而这些浏览数绝大多数分布在0-20000之间，这时候通过适当增大REDIS_SHARED_INTEGERS参数，便可以利用共享对象节省内存空间。

(4) 缩短键值对的存储长度

键值对的长度是和性能成反比的，比如我们来做一组写入数据的性能测试，执行结果如下：

数据量	key 大小	value 大小	string:set 平均耗时	hash:hset 平均耗时
100w	20byte	512byte	1.13微秒	10.28微秒
100w	20byte	200byte	0.74微秒	8.08微秒
100w	20byte	100byte	0.65微秒	7.92微秒
100w	20byte	50byte	0.59微秒	6.74微秒
100w	20byte	20byte	0.55微秒	6.60微秒
100w	20byte	5byte	0.53微秒	6.53微秒

从以上数据可以看出，在key不变的情况下，value值越大操作效率越慢，因为Redis对于同一种数据类型会使用不同的内部编码进行存储，比如字符串的内部编码就有三种：int（整数编码）、raw（优化内存分配的字符串编码）、embstr（动态字符串编码），这是因为Redis的作者是想通过不同编码实现效率和空间的平衡，然而数据量越大使用的内部编码就越复杂，而越是复杂的内部编码存储的性能就越低。

这还只是写入时的速度，当键值对内容较大时，还会带来另外几个问题：

- 内容越大需要的持久化时间就越长，需要挂起的时间越长，Redis的性能就会越低；
- 内容越大在网络上传输的内容就越多，需要的时间就越长，整体的运行速度就越低；
- 内容越大占用的内存就越多，就会更频繁的触发内存淘汰机制，从而给Redis带来了更多的运行负担。

因此在保证完整语义的同时，我们要尽量的缩短键值对的存储长度，必要时要对数据进行序列化和压缩再存储，以Java为例，序列化我们可以使用protostuff或kryo，压缩我们可以使用snappy。

基于Redis内存淘汰机制的优化

设置键值的过期时间

我们应该根据实际的业务情况，对键值设置合理的过期时间，这样 Redis 会帮你自动清除过期的键值对，以节约对内存的占用，以避免键值过多的堆积，频繁的触发内存淘汰策略。

Redis 有四个不同的命令可以用于设置键的生存时间(键可以存在多久)或过期时间(键什么时候会被删除)

- EXPIRE 命令用于将键key 的生存时间为ttl 秒。
- PEXPIRE 命令用于将键key 的生存时间为ttl 毫秒。
- EXPIREAT < timestamp> 命令用于将键key 的过期时间设置为timestamp所指定的秒数时间戳。
- PEXPIREAT < timestamp > 命令用于将键key 的过期时间设置为timestamp所指定的毫秒数时间戳。

```
1 127.0.0.1:6379> set key1 'value1'  
2 OK  
3 127.0.0.1:6379> expire key1 20  
4 (integer) 1  
5 127.0.0.1:6379> get key1 "value1" 127.0.0.1:6379> get key1 "value1"  
127.0.0.1:6379> get key1 (nil) 127.0.0.1:6379>
```

使用 lazy free 特性

lazy free 特性是 Redis 4.0 新增的一个非常使用功能，它可以理解为惰性删除或延迟删除。意思是在删除的时候提供异步延时释放键值的功能，把键值释放操作放在 BIO(Background I/O) 单独的子线程处理中，以减少删除删除对 Redis 主线程的阻塞，可以有效地避免删除 big key 时带来的性能和可用性问题。

lazy free 对应了 4 种场景，默认都是关闭的：

```
1 lazyfree-lazy-eviction no  
2 lazyfree-lazy-expire no  
3 lazyfree-lazy-server-del no  
4 slave-lazy-flush no
```

它们代表的含义如下：

- lazyfree-lazy-eviction：表示当 Redis 运行内存超过 时，是否开启 lazy free 机制删除；
- lazyfree-lazy-expire：表示设置了过期时间的键值，当过期之后是否开启 lazy free 机制删除；
- lazyfree-lazy-server-del：有些指令在处理已存在的键时，会带有一个隐式的 del 键的操作，比如 rename 命令，当目标键已存在，Redis 会先删除目标键，如果这些目标键是一个 big key，就会造成阻塞删除的问题，此配置表示在这种场景中是否开启 lazy free 机制删除；
- slave-lazy-flush：针对 slave(从节点) 进行全量数据同步，slave 在加载 master 的 RDB 文件前，会运行 flushall 来清理自己的数据，它表示此时是否开启 lazy free 机制删除。

建议开启其中的 lazyfree-lazy-eviction、lazyfree-lazy-expire、lazyfree-lazy-server-del 等配置，这样就可以有效的提高主线程的执行效率。

限制 Redis 内存大小 (重点)

最大缓存

maxmemory 1048576

maxmemory 1048576B

maxmemory 1000KB

maxmemory 100MB

maxmemory 1GB

maxmemory 1000K

maxmemory 100M

maxmemory 1G

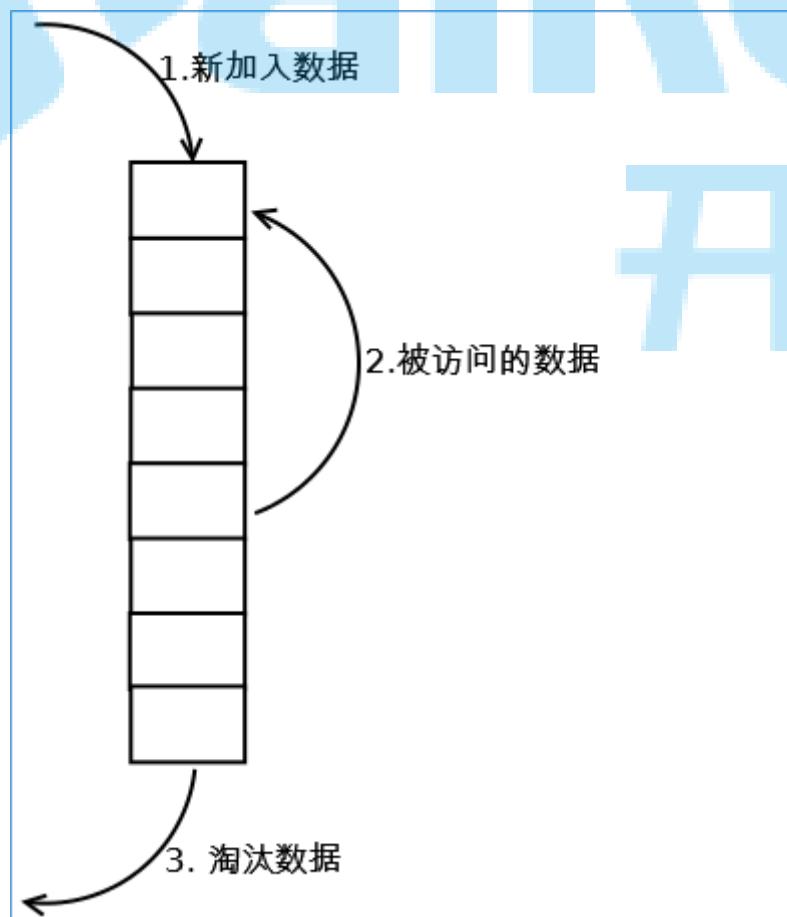
有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一定要设置。

- 在 64 位操作系统中 Redis 的内存大小是没有限制的，也就是配置项 `maxmemory` 是被注释掉的，这样就会导致在物理内存不足时，使用 swap 空间既交换空间，而当操作系统将 Redis 所用的内存分页移至 swap 空间时，将会阻塞 Redis 进程，导致 Redis 出现延迟，从而影响 Redis 的整体性能。因此我们需要限制 Redis 的内存大小为一个固定的值，当 Redis 的运行到达此值时会触发内存淘汰策略，**内存淘汰策略在 Redis 4.0 之后有 8 种**

LRU原理

LRU (Least recently used, 最近最少使用) 算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

最常见的实现是使用一个链表保存缓存数据，详细算法实现如下：

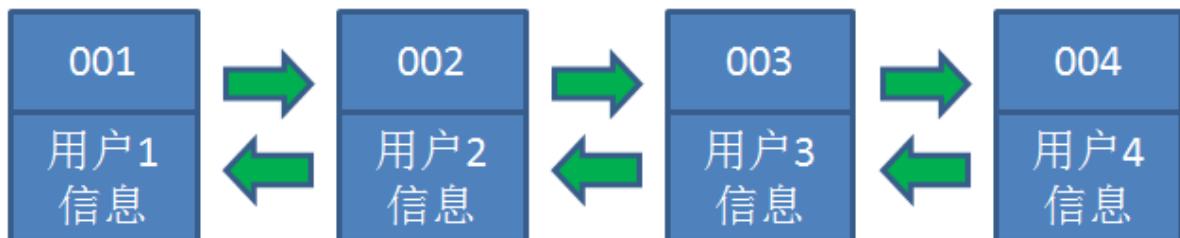


1. 新数据插入到链表头部；
2. 每当缓存命中（即缓存数据被访问），则将数据移到链表头部；
3. 当链表满的时候，将链表尾部的数据丢弃。

案例分析

让我们以用户信息的需求为例，来演示一下LRU算法的基本思路：

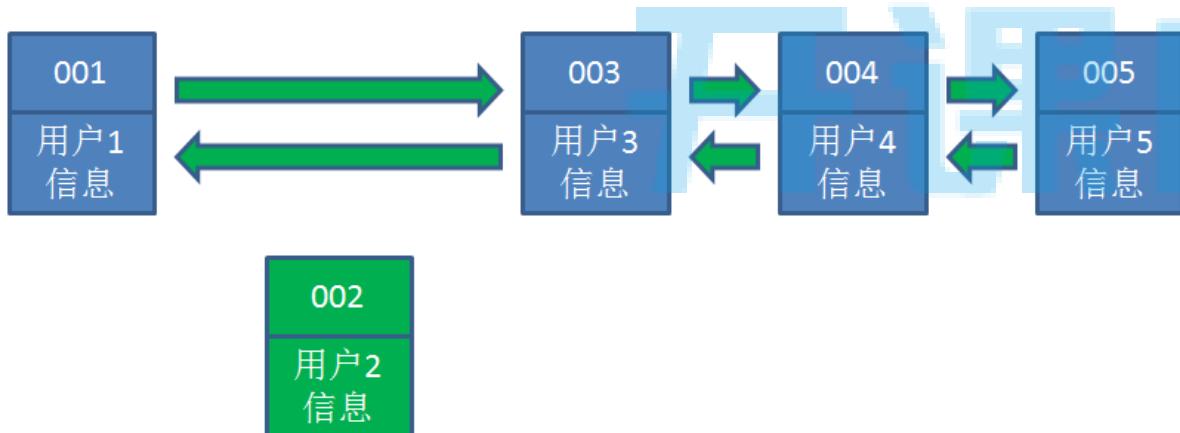
- 1.假设我们使用哈希链表来缓存用户信息，目前缓存了4个用户，这4个用户是按照时间顺序依次从链表右端插入的。



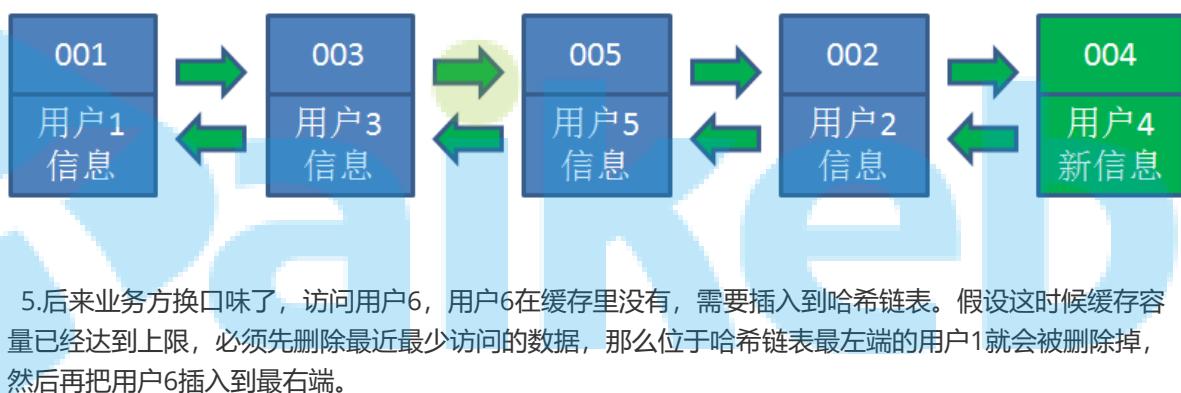
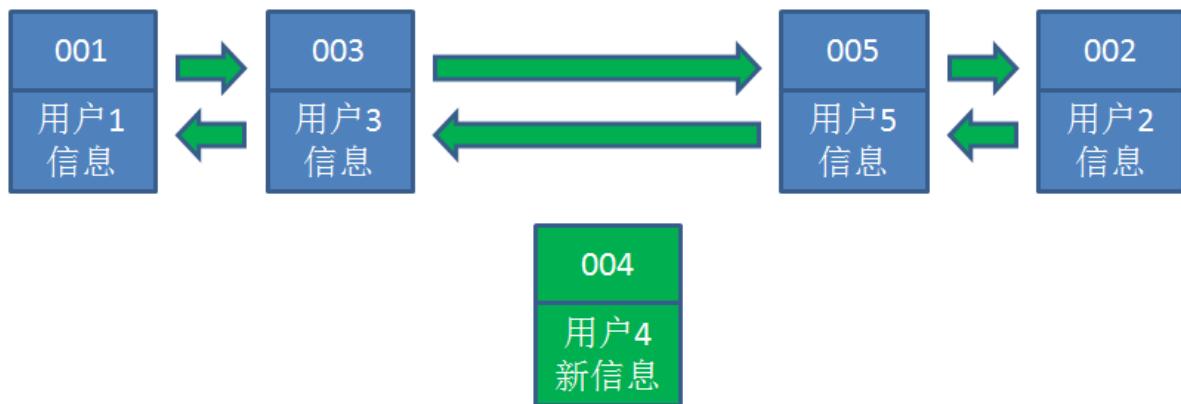
- 2.此时，业务方访问用户5，由于哈希链表中没有用户5的数据，我们从数据库中读取出来，插入到缓存当中。这时候，链表中最右端是最新访问到的用户5，最左端是最近最少访问的用户1。



- 3.接下来，业务方访问用户2，哈希链表中存在用户2的数据，我们怎么做呢？我们把用户2从它的前驱节点和后继节点之间移除，重新插入到链表最右端。这时候，链表中最右端变成了最新访问到的用户2，最左端仍然是最近最少访问的用户1。



4.接下来，业务方请求修改用户4的信息。同样道理，我们把用户4从原来的位置移动到链表最右侧，并把用户信息的值更新。这时候，链表中最右端是最新访问到的用户4，最左端仍然是最近最少访问的用户1。



以上，就是LRU算法的基本思路。

<https://www.itcodemonkey.com/article/11153.html>

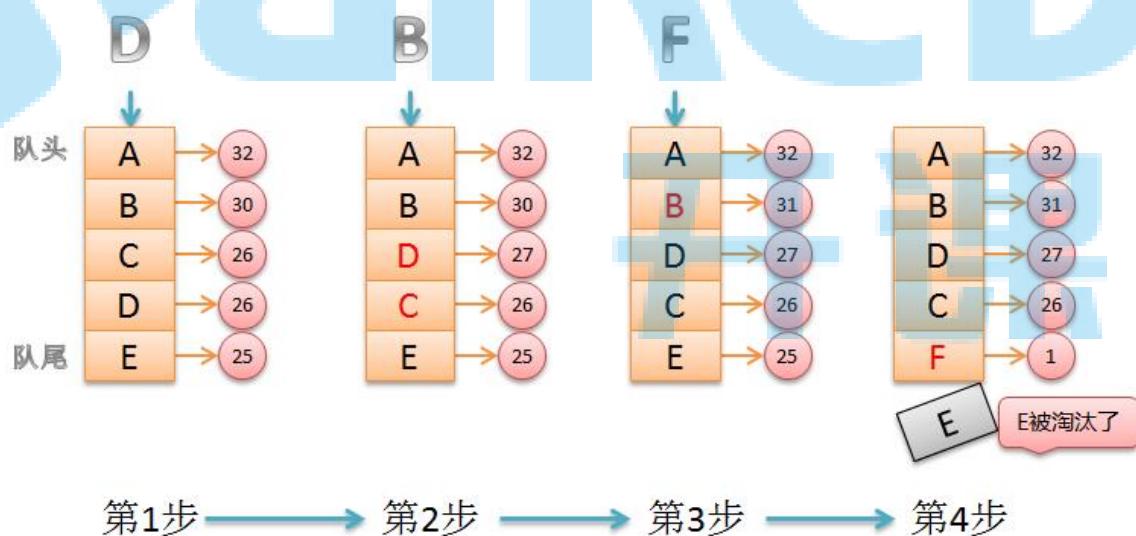
LFU原理

LFU,全称是:Least Frequently Used, 最不经常使用策略,在一段时间内,数据被使用频次最少的,优先被淘汰。*最少使用* (*LFU*) 是一种用于管理计算机内存的缓存算法。主要是记录和追踪内存块的使用次数,当缓存已满并且需要更多空间时,系统将以最低内存块使用频率清除内存.采用LFU算法的最简单方法是为每个加载到缓存的块分配一个计数器。每次引用该块时,计数器将增加一。当缓存达到容量并有一个新的内存块等待插入时,系统将搜索计数器最低的块并将其从缓存中删除(本段摘自维基百科)

LRU缓存淘汰过程



LFU淘汰过程



LRU和LFU侧重点不同, LRU主要体现在对元素的使用时间上,而LFU主要体现在对元素的使用频次上。LFU的缺陷是:在短期的时间内,对某些缓存的访问频次很高,这些缓存会立刻晋升为热点数据,而保证不会淘汰,这样会驻留在系统内存里面。而实际上,这部分数据只是短暂的高频率访问,之后将会长期不访问,瞬时的高频访问将会造成这部分数据的引用频率加快,而一些新加入的缓存很容易被快速删除,因为它们的引用频率很低。

Redis缓存淘汰策略

redis 内存数据集大小上升到一定大小的时候,就会实行数据淘汰策略。

maxmemory-policy volatile-lru, 支持热配置 内存淘汰策略在 Redis 4.0 之后有 8 种*:

1. **noeviction**: 不淘汰任何数据，当内存不足时，新增操作会报错，Redis 默认内存淘汰策略；
2. **allkeys-lru**: 淘汰整个键值中最久未使用的键值；
3. **allkeys-random**: 随机淘汰任意键值；
4. **volatile-lru**: 淘汰所有设置了过期时间的键值中最久未使用的键值；
5. **volatile-random**: 随机淘汰设置了过期时间的任意键值；
6. **volatile-ttl**: 优先淘汰更早过期的键值。

在 Redis 4.0 版本中又新增了 2 种淘汰策略：

1. **volatile-lfu**: 淘汰所有设置了过期时间的键值中，最少使用的键值；
2. **allkeys-lfu**: 淘汰整个键值中最少使用的键值。

其中 allkeys-xxx 表示从所有的键值中淘汰数据，而 volatile-xxx 表示从设置了过期键的键值中淘汰数据。

我们可以根据实际的业务情况进行设置，默认的淘汰策略不淘汰任何数据，在新增时会报错。

Redis 持久化

Redis 是一个内存数据库，为了保证数据的持久性，它提供了三种持久化方案：

- RDB 方式（默认）
- AOF 方式
- 混合持久化模式（4.0增加，5.0默认开启）

RDB

RDB 是 Redis 默认采用的持久化方式。

RDB 方式是通过快照（snapshotting）完成的，当符合一定条件时 Redis 会自动将内存中的数据进行快照并持久化到硬盘。

触发RDB快照的时机

1. 符合指定配置的快照规则
2. 执行 save 或 bgsave 命令
3. 执行 flushall
4. 执行主从复制操作

设置快照规则

save 多少秒内 数据变了多少

save "" : 不使用RDB存储

save 900 1 : 表示15分钟（900秒钟）内至少1个键被更改则进行快照。

save 300 10 : 表示5分钟（300秒）内至少10个键被更改则进行快照。

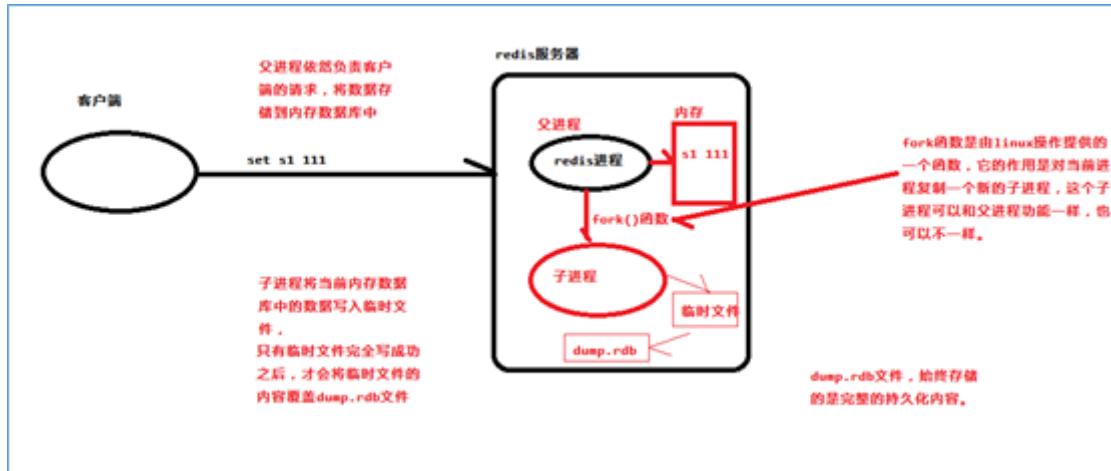
save 60 10000 : 表示1分钟内至少10000个键被更改则进行快照。

漏斗型

RDB快照的实现原理

快照过程

1. Redis 调用系统中的 fork 函数复制一份当前进程的副本(子进程)
2. 父进程继续接收并处理客户端发来的命令，而子进程开始将内存中的数据写入硬盘中的临时文件。
3. 当子进程写入完所有数据后会用该临时文件替换旧的 RDB 文件，至此，一次快照操作完成。



注意事项

1. Redis 在进行快照的过程中不会修改 RDB 文件，只有快照结束后才会将旧的文件替换成新的，也就是说任何时候 RDB 文件都是完整的。
2. 这就使得我们可以通过定时备份 RDB 文件来实现 Redis 数据库的备份，RDB 文件是经过压缩的二进制文件，占用的空间会小于内存中的数据，更加利于传输。

RDB优缺点

- **缺点：**使用 RDB 方式实现持久化，一旦 Redis 异常退出，就会丢失最后一次快照以后更改的所有数据。这个时候我们就需要根据具体的应用场景，通过组合设置自动快照条件的方式来将可能发生的数据损失控制在能够接受范围。如果数据相对来说比较重要，希望将损失降到最小，则可以使用 AOF 方式进行持久化
- **优点：** 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，父进程无需执行任何磁盘 I/O 操作。同时这个也是一个缺点，如果数据集比较大的时候， fork 可以比较耗时，造成服务器在一段时间内停止处理客户端的请求；

AOF

AOF介绍

默认情况下 Redis 没有开启 AOF (append only file) 方式的持久化。

开启 AOF 持久化后，每执行一条会更改 Redis 中的数据的命令，Redis 就会将该命令写入硬盘中的 AOF 文件，这一过程显然会降低 Redis 的性能，但大部分情况下这个影响是能够接受的，另外使用较快的硬盘可以提高 AOF 的性能。

redis.conf :

```
1 # 可以通过修改redis.conf配置文件中的appendonly参数开启  
2 appendonly yes  
3  
4 # AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的。  
5 dir ./  
6  
7 # 默认的文件名是appendonly.aof，可以通过appendfilename参数修改  
8 appendfilename appendonly.aof
```

同步磁盘数据

Redis 每次更改数据的时候，aof 机制都会将命令记录到 aof 文件，但是实际上由于操作系统的**缓存机制**，数据并没有实时写入到硬盘，而是进入**硬盘缓存**。再通过**硬盘缓存机制**去刷新到保存到文件。

参数说明：

```
1 # 每次执行写入都会进行同步，这个是最安全但是效率比较低的方式  
2 appendfsync always  
3  
4 # 每一秒执行(默认)  
5 appendfsync everysec  
6  
7 # 不主动进行同步操作，由操作系统去执行，这个是最快但是最不安全的方式  
8 appendfsync no
```

AOF重写原理（优化AOF文件）

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。

set s1 11

set s1 22

set s1 33

没有优化

set s1 11

set s1 22

set s1 33

优化后

set s1 33

lpush list1 1 2 3

lpush list1 4 5 6

优化后 lpush 1 2 3 4 5 6

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 (RESP) 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。

如何选择RDB和AOF (4.0之前的还需要考虑)

内存数据库 rdb (redis database) +aof 数据不能丢

缓存服务器 rdb

不建议 只使用 aof (性能差)

恢复时：先aof再rdb

混合持久化方式

- Redis 4.0 之后新增的方式，混合持久化是结合了 RDB 和 AOF 的优点，在写入的时候，先把当前的数据以 RDB 的形式写入文件的开头，再将后续的操作命令以 AOF 的格式存入文件，这样既能保证 Redis 重启时的速度，又能减低数据丢失的风险。

RDB 和 AOF 持久化各有利弊，RDB 可能会导致一定时间内的数据丢失，而 AOF 由于文件较大则会影响 Redis 的启动速度，为了能同时拥有 RDB 和 AOF 的优点，Redis 4.0 之后新增了混合持久化的方式，因此我们在必须要进行持久化操作时，应该选择混合持久化的方式。

查询是否开启混合持久化可以使用 config get aof-use-rdb-preamble 命令，执行结果

```
1 127.0.0.1:6379> config get aof-use-rdb-preamble
2 1) "aof-use-rdb-preamble"
3 2) "yes"
```

其中 yes 表示已经开启了混合持久化，no 表示关闭，Redis 5.0 默认值为 yes。如果是其他版本的 Redis 首先需要检查一下，是否已经开启了混合持久化，如果关闭的情况下，可以通过以下两种方式开启：

- 通过命令行开启
- 通过修改 Redis 配置文件开启

① 通过命令行开启

使用命令 config set aof-use-rdb-preamble yes

命令行设置配置的缺点是重启 Redis 服务之后，设置的配置就会失效。

② 通过修改 Redis 配置文件开启

在 Redis 的根路径下找到 redis.conf 文件，把配置文件中的 aof-use-rdb-preamble no 改为 aof-use-rdb-preamble yes

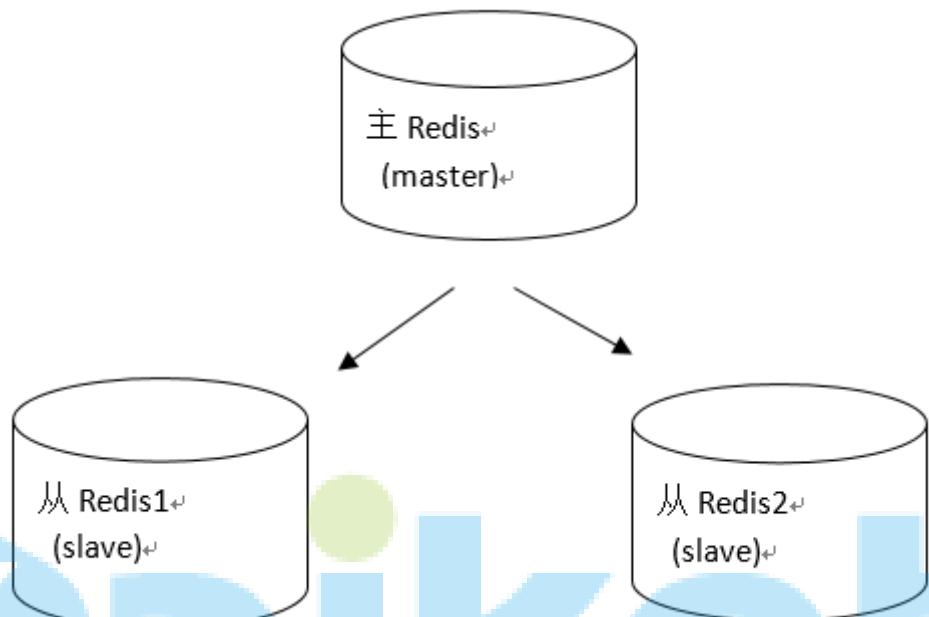
配置完成之后，需要重启 Redis 服务，配置才能生效，但修改配置文件的方式，在每次重启 Redis 服务之后，配置信息不会丢失。

需要注意的是，在非必须进行持久化的业务中，可以关闭持久化，这样可以有效的提升 Redis 的运行速度，不会出现间歇性卡顿的困扰。

Redis 主从复制

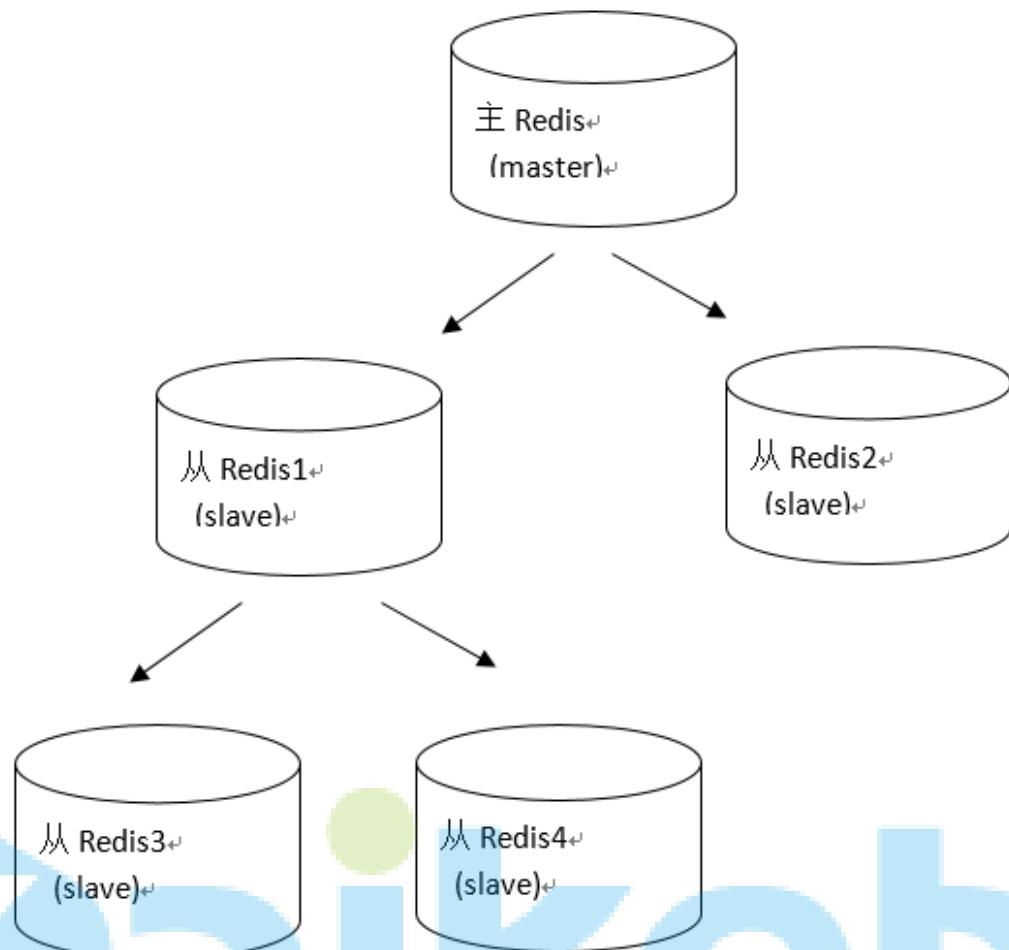
什么是主从复制

持久化保证了即使 Redis 服务重启也不会丢失数据，因为 Redis 服务重启后会将硬盘上持久化的数据恢复到内存中，但是当 Redis 服务器的硬盘损坏了可能会导致数据丢失，不过通过 Redis 的主从复制机制就可以避免这种单点故障，如下图：



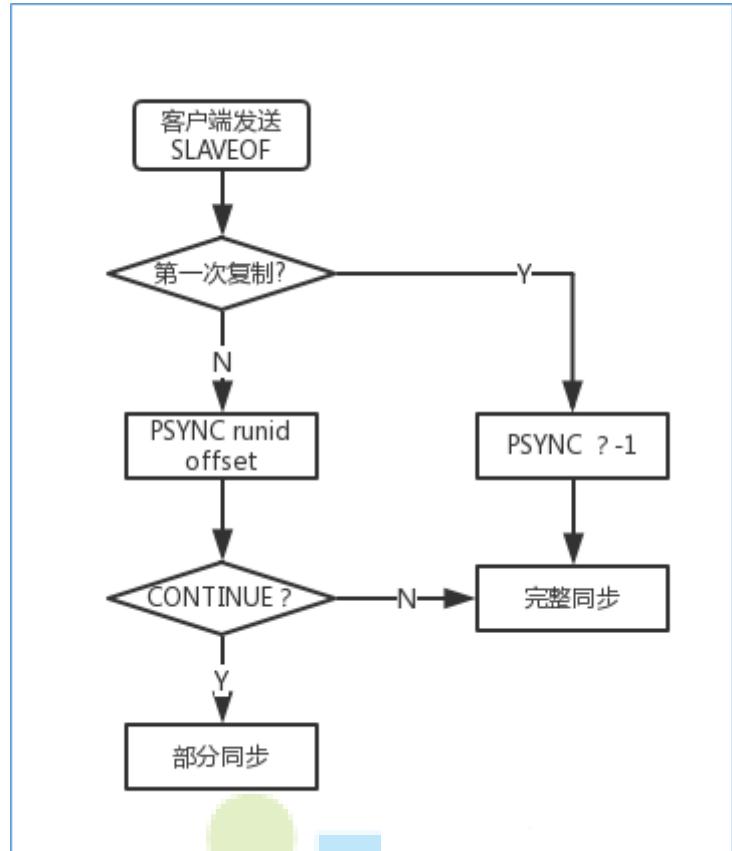
说明：

- 主 Redis 中的数据有两个副本（replication）即从 redis1 和从 redis2，即使一台 Redis 服务器宕机其它两台 Redis 服务也可以继续提供服务。
- 主 Redis 中的数据和从 Redis 上的数据保持实时同步，当主 Redis 写入数据时通过主从复制机制会复制到两个从 Redis 服务上。
- 只有一个主 Redis，可以有多个从 Redis。
- 主从复制不会阻塞 master，在同步数据时，master 可以继续处理 client 请求。
- 一个 Redis 可以即是主又是从，如下图：



实现原理

- Redis 的主从同步，分为全量同步和增量同步。
- 只有从机第一次连接上主机是全量同步。
- 断线重连有可能触发全量同步也有可能是增量同步（master 判断 runid 是否一致）。
- 除此之外的情况都是增量同步。

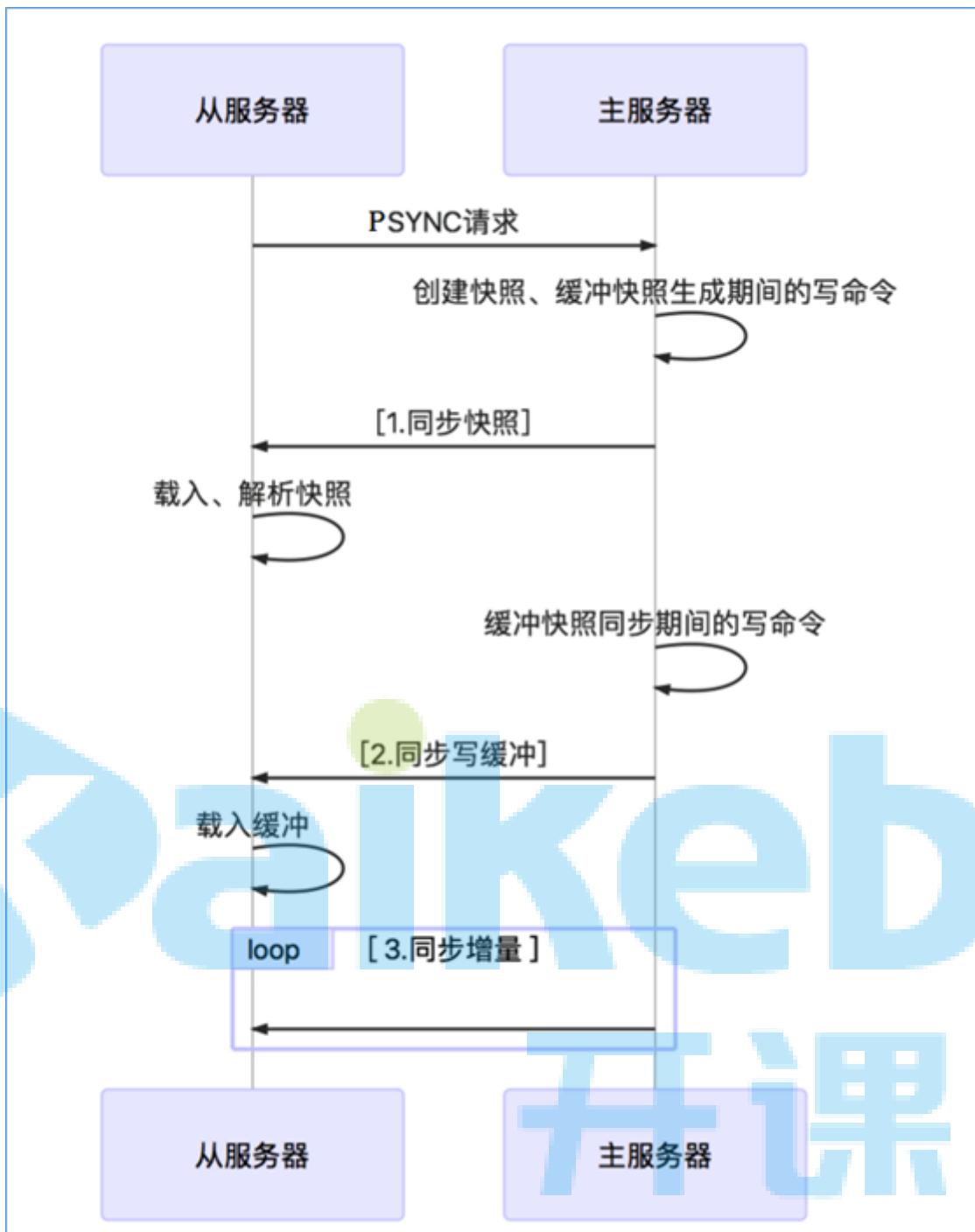


全量同步

Redis 的全量同步过程主要分三个阶段：

- 同步快照阶段：Master 创建并发送快照给 slave，slave 载入并解析快照。Master 同时将此阶段所产生的新的写命令存储到缓冲区。
- 同步写缓冲阶段：Master 向 slave 同步存储在缓冲区的写操作命令。
- 同步增量阶段：Master 向 slave 同步写操作命令。

开课吧



增量同步

- Redis 增量同步主要指 slave 完成初始化后开始正常工作时，master 发生的写操作同步到 slave 的过程。
- 通常情况下，master 每执行一个写命令就会向 slave 发送相同的写命令，然后 slave 接收并执行。

主从配置

主Redis配置

无需特殊配置

从Redis配置

修改从服务器上的 redis.conf 文件:

```
1 # replicaof <masterip> <masterport>
2 # 表示当前【从服务器】对应的【主服务器】的IP是192.168.10.135，端口是6379。
3 #4.0之前只能slaveof 4.0之后默认replicaof, slaveof模式作用
4 slaveof 192.168.133.154 6379
5 replicaof 192.168.133.154 6379
```



课程主题

Redis集群原理与Redis高级用法

课程目标

1. 理解Redis Sentinel 工作原理
2. 掌握Redis哨兵作用
3. 掌握Redis集群框架并清楚Redis集群框架的选择
4. 掌握Redis事务原理及用法（弱事务）
5. 理解lua概念，能够使用Redis和lua整合使用
6. 理解掌握Redis分布式锁，并清楚其优缺点

知识要点

课程主题

课程目标

知识要点

Redis 哨兵机制

Redis Sentinel 工作原理分析

- (1) 为什么要用到哨兵
- (2) 哨兵机制（sentinel）的高可用
- (3) 哨兵的定时监控
- (4) 哨兵leader选举流程
- (5) 自动故障转移机制

哨兵进程的作用

Redis 集群演变

主从复制

Replication+Sentinel * 高可用

工作原理

缺陷

常用方案：

内网DNS

VIP

封装客户端直连 Redis Sentinel 端口

Proxy+Replication+Sentinel (仅仅了解)

工作原理

缺陷：

Redis Cluster * 扩展性

Redis-cluster架构图

Redis-cluster投票:容错

RedisCluster 集群安装

安装RedisCluster

命令客户端连接集群

查看集群的命令

Jedis连接集群

代码实现

使用spring

总结

工作原理

优点：

开课吧

缺点:

Redis 事务

- 事务演示
- 事务失败处理
- Redis乐观锁
- Redis乐观锁实现秒杀

Redis 和lua 整合

- 什么是lua
- Redis中使用lua的好处
- lua的安装和语法
 - Redis整合lua脚本
 - EVAL命令
- lua 脚本调用Redis 命令
 - `redis.call();`
 - `redis.pcall();`
 - `redis-cli --eval`
- Redis+lua 秒杀

Redis分布式锁

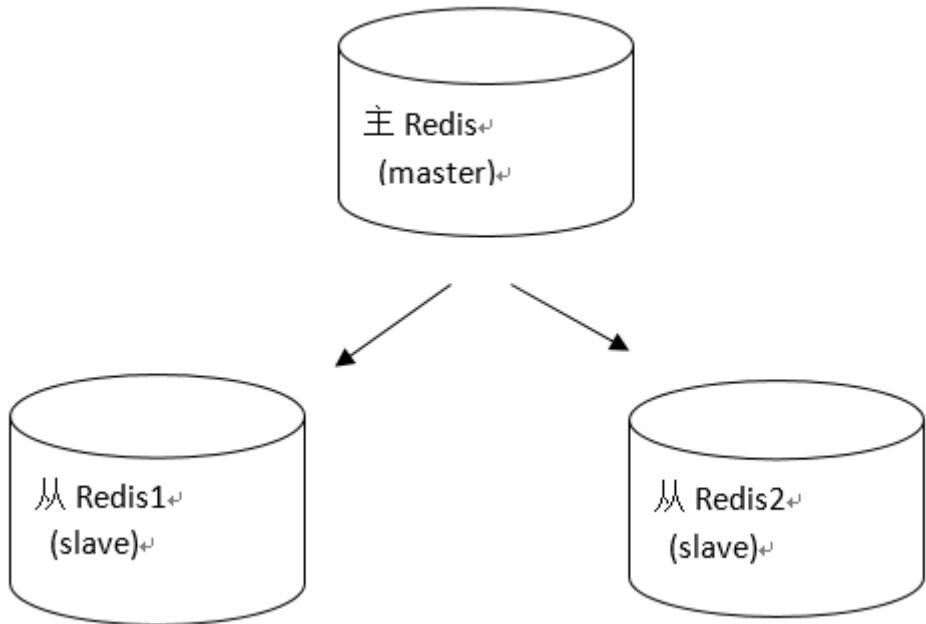
- 业务场景
- 锁的处理
- 分布式锁
 - 流程图
 - 分布式锁的状态
 - 分布式锁特点
 - 分布式锁的实现方式
- Redis方式实现分布式锁
 - 获取锁
 - 释放锁
- Redis分布式锁--优缺点
- 本质分析
 - 本质分析

Redis 哨兵机制

Redis Sentinel 工作原理分析

(1) 为什么要用到哨兵

开课吧



哨兵(Sentinel)主要是为了解决在主从复制架构中出现宕机的情况,主要分为两种情况:

1.从Redis宕机

这个相对而言比较简单,在Redis中从库重新启动后会自动加入到主从架构中,自动完成同步数据。在Redis2.8版本后,主从断线后恢复的情况下实现增量复制。

2.主Redis宕机

这个相对而言就会复杂一些,需要以下2步才能完成

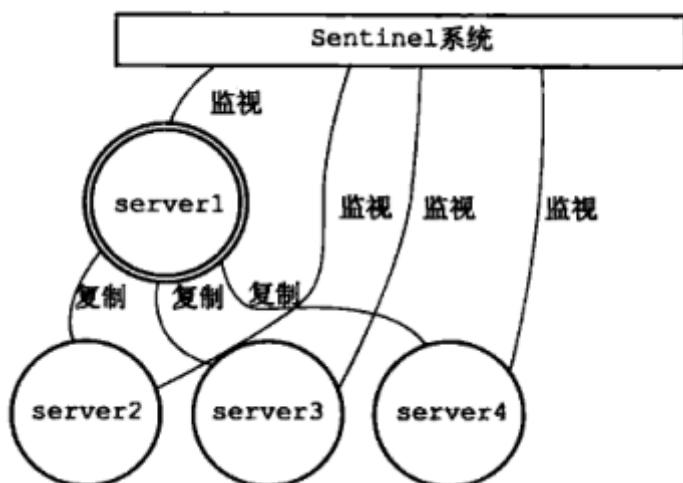
- 在从数据库中执行SLAVEOF NO ONE命令,断开主从关系并且提升为主库继续服务
- 第二步,将主库重新启动后,执行SLAVEOF命令,将其设置为其他库的从库,这时数据就能更新回来

由于这个手动完成恢复的过程其实是比较麻烦的并且容易出错,所以Redis提供的哨兵(sentinel)的功能来解决

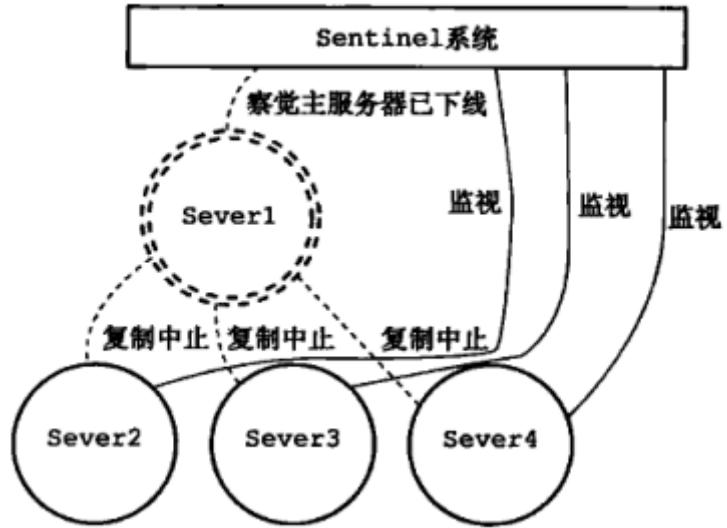
(2) 哨兵机制 (sentinel) 的高可用

Sentinel (哨兵) 是Redis 的高可用性解决方案:由一个或多个Sentinel 实例 组成的Sentinel 系统可以监视任意多个主服务器,以及这些主服务器属下的所有从服务器,并在被监视的主服务器进入下线状态时,自动将下线主服务器属下的某个从服务器升级为新的主服务器。

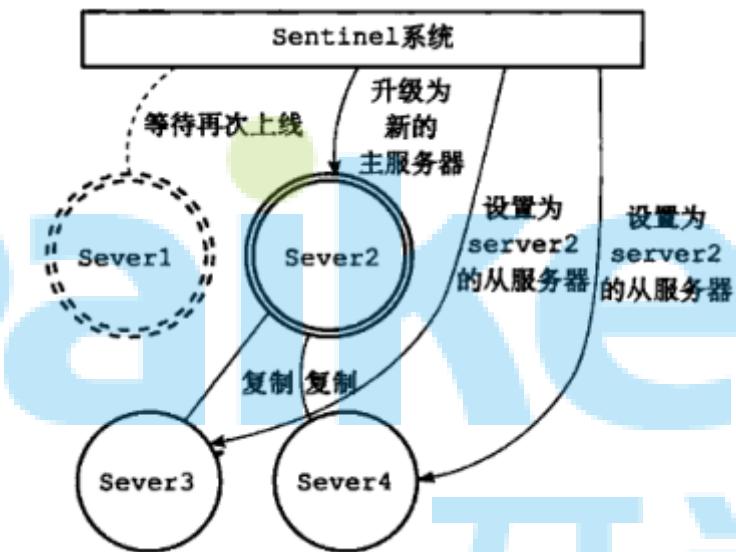
如图所示



在Server1 掉线后:

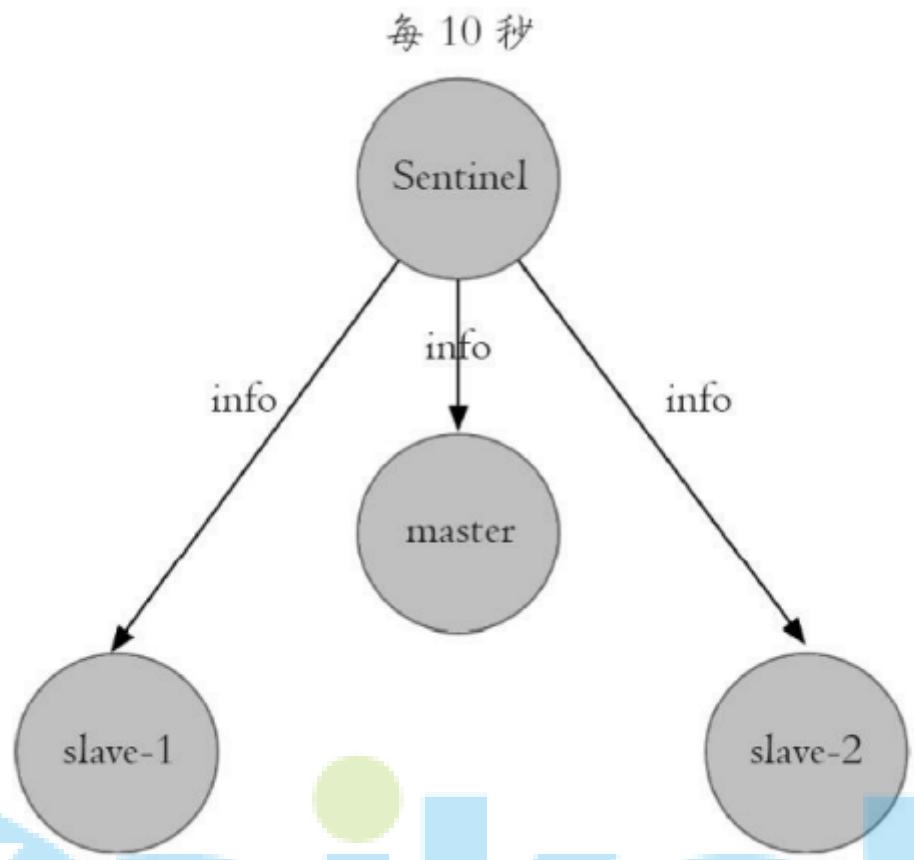


升级Server2 为新的主服务器：

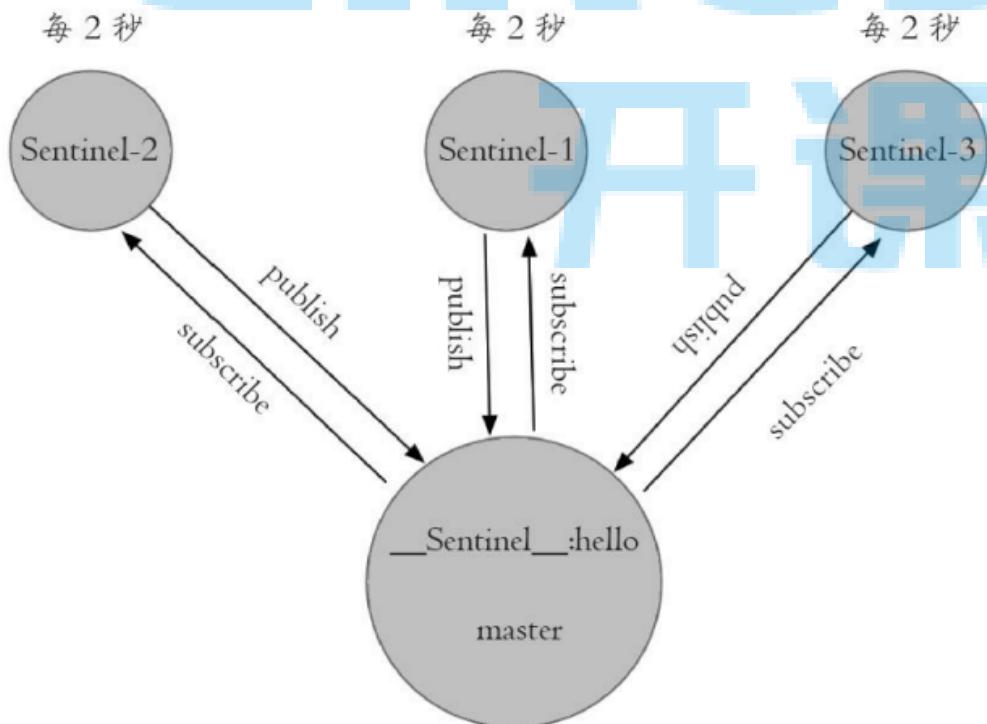


(3) 哨兵的定时监控

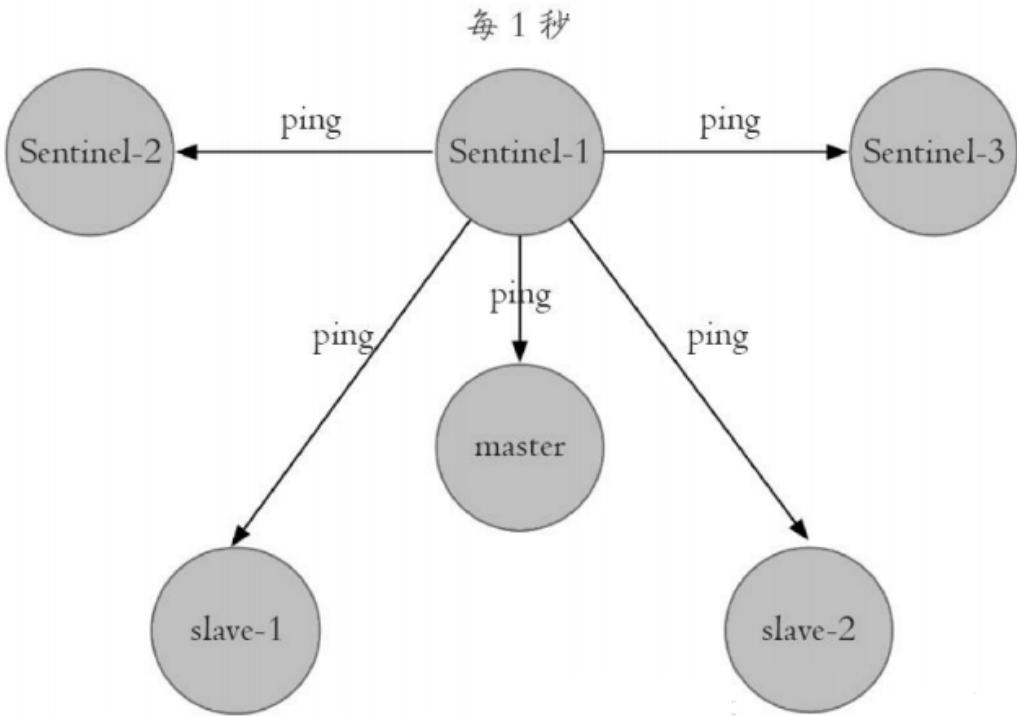
任务1：每个哨兵节点每10秒会向主节点和从节点发送info命令获取最拓扑结构图，哨兵配置时只要配置对主节点的监控即可，通过向主节点发送info，获取从节点的信息，并当有新的从节点加入时可以马上感知到



任务2：每个哨兵节点每隔2秒会向redis数据节点的指定频道上发送该哨兵节点对于主节点的判断以及当前哨兵节点的信息，同时每个哨兵节点也会订阅该频道，来了解其它哨兵节点的信息及对主节点的判断，其实就是通过消息publish和subscribe来完成的



任务3：每隔1秒每个哨兵会向主节点、从节点及其余哨兵节点发送一次ping命令做一次心跳检测，这个也是哨兵用来判断节点是否正常的重要依据

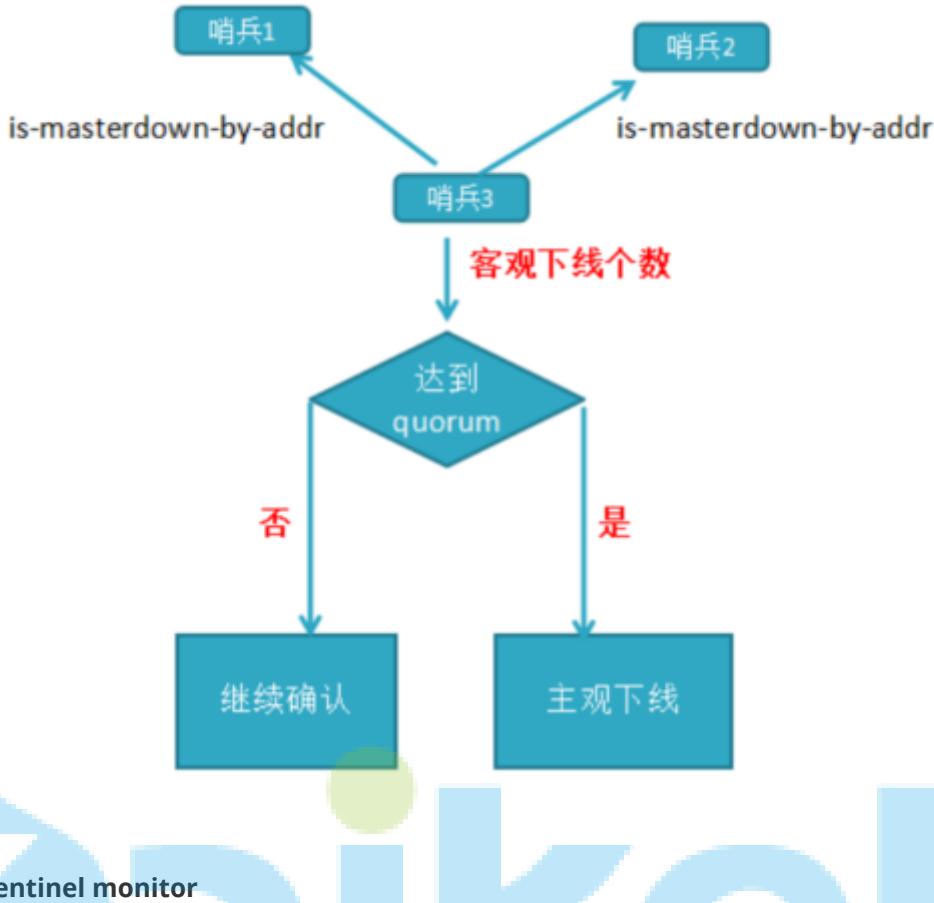


主观下线: 所谓主观下线，就是单个sentinel认为某个服务下线（有可能是接收不到订阅，之间的网络不通等等原因）。SDOWN

sentinel会以每秒一次的频率向所有与其建立了命令连接的实例（master，从服务，其他sentinel）发ping命令，通过判断ping回复是有效回复，还是无效回复来判断实例时候在线（对该sentinel来说是“主观在线”）。

sentinel配置文件中的down-after-milliseconds设置了判断主观下线的时间长度，如果实例在down-after-milliseconds毫秒内，返回的都是无效回复，那么sentinel回认为该实例已（主观）下线，修改其flags状态为SRI_S_DOWN。如果多个sentinel监视一个服务，有可能存在多个sentinel的down-after-milliseconds配置不同，这个在实际生产中要注意。

客观下线: 当主观下线的节点是主节点时，此时该哨兵3节点会通过指令sentinel is-masterdown-by-addr寻求其它哨兵节点对主节点的判断，如果其他的哨兵也认为主节点主观线下了，则当认为客观下线的票数超过了quorum（选举）个数，此时哨兵节点则认为该主节点确实有问题，这样就客观下线了，大部分哨兵节点都同意下线操作，也就是说是客观下线 ODOWN

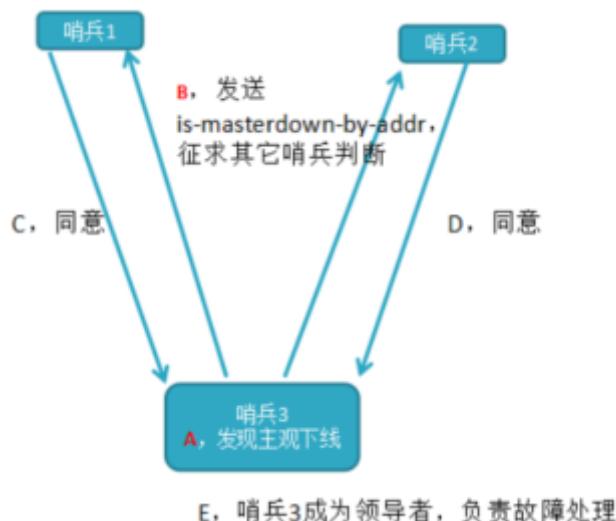


重点: sentinel monitor

(4) 哨兵leader选举流程

如果主节点被判定为客观下线之后，就要选取一个哨兵节点来完成后面的故障转移工作，选举出一个leader的流程如下：

- 每个在线的哨兵节点都可以成为领导者，当它确认（比如哨兵3）主节点下线时，会向其它哨兵发`is-master-down-by-addr`命令，征求判断并要求将自己设置为领导者，由领导者处理故障转移；
- 当其它哨兵收到此命令时，可以同意或者拒绝它成为领导者；
- 如果哨兵3发现自己在选举的票数num 大于等于 $(\text{sentinels})/2+1$ 时，将成为领导者，如果没有超过，继续选举.....

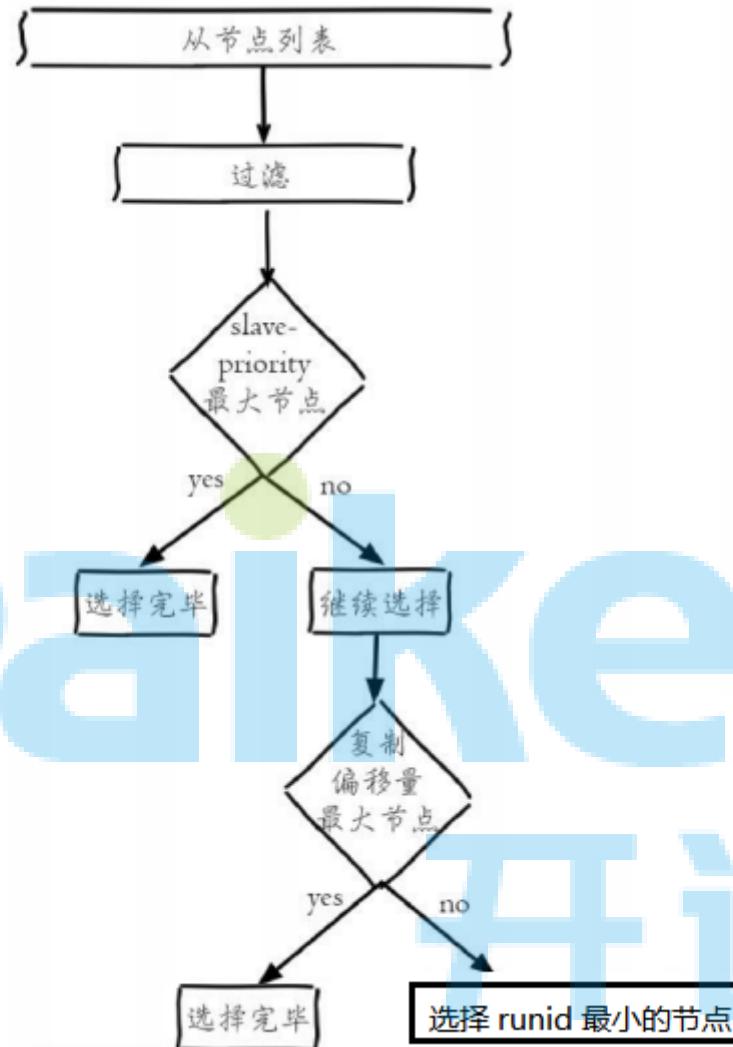


(5) 自动故障转移机制

在从节点中选择新的主节点

sentinel状态数据结构中保存了主服务的所有从服务信息，领头sentinel按照如下的规则从从服务列表中挑选出新的主服务

1. 过滤掉主观下线的节点
2. 选择slave-priority最高的节点，如果有则返回没有就继续选择
3. 选择出复制偏移量最大的从节点，因为复制偏移量越大则数据复制的越完整，如果没有就继续
4. 选择run_id最小的节点



更新主从状态

通过slaveof no one命令，让选出来的从节点成为主节点；并通过slaveof命令让其他节点成为其从节点。

将已下线的主节点设置成新的主节点的从节点，当其回复正常时，复制新的主节点，变成新的主节点的从节点

同理，当已下线的服务重新上线时，sentinel会向其发送slaveof命令，让其成为新主的从

哨兵进程的作用

- **监控(Monitoring)**: 哨兵(sentinel)会不断地检查你的Master和Slave是否运作正常。
- **提醒(Notification)**: 当被监控的某个Redis节点出现问题时，哨兵(sentinel)可以通过API向管理员或者其他应用程序发送通知。

- **自动故障迁移(Automatic failover):** 当一个Master不能正常工作时, 哨兵(sentinel)会开始一次自动故障迁移操作

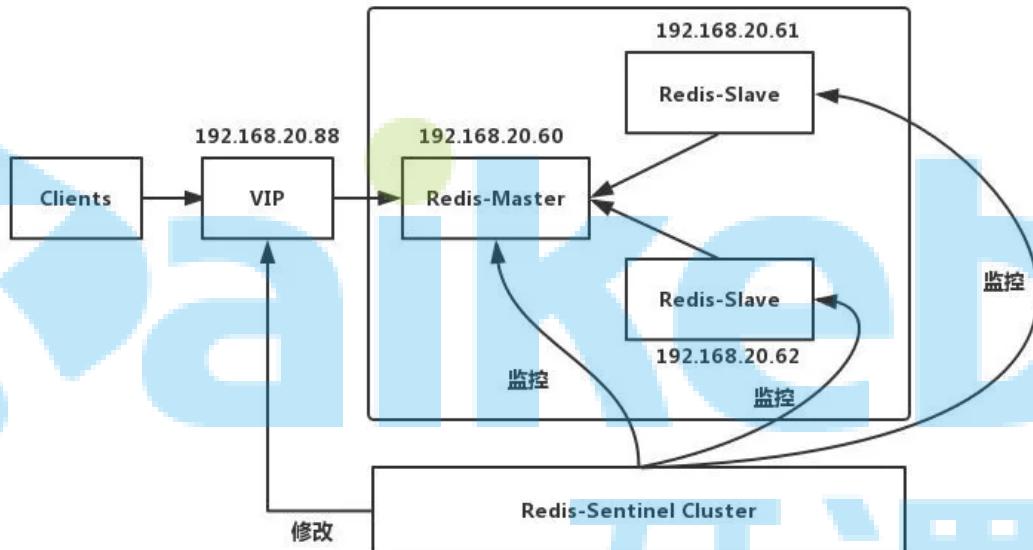
Redis 集群演变

主从复制

参考主从复制

Replication+Sentinel * 高可用

这套架构使用的是社区版本推出的原生高可用解决方案, 其架构图如下!



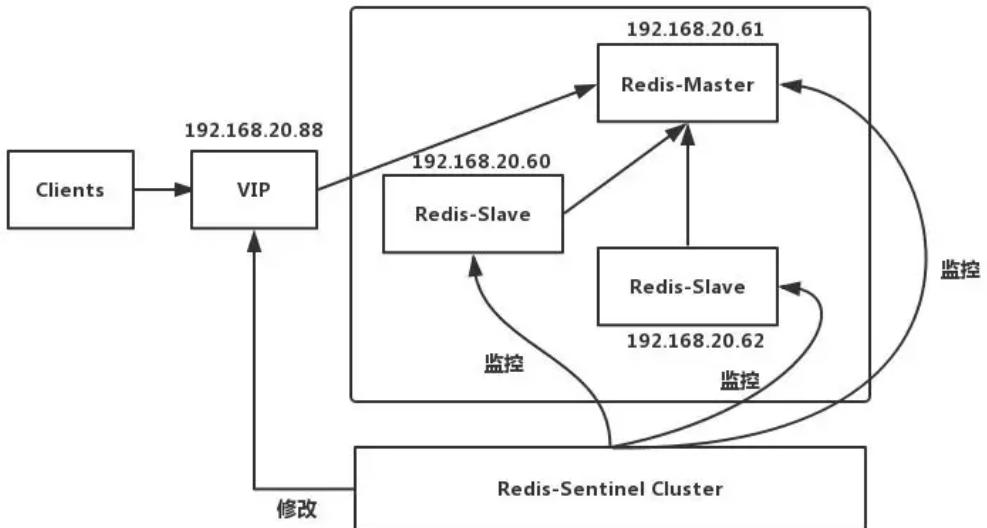
这里Sentinel的作用有三个:

- **监控:** Sentinel 会不断的检查主服务器和从服务器是否正常运行。
- **通知:** 当被监控的某个Redis服务器出现问题, Sentinel通过API脚本向管理员或者其他的应用程序发送通知。
- **自动故障转移:** 当主节点不能正常工作时, Sentinel会开始一次自动的故障转移操作, 它会将与失效主节点是主从关系的其中一个从节点升级为新的主节点, 并且将其他的从节点指向新的主节点。

工作原理

当Master宕机的时候, Sentinel会选举出新的Master, 并根据Sentinel中client-reconfig-script脚本配置的内容, 去动态修改VIP(虚拟IP), 将VIP(虚拟IP)指向新的Master。我们的客户端就连向指定的VIP即可!

故障发生后的转移情况, 可以理解为下图



缺陷

- (1) 主从切换的过程中会丢数据
- (2) Redis只能单点写，不能水平扩容

常用方案：

内网DNS，VIP和 封装客户端直连Redis Sentinel 端口

内网DNS

底层是 Redis Sentinel 集群，代理着 Redis 主从，Web 端连接内网 DNS 提供服务。内网 DNS 按照一定的规则分配，比如 xxxx.redis.cache/queue.port.xxx.xxx，第一个段表示业务简写，第二个段表示这是 Redis 内网域名，第三个段表示 Redis 类型，cache 表示缓存，queue 表示队列，第四个段表示 Redis 端口，第五、第六个段表示内网主域名。当主节点发生故障，比如机器故障、Redis 节点故障或者网络不可达，Sentinel 集群会调用 client-reconfig- 配置的脚本，修改对应端口的内网域名。对应端口的内网域名指向新的 Redis 主节点。

优点：秒级切换，10秒之内；脚本自定义，架构可控；对应用透明，前端不用担心后端发生什么变化

缺点：维护成本高；依赖DNS，存在解析超时；哨兵存在短时间服务不可用；服务时通过外网不可采用

VIP

和第一种方案略有不同，把内网 DNS 换成了虚拟 IP。底层是 Redis Sentinel 集群，代理着 Redis 主从，Web 端通过 VIP 提供服务。在部署 Redis 主从的时候，需要将虚拟 IP 绑定到当前的 Redis 主节点。当主节点发生故障，比如机器故障、Redis 节点故障或者网络不可达，Sentinel 集群会调用 client-reconfig- 配置的脚本，将 VIP 漂移到新的主节点上。

优点：秒级切换，5秒之内；脚本自定义，架构可控；对应用透明，前端不用担心后端发生什么变化

缺点：维护成本更高，使用VIP增加维护成本，并存在IP混乱风险

封装客户端直连 Redis Sentinel 端口

这个主要是因为有些业务只能通过外网访问 Redis，于是衍生出了这种方案。Web 使用客户端连接其中一台 Redis Sentinel 集群中的一台机器的某个端口，然后通过这个端口获取到当前的主节点，然后再连接到真实的 Redis 主节点进行相应的业务员操作。需要注意的是，Redis Sentinel 端口和 Redis 主节点均需要开放访问权限。前端业务使用 Java，有 JedisSentinelPool 可以复用。

优点：服务探测故障及时，DBA维护成本低

缺点：依赖客户端支持 Sentinel；Sentinel 服务器和 Redis 节点需要开放访问权限；对应用有侵入性

Proxy+Replication+Sentinel (仅仅了解)

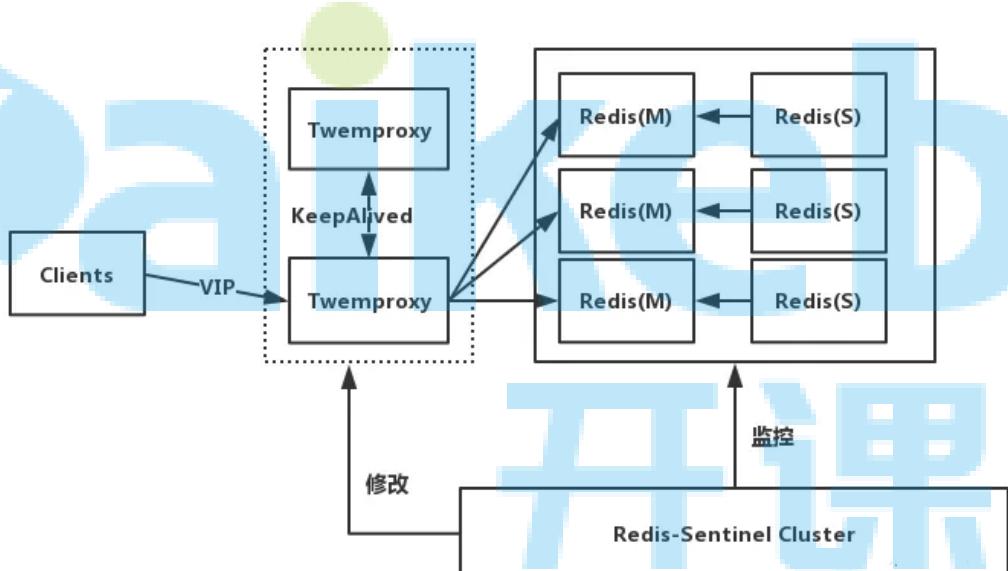
这里的 Proxy 有两种选择：Codis（豌豆荚）和 Twemproxy（推特）。

兔哥经历这套架构的时间为 2015 年，原因有二：

- 因为 Codis 开源的比较晚，考虑到更换组件的成本问题。毕竟本来运行好好的东西，你再去换组件，风险是很大的。
- Redis Cluster 在 2015 年还是试用版，不保证会遇到什么问题，因此不敢尝试。

所以我没接触过 Codis，之前一直用的是 Twemproxy 作为 Proxy。

这里以 Twemproxy 为例说明，如下图所示



工作原理

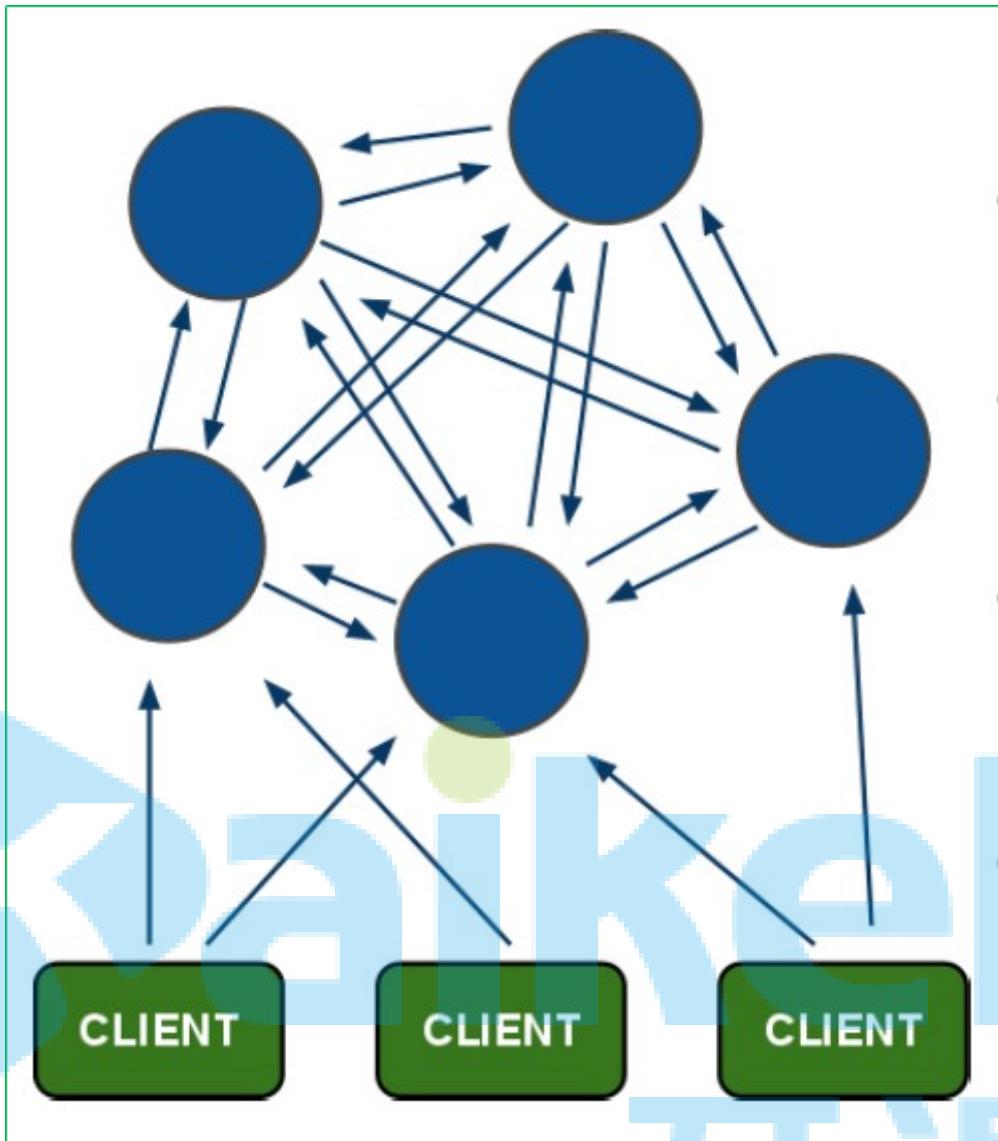
- 前端使用 Twemproxy+KeepAlived 做代理，将其后端的多台 Redis 实例分片进行统一管理与分配
- 每一个分片节点的 Slave 都是 Master 的副本且只读
- Sentinel 持续不断的监控每个分片节点的 Master，当 Master 出现故障且不可用状态时，Sentinel 会通知/启动自动故障转移等动作
- Sentinel 可以在发生故障转移动作后触发相应脚本（通过 client-reconfig-script 参数配置），脚本获取到最新的 Master 来修改 Twemproxy 配置

缺陷：

- (1) 部署结构超级复杂
- (2) 可扩展性差，进行扩缩容需要手动干预
- (3) 运维不方便

Redis Cluster * 扩展性

Redis-cluster架构图

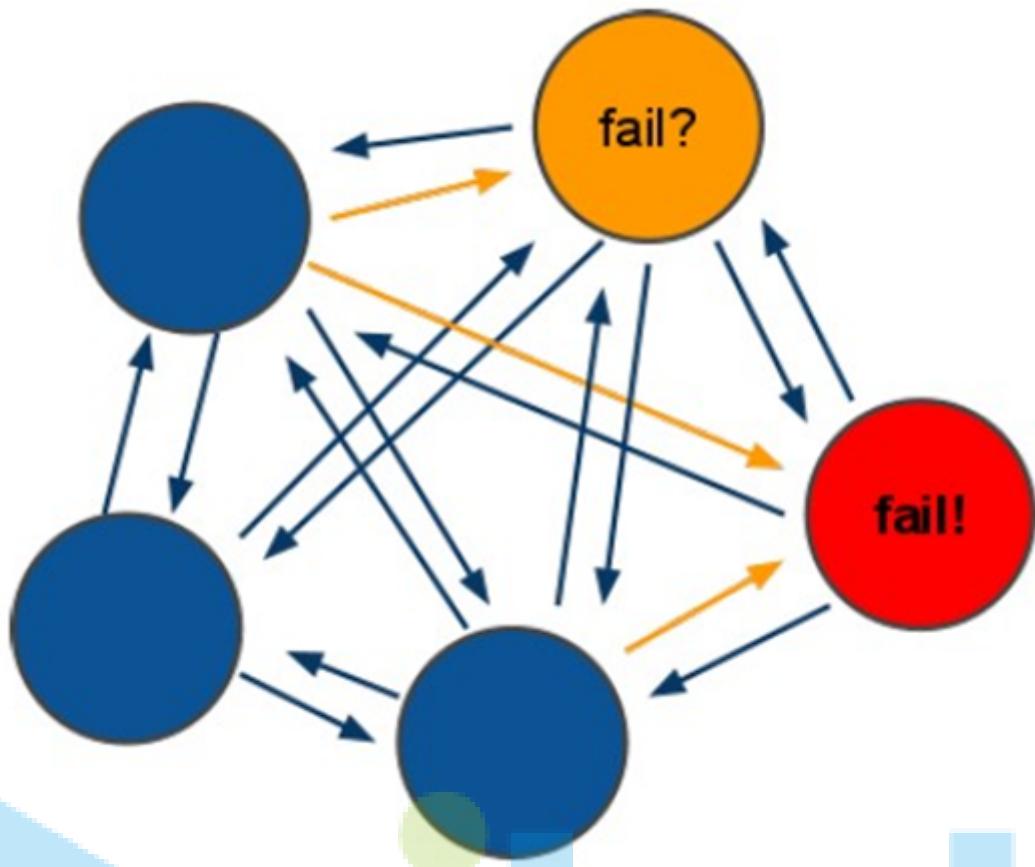


架构细节:

- (1)所有的redis节点彼此互联(**PING-PONG机制**),内部使用二进制协议优化传输速度和带宽.
- (2)节点的fail是通过集群中超过半数的节点检测失效时才生效.
- (3)客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可
- (4)redis-cluster把所有的物理节点映射到[0-16383]slot上,cluster 负责维护node<->slot<->value

1 | Redis 集群中内置了 16384个哈希槽, 当需要在 Redis 集群中放置一个 key-value 时, redis 先对 key 使用 crc16 算法算出一个结果, 然后把结果对 16384 求余数, 这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽, redis 会根据节点数量大致均等的将哈希槽映射到不同的节点

Redis-cluster投票:容错



(1) **节点失效判断**: 集群中所有master参与投票,如果半数以上master节点与其中一个master节点通信超过(cluster-node-timeout),认为该master节点挂掉.

(2) **集群失效判断**:什么时候整个集群不可用(cluster_state:fail)?

- 如果集群任意master挂掉,且当前master没有slave, 则集群进入fail状态。也可以理解成集群的[0-16383]slot映射不完全时进入fail状态。
- 如果集群超过半数以上master挂掉, 无论是否有slave, 集群进入fail状态。

需要开启防火墙, 或者直接关闭防火墙。

```
1 | service iptables stop
```

RedisCluster 集群安装

安装RedisCluster

1 Redis集群最少需要三台主服务器, 三台从服务器。

2

3 端口号分别为: 8001~8006

- 第一步: 创建8001实例, 并编辑redis.conf文件, 修改port为8001。

注意: 创建实例, 即拷贝单机版安装时, 生成的bin目录, 为8001目录。

- 第二步: 修改redis.conf配置文件, 打开cluster-enable yes

- 第三步: 复制8001, 创建8002~8006实例, 注意端口修改。

```
1 | [root@localhost redis]# cp 8001/ 8002 -r
```

- 第四步：启动所有的实例
- 第五步：创建Redis集群

```
1 | [root@localhost 8001]# ./redis-cli --cluster create 127.0.0.1:8001  
2 | 127.0.0.1:8002 127.0.0.1:8003 127.0.0.1:8004 127.0.0.1:8005 127.0.0.1:8006  
3 | --cluster-replicas 1  
4 | >>> Performing hash slots allocation on 6 nodes...  
5 | Master[0] -> Slots 0 - 5460  
6 | Master[1] -> Slots 5461 - 10922  
7 | Master[2] -> Slots 10923 - 16383  
8 | Adding replica 127.0.0.1:8005 to 127.0.0.1:8001  
9 | Adding replica 127.0.0.1:8006 to 127.0.0.1:8002  
10 | Adding replica 127.0.0.1:8004 to 127.0.0.1:8003  
11 | >>> Trying to optimize slaves allocation for anti-affinity  
12 | [WARNING] Some slaves are in the same host as their master  
13 | M: ab492f8084052c670c59a1594ac19583df6be0a6 127.0.0.1:8001  
14 |   slots:[0-5460] (5461 slots) master  
15 | M: c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9 127.0.0.1:8002  
16 |   slots:[5461-10922] (5462 slots) master  
17 | M: 40585c9d836cb31aae27ba29d24803ab2f221063 127.0.0.1:8003  
18 |   slots:[10923-16383] (5461 slots) master  
19 | S: dd19a1123de6b4d7ebe9f5629a45f562bdb3b054 127.0.0.1:8004  
20 |   replicates ab492f8084052c670c59a1594ac19583df6be0a6  
21 | S: 7daf9dd1c8f8b90f6861717c9f9832dfffe8042f6 127.0.0.1:8005  
22 |   replicates c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9  
23 | S: 009cf3bde8b9d67ba051ddb8b80178bd57d30e4e 127.0.0.1:8006  
24 |   replicates 40585c9d836cb31aae27ba29d24803ab2f221063  
Can I set the above configuration? (type 'yes' to accept): yes
```

```
1 | >>> Nodes configuration updated  
2 | >>> Assign a different config epoch to each node  
3 | >>> Sending CLUSTER MEET messages to join the cluster  
4 | Waiting for the cluster to join  
5 | ..  
6 | >>> Performing Cluster Check (using node 127.0.0.1:8001)  
7 | M: ab492f8084052c670c59a1594ac19583df6be0a6 127.0.0.1:8001  
8 |   slots:[0-5460] (5461 slots) master  
9 |   1 additional replica(s)  
10 | M: c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9 127.0.0.1:8002  
11 |   slots:[5461-10922] (5462 slots) master  
12 |   1 additional replica(s)  
13 | M: 40585c9d836cb31aae27ba29d24803ab2f221063 127.0.0.1:8003  
14 |   slots:[10923-16383] (5461 slots) master  
15 |   1 additional replica(s)  
16 | S: 009cf3bde8b9d67ba051ddb8b80178bd57d30e4e 127.0.0.1:8006  
17 |   slots: (0 slots) slave  
18 |   replicates 40585c9d836cb31aae27ba29d24803ab2f221063  
19 | S: 7daf9dd1c8f8b90f6861717c9f9832dfffe8042f6 127.0.0.1:8005  
20 |   slots: (0 slots) slave  
21 |   replicates c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9  
22 | S: dd19a1123de6b4d7ebe9f5629a45f562bdb3b054 127.0.0.1:8004  
23 |   slots: (0 slots) slave  
24 |   replicates ab492f8084052c670c59a1594ac19583df6be0a6
```

```
25 [OK] All nodes agree about slots configuration.  
26 >>> Check for open slots...  
27 >>> Check slots coverage...  
28 [OK] All 16384 slots covered.  
29
```

命令客户端连接集群

```
1 ./redis-cli -h 127.0.0.1 -p 8001 -c
```

```
1 [root@localhost 8001]# ./redis-cli -p 8006 -c  
2
```

查看集群的命令

- 查看集群状态

```
1 127.0.0.1:8006> cluster info  
2 cluster_state:ok  
3 cluster_slots_assigned:16384  
4 cluster_slots_ok:16384  
5 cluster_slots_pfail:0  
6 cluster_slots_fail:0  
7 cluster_known_nodes:6  
8 cluster_size:3  
9 cluster_current_epoch:6  
10 cluster_my_epoch:3  
11 cluster_stats_messages_ping_sent:10603  
12 cluster_stats_messages_pong_sent:10272  
13 cluster_stats_messages_meet_sent:3  
14 cluster_stats_messages_sent:20878  
15 cluster_stats_messages_ping_received:10270  
16 cluster_stats_messages_pong_received:10606  
17 cluster_stats_messages_meet_received:2  
18 cluster_stats_messages_received:20878  
19
```

- 查看集群中的节点:

```
1 127.0.0.1:8006> cluster nodes  
2 40585c9d836cb31aae27ba29d24803ab2f221063 127.0.0.1:8003@18003 master - 0  
1590020495000 3 connected 10923-16383  
3 009cf3bde8b9d67ba051ddb8b80178bd57d30e4e 127.0.0.1:8006@18006 myself,slave  
40585c9d836cb31aae27ba29d24803ab2f221063 0 1590020494000 6 connected  
4 dd19a1123de6b4d7ebe9f5629a45f562bdb3b054 127.0.0.1:8004@18004 slave  
ab492f8084052c670c59a1594ac19583df6be0a6 0 1590020494000 4 connected  
5 7daf9dd1c8f8b90f6861717c9f9832dff8042f6 127.0.0.1:8005@18005 slave  
c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9 0 1590020496382 2 connected  
6 ab492f8084052c670c59a1594ac19583df6be0a6 127.0.0.1:8001@18001 master - 0  
1590020496000 1 connected 0-5460  
7 c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9 127.0.0.1:8002@18002 master - 0  
1590020495376 2 connected 5461-10922
```

Jedis连接集群

代码实现

创建JedisCluster类连接redis集群。

```
1  @Test
2  public void testJedisCluster() throws Exception {
3      //创建一连接, JedisCluster对象,在系统中是单例存在
4      Set<HostAndPort> nodes = new HashSet<>();
5      nodes.add(new HostAndPort("192.168.10.133", 7001));
6      nodes.add(new HostAndPort("192.168.10.133", 7002));
7      nodes.add(new HostAndPort("192.168.10.133", 7003));
8      nodes.add(new HostAndPort("192.168.10.133", 7004));
9      nodes.add(new HostAndPort("192.168.10.133", 7005));
10     nodes.add(new HostAndPort("192.168.10.133", 7006));
11     JedisCluster cluster = new JedisCluster(nodes);
12     //执行JedisCluster对象中的方法, 方法和redis一一对应。
13     cluster.set("cluster-test", "my jedis cluster test");
14     String result = cluster.get("cluster-test");
15     System.out.println(result);
16     //程序结束时需要关闭JedisCluster对象
17     cluster.close();
18 }
19 }
```

使用spring

Ø 配置applicationContext.xml

```
1  <!-- 连接池配置 -->
2  <bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
3      <!-- 最大连接数 -->
4      <property name="maxTotal" value="30" />
5      <!-- 最大空闲连接数 -->
6      <property name="maxIdle" value="10" />
7      <!-- 每次释放连接的最大数目 -->
8      <property name="numTestsPerEvictionRun" value="1024" />
9      <!-- 释放连接的扫描间隔(毫秒) -->
10     <property name="timeBetweenEvictionRunsMillis" value="30000" />
11     <!-- 连接最小空闲时间 -->
12     <property name="minEvictableIdleTimeMillis" value="1800000" />
13     <!-- 连接空闲多久后释放, 当空闲时间>该值 且 空闲连接>最大空闲连接数 时直接释放 -->
14     <property name="softMinEvictableIdleTimeMillis" value="10000" />
15     <!-- 获取连接时的最大等待毫秒数, 小于零:阻塞不确定的时间,默认-1 -->
16     <property name="maxWaitMillis" value="1500" />
17     <!-- 在获取连接的时候检查有效性, 默认false -->
18     <property name="testOnBorrow" value="true" />
19     <!-- 在空闲时检查有效性, 默认false -->
20     <property name="testWhileIdle" value="true" />
21     <!-- 连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true -->
22     <property name="blockWhenExhausted" value="false" />
23 </bean>
24 <!-- redis集群 -->
25 <bean id="jedisCluster" class="redis.clients.jedis.JedisCluster">
```

```
26     <constructor-arg index="0">
27         <set>
28             <bean class="redis.clients.jedis.HostAndPort">
29                 <constructor-arg index="0" value="192.168.101.3">
30             </constructor-arg>
31         </set>
32     <constructor-arg>
33         <bean class="redis.clients.jedis.HostAndPort">
34             <constructor-arg index="0" value="192.168.101.3">
35         </constructor-arg>
36     <constructor-arg>
37         <bean class="redis.clients.jedis.HostAndPort">
38             <constructor-arg index="0" value="192.168.101.3">
39         </constructor-arg>
40     <constructor-arg>
41         <bean class="redis.clients.jedis.HostAndPort">
42             <constructor-arg index="0" value="192.168.101.3">
43         </constructor-arg>
44     <constructor-arg>
45         <bean class="redis.clients.jedis.HostAndPort">
46             <constructor-arg index="0" value="192.168.101.3">
47         </constructor-arg>
48     <constructor-arg>
49         <bean class="redis.clients.jedis.HostAndPort">
50             <constructor-arg index="0" value="192.168.101.3">
51         </constructor-arg>
52     </set>
53 </constructor-arg>
54 <constructor-arg index="1" ref="jedisPoolConfig"></constructor-arg>
55 </bean>
56
```

1 |

Ø 测试代码

```
1 private ApplicationContext applicationContext;
2     @Before
3     public void init() {
4         applicationContext = new ClassPathXmlApplicationContext(
5             "classpath:applicationContext.xml");
6     }
7
8     // redis集群
9     @Test
10    public void testJedisCluster() {
11        JedisCluster jediscluster = (JedisCluster) applicationContext
```

```
12         .getBean("jedisCluster");
13
14     jedisCluster.set("name", "zhangsan");
15     String value = jedisCluster.get("name");
16     System.out.println(value);
17 }
18
```

总结

工作原理

- 客户端与Redis节点直连,不需要中间Proxy层, 直接连接任意一个Master节点
- 根据公式HASH_SLOT=CRC16(key) mod 16384, 计算出映射到哪个分片上, 然后Redis会去相应的节点进行操作

优点:

- (1)无需Sentinel哨兵监控, 如果Master挂了, Redis Cluster内部自动将Slave切换Master
- (2)可以进行水平扩容
- (3)支持自动化迁移, 当出现某个Slave宕机了, 那么就只有Master了, 这时候的高可用性就无法很好的保证了, 万一Master也宕机了, 咋办呢? 针对这种情况, 如果说其他Master有多余的Slave, 集群自动把多余的Slave迁移到没有Slave的Master 中。

缺点:

- (1)批量操作是个坑
- (2)资源隔离性较差, 容易出现相互影响的情况。

Redis 事务

事务演示

```
1 127.0.0.1:6379> multi
2 OK
3 127.0.0.1:6379> set s1 111
4 QUEUED
5 127.0.0.1:6379> hset set1 name zhangsan
6 QUEUED
7 127.0.0.1:6379> exec
8 1) OK
9 2) (integer) 1
10 127.0.0.1:6379> multi
11 OK
12 127.0.0.1:6379> set s2 222
13 QUEUED
14 127.0.0.1:6379> hset set2 age 20
15 QUEUED
16 127.0.0.1:6379> discard
17 OK
18 127.0.0.1:6379> exec
19 (error) ERR EXEC without MULTI
20
21 127.0.0.1:6379> watch s1
22 OK
```

```
23 127.0.0.1:6379> multi
24 OK
25 127.0.0.1:6379> set s1 555
26 QUEUED
27 127.0.0.1:6379> exec          # 此时在没有exec之前，通过另一个命令窗口对监控的s1字段
进行修改
28 (nil)
29 127.0.0.1:6379> get s1
30 111
```

事务失败处理

- Redis 语法错误

整个事务的命令在队列里都清除

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> sets s1 111
(error) ERR unknown command 'sets'
127.0.0.1:6379> set s1
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379> set s4 444
QUEUED
127.0.0.1:6379> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get s4
(nil)
```

- Redis 运行错误

在队列里正确的命令可以执行 (弱事务性)

弱事务性：

- 在队列里正确的命令可以执行 (非原子操作)
- 不支持回滚

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s4 444
QUEUED
127.0.0.1:6379> lpush s4 111 222
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6379> get s4
"444"
127.0.0.1:6379>
```

- Redis 不支持事务回滚 (为什么呢)

- 大多数事务失败是因为语法错误或者类型错误，这两种错误，在开发阶段都是可以预见的
- Redis 为了性能方面就忽略了事务回滚。 (回滚记录历史版本)

Redis乐观锁

乐观锁基于CAS (Compare And Swap) 思想 (比较并替换)，是不具有互斥性，不会产生锁等待而消耗资源，但是需要反复的重试，但也是因为重试的机制，能比较快的响应。因此我们可以利用redis来实现乐观锁。具体思路如下：

- 1、利用redis的watch功能，监控这个redisKey的状态值
- 2、获取redisKey的值
- 3、创建redis事务
- 4、给这个key的值+1
- 5、然后去执行这个事务，如果key的值被修改过则回滚，key不加1

```

1 public void watch() {
2     try {
3         String watchKeys = "watchKeys";
4         //初始值 value=1
5         jedis.set(watchKeys, 1);
6         //监听key为watchKeys的值
7         jedis.watch(watchKeys);
8
9         //开启事务
10        Transaction tx = jedis.multi();
11
12        //watchKeys自增加一
13        tx.incr(watchKeys);
14
15        //执行事务，如果其他线程对watchKeys中的value进行修改，则该事务将不会执行
16        //通过redis事务以及watch命令实现乐观锁
17        List<Object> exec = tx.exec();
18        if (exec == null) {
19            System.out.println("事务未执行");
20        } else {
21            System.out.println("事务成功执行，watchKeys的value成功修改");
22        }
23    } catch (Exception e) {
24        e.printStackTrace();
25    } finally {
26        jedis.close();
27    }
28 }

```

Redis乐观锁实现秒杀

```

1 public class Seckill {
2     public static void main(String[] args) {
3         String redisKey = "second";
4
5         ExecutorService executorService = Executors.newFixedThreadPool(20);
6         try {
7             Jedis jedis = new Jedis("127.0.0.1", 6378);
8             // 初始值
9             jedis.set(redisKey, "0");
10            jedis.close();
11        } catch (Exception e) {
12            e.printStackTrace();
13        }
14
15        for (int i = 0; i < 1000; i++) {
16

```

```
17     executorService.execute(() -> {
18
19         Jedis jedis1 = new Jedis("127.0.0.1", 6378);
20         try {
21             jedis1.watch(redisKey);
22             String redisvalue = jedis1.get(redisKey);
23             int valInteger = Integer.valueOf(redisvalue);
24             String userInfo = UUID.randomUUID().toString();
25
26             // 没有秒完
27             if (valInteger < 20) {
28                 Transaction tx = jedis1.multi();
29                 tx.incr(redisKey);
30                 List list = tx.exec();
31                 // 秒成功 失败返回空list而不是空
32                 if (list != null && list.size() > 0) {
33
34                     System.out.println("用户: " + userInfo + ", 秒杀成
功! 当前成功人数: " + (valInteger + 1));
35
36                 }
37                 // 版本变化, 被别人抢了。
38             } else {
39                 System.out.println("用户: " + userInfo + ", 秒杀失
败");
40             }
41             // 秒完了
42             else {
43                 System.out.println("已经有20人秒杀成功, 秒杀结束");
44
45             }
46         } catch (Exception e) {
47             e.printStackTrace();
48         } finally {
49             jedis1.close();
50         }
51     });
52 });
53 }
54
55 executorService.shutdown();
56
57 }
58
59 }
```

Redis 和lua 整合

什么是lua

lua是一种轻量小巧的脚本语言，用标准C语言编写并以源代码形式开放，其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Redis中使用lua的好处

1. **减少网络开销**, 在Lua脚本中可以把多个命令放在同一个脚本中运行
2. **原子操作**, redis会将整个脚本作为一个整体执行, 中间不会被其他命令插入。换句话说, 编写脚本的过程中无需担心会出现竞态条件
3. **复用性**, 客户端发送的脚本会永远存储在redis中, 这意味着其他客户端可以复用这一脚本来完成同样的逻辑

lua的安装和语法

lua 教程 <https://www.runoob.com/lua/lua-tutorial.html>

Redis整合lua脚本

从Redis2.6.0版本开始, 通过**内置的lua编译/解释器**, 可以使用EVAL命令对lua脚本进行求值。

EVAL命令

- 在redis客户端中, 执行以下命令:

```
1 | EVAL script numkeys key [key ...] arg [arg ...]
```

- 命令说明:

- **script参数**: 是一段Lua脚本程序, 它会被运行在Redis服务器上下文中, 这段脚本不必(也不应该)定义为一个Lua函数。
- **numkeys参数**: 用于指定键名参数的个数。
- **key [key ...]参数**: 从EVAL的第三个参数开始算起, 使用了numkeys个键(key), 表示在脚本中所用到的那些Redis键(key), 这些键名参数可以在Lua中通过全局变量KEYS数组, 用1为基址的形式访问(KEYS[1], KEYS[2], 以此类推)。
- **arg [arg ...]参数**: 可以在Lua中通过全局变量ARGV数组访问, 访问的形式和KEYS变量类似(ARGV[1]、ARGV[2], 诸如此类)。

- 例如:

```
1 | ./redis-cli
2 | > eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
3 |
4 | 1) "key1"
5 |
6 | 2) "key2"
7 |
8 | 3) "first"
9 |
10 | 4) "second"
```

lua 脚本调用Redis 命令

redis.call();

返回值就是redis命令执行的返回值

如果出错, 返回错误信息, 不继续执行

redis.pcall();

返回值就是redis命令执行的返回值

如果出错了 记录错误信息，继续执行

注意事项

- 在脚本中，使用return语句将返回值返回给客户端，如果没有return，则返回nil
- 示例：

```
1 | 127.0.0.1:6379> eval "return redis.call('set',KEYS[1],'bar')" 1 foo
2 |
3 | OK
```

redis-cli --eval

可以使用redis-cli --eval命令指定一个lua脚本文件去执行。

脚本文件(redis.lua)，内容如下：

```
1 | local num = redis.call('GET', KEYS[1]);
2 |
3 | if not num then
4 |   return 0;
5 | else
6 |   local res = num * ARGV[1];
7 |   redis.call('SET',KEYS[1], res);
8 |   return res;
9 | end
```

在redis客户端，执行脚本命令：

```
1 | [root@localhost bin]# ./redis-cli --eval redis.lua lua:incrbymul , 8
2 | (integer) 0
3 | [root@localhost bin]# ./redis-cli incr lua:incrbymul
4 | (integer) 1
5 | [root@localhost bin]# ./redis-cli --eval redis.lua lua:incrbymul , 8
6 | (integer) 8
7 | [root@localhost bin]# ./redis-cli --eval redis.lua lua:incrbymul , 8
8 | (integer) 64
9 | [root@localhost bin]# ./redis-cli --eval redis.lua lua:incrbymul , 2
10 | (integer) 128
11 | [root@localhost bin]# ./redis-cli
```

命令格式说明：

```
1 | --eval: 告诉redis客户端去执行后面的lua脚本
2 | redis.lua: 具体的lua脚本文件名称
3 | lua:incrbymul : lua脚本中需要的key
4 | 8: lua脚本中需要的value
```

- 注意事项：

```
1 | 上面命令中keys和values中间需要使用逗号隔开，并且逗号两边都要有空格
```

Redis+lua 秒杀

秒杀场景经常使用这个东西，主要利用他的原子性

1.首先定义redis数据结构

```
1 | goodId:  
2 | {  
3 |   "total":100,  
4 |   "released":0;  
5 | }
```

- 其中goodId为商品id号，可根据此来查询相关的数据结构信息，total为总数，released为发放出去的数量，可使用数为total-released

2.编写lua脚本

```
1 | local n = tonumber(ARGV[1])  
2 | if not n or n == 0 then  
3 | return 0  
4 | end  
5 | local vals = redis.call("HGET", KEYS[1], "total", "released");  
6 | local total = tonumber(vals[1])  
7 | local blocked = tonumber(vals[2])  
8 | if not total or not blocked then  
9 | return 0  
10 | end  
11 | if blocked + n <= total then  
12 |   redis.call("HINCRBY", KEYS[1], "released", n)  
13 |   return n;  
14 | end  
15 | return 0
```

- 执行脚本命令 EVAL script_string 1 goodId apply_count
- 若库存足够则返回申请的数量，否则返回0，不返回可满足的剩余数

3.spring boot 调用

- pom dependency

```
1 | <groupId>org.springframework.boot</groupId>  
2 | <artifactId>spring-boot-starter-data-redis</artifactId>  
3 | <version>2.0.1.RELEASE</version>
```

```
1     long count = redisHelper.getStrCache().execute(new
2         RedisCallback<Long>() {
3             @Nullable
4             @Override
5             public Long doInRedis(RedisConnection redisConnection) throws
6                 DataAccessException {
7                 long ret =
8                     redisConnection.eval(script.getScriptAsString().getBytes(),
9                         ReturnType.INTEGER, 1, key.getBytes(), String.valueOf(count).getBytes());
10                return ret;
11            }
12        });
13    }
```

4.redis->database

针对redis到databases的更新，思考了很久，没有找到较好的解决办法，先采用定时任务异步更新。至于数据是否丢失的问题，如果redis挂了，重启后redis会恢复数据，等下次定时任务就可以将数据库中的数据保持一致，缺点是redis挂了秒杀活动会失败。

至于redis到database更新方案：

- redis存一份相关hash键名单表，通过读取名单表来读取更新
- 通过流式读取databases中的表来读取更新。

Redis分布式锁

业务场景

- 1、库存超卖
- 2、防止用户重复下单
- 3、MQ消息去重
- 4、订单操作变更

分析：

业务场景共性：

共享资源竞争

用户id、订单id、商品id。。。

解决方案

共享资源互斥

共享资源串行化

问题转化

锁的问题（将需求抽象后得到问题的本质）

锁的处理

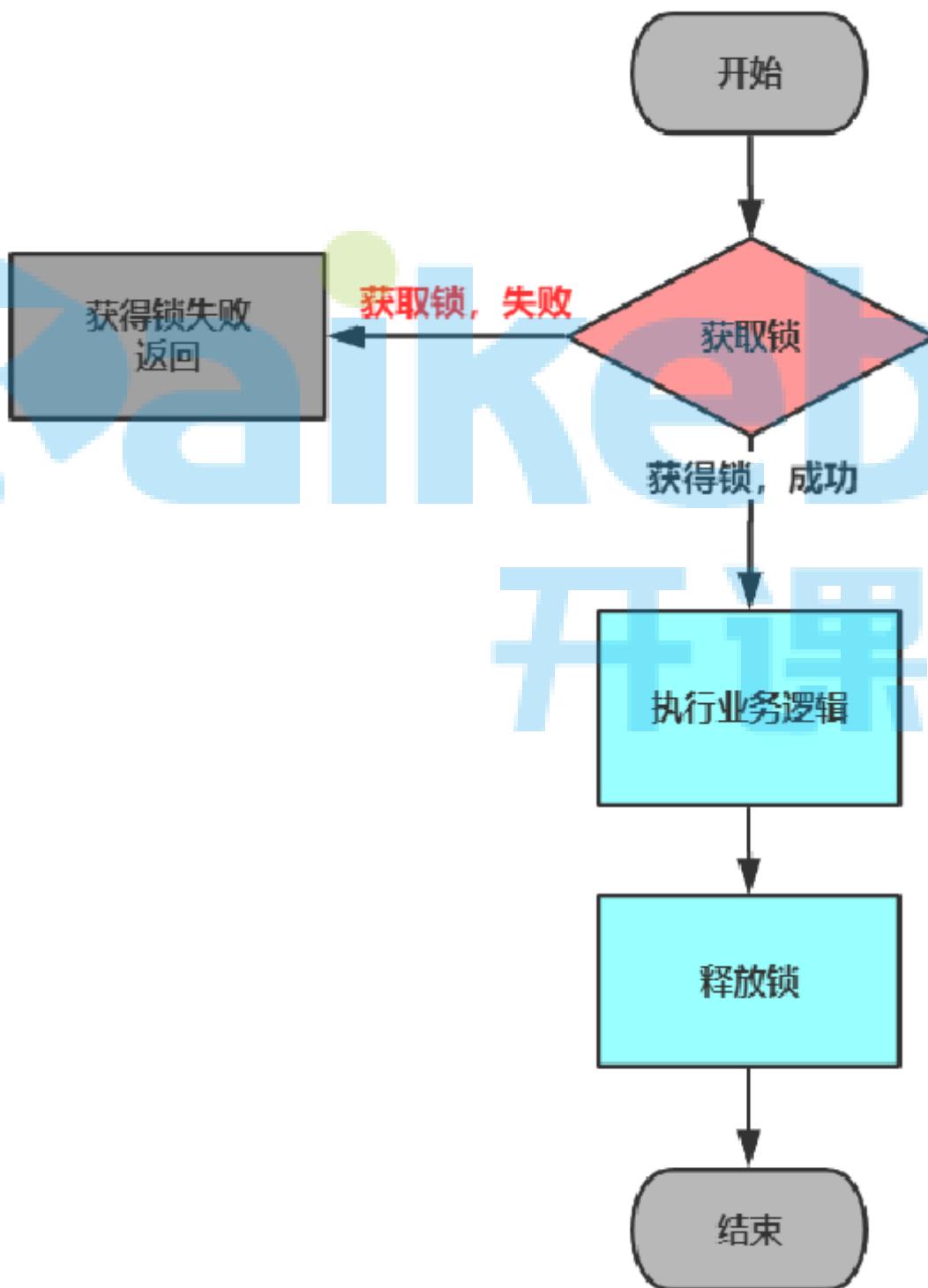
- 单应用中使用锁：（单进程多线程）

- 分布式应用中使用锁：（多进程多线程）

1 | 分布式锁是控制分布式系统之间同步访问共享资源的一种方式。

分布式锁

流程图



分布式锁的状态

1. 客户端通过竞争获取锁才能对共享资源进行操作(①获取锁);
2. 当持有锁的客户端对共享资源进行操作时 (②占有锁)
3. 其他客户端都不可以对这个资源进行操作 (③阻塞)
4. 直到持有锁的客户端完成操作(④释放锁);

分布式锁特点

互斥性

在任意时刻，只有一个客户端可以持有锁（排他性）

高可用，具有容错性

只要锁服务集群中的大部分节点正常运行，客户端就可以进行加锁解锁操作

避免死锁

具备失效机制，锁在一段时间之后一定会释放。（正常释放或超时释放）

加锁和解锁为同一个客户端

一个客户端不能释放其他客户端加的锁了

分布式锁的实现方式

基于数据库实现分布式锁

基于zookeeper时节点的分布式锁

基于Redis的分布式锁

基于Etcd的分布式锁

Redis方式实现分布式锁

只留下正确的方式

获取锁

开课吧

Redis2.6.12版本之前，使用Lua脚本保证原子性，获取锁代码

```
// 加锁脚本
private static final String SCRIPT_TRYLOCK = "if redis.call('setnx', KEYS[1], ARGV[1]) == 1 then redis.call('pexpire', KEYS[1], ARGV[2]) return 1 else return 0 end";
/**
 * 使用Lua脚本，尝试获取redis分布式锁
 * @param jedis
 * @param lockKey 锁
 * @param lockValue 锁值
 * @param expireTime 超期时间，单位秒
 * @return 是否获取成功
 */
public static boolean tryLockLua(Jedis jedis, String lockKey, String lockValue,
int expireTime) {
    int result =
(jint)jedis.eval(SCRIPT_TRYLOCK, 1, lockKey, lockValue, String.valueOf(expireTime)); //设置锁
    if (result == 1) {//获取锁成功
        return true;
    }
    return false;
}
```

从Redis2.6.12版本开始，使用Set一个命令实现加锁，获取锁代

```
/**
 * 获取redis分布式锁
 * @param jedis
 * @param lockKey 锁
 * @param lockValue 锁值
 * @param expireTime 超期时间，单位毫秒
 * @return 是否获取成功
 */
public static boolean tryLock(Jedis jedis, String lockKey, String lockValue, int
expireTime) {
    String result = jedis.set(lockKey, lockValue, "NX", "PX", expireTime);

    if ("OK".equals(result)) {
        return true;
    }
    return false;
}
```

释放锁

redis+lua 脚本

```
1 public static boolean releaseLock(String lockKey, String requestId) {
2     String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return
redis.call('del', KEYS[1]) else return 0 end";
3     Object result = jedis.eval(script,
4         Collections.singletonList(lockKey), Collections.singletonList(requestId));
5     if (result.equals(1L)) {
6         return true;
7     }
8 }
```

Redis分布式锁--优缺点

优点

Redis是基于内存存储，并发性能好。

缺点

1. 需要考虑原子性、超时、误删等情形。
2. 获锁失败时，客户端只能自旋等待，在高并发情况下，性能消耗比较大。

本质分析

本质分析

CAP模型分析

在分布式环境下不可能满足三者共存，只能满足其中的两者共存，在分布式下P不能舍弃(舍弃P就是单机了)。

所以只能是CP（强一致性模型）和AP（高可用模型）。

分布式锁是CP模型，Redis集群是AP模型。（base）

为什么还可以用Redis实现分布式锁？

与业务有关

当业务不需要数据强一致性时，比如：社交场景，就可以使用Redis实现分布式锁

当业务必须要数据的强一致性，即不允许重复获得锁，比如金融场景（重复下单，重复转账）就不要使用

可以使用CP模型实现，比如：zookeeper和etcd。

开课吧

课程主题

Redis性能调优及缓存常见问题

课程目标

1. 掌握 redis客户端优化方案
2. 掌握使用分布式架构来增加读写速度
3. 熟悉redis外部调优方案
4. 理解缓存雪崩，缓存击穿和缓存穿透原理并掌握其解决方案
5. 理解双写不一致的原因及解决方案

知识要点

课程主题

课程目标

知识要点

Redis分布式锁

- 生产环境中的分布式锁
- Redisson分布式锁的实现原理
- 加锁机制
- 锁互斥机制
- 自动延时机制
- 可重入锁机制
- 释放锁机制
- Redisson分布式锁的使用
- 加入jar包的依赖
- 配置Redisson
- 锁的获取和释放
- 业务逻辑中使用分布式锁

Redis 性能调优

- 禁用长耗时的查询命令
- 使用 slowlog 优化耗时命令
- 避免大量数据同时失效
- 使用 Pipeline 批量操作数据
- 客户端使用优化
- 使用分布式架构来增加读写速度
- 使用物理机而非虚拟机
- 禁用 THP 特性

缓存常见问题

- 缓存雪崩
- 缓存击穿
- 缓存穿透
- 缓存预热
- 缓存更新
- 缓存降级
- 缓存双写一致性
 - 先更新数据库再更新缓存(不建议使用)
 - 先更新数据库再删除缓存
 - 先删除缓存再更新数据库

Redis 常见面试问题

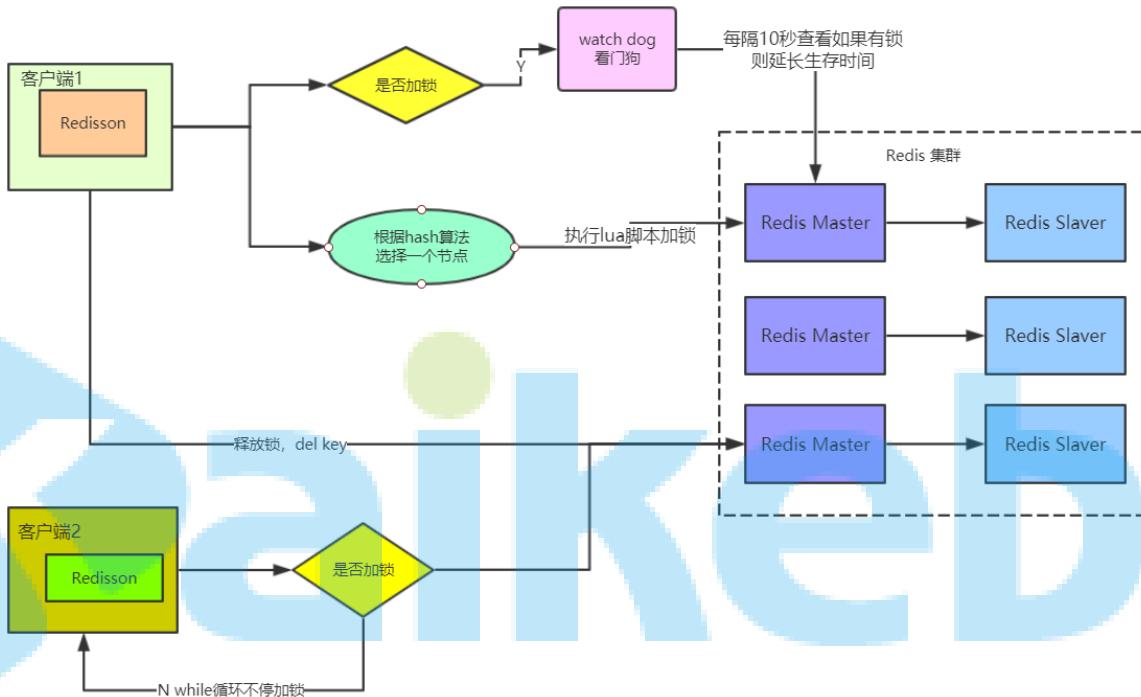
开课吧

Redis分布式锁

生产环境中的分布式锁

目前落地生产环境用分布式锁，一般采用开源框架，比如Redisson。下面来讲一下Redisson对Redis分布式锁的实现。

Redisson分布式锁的实现原理



加锁机制

如果该客户端面对的是一个redis cluster集群，他首先会根据hash节点选择一台机器。

发送lua脚本到redis服务器上，脚本如下：

```
1  "if (redis.call('exists',KEYS[1])==0) then "+  
2    "redis.call('hset',KEYS[1],ARGV[2],1) ; "+  
3    "redis.call('pexpire',KEYS[1],ARGV[1]) ; "+  
4    "return nil; end ;"+  
5  "if (redis.call('hexists',KEYS[1],ARGV[2]) ==1 ) then "+  
6    "redis.call('hincrby',KEYS[1],ARGV[2],1) ; "+  
7    "redis.call('pexpire',KEYS[1],ARGV[1]) ; "+  
8    "return nil; end ;"+  
9  "return redis.call('pttl',KEYS[1]) ;"
```

lua的作用：保证这段复杂业务逻辑执行的原子性。

lua的解释：

KEYS[1]： 加锁的key

ARGV[1]： key的生存时间， 默认为30秒

ARGV[2]： 加锁的客户端ID (`UUID.randomUUID() + ":" + threadId`)

第一段if判断语句，就是用“exists myLock”命令判断一下，如果你要加锁的那个锁key不存在的话，你就进行加锁。如何加锁呢？很简单，用下面的命令：

```
hset myLock
```

```
8743c9c0-0795-4907-87fd-6c719a6b4586:1 1
```

通过这个命令设置一个hash数据结构，这行命令执行后，会出现一个类似下面的数据结构：

```
myLock :{"8743c9c0-0795-4907-87fd-6c719a6b4586:1":1 }
```

上述就代表“8743c9c0-0795-4907-87fd-6c719a6b4586:1”这个客户端对“myLock”这个锁key完成了加锁。

接着会执行“pexpire myLock 30000”命令，设置myLock这个锁key的生存时间是30秒。

锁互斥机制

那么在这个时候，如果客户端2来尝试加锁，执行了同样的一段lua脚本，会咋样呢？

很简单，第一个if判断会执行“exists myLock”，发现myLock这个锁key已经存在了。

接着第二个if判断，判断一下，myLock锁key的hash数据结构中，是否包含客户端2的ID，但是明显不是的，因为那里包含的是客户端1的ID。

所以，客户端2会获取到pttl myLock返回的一个数字，这个数字代表了myLock这个锁key的剩余生存时间。比如还剩15000毫秒的生存时间。

此时客户端2会进入一个while循环，不停的尝试加锁。

自动延时机制

只要客户端1一旦加锁成功，就会启动一个watch dog看门狗，他是一个后台线程，会每隔10秒检查一下，如果客户端1还持有锁key，那么就会不断的延长锁key的生存时间。

可重入锁机制

第一个if判断肯定不成立，“exists myLock”会显示锁key已经存在了。

第二个if判断会成立，因为myLock的hash数据结构中包含的那个ID，就是客户端1的那个ID，也就是“8743c9c0-0795-4907-87fd-6c719a6b4586:1”

此时就会执行可重入加锁的逻辑，他会用：

```
incrby myLock
```

```
8743c9c0-0795-4907-87fd-6c719a6b4586:1 1
```

通过这个命令，对客户端1的加锁次数，累加1。数据结构会变成：

```
myLock :{"8743c9c0-0795-4907-87fd-6c719a6b4586:1":2 }
```

释放锁机制

执行lua脚本如下：

```
1 #如果key已经不存在，说明已经被解锁，直接发布（publish）redis消息
2 "if (redis.call('exists', KEYS[1]) == 0) then " +
3     "redis.call('publish', KEYS[2], ARGV[1]); " +
4     "return 1; " +
5     "end;" +
6 # key和field不匹配，说明当前客户端线程没有持有锁，不能主动解锁。
```

```

7          "if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then
8          " +
9                  "return nil;" +
10             "# 将value减1
11             "local counter = redis.call('hincrby', KEYS[1],
12               ARGV[3], -1); " +
13             "# 如果counter>0说明锁在重入, 不能删除key
14             "if (counter > 0) then " +
15                 "redis.call('pexpire', KEYS[1], ARGV[2]); " +
16                 "return 0; " +
17             "# 删除key并且publish 解锁消息
18             "else " +
19                 "redis.call('del', KEYS[1]); " +
20                 "redis.call('publish', KEYS[2], ARGV[1]); " +
21                 "return 1; " +
22             "end; " +
23             "return nil;";

```

- KEYS[1]：需要加锁的key，这里需要是字符串类型。
 - KEYS[2]：redis消息的ChannelName,一个分布式锁对应唯一的一个channelName:
“redisson_lockchannel{” + getName() + “}”
 - ARGV[1]：reids消息体，这里只需要一个字节的标记就可以，主要标记redis的key已经解锁，再结合redis的Subscribe，能唤醒其他订阅解锁消息的客户端线程申请锁。
 - ARGV[2]：锁的超时时间，防止死锁
 - ARGV[3]：锁的唯一标识，也就是刚才介绍的 id (UUID.randomUUID() + ":" + threadId)
- 如果执行lock.unlock(), 就可以释放分布式锁，此时的业务逻辑也是非常简单的。

其实说白了，就是每次都对myLock数据结构中的那个加锁次数减1。

如果发现加锁次数是0了，说明这个客户端已经不再持有锁了，此时就会用：

“del myLock”命令，从redis里删除这个key。

然后呢，另外的客户端2就可以尝试完成加锁了。

Redisson分布式锁的使用

加入jar包的依赖

```

1 <dependency>
2     <groupId>org.redisson</groupId>
3     <artifactId>redisson</artifactId>
4     <version>2.7.0</version>
5 </dependency>

```

配置Redisson

```

1 public class RedissonManager {
2     private static Config config = new Config();
3     //声明redisso对象
4     private static Redisson redisson = null;
5     //实例化redisson
6     static{

```

```

7     config.useClusterServers()
8
9     // 集群状态扫描间隔时间，单位是毫秒
10    .setScanInterval(2000)
11
12    //cluster方式至少6个节点(3主3从, 3主做sharding, 3从用来保证主宕机后可以高可用)
13
14    .addNodeAddress("redis://127.0.0.1:6379" )
15
16    .addNodeAddress("redis://127.0.0.1:6380")
17
18    .addNodeAddress("redis://127.0.0.1:6381")
19
20    .addNodeAddress("redis://127.0.0.1:6382")
21
22    .addNodeAddress("redis://127.0.0.1:6383")
23
24    .addNodeAddress("redis://127.0.0.1:6384");
25
26
27    //得到redisson对象
28    redisson = (Redisson) Redisson.create(config);
29
30 }
31
32 //获取redisson对象的方法
33 public static Redisson getRedisson(){
34     return redisson;
35 }
36 }
```

锁的获取和释放

```

1 public class DistributedRedisLock {
2     //从配置类中获取redisson对象
3     private static Redisson redisson = RedissonManager.getRedisson();
4     private static final String LOCK_TITLE = "redisLock_";
5     //加锁
6     public static boolean acquire(String lockName){
7         //声明key对象
8         String key = LOCK_TITLE + lockName;
9         //获取锁对象
10        RLock mylock = redisson.getLock(key);
11        //加锁，并且设置锁过期时间3秒，防止死锁的产生      uuid+threadID
12        mylock.lock(3,TimeUtil.SECOND);
13        //加锁成功
14        return true;
15    }
16    //锁的释放
17    public static void release(String lockName){
18        //必须是和加锁时的同一个key
19        String key = LOCK_TITLE + lockName;
20        //获取锁对象
21        RLock mylock = redisson.getLock(key);
22        //释放锁（解锁）
23        mylock.unlock();
24    }
}
```

```
25 }  
26 }  
27 }
```

业务逻辑中使用分布式锁

```
1 public String discount() throws IOException{  
2     String key = "test123";  
3     //加锁  
4     DistributedRedisLock.acquire(key);  
5     //执行具体业务逻辑  
6     dosoming  
7     //释放锁  
8     DistributedRedisLock.release(key);  
9     //返回结果  
10    return soming;  
11 }
```

Redis 性能调优

禁用长耗时的查询命令

Redis 绝大多数读写命令的时间复杂度都在 $O(1)$ 到 $O(N)$ 之间，在官方文档对每命令都有时间复杂度说明，如下图

KEYS pattern

Available since 1.0.0.

Time complexity: $O(N)$ with N being the number of keys in the database, under the assumption that the key names in the database and the given pattern have limited length.

其中 $O(1)$ 表示可以安全使用的，而 $O(N)$ 就应该当心了， N 表示不确定，数据越大查询的速度可能会越慢。因为 Redis 只用一个线程来做数据查询，如果这些指令耗时很长，就会阻塞 Redis，造成大量延时。

要避免 $O(N)$ 命令对 Redis 造成的影响，可以从以下几个方面入手改造：

- 决定禁止使用 keys 命令；
- 避免一次查询所有的成员，要使用 scan 命令进行分批的，游标式的遍历；
- 通过机制严格控制 Hash、Set、Sorted Set 等结构的数据大小；
- 将排序、并集、交集等操作放在客户端执行，以减少 Redis 服务器运行压力；
- 删除 (del) 一个大数据的时候，可能会需要很长时间，所以建议用异步删除的方式 unlink，它会启动一个新的线程来删除目标数据，而不阻塞 Redis 的主线程。

使用 slowlog 优化耗时命令

我们可以使用 slowlog 功能找出最耗时的 Redis 命令进行相关的优化，以提升 Redis 的运行速度，慢查询有两个重要的配置项：

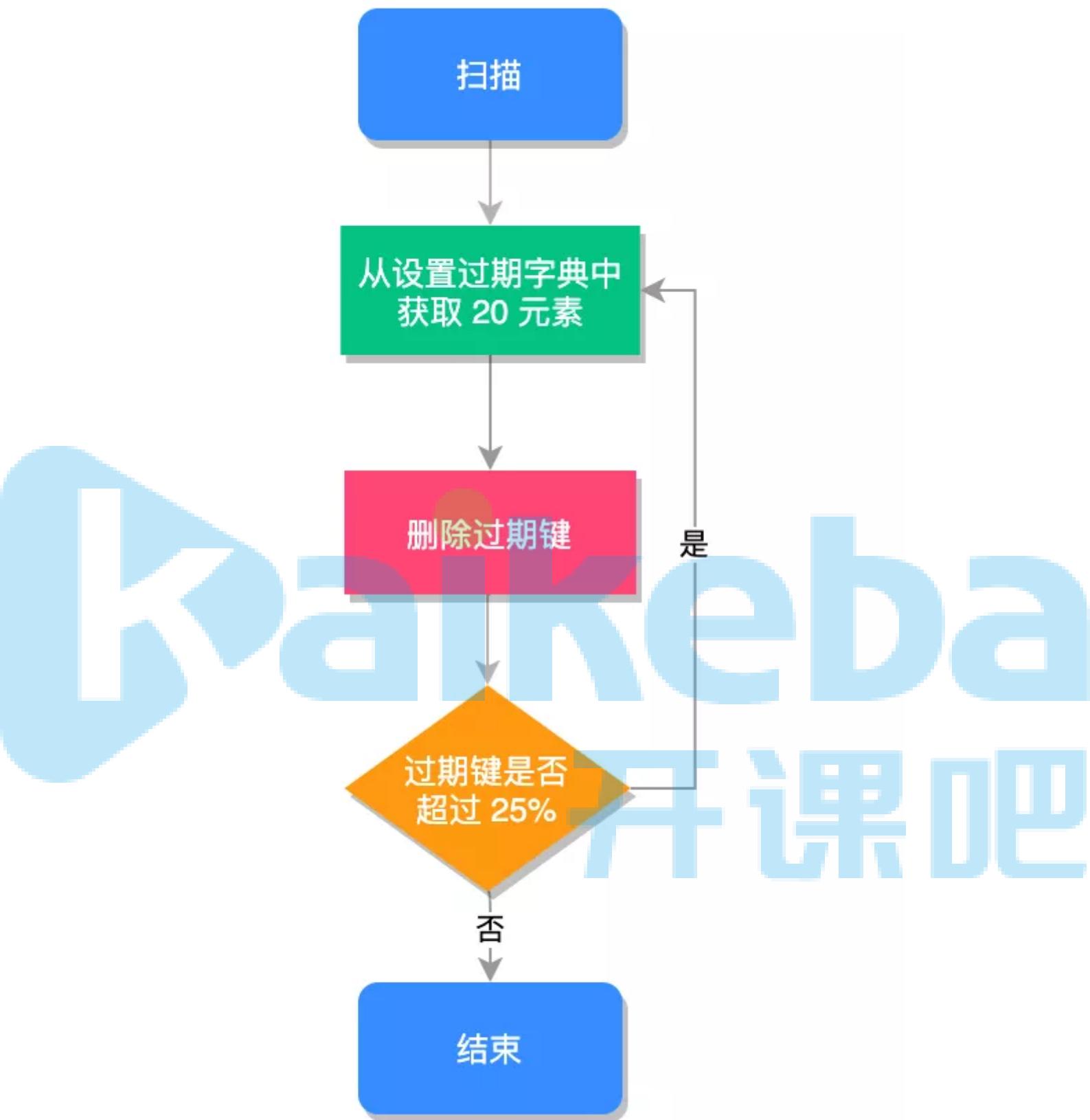
- `slowlog-log-slower-than`：用于设置慢查询的评定时间，也就是说超过此配置项的命令，将会被当成慢操作记录在慢查询日志中，它执行单位是微秒（1 秒等于 1000000 微秒）；
- `slowlog-max-len`：用来配置慢查询日志的最大记录数。

我们可以根据实际的业务情况进行相应的配置，其中慢日志是按照插入的顺序倒序存入慢查询日志中，我们可以使用 `slowlog get n` 来获取相关的慢查询日志，再找到这些慢查询对应的业务进行相关的优化。

避免大量数据同时失效

Redis 过期键值删除使用的是贪心策略，它每秒会进行 10 次过期扫描，此配置可在 redis.conf 进行配置，默认值是 `hz 10`，Redis 会随机抽取 20 个值，删除这 20 个键中过期的键，如果过期 key 的比例超过 25%，重复执行此流程，如下图所示：





如果在大型系统中有大量缓存在同一时间同时过期，那么会导致 Redis 循环多次持续扫描删除过期字典，直到过期字典中过期键值被删除的比较稀疏为止，而在整个执行过程会导致 Redis 的读写出现明显的卡顿，卡顿的另一种原因是内存管理器需要频繁回收内存页，因此也会消耗一定的 CPU。

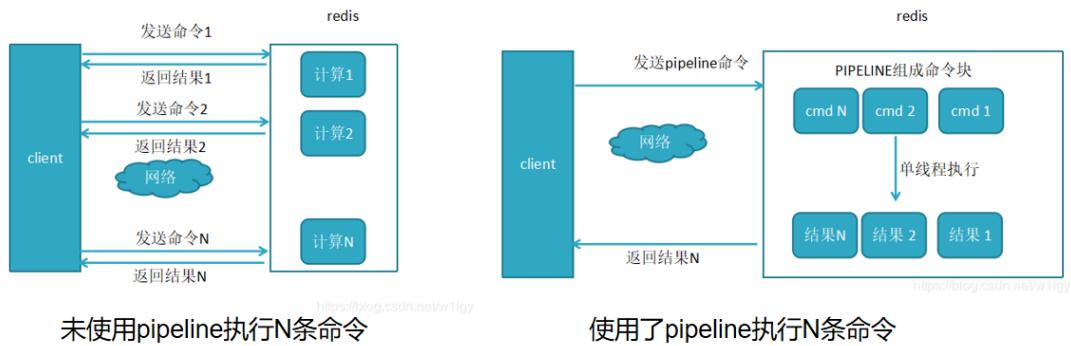
为了避免这种卡顿现象的产生，我们需要预防大量的缓存在同一时刻一起过期，就简单的解决方案就是在过期时间的基础上添加一个指定范围的随机数。

使用 Pipeline 批量操作数据

Pipeline (管道技术) 是客户端提供的一种批处理技术

可以批量执行一组指令，一次性返回全部结果，

可以减少频繁的请求应答



客户端使用优化

在客户端的使用上我们除了要尽量使用 Pipeline 的技术外，还需要注意要尽量使用 Redis 连接池，而不是频繁创建销毁 Redis 连接，这样就可以减少网络传输次数和减少了非必要调用指令。

```
import redis.clients.jedis.JedisPool;  
import redis.clients.jedis.JedisPoolConfig;
```

使用分布式架构来增加读写速度

Redis 分布式架构有重要的手段：

- 主从同步
- 哨兵模式
- Redis Cluster 集群

使用主从同步功能我们可以把写入放到主库上执行，把读功能转移到从服务上，因此就可以在单位时间内处理更多的请求，从而提升的 Redis 整体的运行速度。

而哨兵模式是对于主从功能的升级，但当主节点崩溃之后，无需人工干预就能自动恢复 Redis 的正常使用。

Redis Cluster 是 Redis 3.0 正式推出的，Redis 集群是通过将数据库分散存储到多个节点上来平衡各个节点的负载压力。

Redis Cluster 采用虚拟哈希槽分区，所有的键根据哈希函数映射到 0 ~ 16383 整数槽内，计算公式：
 $slot = CRC16(key) \& 16383$ ，每一个节点负责维护一部分槽以及槽所映射的键值数据。这样 Redis 就可以把读写压力从一台服务器，分散给多台服务器了，因此性能会有很大的提升。

Redis Cluster 应该是首选的实现方案，它可以把读写压力自动的分担给更多的服务器，并且拥有自动容灾的能力。

使用物理机而非虚拟机

在虚拟机中运行 Redis 服务器，因为和物理机共享一个物理网口，并且一台物理机可能有多个虚拟机在运行，因此在内存占用上和网络延迟方面都会有很糟糕的表现，我们可以通过 `./redis-cli --intrinsic-latency 100` 命令查看延迟时间，如果对 Redis 的性能有较高要求的话，应尽可能在物理机上直接部署 Redis 服务器。

禁用 THP 特性

Linux kernel 在 2.6.38 内核增加了 Transparent Huge Pages (THP) 特性，支持大内存页 2MB 分配，默认开启。

当开启了 THP 时，`fork` 的速度会变慢，`fork` 之后每个内存页从原来 4KB 变为 2MB，会大幅增加重写期间父进程内存消耗。同时每次写命令引起的复制内存页单位放大了 512 倍，会拖慢写操作的执行时间，导致大量写操作慢查询。例如简单的 `incr` 命令也会出现在慢查询中，因此 Redis 建议将此特性进行禁用，禁用方法如下：

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

为了使机器重启后 THP 配置依然生效，可以在 `/etc/rc.local` 中追加 `echo never > /sys/kernel/mm/transparent_hugepage/enabled`。

缓存常见问题

缓存雪崩

- 什么叫缓存雪崩？

1 当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统（比如DB）带来很大压力。

- 如何解决？

- 1: 在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。
- 2: 不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。
- 3: 做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期（此点为补充）

缓存击穿

- 什么叫缓存击穿？

- 1 对于一些设置了过期时间的key，如果这些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，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。
- 2 缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回写到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

- 如何解决？

- 1 使用redis的setnx互斥锁先进行判断，这样其他线程就处于等待状态，保证不会有大并发操作去操作数据库。

```
1 if(redis.setnx()==1){  
2     //先查询缓存  
3     //查询数据库  
4     //加入缓存  
5 }  
6  
7 }
```

缓存穿透

- 什么叫缓存穿透？

- 1 一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造成很大的压力。
- 2 也就是说，对不存在的key进行高并发访问，导致数据库压力瞬间增大，这就叫做【缓存穿透】。

- 如何解决？

- 1 对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert了之后清理缓存。
- 2 对一定不存在的key进行过滤。可以把所有的可能存在的key放到一个大的Bitmap中，查询时通过该bitmap过滤。（布隆表达式）

缓存预热

1. 直接写个缓存刷新页面，上线前手工操作一下
2. 数据量不大的时候，可以在项目启动的时候自动加载
3. 定时刷新缓存

缓存更新

自定义的缓存淘汰策略：

1. 定期去清理过期的缓存
2. 当有用户请求过来时，先判断这个请求用到的缓存是否过期，过期的话就去底层系统得到新数据进行缓存更新

缓存降级

当访问量出现剧增、服务出现问题（相应时间慢或者不响应）或非核心业务影响到核心流程的性能，还需要保证服务的可用性，即便有损服务。

方式：系统根据一些关键数据进行降级

配置开关实现人工降级

有些服务时无法降级（加入购物车，结算）

参考日志级别：一般：ex有些服务偶尔网络抖动或者服务正在上线超时，可以自定降级

警告：有些服务在一端时间内有波动（95%-100%），可以自定降级或人工降级，还有发送告警

错误：可利用率低于90%，redis连接池被打爆了，数据库连接池被打爆，或者访问量突然猛增到系统能承受的最大阈值，这时候根据情况自动降级或人工降级

严重错误：比如因为特殊原因数据错误了，需要紧急人工降级。

redis服务出问题了，不去查数据库，而是直接返回一个默认值（自定义一些随机值）

缓存双写一致性

一般来说，在读取缓存方面，我们都是先读取缓存，再读取数据库的。

但是，在更新缓存方面，我们是需要先更新缓存，再更新数据库？还是先更新数据库，再更新缓存？还是说有其他的方案？

先更新数据库再更新缓存（不建议使用）

- 操作步骤（线程A和线程B都对同一数据进行更新操作）：

1. 线程A更新了数据库
2. 线程B更新了数据库
3. 线程B更新了缓存
4. 线程A更新了缓存

- 问题1：脏读
- 问题2：浪费性能

先更新数据库再删除缓存

- 操作步骤（线程A更新、线程B读）

- 请求A进行写操作，删除缓存，此时A的写操作还没有执行完
- 请求B查询发现缓存不存在
- 请求B去数据库查询得到旧值
- 请求B将旧值写入缓存
- 请求A将新值写入数据库

- 解决方案1：延时双删策略，伪代码如下：

```
1 public** *void** write(**String** key, **Object** data){``  
2  
3     db.updateData(data);`  
4     redis.delKey(key);``  
5     Thread.sleep(1000);``  
6     redis.deleteKey(key);``  
7 }
```

- 解决方案2：使用消息队列

先删除缓存再更新数据库

- 操作步骤

1. 用户A删除缓存失败
2. 用户A成功更新了数据

或者

1. 用户A删除了缓存;
2. 用户B读取缓存，缓存不存在;
3. 用户B从数据库拿到旧数据;
4. 用户B更新了缓存;
5. 用户A更新了数据。

- 问题：脏数据
- 解决方案1：设置缓存有效时间（最简单）
- 解决方案2：使用消息队列：通过数据库的binlog来异步淘汰key，利用工具(canal)将binlog日志采集发送到MQ中，然后通过ACK机制确认处理删除缓存
- 要求“缓存+数据库”必须保持一致，读请求和写请求进行串行化，串到一个内存队列中 最好不要坐这种方案

Redis 常见面试问题

1. Redis 有哪些数据结构 8-9种 day01
2. 能说一下他们的特性，还有分别的使用场景么？

day01

String 缓存功能 计算器 共享Session

Hash :地理分析，属性简单的爆品

List: 粉丝列表，文章评论 Range 做高性能分页 简单的消息队列

Set: 共同好友

Zset :排行榜 热搜

BitMap: IP注册数，在线人数

HyperLogLog:

Geo:附近的人 或者地理位置

Stream：数据分析，即时通信，客户端扩展

3. 单机会有瓶颈，怎么解决这个瓶颈？

day02 day03

集群：主从，哨兵，cluster

4. 哪他们之间是如何进行数据交互的？Redis是如何进行持久化的？Redis数据都在内存中，一断电或重启不就没有了吗？

day02

全量同步和增量同步，基于RDB

RDB

AOF

混合持久化

5. 哪你是如何选择持久方式的？

day02

版本问题：4.0之前 做内存数据库：RDB+AOF 纯缓存：RDB

4.0 做混合持久化

6. Redis还有其他保证集群高可用的方式吗？

day03

哨兵

7. 数据传输的时候网络断了或者服务器断了，怎么办？

day 03

哨兵

8. 能说一下Redis的内存淘汰机制么？

day02

主动删除 延迟延迟删除

9. 如果的如果，定期没删，我也没查询，那可咋整

设置最大内存，缓存淘汰策略 day02

1、读从写主，客户端怎么知道两种地址 ---主从配置，需要客户端配置 哨兵-内网DNS 在配置文件写

VIP-动态IP指定，客户端封装哨兵，交给哨兵处理 day03

10. 哨兵机制的原理是什么？

day03

定时监控，故障判定（SDOWN, ODOWN），自动故障迁移

11. 哨兵组件的主要功能是什么？

监控

告警

自动故障迁移 day03

12. Redis的事务原理是什么？

普通版：说一下Redis的事务命令并解释命令 day03

高调版：我们生产上一直用的redis cluster集群，事务是不起作用的，所以我们根本不用redis事务

13. Redis事务为什么是“弱”事务？

第一，不是原子性的

第二，没有回滚操作 day03

14. Redis为什么没有回滚操作？

1. 大多数错误都是 操作或指令错误

2. 为了性能 day03

15. 在Redis中整合lua有几种方式，你认为哪种更好，为什么？

三种方式 最好的是客户端调用 eval 利用原子性，做秒杀 day03

16. lua如何解决Redis并发问题？

利用lua原子性 day03

17. 介绍Redis下的分布式锁实现原理、优势劣势和使用场景

获取锁 和释放锁

优势：性能高

缺点：考虑原子性，排他性，延时性等 AP

要求强一致 不能用 day03

18. Redis-Cluster和Redis主从+哨兵的区别，你认为哪个更好，为什么。

Redis-Cluster 不仅可拓展 而且高可用

主从+哨兵 主要解决的是高可用问题 单点故障（共享session服务器）一般用主从+哨兵 day03

19. 什么情况下会造成缓存穿透，如何解决？

redis 和数据库都没有数据 又有大量访问 bitmap 随机数。 day04

20. 什么情况下会造成缓存雪崩，如何解决？

缓存同时失效并有大量访问 设置随机数时间，不一起失效；缓存降级；多级缓存置于前方
(CDN,Nginx)

21. 什么是缓存击穿，如何解决

热点数据失效 热点设置为持久化数据 多级缓存置于前方 (CDN,Nginx)

22. 什么情况下会造成数据库与Redis缓存数据不一致，如何解决？

day04

数据库双存储

最终一致性：设置缓存时间（最简单），使用消息队列

强一致性：读写串行化，不建议使用，如使用请加机器

23. 那你了解的最经典的KV，DB读写模式什么样

读的时候 先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应

更新的时候，**先更新数据库，然后再删除缓存**

24. 为什么是删除缓存，而不是更新缓存？

lazy 计算思想：不要每次都做重新的复杂计算，不管它会不会用到，都是让它用到的时候再重新计算

Redis 和Memcache的区别

Redis 支持复杂的数据结构

Redis 拥有更多的数据结构，能支持更丰富的操作

Reis 原生支持集群

性能对比：存储性能

R 只使用单核 而M 使用多核 平均到每个核上 redis存储性能更高， 100K 以上的数据M性能是高于R。



课前准备

- 准备redis安装包

课堂主题

Redis单机安装及数据类型分析、消息模式及事务、Java客户端的使用

课堂目标

- Redis是什么？
- Redis的主要应用场景有哪些？
- Redis单机安装要掌握？
- Redis数据类型有哪些？
- Redis的数据类型各自的使用场景及注意事项是什么？
- Redis的消息模式是如何实现的？
- Redis的事务是如何实现的？
- Java的客户端Jedis是如何使用的？

知识要点

课前准备

课堂主题

课堂目标

知识要点

Redis介绍

什么是Redis？

Redis官网

什么是NoSQL？

NoSQL数据库分类

Redis发展历史

Redis应用场景

Redis单机版安装配置

Redis下载

Redis安装环境

Redis安装

Redis启动

前端启动

后端启动（守护进程启动）

后端启动的关闭方式

其他命令说明

Redis客户端

Redis命令行客户端

Java客户端Jedis

Jedis介绍

添加依赖
单实例连接
连接池连接
连接redis集群
Jedis整合spring

Redis数据类型

string类型

命令
 赋值
 取值
 取值并赋值
 数值增减
 注意事项
 递增数字
 增加指定的整数
 递减数值
 减少指定的整数
 仅当不存在时赋值
 其它命令
 向尾部追加值
 获取字符串长度
 同时设置/获取多个键值

应用场景之自增主键

hash类型

hash类型介绍
命令
 赋值
 设置一个字段值
 设置多个字段值
 当字段不存在时赋值
 取值
 获取一个字段值
 获取多个字段值
 获取所有字段值
 删除字段
 增加数字
 其它命令(自学)
 判断字段是否存在
 只获取字段名或字段值
 获取字段数量
 获取所有字段

string类型和hash类型的区别

应用之存储商品信息

list类型

ArrayList与LinkedList的区别

list类型介绍
命令
 LPUSH/RPUSH
 LRANGE
 LPOP/RPOP
 LLEN
 其它命令(自学)
 LREM

LINDEX
LTRIM
LINSERT
RPOPLPUSH

应用之商品评论列表

set类型

set类型介绍

命令

SADD/SREM
SMEMBERS
SISMEMBER

集合运算命令

SDIFF
SINTER
SUNION

其它命令(自学)

SCARD
SPOP

zset类型 (sortedset)

zset介绍

命令

ZADD
ZRANGE/ZREVRANGE
ZSCORE
ZREM

其它命令(自学)

ZRANGEBYSCORE
ZINCRBY
ZCARD
ZCOUNT
ZREMRANGEBYRANK
ZREMRANGEBYSCORE
ZRANK/ZREVRANK

应用之商品销售排行榜

通用命令

keys
del
exists
expire (重点)
rename
type

Redis消息模式

队列模式
发布订阅模式

Redis事务

Redis事务介绍

事务命令

MULTI
EXEC
DISCARD
WATCH
UNWATCH

事务演示

[事务失败处理](#)

[扩展点](#)

[本课总结](#)

[布置作业](#)

[课后互动问答](#)

[下节预告](#)

Redis介绍

什么是Redis？

- Redis是用C语言开发的一个开源的高性能键值对(key-value)内存数据库，它是一种NoSQL数据库。
- 它是【单进程单线程】的内存数据库，所以说不存在线程安全问题。
- 它可以支持并发10W QPS，所以说性能非常优秀。之所以单进程单线程性能还这么好，是因为底层采用了【IO多路复用(NIO思想)】
- 相比Memcache这种专业缓存技术，它有更优秀的读写性能，及丰富的数据类型。
- 它提供了五种数据类型来存储【值】：字符串类型(string)、散列类型(hash)、列表类型(list)、集合类型(set)、有序集合类型(sortedset、zset)

Redis官网

- 官网地址：<http://redis.io/>
- 中文官网地址：<http://www.redis.cn/>
- 下载地址：<http://download.redis.io/releases/>

什么是NoSQL？

- NoSQL，即 Not-Only SQL（不仅仅是SQL），泛指非关系型的数据库。
- 什么是关系型数据库？数据结构是一种有行有列的数据库
- NoSQL数据库是为了解决高并发、高可用、高扩展、大数据存储问题而产生的数据库解决方案。
- NoSQL可以作为关系型数据库的良好补充，但是不能替代关系型数据库。

MySQL(关系型数据库) ----> NoSQL ---> NewSQL(TiDB)

NoSQL数据库分类

- 键值(Key-Value)存储数据库

相关产品：[Tokyo Cabinet/Tyrant](#)、[Redis](#)、 [Voldemort](#)、[Berkeley DB](#)

典型应用：内容缓存，主要用于处理大量数据的高访问负载。

数据模型：一系列键值对

优势：快速查询

劣势：存储的数据缺少结构化

- 列存储数据库

相关产品：`Cassandra`，`HBase`，`Riak`

典型应用：分布式的文件系统

数据模型：以列簇式存储，将同一列数据存在一起

优势：查找速度快，可扩展性强，更容易进行分布式扩展

劣势：功能相对局限

- 文档型数据库

相关产品：`CouchDB`、`MongoDB`

典型应用：`web` 应用（与 `key-value` 类似，`value` 是结构化的）

数据模型：一系列键值对

优势：数据结构要求不严格

劣势：

- 图形(`Graph`)数据库

相关数据库：`Neo4J`、`InfoGrid`、`Infinite Graph`

典型应用：社交网络

数据模型：图结构

优势：利用图结构相关算法。

劣势：需要对整个图做计算才能得出结果，不容易做分布式的集群方案。

Redis发展历史

2008年，意大利的一家创业公司 Merzia 推出了一款基于 MySQL 的网站实时统计系统 `LLOOGG`，然而没过多久该公司的创始人 `Salvatore Sanfilippo` 便对 MySQL 的性能感到失望，于是他决定亲自为 `LLOOGG` 量身定做一个数据库，并于2009年开发完成，这个数据库就是 `Redis`。

不过 `Salvatore Sanfilippo` 并不满足只将 `Redis` 用于 `LLOOGG` 这一款产品，而是希望更多的人使用它，于是在同一年 `Salvatore Sanfilippo` 将 `Redis` 开源发布，并开始和 `Redis` 的另一名主要的代码贡献者 `Pieter Noordhuis` 一起继续着 `Redis` 的开发，直到今天。

`Salvatore Sanfilippo` 自己也没有想到，短短的几年时间，`Redis` 就拥有了庞大的用户群体。`Hacker News` 在2012年发布了一份数据库的使用情况调查，结果显示有近12%的公司在使用`Redis`。国内如新浪微博、街旁网、知乎网，国外如 `Github`、`Stack Overflow`、`Flickr` 等都是 `Redis` 的用户。

`VMware` 公司从2010年开始赞助 `Redis` 的开发，`Salvatore Sanfilippo` 和 `Pieter Noordhuis` 也分别在3月和5月加入 `VMware`，全职开发 `Redis`。

Redis应用场景

- 内存数据库（登录信息、购物车信息、用户浏览记录等）
- 缓存服务器（商品数据、广告数据等等）（最多使用）
- 解决分布式集群架构中的 session 分离问题（session 共享）
- 任务队列（秒杀、抢购、12306等等）
- 分布式锁的实现
- 支持发布订阅的消息模式
- 应用排行榜(有序集合)
- 网站访问统计
- 数据过期处理（可以精确到毫秒）

Redis单机版安装配置

Redis下载

- 官网地址：<http://redis.io/>
- 中文官网地址：<http://www.redis.cn/>
- 下载地址：<http://download.redis.io/releases/>

Redis安装环境

Redis 没有官方的 windows 版本，所以建议在 Linux 系统上安装运行，我们使用 centos 7 (Linux 操作系统的一个系列) 作为安装环境。

Windows版本

Redis没有官方的Windows版本，但是微软开源技术团队 (Microsoft Open Tech group) 开发和维护着这个Win64的版本。更多信息请参考[这里](#)。

Redis安装

第一步：安装 c 语言需要的 GCC 环境

```
1 | yum install -y gcc-c++  
2 | yum install -y wget
```

第二步：下载并解压缩 Redis 源码压缩包

```
1 | wget http://download.redis.io/releases/redis-5.0.4.tar.gz  
2 | tar -zxf redis-5.0.4.tar.gz
```

第三步：编译 Redis 源码，进入 redis-3.2.9 目录，执行编译命令

```
1 | cd redis-5.0.4  
2 | make
```

第四步：安装 Redis，需要通过 PREFIX 指定安装路径

```
1 | make install PREFIX=/kkb/server/redis
```

Redis启动

前端启动

- 启动命令：`redis-server`，直接运行 bin/redis-server 将以前端模式启动

```
1 | ./redis-server
```

- 关闭命令：`ctrl+c`
- 启动缺点：客户端窗口关闭则 redis-server 程序结束，不推荐使用此方法
- 启动图例：

The terminal output shows the Redis configuration file being read and the server starting:

```
***** GENERAL *****  
# By default Redis does not run as a daemon, use 'yes' if you need it.  
# Note that Redis will write a pid file in /var/run/redis.pid when daemonized.  
daemonize no  
# When running daemonized, Redis writes a pid file in /var/run/redis.pid by  
# default. You can specify a custom pid file location here.  
pidfile /var/run/redis.pid  
# Accept connections on the specified port, default is 6379.  
# If port 0 is specified Redis will not listen on a TCP socket.  
port 6379  
#./redis.conf" 938L 41403C written  
[root@server01 bin]# ./redis-server  
2583:M 17 Apr 17:29:28.541 * Warning: no config file specified, using the default config. In order to specify a config file use ./redis-server /path/to/redis.conf  
2583:M 17 Apr 17:29:28.545 * Increased maximum number of open files to 10032 (it was originally set to 1024).  
2583:M 17 Apr 17:29:28.590 # warning: 32 bit instance detected but no memory limit set. Setting 3 GB maxmemory limit with 'noeviction' policy now.  
  
Redis 3.0.0 (00000000/0) 32 bit  
Running in standalone mode  
Port: 6379  
PID: 2583  
  
http://redis.io  
  
2583:M 17 Apr 17:29:28.599 # Server started, Redis version 3.0.0  
2583:M 17 Apr 17:29:28.600 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue  
dd 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect  
2583:M 17 Apr 17:29:28.601 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is 128.  
2583:M 17 Apr 17:29:28.601 * The server is now ready to accept connections on port 6379
```

后端启动（守护进程启动）

- 第一步：拷贝 redis-5.0.4/redis.conf 配置文件到 Redis 安装目录的 bin 目录

```
1 | cp /root/redis-5.0.4/redis.conf /kkb/server/redis/bin/
```

- 第二步：修改 redis.conf

```
1 | vim redis.conf
```

```
1 # 将`daemonize`由`no`改为`yes`
2 daemonize yes
3
4 # 默认绑定的是回环地址，默认不能被其他机器访问
5 # bind 127.0.0.1
6
7 # 是否开启保护模式，由yes该为no
8 protected-mode no
```

- 第三步：启动服务

```
1 | ./redis-server redis.conf
```

后端启动的关闭方式

```
1 | ./redis-cli shutdown
```

其他命令说明

- `redis-server` : 启动 redis 服务
- `redis-cli` : 进入 redis 命令客户端
- `redis-benchmark` : 性能测试的工具
- `redis-check-aof` : aof 文件进行检查的工具
- `redis-check-dump` : rdb 文件进行检查的工具
- `redis-sentinel` : 启动哨兵监控服务

Redis客户端

Redis命令行客户端



```
-r--r--r--. 1 root root 18 9月 30 10:00 dump.rdb
-rw-r--r--. 1 root root 4171542 9月 30 09:49 redis-benchmark
-rw-r--r--. 1 root root 16415 9月 30 09:49 redis-check-aof
-rw-r--r--. 1 root root 37647 9月 30 09:49 redis-check-dump
-rw-r--r--. 1 root root 4260308 9月 30 09:49 redis-cli
-rw-r--r--. 1 root root 41404 9月 30 09:57 redis.conf
1rw-rwxrwx. 1 root root 12 9月 30 09:49 redis-sentinel -> redis-server
-rw-r--r--. 1 root root 5690105 9月 30 09:49 redis-server
[root@localhost-0723 bin]# ps aux|grep redis
```

- 命令格式

```
1 | ./redis-cli -h 127.0.0.1 -p 6379
```

- 参数说明

```
1 | -h : redis服务器的ip地址
2 | -p : redis实例的端口号
```

- 默认方式

如果不指定主机和端口也可以

- 默认主机地址是127.0.0.1
- 默认端口是6379

```
1 | ./redis-cli
```

Java客户端Jedis

Jedis介绍

Redis不仅使用命令来操作，而且可以使用程序客户端操作。现在基本上主流的语言都有客户端支持，比如java、C、C#、C++、php、Node.js、Go等。

在官方网站里列一些Java的客户端，有Jedis、Redisson、Jredis、JDBC-Redis、等其中官方推荐使用Jedis和Redisson。在企业中用的最多的就是Jedis，下面我们就重点学习下Jedis。

Jedis同样也是托管在github上，地址：<https://github.com/xetorthio/jedis>

添加依赖

```
1 <dependencies>
2   <dependency>
3     <groupId>redis.clients</groupId>
4     <artifactId>jedis</artifactId>
5     <version>2.9.0</version>
6   </dependency>
7
8   <dependency>
9     <groupId>org.springframework</groupId>
10    <artifactId>spring-context</artifactId>
11    <version>5.0.7.RELEASE</version>
12  </dependency>
13
14  <dependency>
15    <groupId>org.springframework</groupId>
16    <artifactId>spring-test</artifactId>
17    <version>5.0.7.RELEASE</version>
18  </dependency>
19
20  <!-- 单元测试JUnit -->
21  <dependency>
22    <groupId>junit</groupId>
23    <artifactId>junit</artifactId>
24    <version>4.12</version>
25  </dependency>
26 </dependencies>
27 <build>
28   <plugins>
```

```
29      <!-- 配置Maven的JDK编译级别 -->
30      <plugin>
31          <groupId>org.apache.maven.plugins</groupId>
32          <artifactId>maven-compiler-plugin</artifactId>
33          <version>3.2</version>
34          <configuration>
35              <source>1.8</source>
36              <target>1.8</target>
37              <encoding>UTF-8</encoding>
38          </configuration>
39      </plugin>
40  </plugins>
41 </build>
42
```

单实例连接

```
1  @Test
2  public void testJedis() {
3      //创建一个Jedis的连接
4      Jedis jedis = new Jedis("127.0.0.1", 6379);
5      //执行redis命令
6      jedis.set("mytest", "hello world, this is jedis client!");
7      //从redis中取值
8      String result = jedis.get("mytest");
9      //打印结果
10     System.out.println(result);
11     //关闭连接
12     jedis.close();
13
14 }
15
```

连接池连接

```
1  @Test
2  public void testJedisPool() {
3      //创建一连接池对象
4      JedisPool jedisPool = new JedisPool("127.0.0.1", 6379);
5      //从连接池中获得连接
6      Jedis jedis = jedisPool.getResource();
7      String result = jedis.get("mytest") ;
8      System.out.println(result);
9      //关闭连接
10     jedis.close();
11
12     //关闭连接池
13     jedisPool.close();
14 }
15
```

连接redis集群

创建JedisCluster类连接redis集群。

```
1  @Test
2  public void testJedisCluster() throws Exception {
3      //创建一连接，JedisCluster对象，在系统中是单例存在
4      Set<HostAndPort> nodes = new HashSet<>();
5      nodes.add(new HostAndPort("192.168.242.129", 7001));
6      nodes.add(new HostAndPort("192.168.242.129", 7002));
7      nodes.add(new HostAndPort("192.168.242.129", 7003));
8      nodes.add(new HostAndPort("192.168.242.129", 7004));
9      nodes.add(new HostAndPort("192.168.242.129", 7005));
10     nodes.add(new HostAndPort("192.168.242.129", 7006));
11     JedisCluster cluster = new JedisCluster(nodes);
12     //执行JedisCluster对象中的方法，方法和redis一一对应。
13     cluster.set("cluster-test", "my jedis cluster test");
14     String result = cluster.get("cluster-test");
15     System.out.println(result);
16     //程序结束时需要关闭JedisCluster对象
17     cluster.close();
18 }
19 }
```

Jedis整合spring

配置spring配置文件applicationContext.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5          http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7      <!-- 连接池配置 -->
8      <bean id="jedisPoolConfig"
9          class="redis.clients.jedis.JedisPoolConfig">
10         <!-- 最大连接数 -->
11         <property name="maxTotal" value="30" />
12         <!-- 最大空闲连接数 -->
13         <property name="maxIdle" value="10" />
14         <!-- 每次释放连接的最大数目 -->
15         <property name="numTestsPerEvictionRun" value="1024" />
16         <!-- 释放连接的扫描间隔(毫秒) -->
17         <property name="timeBetweenEvictionRunsMillis" value="30000" />
18         <!-- 连接最小空闲时间 -->
19         <property name="minEvictableIdleTimeMillis" value="1800000" />
20         <!-- 连接空闲多久后释放，当空闲时间>该值且空闲连接>最大空闲连接数时直接释放 -->
21         <property name="softMinEvictableIdleTimeMillis" value="10000" />
22         <!-- 获取连接时的最大等待毫秒数，小于零：阻塞不确定的时间，默认-1 -->
```

```
23     <property name="maxWaitMillis" value="1500" />
24     <!-- 在获取连接的时候检查有效性, 默认false -->
25     <property name="testOnBorrow" value="true" />
26     <!-- 在空闲时检查有效性, 默认false -->
27     <property name="testWhileIdle" value="true" />
28     <!-- 连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true -->
29     <property name="blockWhenExhausted" value="false" />
30 </bean>
31
32     <!-- redis单机 通过连接池 -->
33     <bean id="jedisPool" class="redis.clients.jedis.JedisPool"
34         destroy-method="close">
35         <constructor-arg name="poolConfig"
36             ref="jedisPoolConfig" />
37         <constructor-arg name="host" value="192.168.10.135" />
38         <constructor-arg name="port" value="6379" />
39     </bean>
40
41     <!-- redis集群 -->
42     <bean id="jedisCluster" class="redis.clients.jedis.JedisCluster">
43         <constructor-arg index="0">
44             <set>
45                 <bean class="redis.clients.jedis.HostAndPort">
46                     <constructor-arg index="0" value="192.168.10.135"></constructor-
47 arg>
48                 <constructor-arg index="1" value="7001"></constructor-arg>
49             </bean>
50             <bean class="redis.clients.jedis.HostAndPort">
51                 <constructor-arg index="0" value="192.168.10.135"></constructor-
52 arg>
53                 <constructor-arg index="1" value="7002"></constructor-arg>
54             </bean>
55             <bean class="redis.clients.jedis.HostAndPort">
56                 <constructor-arg index="0" value="192.168.10.135"></constructor-
57 arg>
58                 <constructor-arg index="1" value="7003"></constructor-arg>
59             </bean>
60             <bean class="redis.clients.jedis.HostAndPort">
61                 <constructor-arg index="0" value="192.168.10.135"></constructor-
62 arg>
63                 <constructor-arg index="1" value="7004"></constructor-arg>
64             </bean>
65             <bean class="redis.clients.jedis.HostAndPort">
66                 <constructor-arg index="0" value="192.168.10.135"></constructor-
67 arg>
68                 <constructor-arg index="1" value="7005"></constructor-arg>
69             </bean>
</set>
```

```
70     </constructor-arg>
71     <constructor-arg index="1" ref="jedisPoolConfig"></constructor-arg>
72   </bean>
73 </beans>
```

测试代码

```
1 package com.kkb.redis.test;
2
3 import javax.annotation.Resource;
4
5 import org.junit.Test;
6 import org.junit.runner.RunWith;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.test.context.ContextConfiguration;
9 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
10
11 import redis.clients.jedis.Jedis;
12 import redis.clients.jedis.JedisCluster;
13 import redis.clients.jedis.JedisPool;
14
15 @RunWith(SpringJUnit4ClassRunner.class)
16 @ContextConfiguration(locations = "classpath:application.xml")
17 public class TestJedis2 {
18
19     @Autowired
20     private JedisPool jedisPool;
21
22     @Resource
23     private JedisCluster cluster;
24
25     @Test
26     public void testJedisPool() {
27         // 从连接池中获得连接
28         Jedis jedis = jedisPool.getResource();
29         String result = jedis.get("mytest");
30         System.out.println(result);
31         // 关闭连接
32         jedis.close();
33     }
34
35     @Test
36     public void testJedisCluster() throws Exception {
37         // 执行JedisCluster对象中的方法，方法和redis一一对应。
38         cluster.set("cluster-test", "my jedis cluster test");
39         String result = cluster.get("cluster-test");
40         System.out.println(result);
41     }
42 }
43 }
44 }
```

Redis数据类型

官方命令大全网址：<http://www.redis.cn/commands.html>

Redis 中存储数据是通过 key-value 格式存储数据的，其中 value 可以定义五种数据类型：

- String (字符类型)
- Hash (散列类型)
- List (列表类型)
- Set (集合类型)
- SortedSet (有序集合类型，简称zset)

注意：在 redis 中的命令语句中，命令是忽略大小写的，而 key 是不忽略大小写的。

string类型

命令

赋值

- 语法：

```
1 | SET key value
```

- 示例：

```
1 | 127.0.0.1:6379> set test 123
2 | OK
```

取值

- 语法：

```
1 | GET key
```

```
127.0.0.1:6379> get test "123"
```

- 示例：

```
1 | 127.0.0.1:6379> get test
2 | "123"
```

取值并赋值

语法：

```
1 | GETSET key value
```

示例：

```
1 | 127.0.0.1:6379> getset s2 222
2 | "111"
3 | 127.0.0.1:6379> get s2
4 | "222"
```

数值增减

注意事项

- ```
1 | 1、当value为整数数据时，才能使用以下命令操作数值的增减。
2 | 2、数值递增都是【原子】操作。
3 | 3、redis中的每一个单独的命令都是原子性操作。当多个命令一起执行的时候，就不能保证原子性，不过我们可以使用事务和lua脚本来保证这一点。
```

非原子性操作示例：

```
1 | int i = 1;
2 | i++;
3 | System.out.println(i)
```

### 递增数字

- 语法 ( increment ) :

```
1 | INCR key
```

- 示例：

```
1 | 127.0.0.1:6379> incr num
2 | (integer) 1
3 | 127.0.0.1:6379> incr num
4 | (integer) 2
5 | 127.0.0.1:6379> incr num
6 | (integer) 3
```

### 增加指定的整数

- 语法：

```
1 | INCRBY key increment
```

- 示例：

```
1 | 127.0.0.1:6379> incrby num 2
2 | (integer) 5
3 | 127.0.0.1:6379> incrby num 2
4 | (integer) 7
5 | 127.0.0.1:6379> incrby num 2
6 | (integer) 9
```

## 递增数值

- 语法：

```
1 | DECR key
```

- 示例：

```
1 | 127.0.0.1:6379> incr num
2 | (integer) 1
3 | 127.0.0.1:6379> incr num
4 | (integer) 2
5 | 127.0.0.1:6379> incr num
6 | (integer) 3
```

## 减少指定的整数

- 语法：

```
1 | DECRBY key decrement
```

- 示例

```
1 | 127.0.0.1:6379> decr num
2 | (integer) 6
3 | 127.0.0.1:6379> decr num
4 | (integer) 5
5 | 127.0.0.1:6379> decrby num 3
6 | (integer) 2
7 | 127.0.0.1:6379> decrby num 3
8 | (integer) -1
9 |
```

## 仅当不存在时赋值

使用该命令可以实现【分布式锁】的功能，后续讲解！！！

- 语法：

```
1 | setnx key value
```

- 示例：

```
1 | redis> EXISTS job # job 不存在
2 | (integer) 0
3 | redis> SETNX job "programmer" # job 设置成功
4 | (integer) 1
5 | redis> SETNX job "code-farmer" # 尝试覆盖 job , 失败
6 | (integer) 0
7 | redis> GET job # 没有被覆盖
8 | "programmer"
```

## 其它命令

### 向尾部追加值

APPEND 命令，向键值的末尾追加 value。

如果键不存在则将该键的值设置为 value，即相当于 SET key value。返回值是追加后字符串的总长度。

- 语法：

```
1 | APPEND key value
```

- 示例：

```
1 | 127.0.0.1:6379> set str hello
2 | OK
3 | 127.0.0.1:6379> append str " world!"
4 | (integer) 12
5 | 127.0.0.1:6379> get str
6 | "hello world!"
```

### 获取字符串长度

STRLEN 命令，返回键值的长度，如果键不存在则返回0。

- 语法：

```
1 | STRLEN key
```

- 示例：

```
1 127.0.0.1:6379> strlen str
2 (integer) 0
3 127.0.0.1:6379> set str hello
4 OK
5 127.0.0.1:6379> strlen str
6 (integer) 5
```

## 同时设置/获取多个键值

- 语法：

```
1 MSET key value [key value ...]
2
3 MGET key [key ...]
```

- 示例：

```
1 127.0.0.1:6379> mset k1 v1 k2 v2 k3 v3
2 OK
3 127.0.0.1:6379> get k1
4 "v1"
5 127.0.0.1:6379> mget k1 k3
6 1) "v1"
7 2) "v3"
```

## 应用场景之自增主键

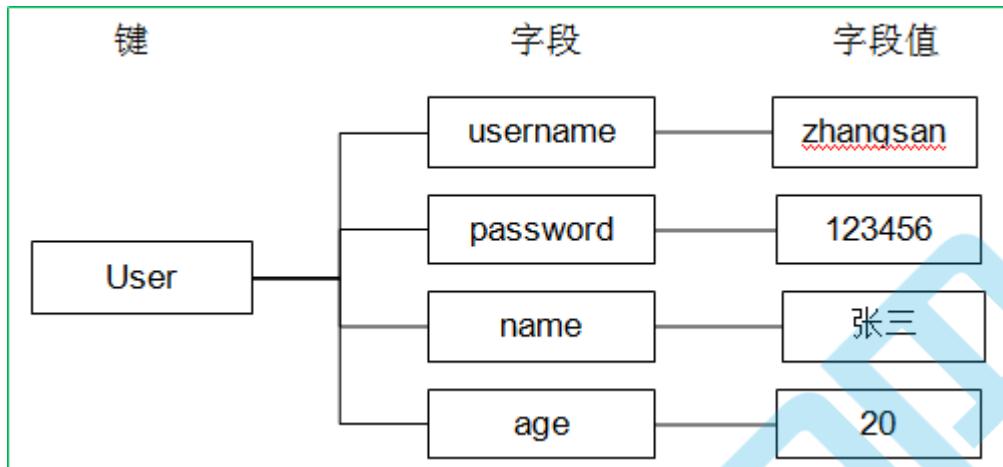
- 需求：商品编号、订单号采用 INCR 命令生成。
- 设计：key 命名要有一定的设计
- 实现：定义商品编号 key : items:id

```
1 192.168.101.3:7003> INCR items:id
2 (integer) 2
3 192.168.101.3:7003> INCR items:id
4 (integer) 3
```

## hash类型

### hash类型介绍

hash 类型也叫散列类型，它提供了字段和字段值的映射。字段值只能是字符串类型，不支持散列类型、集合类型等其它类型。如下：



## 命令

### 赋值

HSET 命令不区分插入和更新操作，当执行插入操作时 HSET 命令返回 1，当执行更新操作时返回 0。

#### 设置一个字段值

- 语法：

```
1 | HSET key field value
```

- 示例：

```
1 | 127.0.0.1:6379> hset user username zhangsan
2 | (integer) 1
```

#### 设置多个字段值

- 语法：

```
1 | HMSET key field value [field value ...]
```

- 示例：

```
1 | 127.0.0.1:6379> hmset user age 20 username lisi
2 | OK
```

#### 当字段不存在时赋值

类似 HSET，区别在于如果字段存在，该命令不执行任何操作

- 语法：

```
1 | HSETNX key field value
```

- 示例：

```
1 | 127.0.0.1:6379> hsetnx user age 30 # 如果user中没有age字段则设置age值为30，否则不做任何操作
2 | (integer) 0
3 |
```

## 取值

获取一个字段值

- 语法：

```
1 | HGET key field
```

- 示例：

```
1 | 127.0.0.1:6379> hget user username
2 | "zhangsan"
```

获取多个字段值

- 语法：

```
1 | HMGET key field [field ...]
```

- 示例：

```
1 | 127.0.0.1:6379> hmget user age username
2 | 1) "20"
3 | 2) "lisi"
```

获取所有字段值

- 语法：

```
1 | HGETALL key
```

- 示例：

```
1 | 127.0.0.1:6379> hgetall user
2 | 1) "age"
3 | 2) "20"
4 | 3) "username"
5 | 4) "lisi"
```

## 删除字段

可以删除一个或多个字段，返回值是被删除的字段个数

- 语法：

```
1 | HDEL key field [field ...]
```

- 示例：

```
1 | 127.0.0.1:6379> hdel user age
2 | (integer) 1
3 | 127.0.0.1:6379> hdel user age name
4 | (integer) 0
5 | 127.0.0.1:6379> hdel user age username
6 | (integer) 1
```

## 增加数字

- 语法：

```
1 | HINCRBY key field increment
```

- 示例：

```
1 | 127.0.0.1:6379> hincrby user age 2 # 将用户的年龄加2
2 | (integer) 22
3 | 127.0.0.1:6379> hget user age # 获取用户的年龄
4 | "22"
```

## 其它命令(自学)

### 判断字段是否存在

- 语法：

```
1 | HEXISTS key field
```

- 示例：

```
1 | 127.0.0.1:6379> hexists user age 查看user中是否有age字段
2 | (integer) 1
3 | 127.0.0.1:6379> hexists user name 查看user中是否有name字段
4 | (integer) 0
```

## 只获取字段名或字段值

- 语法：

```
1 | HKEYS key
2 | HVALS key
```

- 示例：

```
1 | 127.0.0.1:6379> hmset user age 20 name lisi
2 | OK
3 | 127.0.0.1:6379> hkeys user
4 | 1) "age"
5 | 2) "name"
6 | 127.0.0.1:6379> hvals user
7 | 1) "20"
8 | 2) "lisi"
```

## 获取字段数量

- 语法：

```
1 | HLEN key
```

- 示例：

```
1 | 127.0.0.1:6379> hlen user
2 | (integer) 2
```

## 获取所有字段

获得 hash 的所有信息，包括 key 和 value

- 语法：

```
1 | hgetall key
```

## string类型和hash类型的区别

hash类型适合存储那些对象数据，特别是对象属性经常发生【增删改】操作的数据。string类型也可以存储对象数据，将java对象转成json字符串进行存储，这种存储适合【查询】操作。

## 应用之存储商品信息

- 商品信息字段

```
1 | 【商品id、商品名称、商品描述、商品库存、商品好评】
```

- 定义商品信息的key

```
1 | 商品ID为1001的信息在 Redis中的key为 : [items:1001]
```

- 存储商品信息

```
1 | 192.168.101.3:7003> HMSET items:1001 id 3 name apple price 999.9
2 | OK
3 |
```

- 获取商品信息

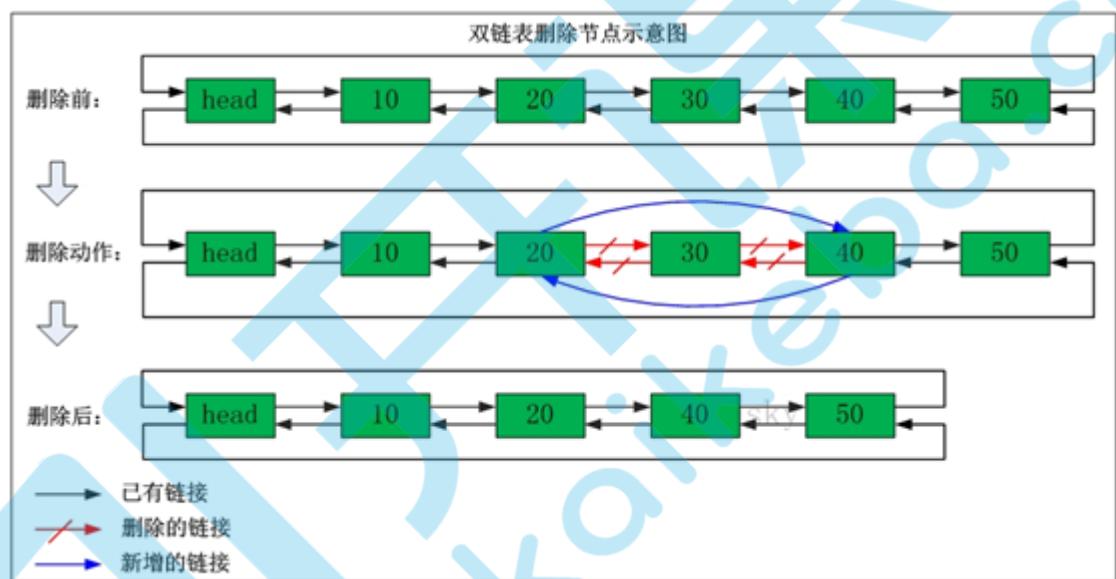
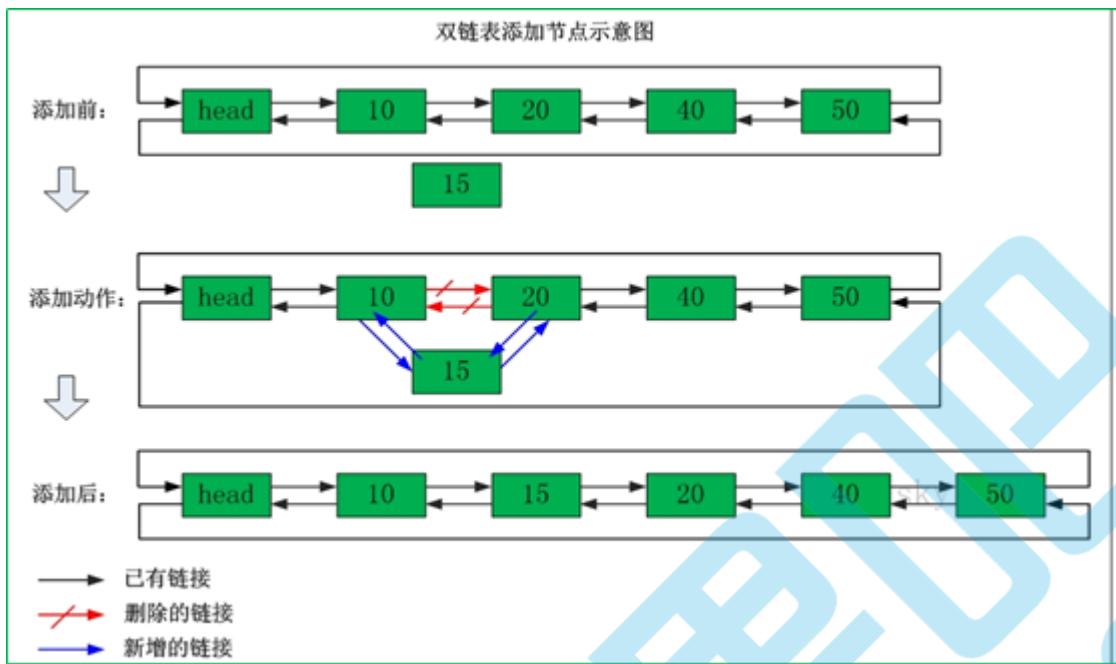
```
1 | 192.168.101.3:7003> HGET items:1001 id
2 | "3"
3 | 192.168.101.3:7003> HGETALL items:1001
4 | 1) "id"
5 | 2) "3"
6 | 3) "name"
7 | 4) "apple"
8 | 5) "price"
9 | 6) "999.9"
10|
```

## list类型

### ArrayList与LinkedList的区别

ArrayList 使用数组方式存储数据，所以根据索引查询数据速度快，而新增或者删除元素时需要设计到位移操作，所以比较慢。

LinkedList 使用双向链表方式存储数据，每个元素都记录前后元素的指针，所以插入、删除数据时只是更改前后元素的指针指向即可，速度非常快。然后通过下标查询元素时需要从头开始索引，所以比较慢，但是如果查询前几个元素或后几个元素速度比较快。



## list类型介绍

Redis 的列表类型 (list 类型) 可以存储一个有序的字符串列表，常用的操作是向列表两端添加元素，或者获得列表的某一个片段。

列表类型内部是使用双向链表 (double linked list) 实现的，所以向列表两端添加元素的时间复杂度为 O(1)，获取越接近两端的元素速度就越快。这意味着即使是一个有几千万个元素的列表，获取头部或尾部的10条记录也是极快的。

## 命令

### LPUSH/RPUSH

- 语法：

```
1 | LPUSH key value [value ...]
2 | RPUSH key value [value ...]
```

- 示例：

```
1 | 127.0.0.1:6379> lpush list:1 1 2 3
2 | (integer) 3
3 | 127.0.0.1:6379> rpush list:1 4 5 6
4 | (integer) 3
```

## LRANGE

```
1 | 获取列表中的某一片段。将返回`start`、`stop`之间的所有元素（包含两端的元素），索引从`0`开始。索引可以是负数，如：“`-1`”代表最后边的一个元素。
```

- 语法：

```
1 | LRANGE key start stop
```

- 示例：

```
1 | 127.0.0.1:6379> lrange list:1 0 2
2 | 1) "2"
3 | 2) "1"
4 | 3) "4"
```

## LPOP/RPOP

```
1 | 从列表两端弹出元素
```

从列表左边弹出一个元素，会分两步完成：

- 第一步是将列表左边的元素从列表中移除
- 第二步是返回被移除的元素值。

语法：

```
1 | LPOP key
2 | RPOP key
```

## 示例

```
1 | 127.0.0.1:6379>lpop list:1
2 | "3"
3 | 127.0.0.1:6379>rpop list:1
4 | "6"
```

## LLEN

```
1 | 获取列表中元素的个数
```

- 语法：

```
1 | llen key
```

- 示例：

```
1 | 127.0.0.1:6379> llen list:1
2 | (integer) 2
```

## 其它命令(自学)

### LREM

```
1 | 删除列表中指定个数的值
```

LREM 命令会删除列表中前 count 个值为 value 的元素，返回实际删除的元素个数。根据 count 值的不同，该命令的执行方式会有所不同：

- ```
1 | - 当count>0时， LREM会从列表左边开始删除。
2 | - 当count<0时， LREM会从列表后边开始删除。
3 | - 当count=0时， LREM删除所有值为value的元素。
```

- 语法：

```
1 | LREM key count value
```

LINDEX

```
1 | 获得指定索引的元素值
```

- 语法：

```
1 | LINDEX key index
```

- 示例：

```
1 | 127.0.0.1:6379>lindex 1:list 2
2 | "1"
```

Ø 设置指定索引的元素值

语法：LSET key index value

```
127.0.0.1:6379> lset l:list 2 2 OK 127.0.0.1:6379> lrange l:list 0 -1 1) "6" 2) "5" 3) "2" 4)  
"2"
```

LTRIM

1 | 只保留列表指定片段,指定范围和LRANGE一致

- 语法：

1 | LTRIM key start stop

- 示例：

```
1 | 127.0.0.1:6379> lrange l:list 0 -1  
2 | 1) "6"  
3 | 2) "5"  
4 | 3) "0"  
5 | 4) "2"  
6 | 127.0.0.1:6379> ltrim l:list 0 2  
7 | OK  
8 | 127.0.0.1:6379> lrange l:list 0 -1  
9 | 1) "6"  
10 | 2) "5"  
11 | 3) "0"
```

LINSERT

1 | 向列表中插入元素。

2 | 该命令首先会在列表中从左到右查找值为pivot的元素，然后根据第二个参数是BEFORE还是AFTER来决定将value插入到该元素的前面还是后面。

语法：

1 | LINSERT key BEFORE|AFTER pivot value

示例：

```
1 | 127.0.0.1:6379> lrange list 0 -1
2 | 1) "3"
3 | 2) "2"
4 | 3) "1"
5 | 127.0.0.1:6379> linsert list after 3 4
6 | (integer) 4
7 | 127.0.0.1:6379> lrange list 0 -1
8 | 1) "3"
9 | 2) "4"
10 | 3) "2"
11 | 4) "1"
12 |
```

RPOPLPUSH

```
1 | 将元素从一个列表转移到另一个列表中
```

- 语法：

```
1 | RPOPLPUSH source destination
```

- 示例：

```
1 | 127.0.0.1:6379> rpoplpush list newlist
2 | "1"
3 | 127.0.0.1:6379> lrange newlist 0 -1
4 | 1) "1"
5 | 127.0.0.1:6379> lrange list 0 -1
6 | 1) "3"
7 | 2) "4"
8 | 3) "2"
```

应用之商品评论列表

- 需求：

```
1 | 用户针对某一商品发布评论，一个商品会被不同的用户进行评论，存储商品评论时，要按时间顺序排序。
2 |
3 | 用户在前端页面查询该商品的评论，需要按照时间顺序降序排序。
```

- 分析：

```
1 | 使用list存储商品评论信息，KEY是该商品的ID，VALUE是商品评论信息列表
```

- 实现：

商品编号为 1001 的商品评论 key 【 items: comment:1001 】

```
1 | 192.168.101.3:7001> LPUSH items:comment:1001 '{"id":1,"name":"商品不错，很好！！！","date":1430295077289}'
```

set类型

set类型介绍

set 类型即集合类型，其中的数据是不重复且没有顺序。

集合类型和列表类型的对比：

	集合类型	列表类型
存储内容	至多 $2^{32}-1$ 个字符串	至多 $2^{32}-1$ 个字符串
有序性	否	是
唯一性	是	否

集合类型的常用操作是向集合中加入或删除元素、判断某个元素是否存在等，由于集合类型的 Redis 内部是使用值为空的散列表实现，所有这些操作的时间复杂度都为 O(1)。

Redis 还提供了多个集合之间的交集、并集、差集的运算。

命令

SADD/SREM

```
1 | 添加元素/删除元素
```

语法：

```
1 | SADD key member [member ...]  
2 | SREM key member [member ...]
```

示例：

```
1 | 127.0.0.1:6379> sadd set a b c  
2 | (integer) 3  
3 | 127.0.0.1:6379> sadd set a  
4 | (integer) 0  
5 | 127.0.0.1:6379> srem set c d  
6 | (integer) 1
```

SMEMBERS

```
1 | 获得集合中的所有元素
```

语法：

```
1 | SMEMBERS key
```

示例：

```
1 | 127.0.0.1:6379> smembers set
2 | 1) "b"
3 | 2) "a"
```

SISMEMBER

```
1 | 判断元素是否在集合中
```

语法：

```
1 | SISMEMBER key member
```

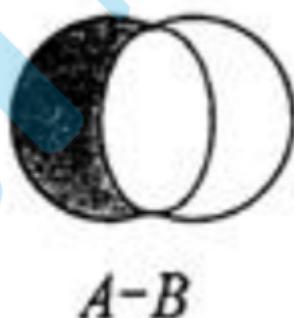
示例：

```
1 | 127.0.0.1:6379>sismember set a
2 | (integer) 1
3 | 127.0.0.1:6379>sismember set b
4 | (integer) 0
```

集合运算命令

SDIFF

```
1 | 集合的差集运算 A-B : 属于A并且不属于B的元素构成的集合。
```



语法：

```
1 | SDIFF key [key ...]
```

示例：

```
1 | 127.0.0.1:6379> sadd setA 1 2 3  
2 | (integer) 3  
3 | 127.0.0.1:6379> sadd setB 2 3 4  
4 | (integer) 3  
5 | 127.0.0.1:6379> sdiff setA setB  
6 | 1) "1"  
7 | 127.0.0.1:6379> sdiff setB setA  
8 | 1) "4"
```

SINTER

1 | 集合的交集运算 $A \cap B$: 属于A且属于B的元素构成的集合。



语法 :

```
1 | SINTER key [key ...]
```

示例 :

```
1 | 127.0.0.1:6379> sinter setA setB  
2 | 1) "2"  
3 | 2) "3"
```

SUNION

1 | 集合的并集运算 $A \cup B$: 属于A或者属于B的元素构成的集合

语法 :

```
1 | SUNION key [key ...]
```

示例 :

```
1 | 127.0.0.1:6379> sunion setA setB  
2 | 1) "1"  
3 | 2) "2"  
4 | 3) "3"  
5 | 4) "4"
```

其它命令(自学)

SCARD

1 | 获得集合中元素的个数

语法：

1 | SCARD key

示例：

```
1 | 127.0.0.1:6379> smembers setA
2 | 1) "1"
3 | 2) "2"
4 | 3) "3"
5 | 127.0.0.1:6379> scard setA
6 | (integer) 3
7 | 
```

SPOP

1 | 从集合中弹出一个元素。

2 | 注意：由于集合是无序的，所有SPOP命令会从集合中随机选择一个元素弹出

语法：

1 | SPOP key

示例：

```
1 | 127.0.0.1:6379> spop setA
2 | "1"
```

zset类型 (sortedset)

zset介绍

在 set 集合类型的基础上，有序集合类型为集合中的每个元素都关联一个分数，这使得我们不仅可以完成插入、删除和判断元素是否存在在集合中，还能够获得分数最高或最低的前N个元素、获取指定分数范围内的元素等与分数有关的操作。

在某些方面有序集合和列表类型有些相似：

- 1 | 1、二者都是有序的。
- 2 |
- 3 | 2、二者都可以获得某一范围的元素。

但是，二者有着很大区别：

- 1 | 1、列表类型是通过链表实现的，获取靠近两端的数据速度极快，而当元素增多后，访问中间数据的速度会变慢。
- 2 |
- 3 | 2、有序集合类型使用散列表实现，所有即使读取位于中间部分的数据也很快。
- 4 |
- 5 | 3、列表中不能简单的调整某个元素的位置，但是有序集合可以（通过更改分数实现）
- 6 |
- 7 | 4、有序集合要比列表类型更耗内存。

命令

ZADD

- 1 | 增加元素。
- 2 |
- 3 | 向有序集合中加入一个元素和该元素的分数，如果该元素已经存在则会用新的分数替换原有的分数。返回值是新加入到集合中的元素个数，不包含之前已经存在的元素。

语法：

```
1 | ZADD key score member [score member ...]
```

示例：

```
1 | 127.0.0.1:6379> zadd scoreboard 80 zhangsan 89 lisi 94 wangwu  
2 | (integer) 3  
3 | 127.0.0.1:6379> zadd scoreboard 97 lisi  
4 | (integer) 0  
5 |
```

ZRANGE/ZREVRANGE

- 1 | 获得排名在某个范围的元素列表。
 - ZRANGE：按照元素分数从小到大的顺序返回索引从start到stop之间的所有元素（包含两端的元素）
 - ZREVRANGE：按照元素分数从大到小的顺序返回索引从start到stop之间的所有元素（包含两端的元素）
- 2 |
- 3 |

语法：

```
1 | ZRANGE key start stop [WITHSCORES]  
2 | ZREVRANGE key start stop [WITHSCORES]
```

示例：

```
1 | 127.0.0.1:6379> zrange scoreboard 0 2
2 | 1) "zhangsan"
3 | 2) "wangwu"
4 | 3) "lisi"
5 | 127.0.0.1:6379> zrevrange scoreboard 0 2
6 | 1) " lisi "
7 | 2) "wangwu"
8 | 3) " zhangsan "
```

- 如果需要获得元素的分数的可以在命令尾部加上 WITHSCORES 参数

```
1 | 127.0.0.1:6379> zrange scoreboard 0 1 WITHSCORES
2 | 1) "zhangsan"
3 | 2) "80"
4 | 3) "wangwu"
5 | 4) "94"
```

ZSCORE

```
1 | 获取元素的分数。
```

语法：

```
1 | ZSCORE key member
```

示例：

```
1 | 127.0.0.1:6379> zscore scoreboard lisi
2 | "97"
```

ZREM

```
1 | 删除元素。
2 | 移除有序集合key中的一个或多个成员，不存在的成员将被忽略。
3 | 当key存在但不是有序集类型时，返回一个错误。
```

语法：

```
1 | ZREM key member [member ...]
```

示例：

```
1 | 127.0.0.1:6379> zrem scoreboard lisi
2 | (integer) 1
```

其它命令(自学)

ZRANGEBYSCORE

1 | 获得指定分数范围的元素。

语法：

1 | ZRANGEBYSCORE key min max [WITHSCORES]

示例：

```
1 127.0.0.1:6379> ZRANGEBYSCORE scoreboard 90 97 WITHSCORES
2 1) "wangwu"
3 2) "94"
4 3) "lisi"
5 4) "97"
6 127.0.0.1:6379> ZRANGEBYSCORE scoreboard 70 100 limit 1 2
7 1) "wangwu"
8 2) "lisi"
```

ZINCRBY

1 | 增加某个元素的分数。
2 | 返回值是更改后的分数

语法：

1 | ZINCRBY key increment member

示例：

```
1 127.0.0.1:6379> ZINCRBY scoreboard 4 lisi
2 "101"
```

ZCARD

1 | 获得集合中元素的数量。

语法：

1 | ZCARD key

示例：

```
1 | 127.0.0.1:6379> ZCARD scoreboard  
2 | (integer) 3
```

ZCOUNT

```
1 | 获得指定分数范围内的元素个数
```

语法：

```
1 | ZCOUNT key min max
```

示例：

```
1 | 127.0.0.1:6379> ZCOUNT scoreboard 80 90  
2 | (integer) 1
```

ZREMRANGEBYRANK

```
1 | 按照排名范围删除元素
```

语法：

```
1 | ZREMRANGEBYRANK key start stop
```

示例：

```
1 | 127.0.0.1:6379> ZREMRANGEBYRANK scoreboard 0 1  
2 | (integer) 2  
3 | 127.0.0.1:6379> ZRANGE scoreboard 0 -1  
4 | 1) "lisi"  
5 |
```

ZREMRANGEBYSCORE

```
1 | 按照分数范围删除元素
```

语法：

```
1 | ZREMRANGEBYSCORE key min max
```

示例：

```
1 | 127.0.0.1:6379> zadd scoreboard 84 zhangsan  
2 | (integer) 1  
3 | 127.0.0.1:6379> ZREMRANGEBYSCORE scoreboard 80 100  
4 | (integer) 1  
5 |
```

ZRANK/ZREVRANK

```
1 | 获取元素的排名。  
2 | - ZRANK : 从小到大  
3 | - ZREVRANK : 从大到小
```

语法：

```
1 | ZRANK key member  
2 | ZREVRANK key member
```

示例：

```
1 | 127.0.0.1:6379> ZRANK scoreboard lisi  
2 | (integer) 0  
3 | 127.0.0.1:6379> ZREVRANK scoreboard zhangsan  
4 | (integer) 1
```

应用之商品销售排行榜

- 需求：

```
1 | 根据商品销售量对商品进行排行显示
```

- 设计：

```
1 | 定义商品销售排行榜 ( sorted set集合 ) , Key为items:sellsort , 分数为商品销售量。
```

写入商品销售量：

- 商品编号 1001 的销量是 9 , 商品编号 1002 的销量是 10

```
1 | 192.168.101.3:7007> ZADD items:sellsort 9 1001 10 1002
```

- 商品编号 1001 的销量加 1

```
1 | 192.168.101.3:7001> ZINCRBY items:sellsort 1 1001
```

- 商品销量前 10 名：

```
1 | 192.168.101.3:7001> ZREVRANGE items:sellsort 0 9 withscores
```

通用命令

keys

```
1 | 返回满足给定pattern 的所有key
```

语法：

```
1 | keys pattern
```

示例：

```
1 | redis 127.0.0.1:6379> keys mylist*
2 | 1) "mylist"
3 | 2) "mylist5"
4 | 3) "mylist6"
5 | 4) "mylist7"
6 | 5) "mylist8"
```

del

语法：

```
1 | DEL key
```

示例：

```
1 | 127.0.0.1:6379> del test
2 | (integer) 1
```

exists

```
1 | 确认一个key 是否存在
```

语法：

```
1 | exists key
```

示例：从结果来看，数据库中不存在 Hongwan 这个 key，但是 age 这个 key 是存在的

```
1 | redis 127.0.0.1:6379> exists Hongwan
2 | (integer) 0
3 | redis 127.0.0.1:6379> exists age
4 | (integer) 1
5 | redis 127.0.0.1:6379>
```

expire (重点)

1 | Redis在实际使用过程中更多的用作缓存，然而缓存的数据一般都是需要设置生存时间的，即：到期后数据销毁。

语法：

1	EXPIRE key seconds	设置key的生存时间 (单位 : 秒) key在多少秒后会自动删除
2	TTL key	查看key生的生存时间
3	PERSIST key	清除生存时间
4	PEXPIRE key milliseconds	生存时间设置单位为 : 毫秒

示例：

1	192.168.101.3:7002> set test 1	设置test的值为1
2	OK	
3	192.168.101.3:7002> get test	获取test的值
4	"1"	
5	192.168.101.3:7002> EXPIRE test 5	设置test的生存时间为5秒
6	(integer) 1	
7	192.168.101.3:7002> TTL test	查看test的生于生成时间还有1秒删除
8	(integer) 1	
9	192.168.101.3:7002> TTL test	
10	(integer) -2	
11	192.168.101.3:7002> get test	获取test的值，已经删除
12	(nil)	

rename

1 | 重命名key

语法：

1 | rename oldkey newkey

示例：`age` 成功的被我们改名为 `age_new` 了

1	redis 127.0.0.1:6379[1]> keys *	
2	1) "age"	
3	redis 127.0.0.1:6379[1]> rename age age_new	
4	OK	
5	redis 127.0.0.1:6379[1]> keys *	
6	1) "age_new"	
7	redis 127.0.0.1:6379[1]>	

type

1 | 显示指定key的数据类型

语法：

1 | type key

示例：这个方法可以非常简单的判断出值的类型

```
1 redis 127.0.0.1:6379> type addr
2 string
3 redis 127.0.0.1:6379> type myzset2
4 zset
5 redis 127.0.0.1:6379> type mylist
6 list
7 redis 127.0.0.1:6379>
8
```

Redis消息模式

队列模式

使用list类型的lpush和rpop实现消息队列

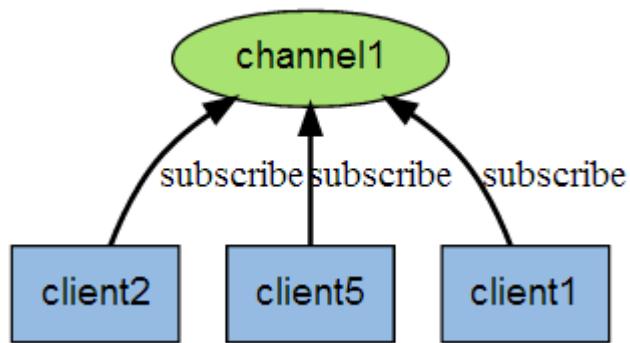


注意事项：

- 消息接收方如果不知道队列中是否有消息，会一直发送rpop命令，如果这样的话，会每一次都建立一次连接，这样显然不好。
- 可以使用brpop命令，它如果从队列中取不出来数据，会一直阻塞，在一定范围内没有取出则返回null、

发布订阅模式

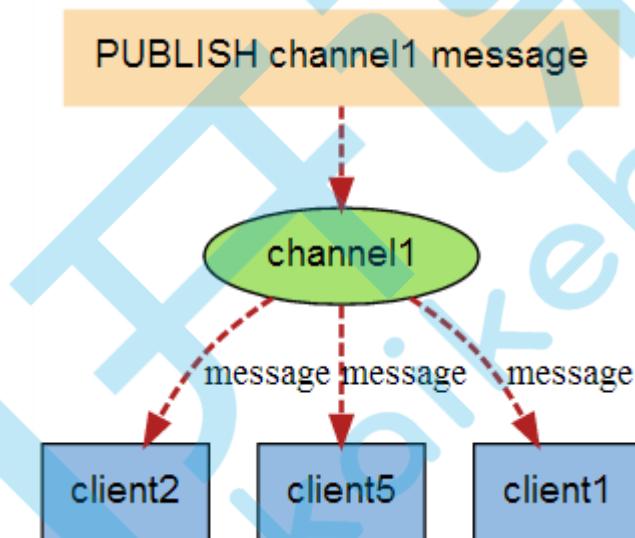
- 订阅消息 (subscribe)



示例：

```
1 | subscribe kkb-channel
```

- 发布消息 (publish)



```
1 | publish kkb-channel "我是灭霸詹"
```

- Redis发布订阅命令

Redis 发布订阅命令

下表列出了 redis 发布订阅常用命令：

序号	命令及描述
1	<code>PSUBSCRIBE pattern [pattern ...]</code> 订阅一个或多个符合给定模式的频道。
2	<code>PUBSUB subcommand [argument [argument ...]]</code> 查看订阅与发布系统状态。
3	<code>PUBLISH channel message</code> 将信息发送到指定的频道。
4	<code>PUNSUBSCRIBE [pattern [pattern ...]]</code> 退订所有给定模式的频道。
5	<code>SUBSCRIBE channel [channel ...]</code> 订阅给定的一个或多个频道的信息。
6	<code>UNSUBSCRIBE [channel [channel ...]]</code> 撤退订给定的频道。

Redis事务

Redis事务介绍

- Redis 的事务是通过 `MULTI`、`EXEC`、`DISCARD` 和 `WATCH`、`UNWATCH` 这五个命令来完成的。
- Redis 的单个命令都是原子性的，所以这里需要确保事务性的对象是命令集合。
- Redis 将命令集合序列化并确保处于同一事务的命令集合连续且不被打断的执行
- Redis 不支持回滚操作。

事务命令

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，然后使用 `EXEC` 命令原子化地执行这个命令序列。

语法：

```
1 | multi
```

EXEC

- 在一个事务中执行所有先前放入队列的命令，然后恢复正常连接状态

语法：

```
1 | exec
```

DISCARD

```
1 | 清除所有先前在一个事务中放入队列的命令，然后恢复正常连接状态。
```

语法：

```
1 | discard
```

WATCH

```
1 | 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]的状态。
```

语法：

```
1 | watch key [key...]
```

注意事项：使用该命令可以实现 Redis 的乐观锁。

UNWATCH

```
1 | 清除所有先前为一个事务监控的键。
```

语法：

```
1 | unwatch
```

事务演示

```
1 127.0.0.1:6379> multi
2 OK
3 127.0.0.1:6379> set s1 111
4 QUEUED
5 127.0.0.1:6379> hset set1 name zhangsan
6 QUEUED
7 127.0.0.1:6379> exec
8 1) OK
9 2) (integer) 1
10 127.0.0.1:6379> multi
11 OK
12 127.0.0.1:6379> set s2 222
13 QUEUED
```

```
14 127.0.0.1:6379> hset set2 age 20
15 QUEUED
16 127.0.0.1:6379> discard
17 OK
18 127.0.0.1:6379> exec
19 (error) ERR EXEC without MULTI
20
21 127.0.0.1:6379> watch s1
22 OK
23 127.0.0.1:6379> multi
24 OK
25 127.0.0.1:6379> set s1 555
26 QUEUED
27 127.0.0.1:6379> exec      # 此时在没有exec之前，通过另一个命令窗口对监控的s1字段进行修改
28 (nil)
29 127.0.0.1:6379> get s1
30 111
```

事务失败处理

- Redis 语法错误 (编译期)

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> sets s1 111
(error) ERR unknown command 'sets'
127.0.0.1:6379> set s1
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379> set s4 444
QUEUED
127.0.0.1:6379> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get s4
(nil)
```

- Redis 运行错误

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s4 444
QUEUED
127.0.0.1:6379> lpush s4 111 222
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6379> get s4
"444"
127.0.0.1:6379>
```

- Redis 不支持事务回滚 (为什么呢)

- 大多数事务失败是因为语法错误或者类型错误，这两种错误，在开发阶段都是可以预见的
- Redis 为了性能方面就忽略了事务回滚。

扩展点

spring-data-redis

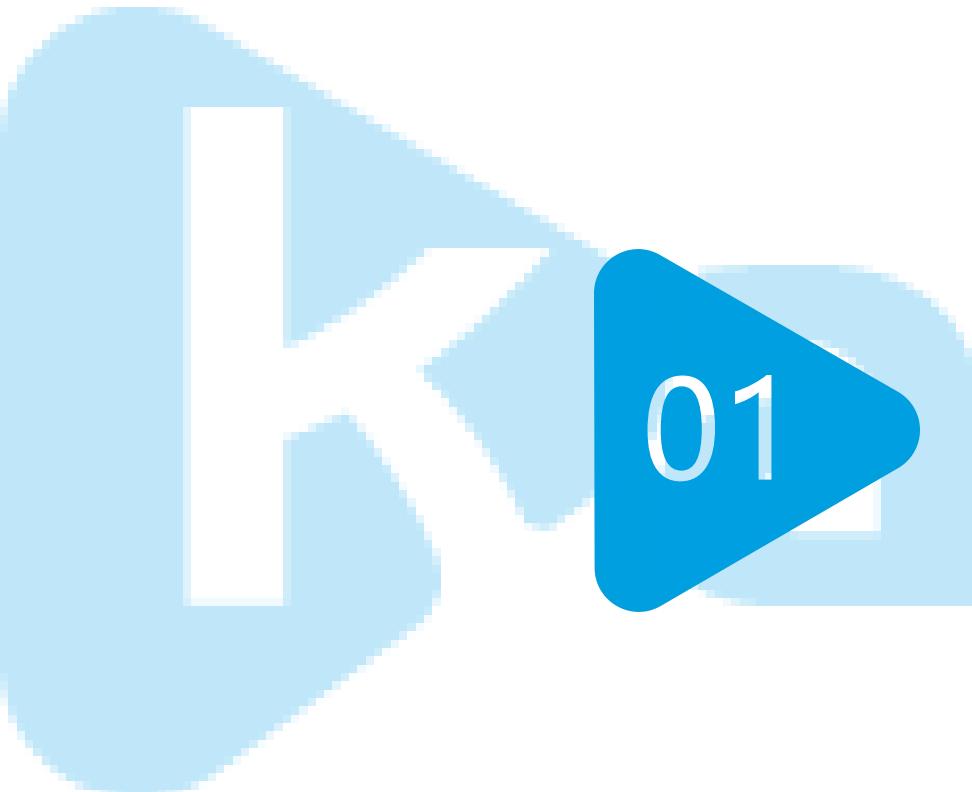
本课总结

布置作业

课后互动问答

下节预告





01

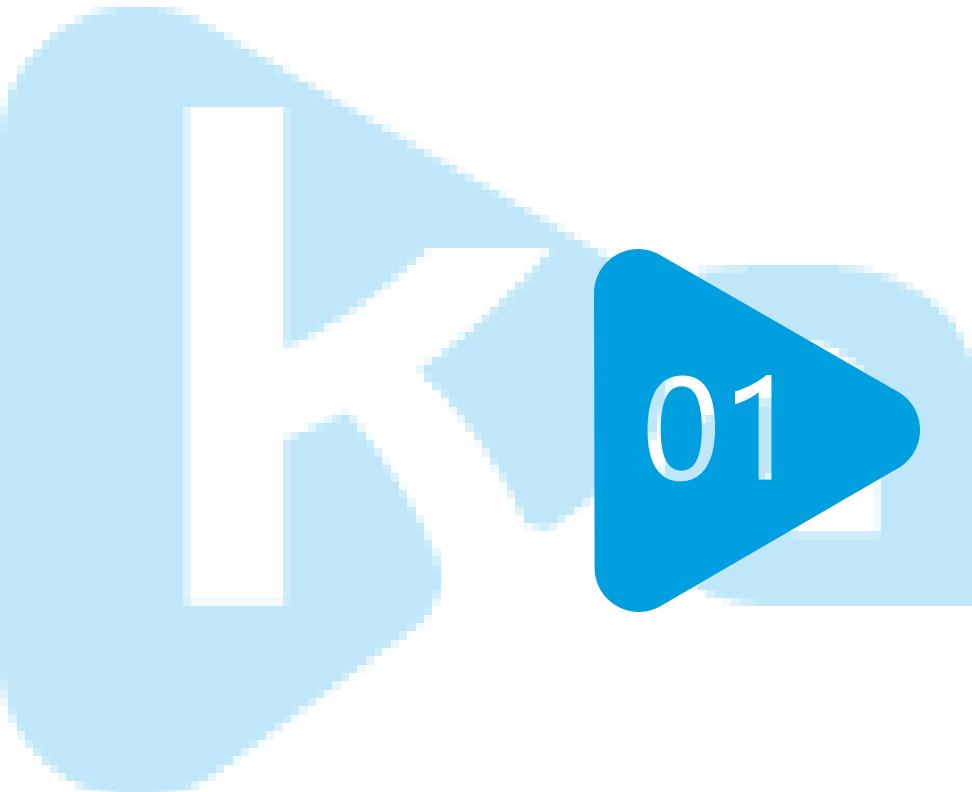


keiba
哨兵机制

开课吧

Redis哨兵机制

- **监控(Monitoring)**: 哨兵(sentinel) 会不断地检查你的 Master 和 Slave 是否运作正常。
- **提醒(Notification)**: 当被监控的某个`Redis`节点出现问题时, 哨兵(sentinel) 可以通过 API 向管理员或者其他应用程序发送通知。
- **自动故障迁移(Automatic failover)**: 当一个Master不能正常工作时, 哨兵(sentinel) 会开始一次自动故障迁移操作



01



开课吧

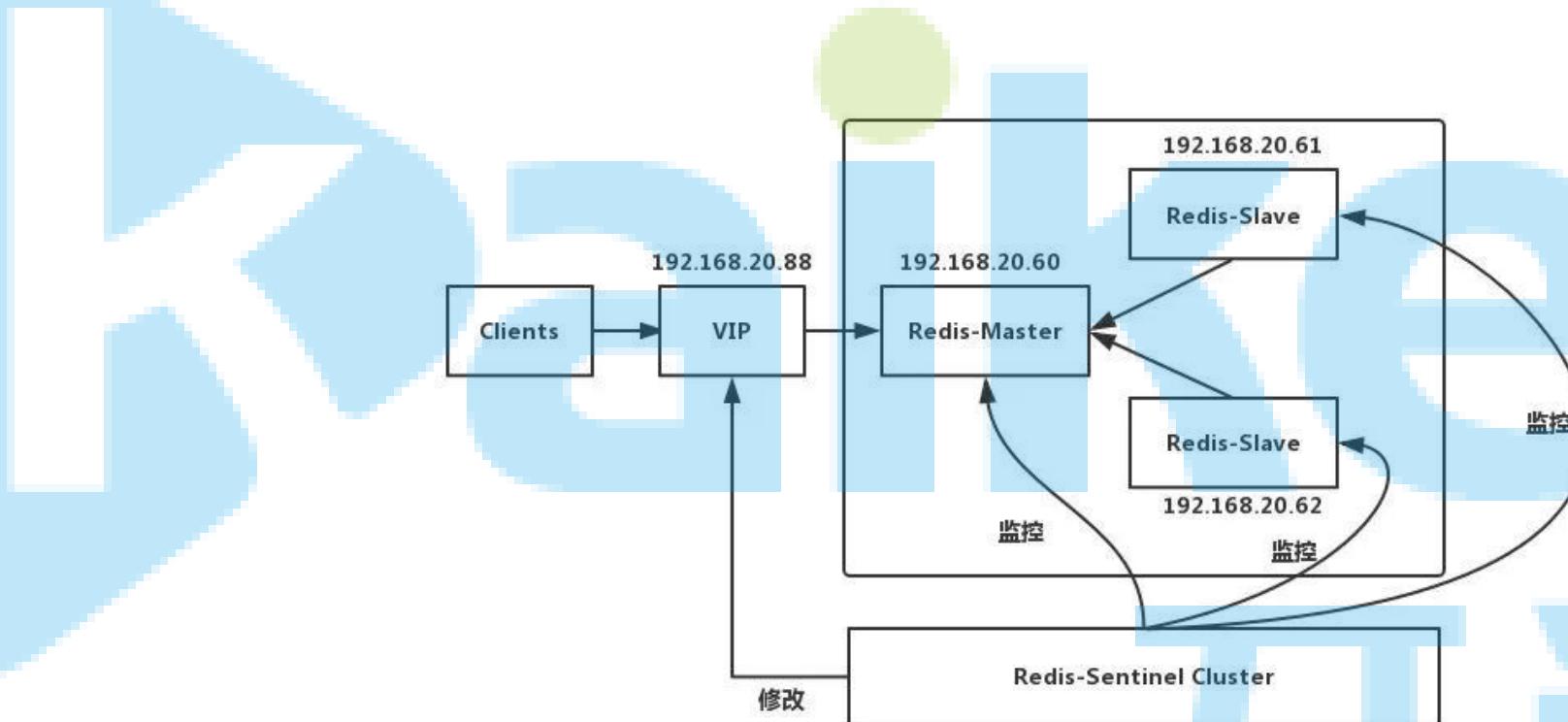
Redis集群演进

1. Redis主从复制
2. Replication+Sentinel
3. Proxy+Replication+Sentinel
4. Redis Cluster



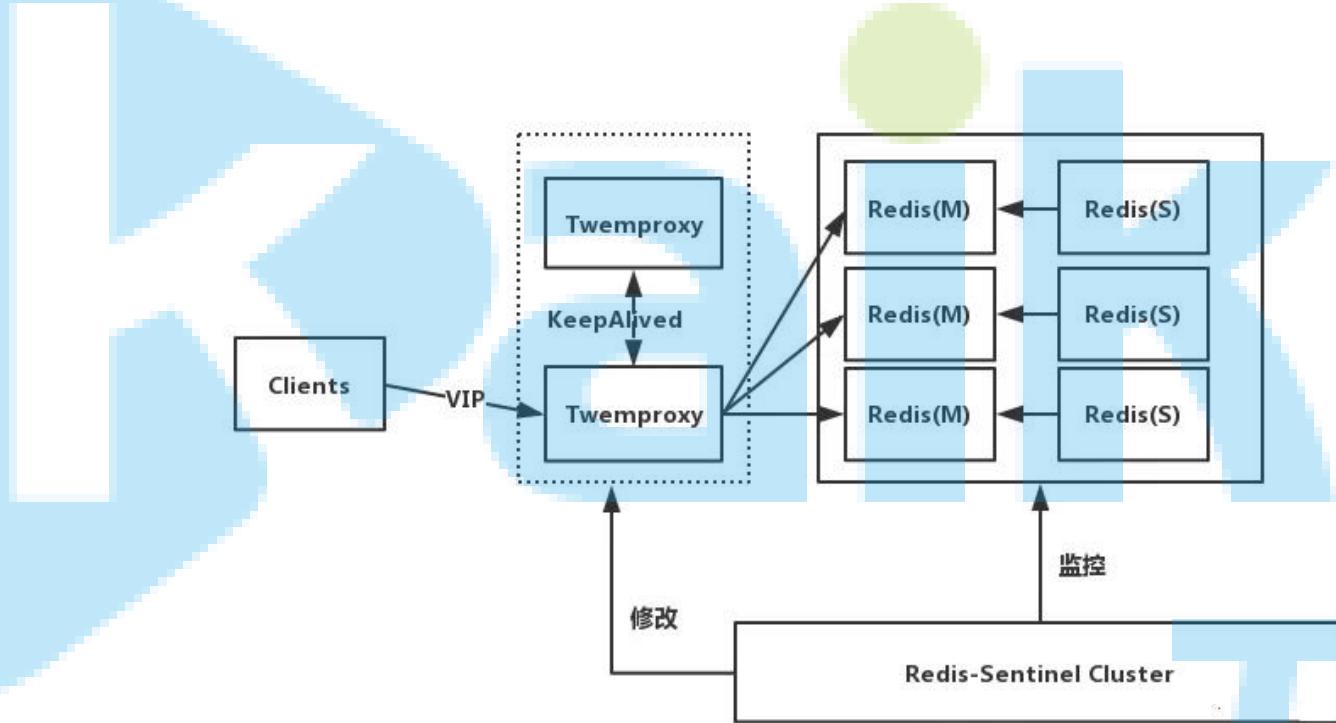
Redis集群演进-Replication+Sentinel

这套架构使用的是社区版本推出的原生高可用解决方案

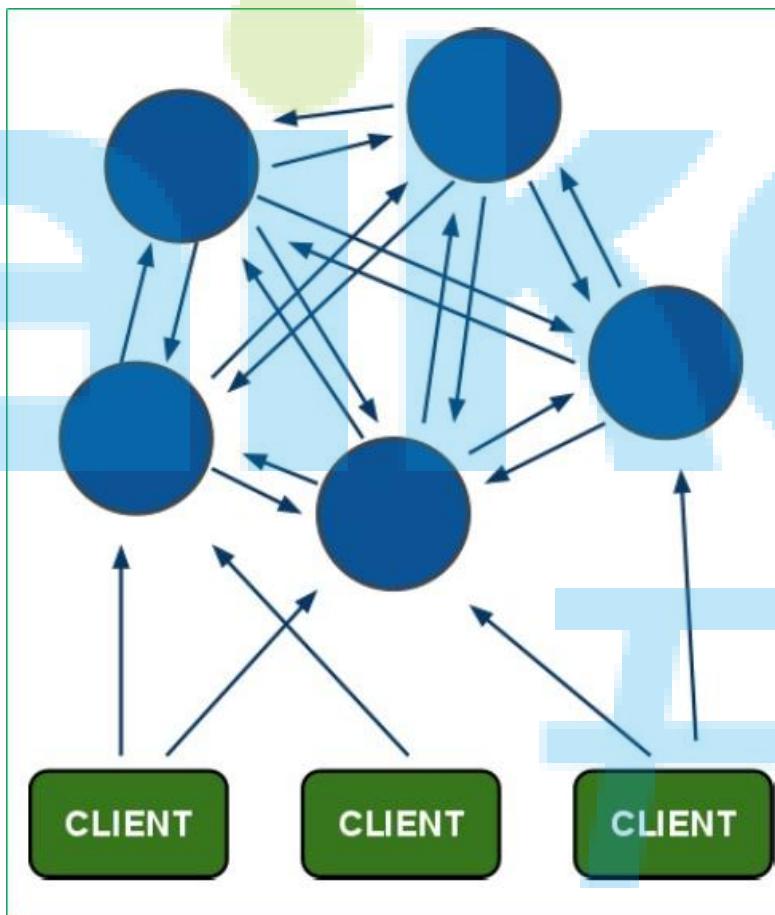


Redis集群演进-Proxy+Replication+Sentinel

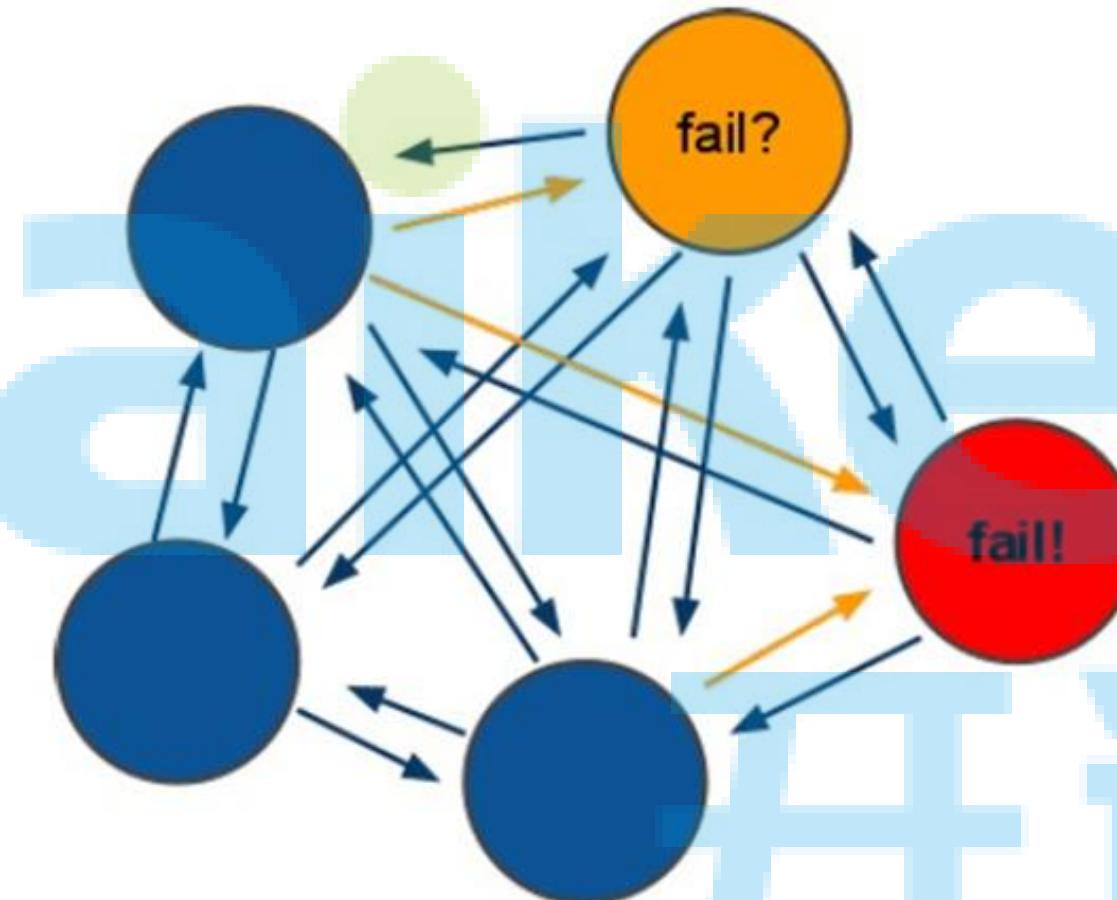
这里的Proxy目前有两种选择:Codis (豌豆莢) 和Twemproxy (推特)



Redis集群演进-Redis Cluster



Redis-cluster投票:容错



JedisCluster类连接redis集群

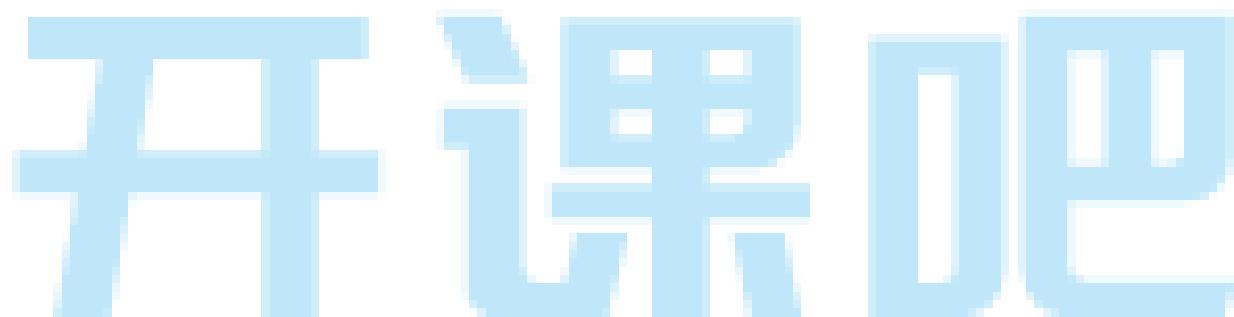
```
1  @Test
2  public void testJedisCluster() throws Exception {
3      //创建一连接, JedisCluster对象,在系统中是单例存在
4      Set<HostAndPort> nodes = new HashSet<>();
5      nodes.add(new HostAndPort("192.168.10.133", 7001));
6      nodes.add(new HostAndPort("192.168.10.133", 7002));
7      nodes.add(new HostAndPort("192.168.10.133", 7003));
8      nodes.add(new HostAndPort("192.168.10.133", 7004));
9      nodes.add(new HostAndPort("192.168.10.133", 7005));
10     nodes.add(new HostAndPort("192.168.10.133", 7006));
11     JedisCluster cluster = new JedisCluster(nodes);
12     //执行JedisCluster对象中的方法, 方法和redis一一对应。
13     cluster.set("cluster-test", "my jedis cluster test");
14     String result = cluster.get("cluster-test");
15     System.out.println(result);
16     //程序结束时需要关闭JedisCluster对象
17     cluster.close();
18 }
19 }
```



03



Redis和lua整合



开课吧

什么是lua

lua是一种轻量小巧的**脚本语言**，用标准**C语言**编写并以源代码形式开放，其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。



开课吧

Redis中使用lua的好处

1. **减少网络开销**, 在Lua脚本中可以把多个命令放在同一个脚本中运行
2. **原子操作**, redis会将整个脚本作为一个整体执行, 中间不会被其他命令插入。换句话说, 编写脚本的过程中无需担心会出现竞态条件
3. **复用性**, 客户端发送的脚本会永远存储在redis中, 这意味着其他客户端可以复用这一脚本来完成同样的逻辑

[lua教程](#)

开课吧

Redis整合lua脚本

- EVAL命令
- lua脚本中调用Redis命令
- EVALSHA
- SCRIPT命令
- redis-cli --eval

开课吧

Redis整合lua脚本

- EVAL命令
- lua脚本中调用Redis命令
- EVALSHA
- SCRIPT命令
- redis-cli --eval

开课吧

Redis结合lua 实现秒杀

1.首先定义redis数据结构

```
1 goodId:  
2 {  
3     "total":100,  
4     "released":0;  
5 }
```

开课吧

2. 编写lua脚本

```
1 local n = tonumber(ARGV[1])
2 if not n or n == 0 then
3     return 0
4 end
5 local vals = redis.call("HMGET", KEYS[1], "total", "released");
6 local total = tonumber(vals[1])
7 local blocked = tonumber(vals[2])
8 if not total or not blocked then
9     return 0
10 end
11 if blocked + n <= total then
12     redis.call("HINCRBY", KEYS[1], "released", n)
13     return n;
14 end
15 return 0
```

3.spring boot 调用

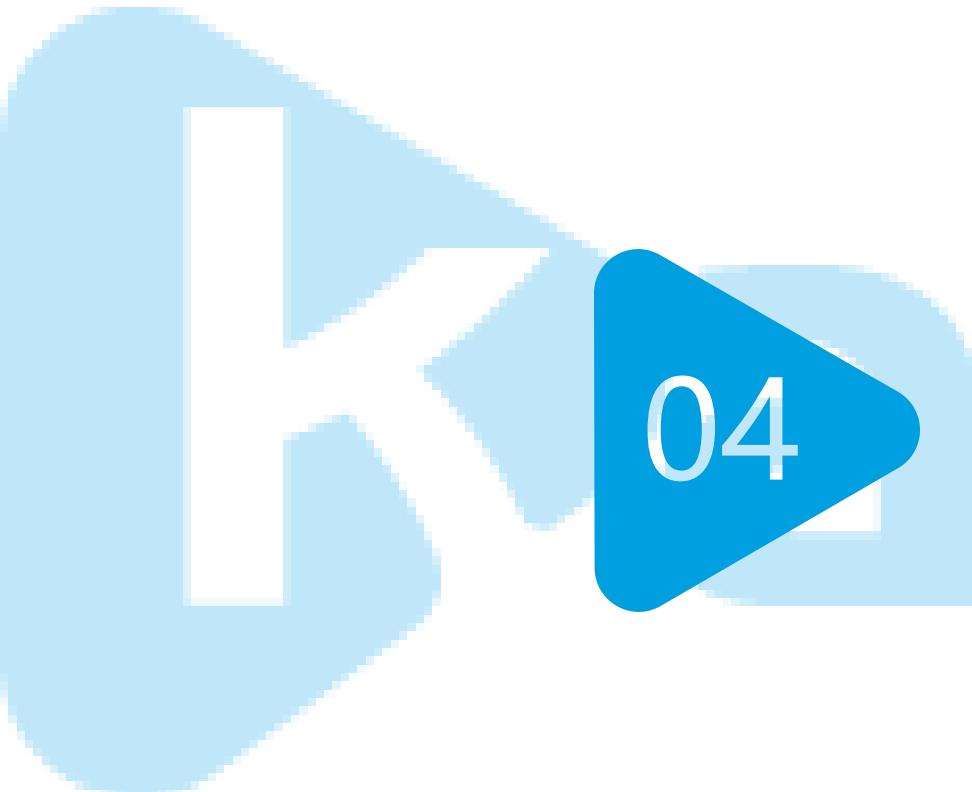
```
1 <groupId>org.springframework.boot</groupId>
2 <artifactId>spring-boot-starter-data-redis</artifactId>
3 <version>2.0.1.RELEASE</version>

1
2     Long count = redisHelper.getStrCache().execute(new RedisCallback<Long>() {
3         @Nullable
4         @Override
5         public Long doInRedis(RedisConnection redisConnection) throws
6             DataAccessException {
7                 long ret = redisConnection.eval(script.getScriptAsString().getBytes(),
8                     ReturnType.INTEGER, 1, key.getBytes(), String.valueOf(count).getBytes());
9                 return ret;
10            }
11        });
12    );
```

4.redis->database

kaikedab

开课吧



04



Redis 事务

开课吧

在生产环境里，经常会利用redis乐观锁来实现秒杀，Redis乐观锁是Redis事务的经典应用。

具体思路如下：

监控锁定量，如果该值被修改成功则表示该请求被通过，反之表示该请求未通过。

从监控到修改到执行都需要在redis里操作，这样就需要用到Redis事务。

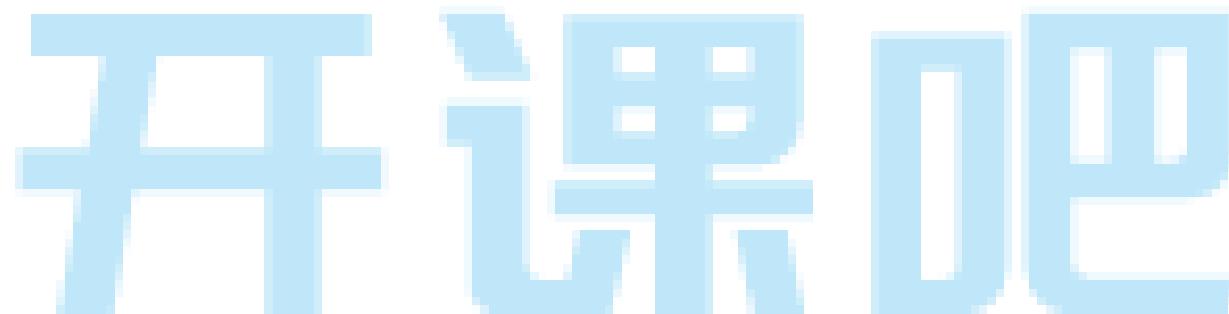
- Redis的事务是通过MULTI、EXEC、DISCARD和WATCH这四个命令来完成的。
- Redis的单个命令都是原子性的，所以这里需要确保事务性的对象是命令集合。
- -Redis将命令集合序列化并确保处于同一事务的命令集合连续且不被打断的执行
- -Redis不支持回滚操作。

开课吧

- Redis的事务是通过MULTI、EXEC、DISCARD和WATCH这四个命令来完成的。
- Redis的单个命令都是原子性的，所以这里需要确保事务性的对象是命令集合。
- -Redis将命令集合序列化并确保处于同一事务的命令集合连续且不被打断的执行
- -Redis不支持回滚操作。

开课吧

- Redis的事务是通过MULTI、EXEC、DISCARD和WATCH这四个命令来完成的。
- Redis的单个命令都是原子性的，所以这里需要确保事务性的对象是命令集合。
- -Redis将命令集合序列化并确保处于同一事务的命令集合连续且不被打断的执行
- -Redis不支持回滚操作。

开课吧

```
1 public void watch() {
2     try {
3         String watchKeys = "watchKeys";
4         //初始值 value=1
5         jedis.set(watchKeys, 1);
6         //监听key为watchKeys的值
7         jedis.watch(watchkeys);
8
9         //开启事务
10        Transaction tx = jedis.multi();
11
12        //watchKeys自增加一
13        tx.incr(watchKeys);
14
15        //执行事务，如果其他线程对watchKeys中的value进行修改，则该事务将不会执行
16        //通过redis事务以及watch命令实现乐观锁
17        List<Object> exec = tx.exec();
18        if (exec == null) {
19            System.out.println("事务未执行");
20        } else {
21            System.out.println("事务成功执行, watchKeys的value成功修改");
22        }
23    } catch (Exception e) {
24        e.printStackTrace();
25    } finally {
26        jedis.close();
27    }
28}
29
```

Redis乐观锁实现秒杀

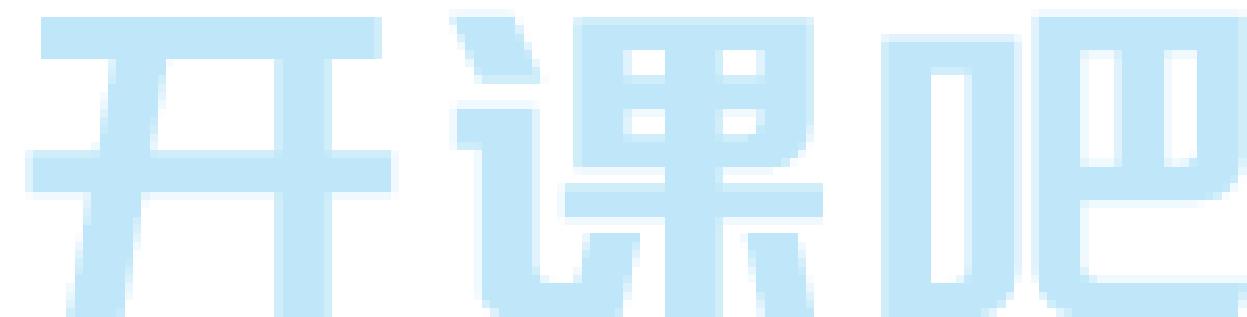
```
1 public class SecKill {
2     public static void main(String[] args) {
3         String redisKey = "second";
4
5         ExecutorService executorService = Executors.newFixedThreadPool(20);
6         try {
7             Jedis jedis = new Jedis("127.0.0.1", 6378);
8             // 初始值
9             jedis.set(redisKey, "0");
10            jedis.close();
11        } catch (Exception e) {
12            e.printStackTrace();
13        }
14
15        for (int i = 0; i < 1000; i++) {
16
17            executorService.execute(() -> {
18
19                Jedis jedis1 = new Jedis("127.0.0.1", 6378);
20                try {
21                    jedis1.watch(redisKey);
22                    String redisValue = jedis1.get(redisKey);
23                    int valInteger = Integer.valueOf(redisValue);
24                    String userInfo = UUID.randomUUID().toString();
25
26                    // 没有秒完
27                    if (valInteger < 20) {
28                        Transaction tx = jedis1.multi();
29                        tx.incr(redisKey);
30                        List list = tx.exec();
31                        // 秒成功    失败返回空list而不是空
32                        if (list != null && list.size() > 0) {
33
34                            System.out.println("用户: " + userInfo + ", 秒杀成功! 当前成功
人数: " + (valInteger + 1));
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59    } } ); } finally { jedis1.close(); } } executorService.shutdown(); }
```



05



Redis实现分布式锁



开课吧

1. 库存超卖
2. 防止用户重复下单
3. MQ消息去重
4. 订单操作变更

开课吧

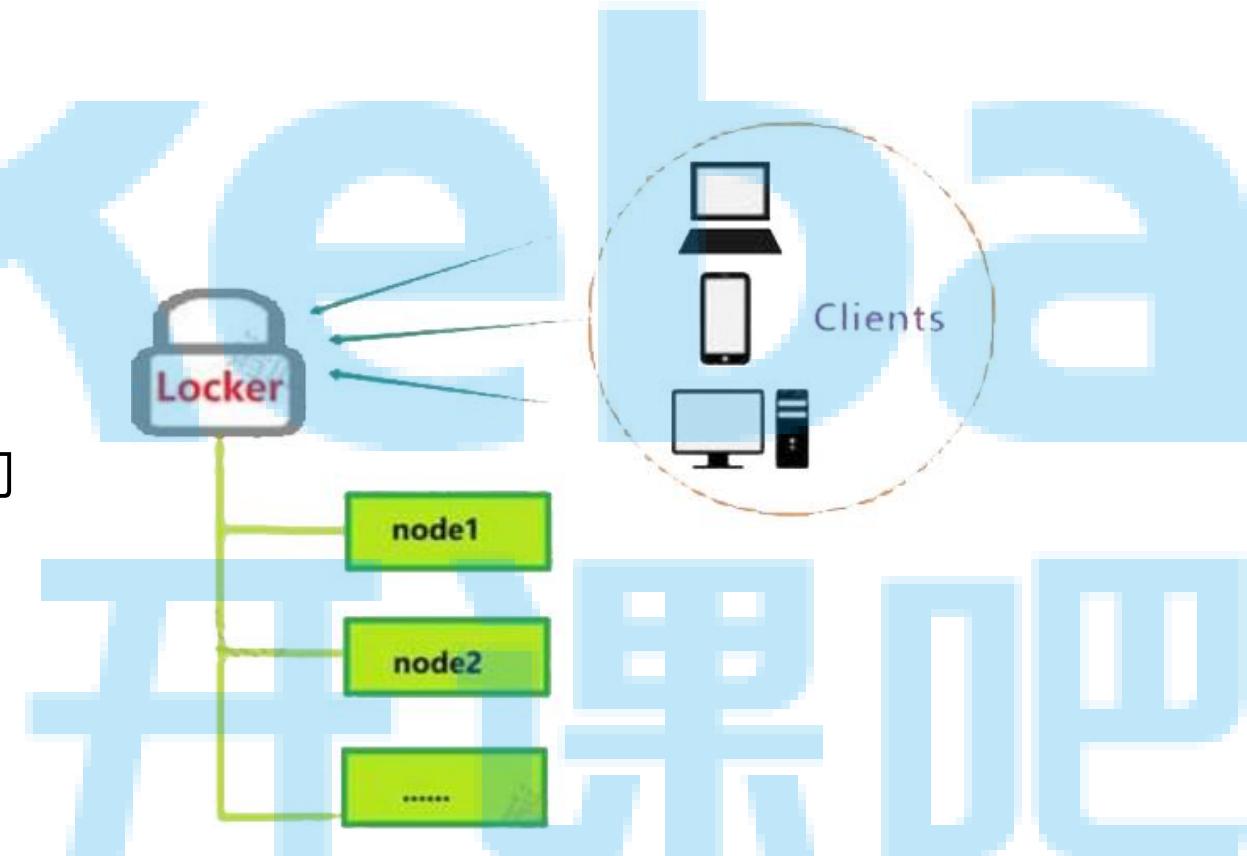
锁的处理

- 单应用中使用锁： (单进程多线程)

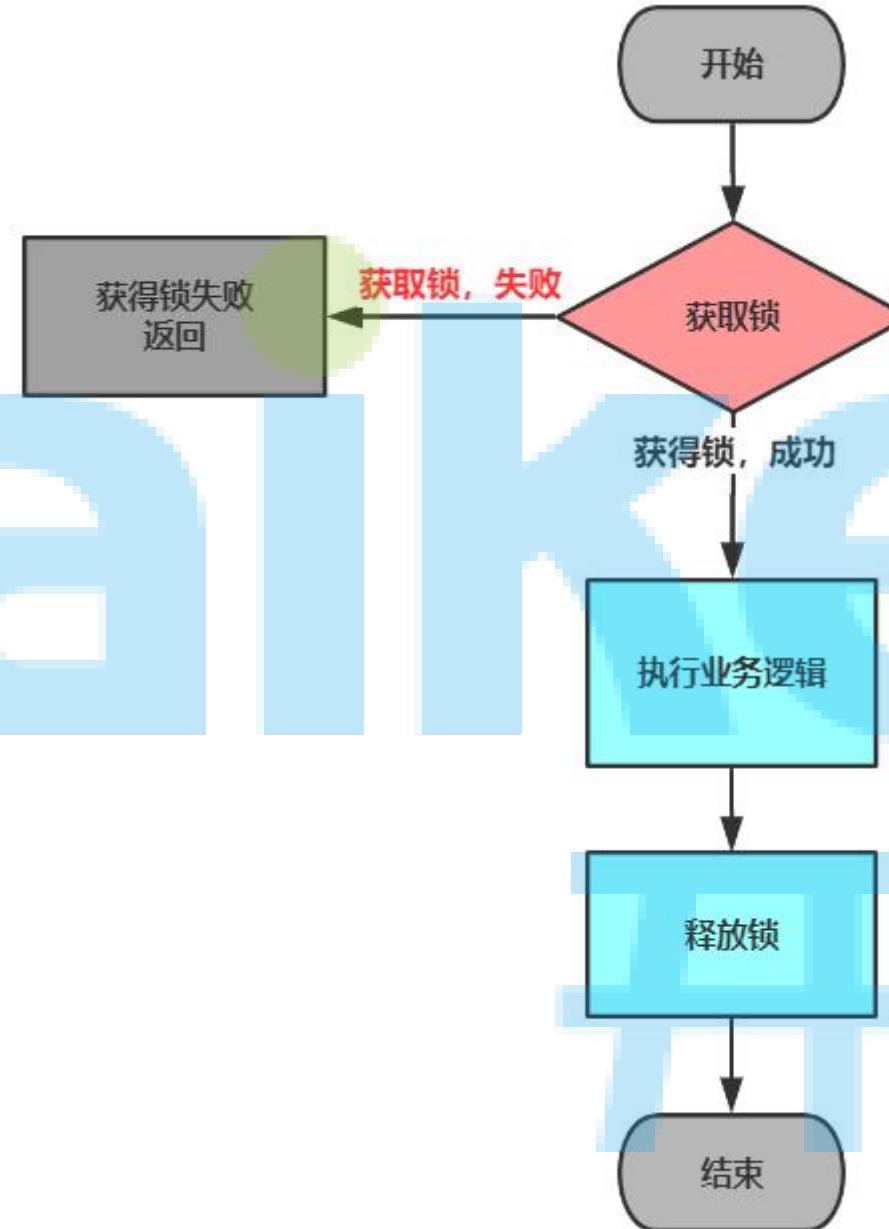
synchronize、ReentrantLock

- 分布式应用中使用锁： (多进程多线程)

分布式锁是控制分布式系统之间同步访问
共享资源的一种方式。



分布式锁流程



分布式锁的状态

1. 客户端通过竞争获取锁才能对共享资源进行操作(①获取锁);
2. 当持有锁的客户端对共享资源进行操作时 (②占有锁)
3. 其他客户端都不可以对这个资源进行操作 (③阻塞)
4. 直到持有锁的客户端完成操作(④释放锁);

开课吧

什么是分布式锁? --分布式锁特点



设计分布式锁时需要考虑一下四点:

- **互斥性**

在任意时刻，只有一个客户端可以持有锁（排他性）

- **高可用，具有容错性**

只要锁服务集群中的大部分节点正常运行，客户端就可以进行加锁解锁操作

- **避免死锁**

具备锁失效机制，锁在一段时间之后一定会释放。（正常释放或超时释放）

- **加锁和解锁为同一个客户端()**

一个客户端不能释放其他客户端加的锁了

| 分布式锁的实现方式



基于数据库实现分布式锁

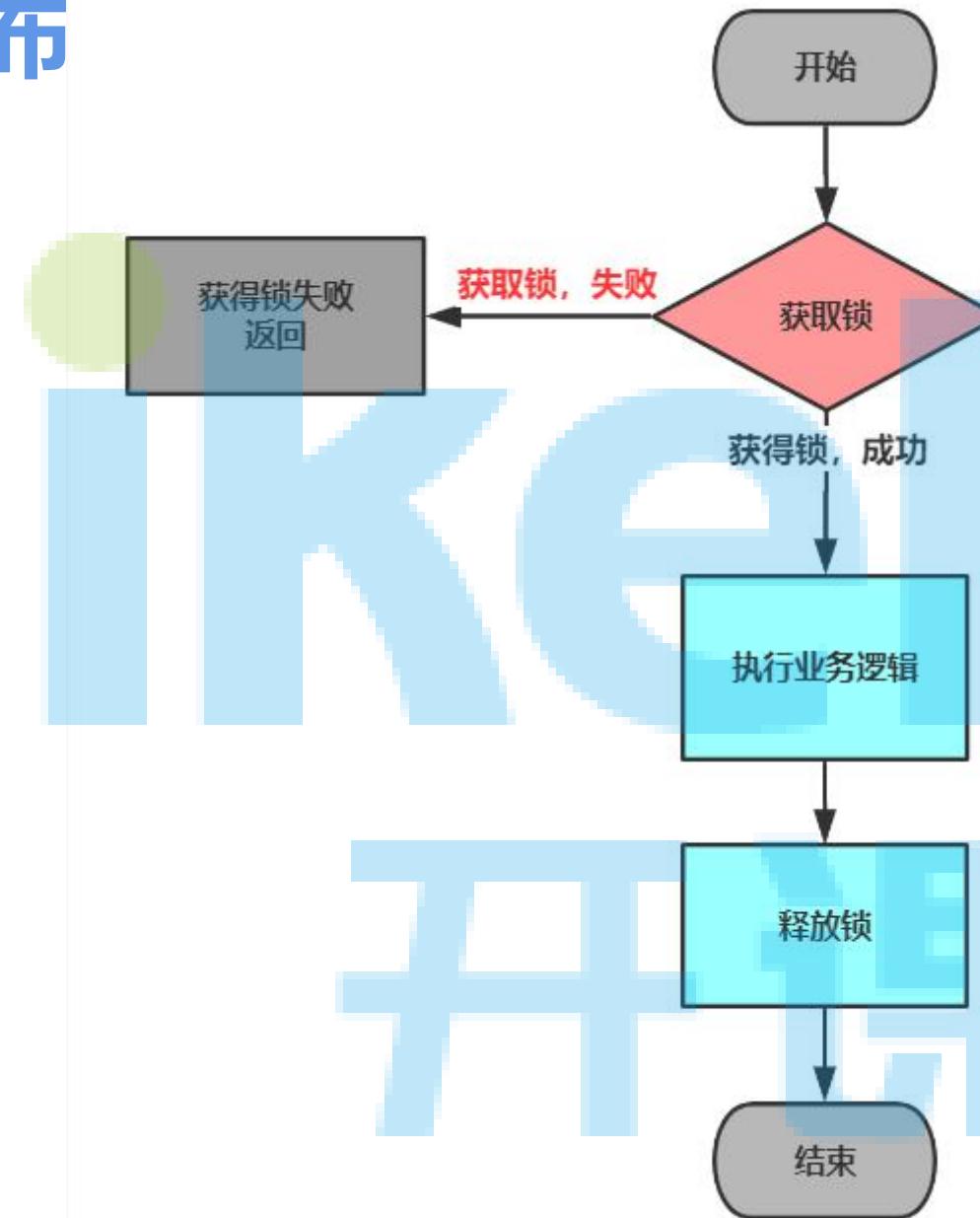
基于Redis的分布式锁

基于zookeeper时节点的分布式锁

基于Etcd的分布式锁

开课吧

Redis方式实现分布



Redis方式实现分布式锁-获取锁



- 使用 setnx + expire 实现加锁动作。

(1) setnx key,value

设置key的值为value，key不存在设置成功返回1，否则设置失败返回0。

(2) expire key timeout: 设置key的有效期。

大家想一下为什么要设置过期时间？

Redis方式实现分布式锁-获取锁



● 获取锁代码-错误方式：

```
/**  
 * 获取redis分布式锁  
 * @param jedis  
 * @param lockKey 锁  
 * @param lockValue 锁值  
 * @param expireTime 超期时间, 单位秒  
 * @return 是否获取成功  
 */  
  
public static boolean tryLock(Jedis jedis, String lockKey, String lockValue, int  
expireTime) {  
    Long result = jedis.setnx(lockKey, lockValue); //设置锁  
    if (result == 1) { //获取锁成功  
        //若在这里线程突然崩溃, 则无法设置过期时间, 将发生死锁  
        jedis.expire(lockKey, expireTime);  
        return true;  
    }  
    return false;  
}
```

Redis方式实现分布式锁-获取锁



使用Lua脚本保证原子性

```
if redis.call('setnx', KEYS[1], ARGV[1]) == 1 then  
    redis.call('expire', KEYS[1], ARGV[2])  
    return 1  
else  
    return 0  
end
```

#测试，在服务器上创建lua脚本文件trylock.lua，可以直接使用redis-cli测试。

```
[root@localhost redis]# ./redis-cli --eval ./trylock.lua "lockKey" , "lockValue" 30  
(integer) 1  
[root@localhost redis]# ./redis-cli --eval ./trylock.lua "lockKey" , "lockValue" 30  
(integer) 0
```

Redis方式实现分布式锁-获取锁



Redis2.6.12版本之前，使用Lua脚本保证原子性，获取锁代码

```
// 加锁脚本
private static final String SCRIPT_TRYLOCK = "if redis.call('setnx', KEYS[1], ARGV[1]) == 1 then redis.call('pexpire', KEYS[1], ARGV[2]) return 1 else return 0 end";
/**
 * 使用Lua脚本，尝试获取redis分布式锁
 * @param jedis
 * @param lockKey 锁
 * @param lockValue 锁值
 * @param expireTime 超期时间，单位秒
 * @return 是否获取成功
 */
public static boolean tryLockLua(Jedis jedis, String lockKey, String lockValue, int expireTime) {
    int result =
(jint)jedis.eval(SCRIPT_TRYLOCK,1,lockKey,lockValue,String.valueOf(expireTime));//设置锁
    if (result == 1){//获取锁成功
        return true;
    }
    return false;
}
```

Redis方式实现分布式锁-获取锁



- 从Redis2.6.12版本开始，可以使用Set一个命令实现加锁

SET key value NX PX miliseconds

如：SET key value NX PX 10000

设置key的值为value并设置10秒的有效期，key不存在设置成功，否则失败。

开课吧

Redis方式实现分布式锁-获取锁



- 从Redis2.6.12版本开始，使用Set一个命令实现加锁，获取锁代码

```
/**  
 * 获取redis分布式锁  
 * @param jedis  
 * @param lockKey 锁  
 * @param lockValue 锁值  
 * @param expireTime 超期时间，单位毫秒  
 * @return 是否获取成功  
 */  
public static boolean tryLock(Jedis jedis, String lockKey, String lockValue, int  
expireTime) {  
    String result = jedis.set(lockKey, lockValue, "NX", "PX", expireTime);  
  
    if ("OK".equals(result)) {  
        return true;  
    }  
    return false;  
}
```

Redis方式实现分布式锁-释放锁



- 使用del命令

del key : 删除key

只使用del命令就足够了吗?

开课吧

Redis方式实现分布式锁-释放锁-错误删除锁

线程A成功得到了锁，并且设置的超时时间是30秒

节点1的JVM进程



节点2的JVM进程



同步代码块

XXXXXX
XXXXX
XXX
XXXXXX

线程A执行的很慢很慢，过了30秒都没执行完，这时候锁过期自动释放，线程B得到了锁。

节点1的JVM进程



节点2的JVM进程



同步代码块

XXXXXX
XXXXX
XXX
XXXXXX

Redis方式实现分布式锁-释放锁-错误删除锁

线程A执行完了任务，线程A接着执行del指令来释放锁。但这时候线程B还没执行完，**线程A实际上删除的是线程B加的锁**

节点1的JVM进程



执行完了，
Del!

节点2的JVM进程



Lock

同步代码块

xxxxxx
xxxxxx
xxx
xxxxxx

Redis方式实现分布式锁-释放锁-正确姿势-保证Value的唯一性

```
//解锁脚本，一定要比较锁的value，防止误解锁
private static final String SCRIPT_UNLOCK = "if redis.call('get', KEYS[1]) == ARGV[1]
then return redis.call('del', KEYS[1]) else return 0 end";

public static boolean unLockLua(Jedis jedis, String lockKey, String lockValue) {
    int result = (int)jedis.eval(SCRIPT_UNLOCK,1,lockKey,lockValue); //删除锁
    if (result == 1) { //释放锁成功
        return true;
    }
    return false;
}
```

锁的value的生成：

- (1) 可以加锁的时候把当前的线程ID（服务标识+线程id）当做value。
- (2) 使用时间戳 + 服务标识的方式生成随机数，大多数情况下也够用。
- (3) 官方推荐从 /dev/urandom 中取 20 个 byte 作为随机数当做value。

Redis方式实现分布式锁-超时时间的设置



设置超时时间预防死锁，存在锁超时时间设置是否合理的问题

1. 锁超时时间过长，持有有锁的线程挂掉，会造成大量线程阻塞；
2. 锁超时时间过短，有锁的线程业务还没有完成，锁被其他线程意外获得。

锁超时时间，需根据自己的业务场景，可根据经验类比，合理的设置锁超时时间，并预留一定的富余时间。

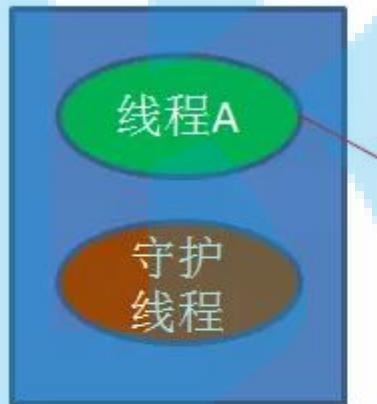
如果持有锁的客户端，如果锁因为达到超时时间，被Redis意外删除，如何解决？

Redis方式实现分布式锁-释放锁-锁续航



获得锁的线程开启一个**守护线程**, 用来给快要过期的锁 “续航”

节点1的JVM进程



同步代码块

XXXXXX
XXXXXX
XXX
XXXXXX

节点2的JVM进程

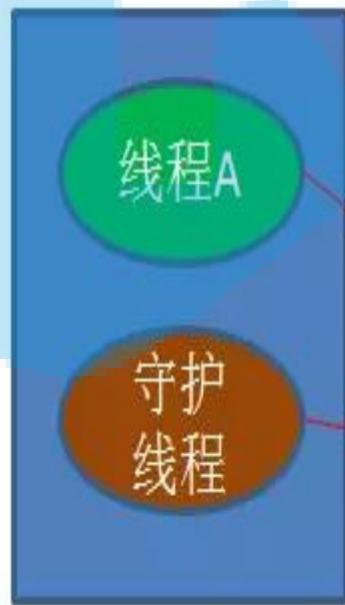


Redis方式实现分布式锁-释放锁-锁续航



过去了29秒，线程A还没执行完，这时候守护线程会执行expire指令，为这把锁“续命”20秒。守护线程从第29秒开始执行，每20秒执行一次

节点1的JVM进程

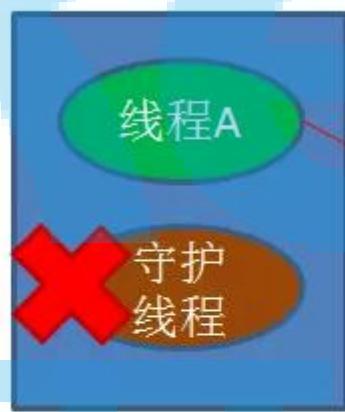


过了29秒，
马上过期
expire
延长20秒

同步代码块
XXXXXX
XXXXX
XXX
XXXXXX

当线程A执行完任务，需同时销毁守护线程。

节点1的JVM进程



终于执行
完了！

节点2的JVM进程



同步代码块

XXXXXX
XXXXX
XXX
XXXXXX

Redis方式实现分布式锁-要点回顾



- 获得锁一定用SET key value NX PX milliseconds 命令
- 锁的value要具有唯一性
- 释放锁一定要使用lua脚本，一定要判断value

开课吧

Redis分布式锁--优缺点



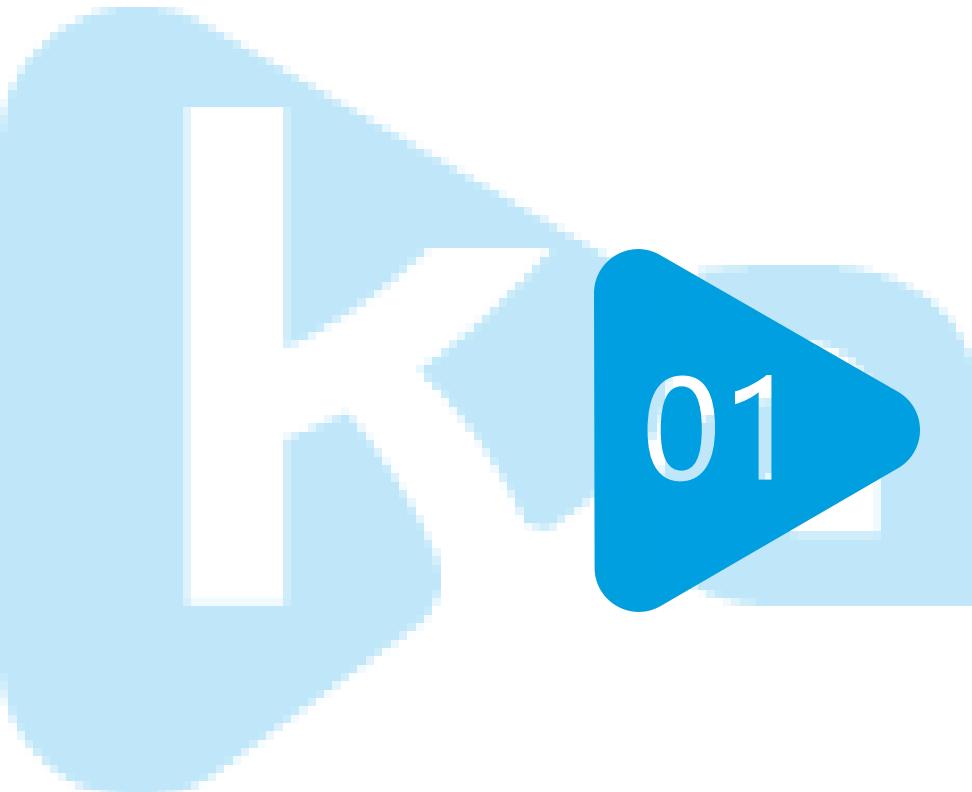
●优点

Redis是基于内存存储，并发性能好。

●缺点

1. 需要考虑原子性、超时、误删等情形。
2. 获锁失败时，客户端只能自旋等待，在高并发情况下，性能消耗比较大。

开课吧



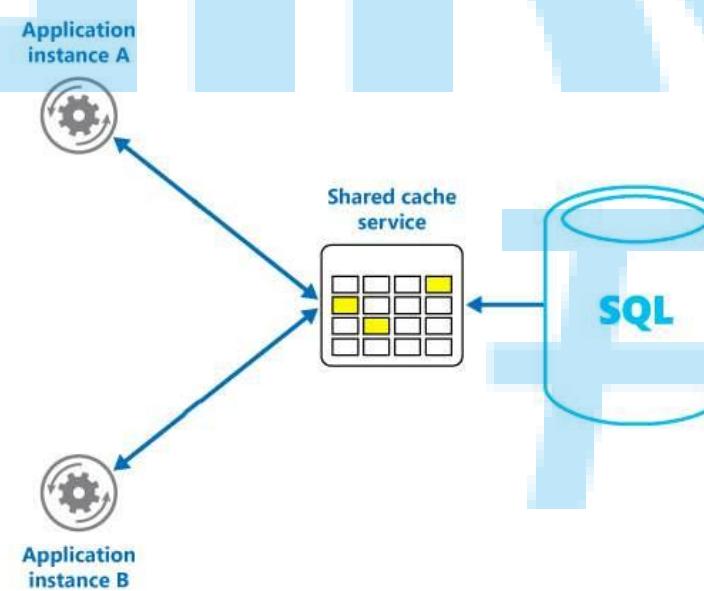
01

缓存通识
keiba

开课吧

| 缓存通识

缓存是高并发场景下提高热点数据访问性能的一个有效手段，在开发项目时会经常使用到。

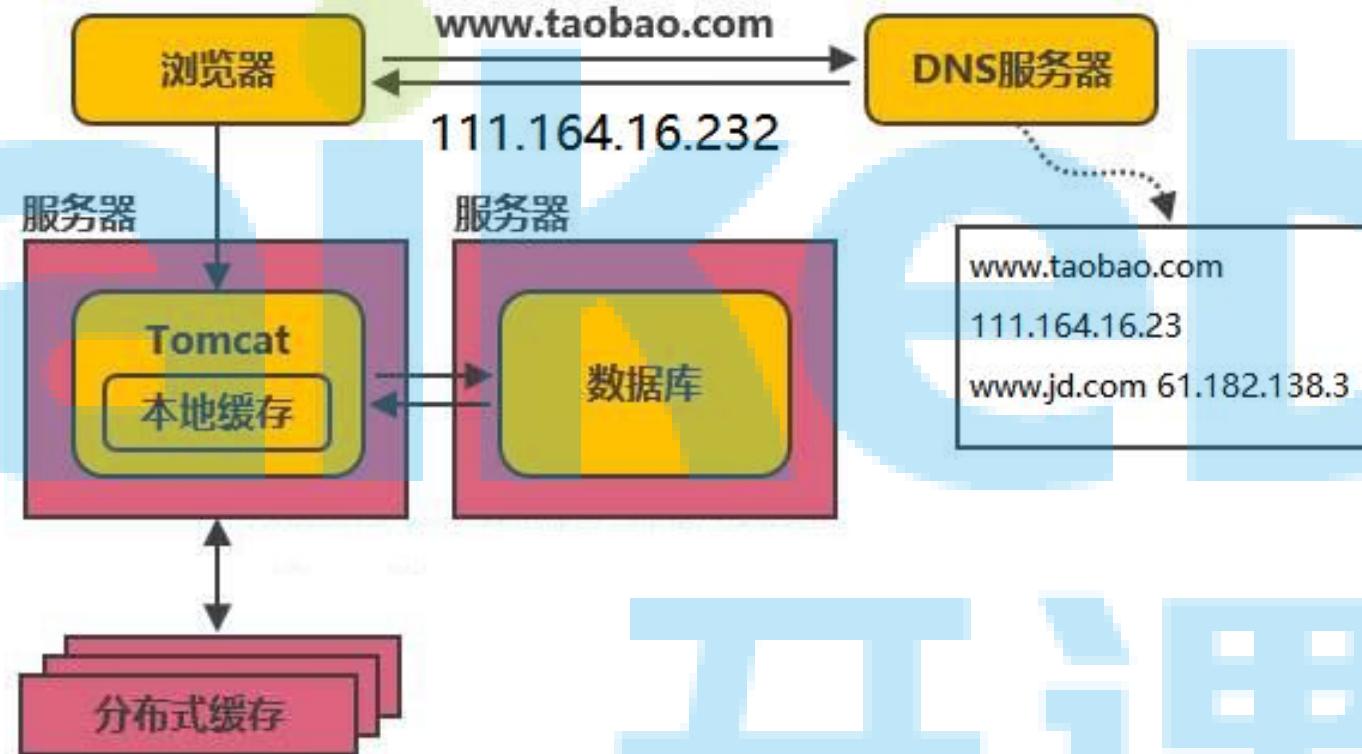


缓存类型

本地缓存

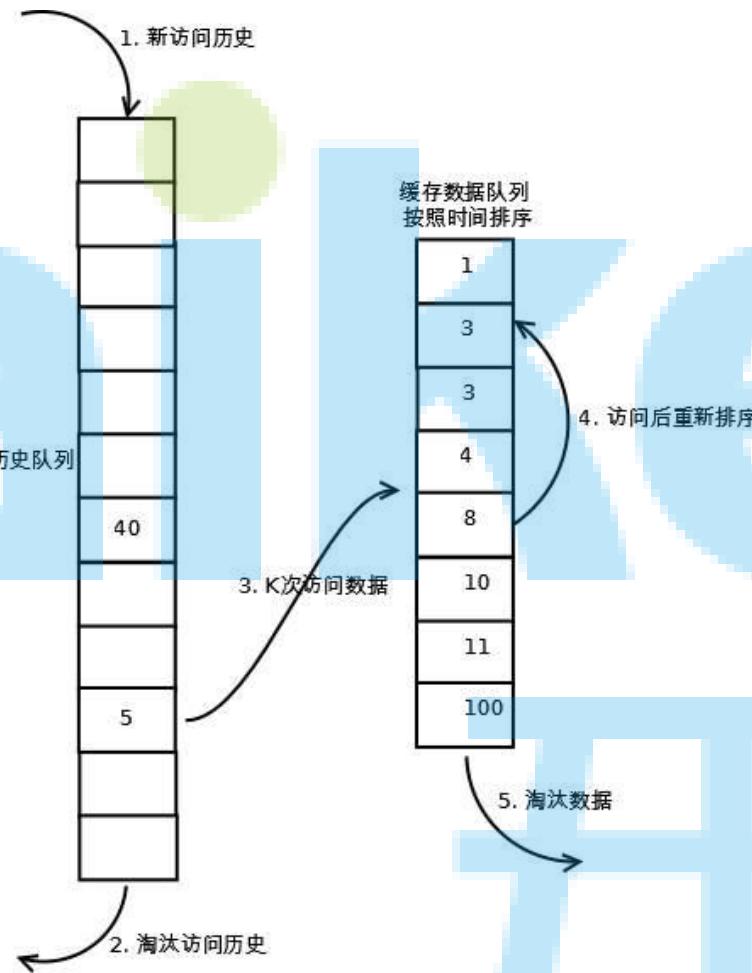
分布式缓存

多级缓存



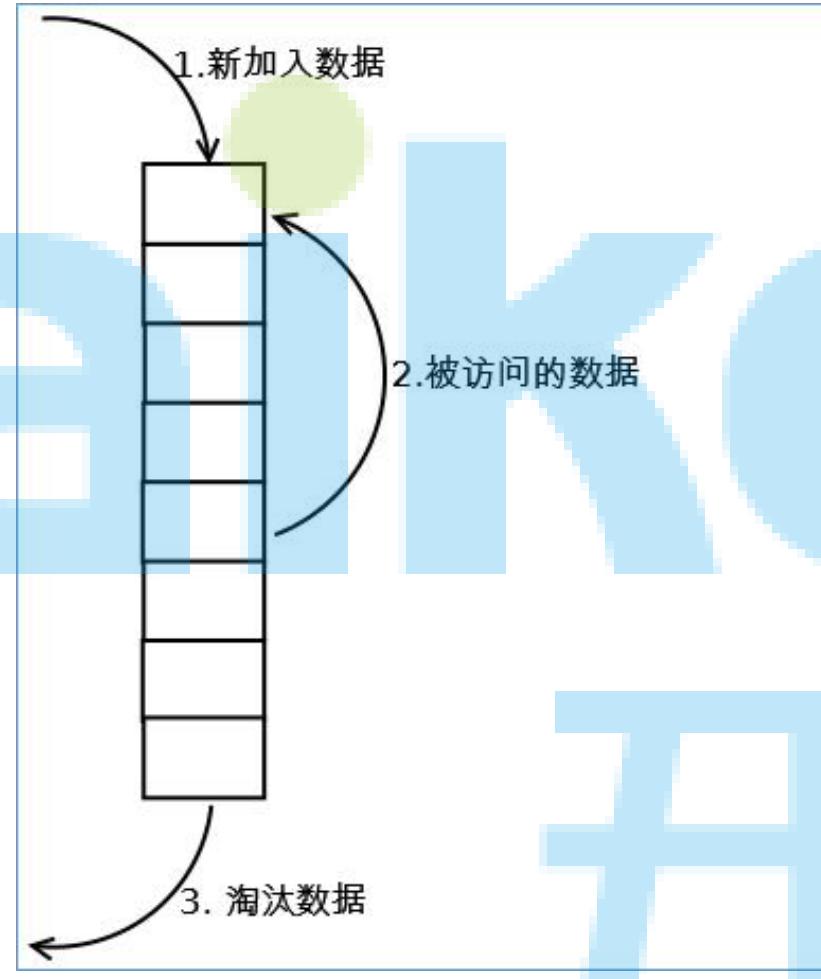
开课吧

淘汰策略-FIFO



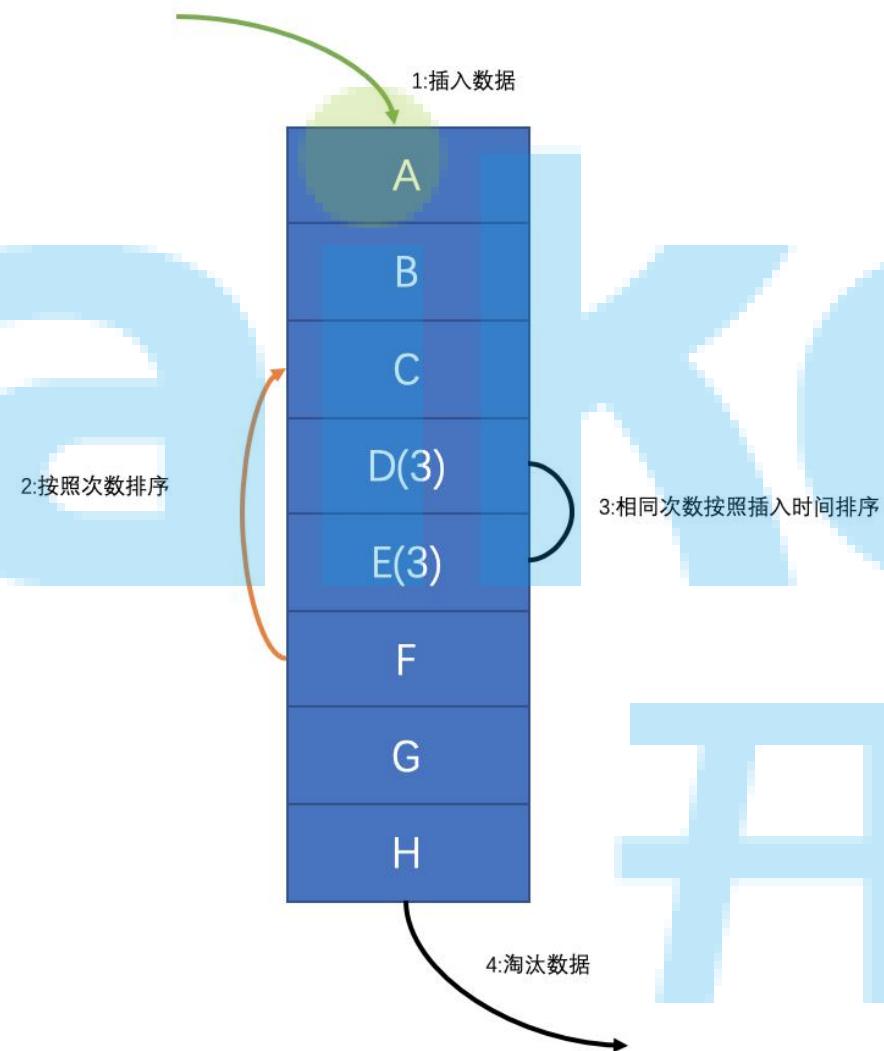
开课吧

| 淘汰策略-LRU



开课吧

淘汰策略-LFU



开课吧

Memcache 和 Redis



redis



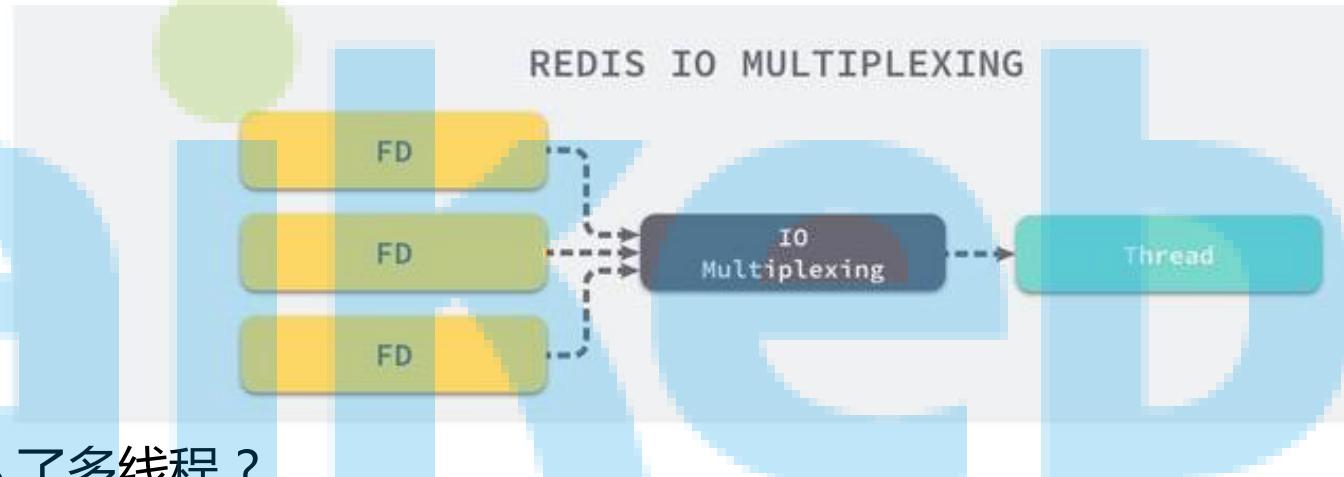
开课吧

Redis6.0 引入了多线程

1 为什么 Redis 一开始选择单线程模型（单线程的好处）？

- 1) IO 多路复用
- 2) 可维护性高
- 3) 基于内存

基于内存而且使用IO多路复用，单线程速度很快，由此保证了多线程的特点。

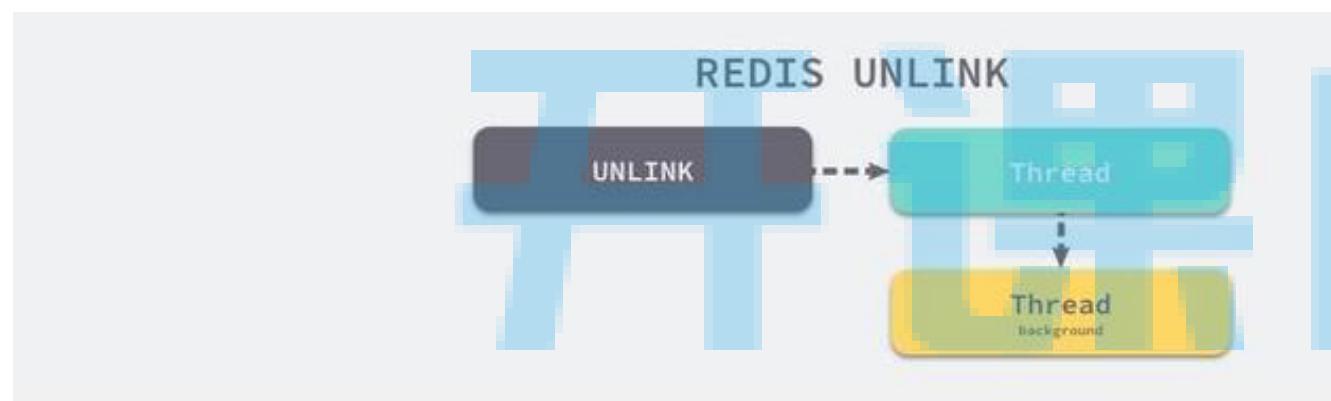


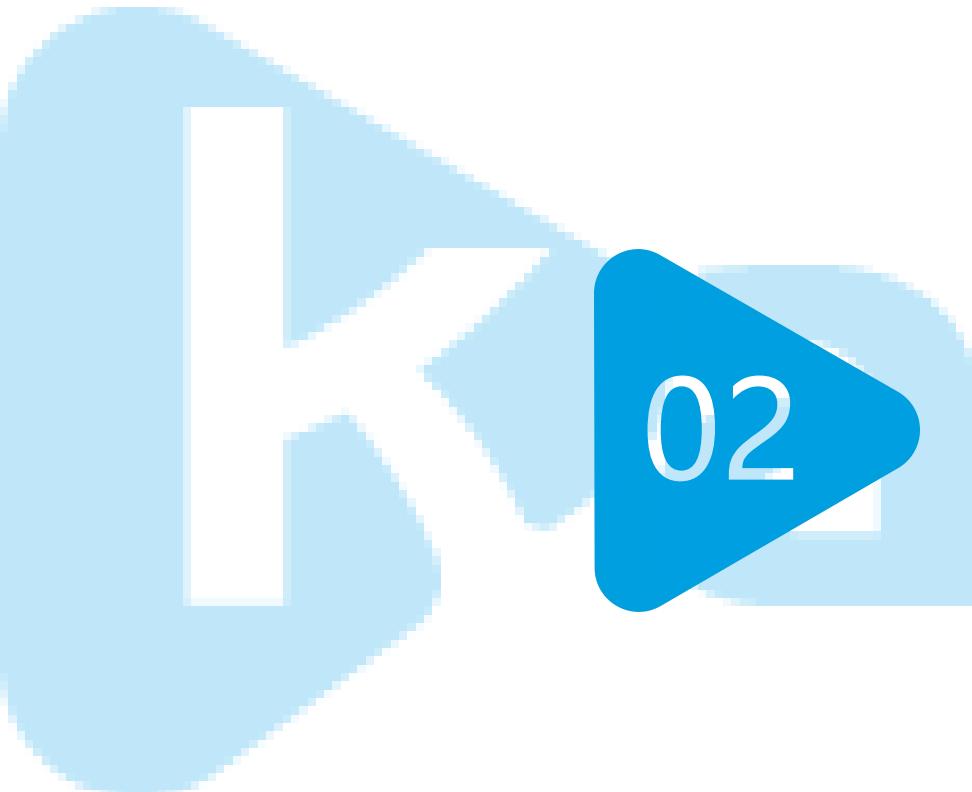
2 为什么 Redis 在 6.0 之后加入了多线程？

网络读写占用了
CPU大部分时间

Redis的多线程 只
是处理网络数据的
读写和协议解析

del



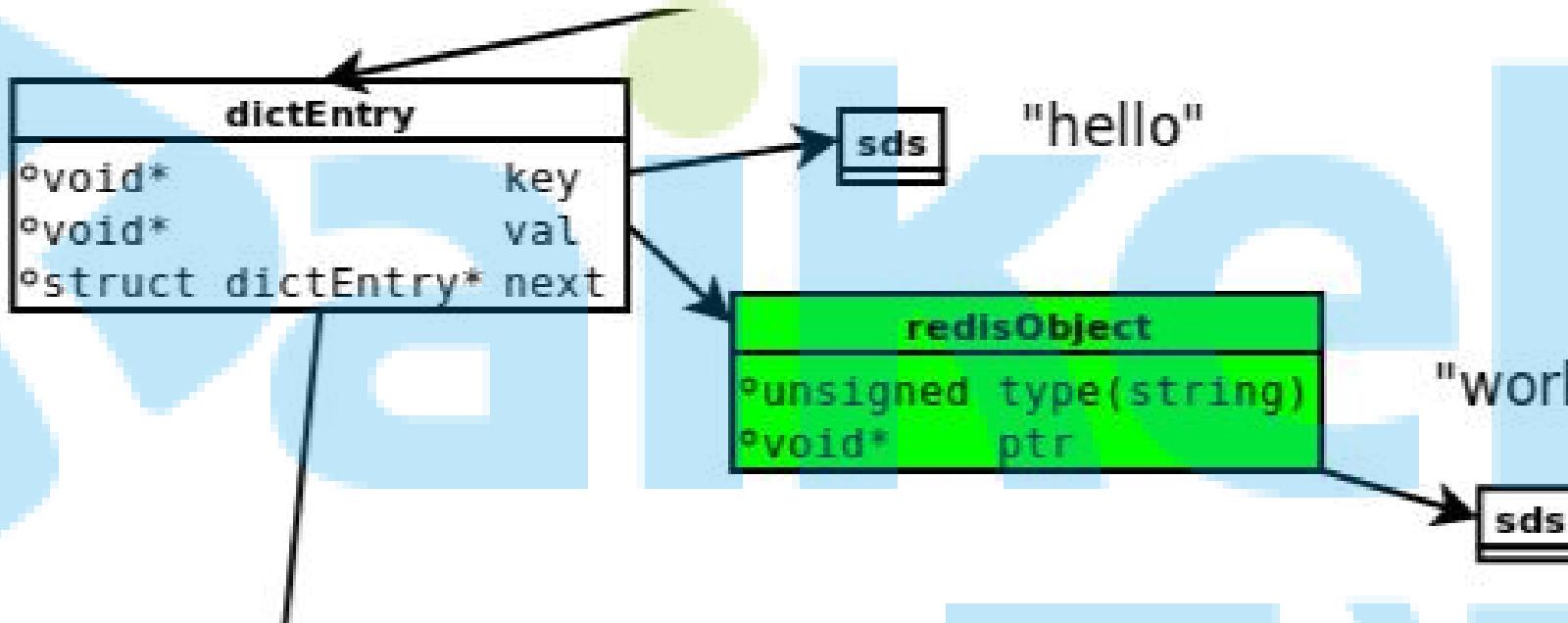


02

设计优化
keiba

开课吧

内存优化-估算Redis内存使用量



开课吧

优化内存占用

- (1) 利用jemalloc特性进行优化
- (2) 使用整型/长整型
- (3) 共享对象
- (4) 缩短键值对的存储长度

开课吧

优化内存占用

数据量	key 大小	value 大小	string:set 平均耗时	hash:hset 平均耗时
100w	20byte	512byte	1.13微秒	10.28微秒
100w	20byte	200byte	0.74微秒	8.08微秒
100w	20byte	100byte	0.65微秒	7.92微秒
100w	20byte	50byte	0.59微秒	6.74微秒
100w	20byte	20byte	0.55微秒	6.60微秒
100w	20byte	5byte	0.53微秒	6.53微秒



03



基于Redis内存 淘汰机制的优化

开课吧

1 设置键值的过期时间

Redis 有四个不同的命令可以用于设置键的生存时间(键可以存在多久)或过期时间(键什么时候会被删除)：

EXPIRE 命令用于将键key 的生存时间设置为ttl 秒。

PEXPIRE 命令用于将键key 的生存时间设置为ttl 毫秒。

EXPIREAT < timestamp> 命令用于将键key 的过期时间设置为timestamp所指定的秒数时间戳。

PEXPIREAT < timestamp > 命令用于将键key 的过期时间设置为timestamp所指定的毫秒数时间戳。

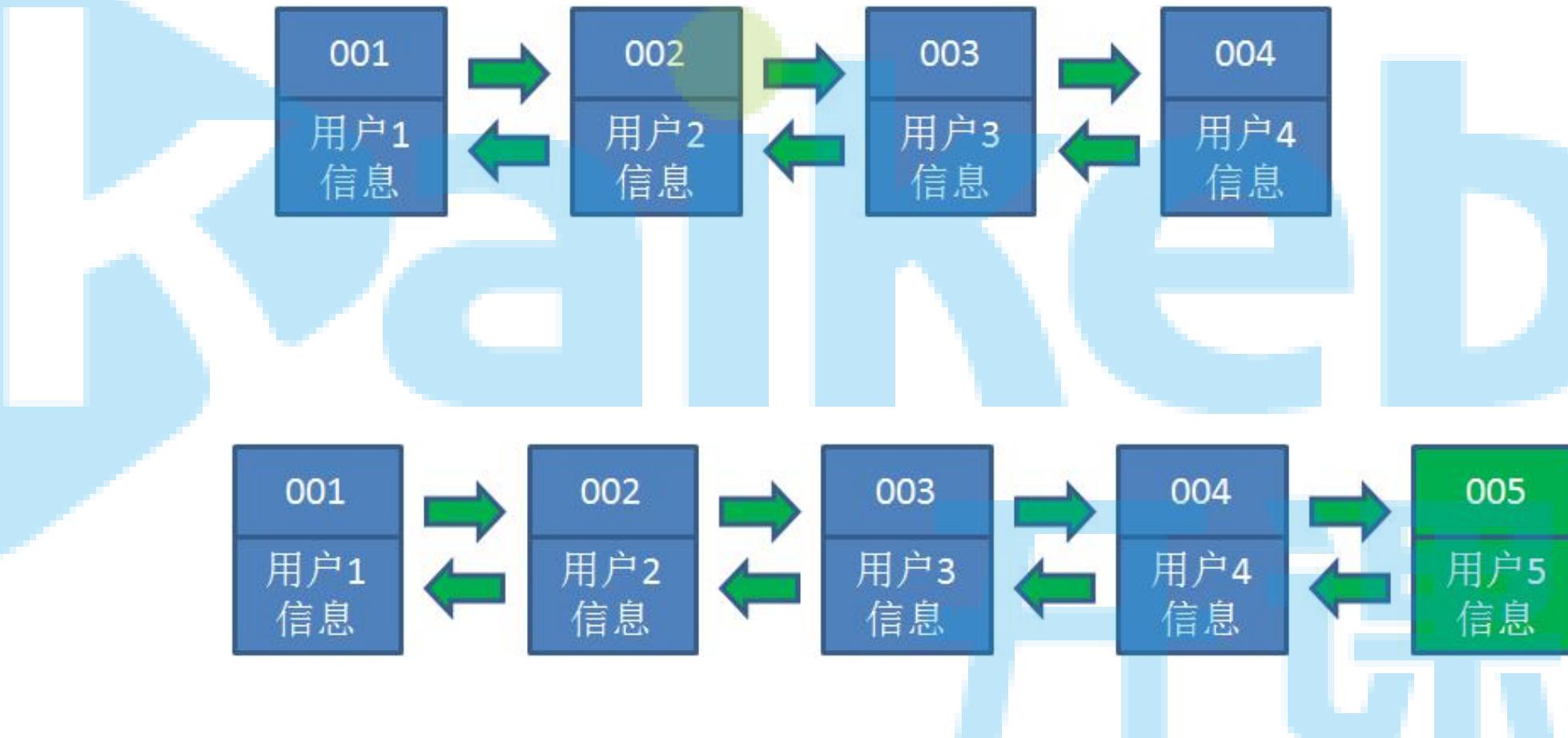
| 2 使用lazy free

lazyfree-lazy-eviction no
lazyfree-lazy-expire no
lazyfree-lazy-server-del no
slave-lazy-flush no

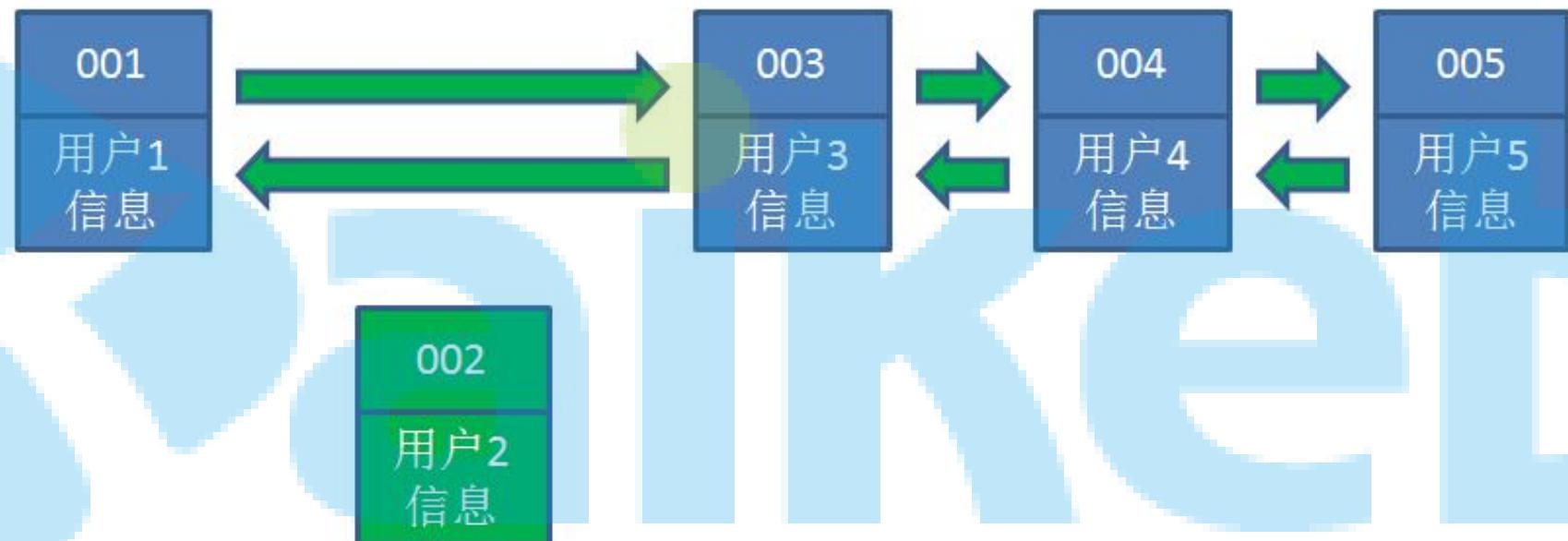
3 限制 Redis 内存大小-最大缓存

- 在 redis 中，允许用户设置最大使用内存大小**maxmemory**，默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一定要设置。
- 在 64 位操作系统中 Redis 的内存大小是没有限制的，也就是配置项 maxmemory 是被注释掉的，这样就会导致在物理内存不足时，使用 swap 空间既交换空间，而当操心系统将 Redis 所用的内存分页移至 swap 空间时，将会阻塞 Redis 进程，导致 Redis 出现延迟，从而影响 Redis 的整体性能。因此我们需要限制 Redis 的内存大小为一个固定的值，当 Redis 的运行到达此值时会触发内存淘汰策略，**内存淘汰策略在 Redis 4.0 之后有 8 种**

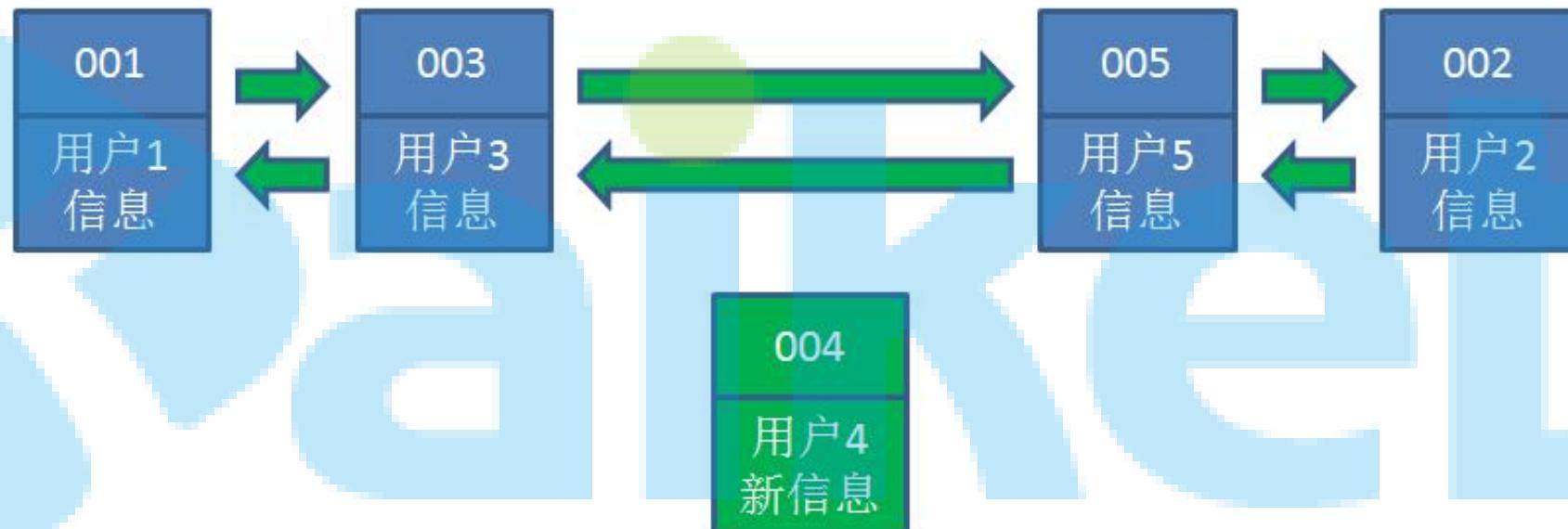
3 限制 Redis 内存大小-缓存淘汰策略



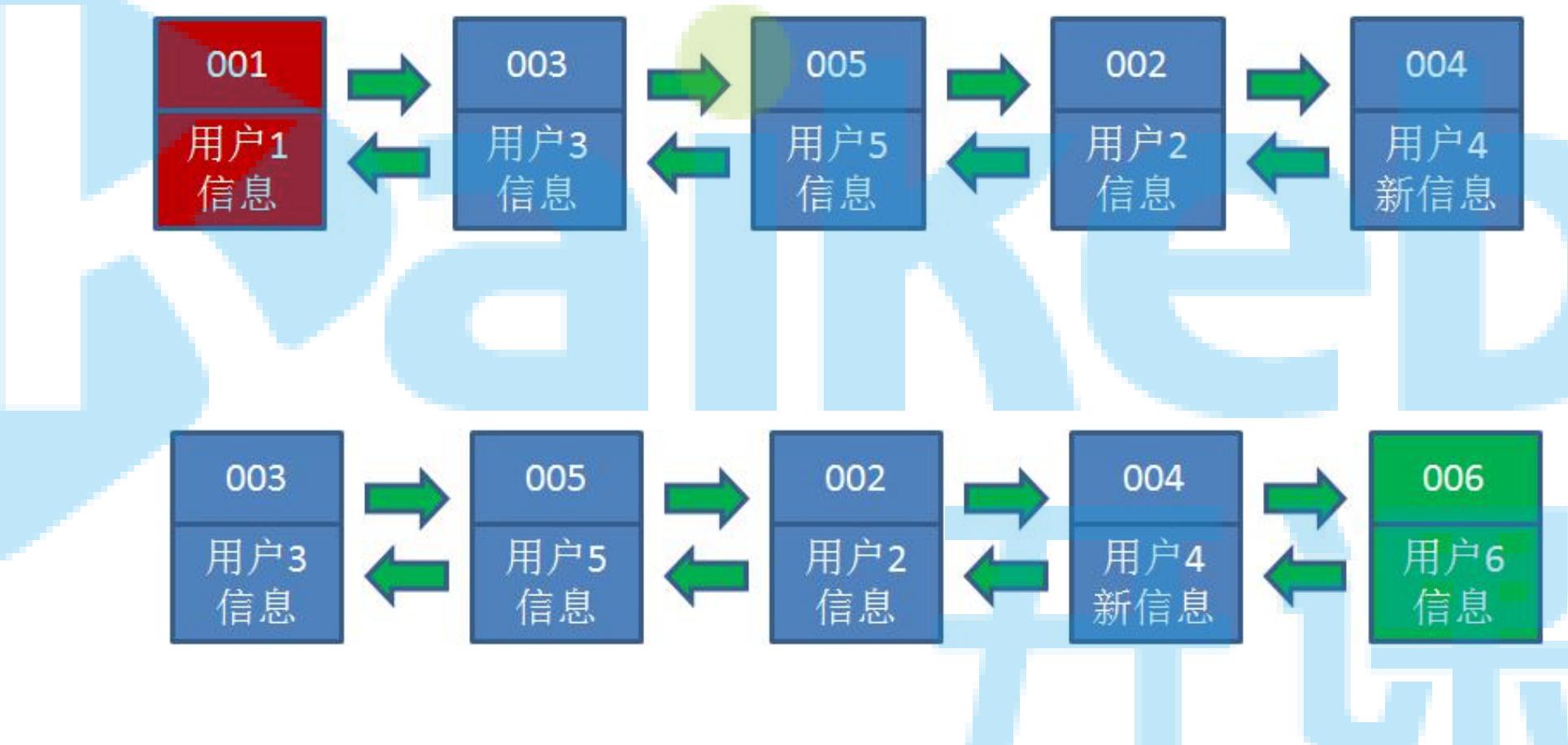
4 限制 Redis 内存大小-缓存淘汰策略



3 限制 Redis 内存大小-缓存淘汰策略



3 限制 Redis 内存大小-缓存淘汰策略



3 限制 Redis 内存大小-缓存淘汰策略

redis淘汰策略配置：maxmemory-policy volatile-lru，支持热配置

1. noeviction: 不淘汰任何数据，当内存不足时，新增操作会报错，Redis 默认内存淘汰策略；
2. allkeys-lru: 淘汰整个键值中最久未使用的键值；
3. allkeys-random: 随机淘汰任意键值；
4. **volatile-lru**: 淘汰所有设置了过期时间的键值中最久未使用的键值；
5. volatile-random: 随机淘汰设置了过期时间的任意键值；
6. volatile-ttl: 优先淘汰更早过期的键值。

在 Redis 4.0 版本中又新增了 2 种淘汰策略：

1. **volatile-lfu**: 淘汰所有设置了过期时间的键值中，最少使用的键值；
2. allkeys-lfu: 淘汰整个键值中最少使用的键值。

其中 allkeys-xxx 表示从所有的键值中淘汰数据，而 volatile-xxx 表示从设置了过期键的键值中淘汰数据。我们可以根据实际的业务情况进行设置，默认的淘汰策略不淘汰任何数据，在新增时会报错。



04



Redis 持久化

开课吧

Redis持久化

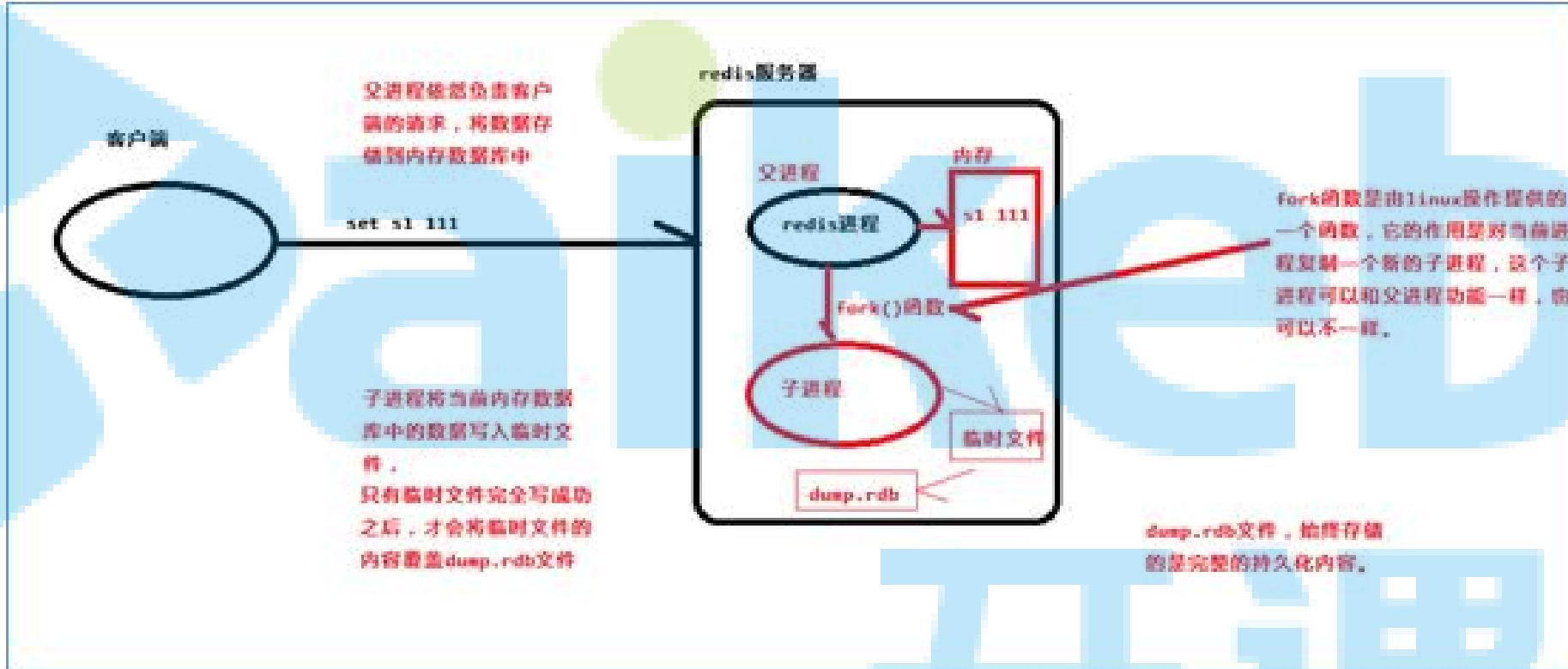
RDB方式 (默认)

AOF方式

混合持久化模式 (5.0后默认开启)

开课吧

Redis持久化-RDB



Redis持久化-AOF

append only file

redis.conf :

```
1 # 可以通过修改redis.conf配置文件中的appendonly参数开启
2 appendonly yes
3
4 # AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的。
5 dir ./ 
6
7 # 默认的文件名是appendonly.aof，可以通过appendfilename参数修改
8 appendfilename appendonly.aof
```

开课吧

Redis持久化-AOF

redis.conf :

```
1 # 可以通过修改redis.conf配置文件中的appendonly参数开启  
2 appendonly yes  
3  
4 # AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的。  
5 dir ./  
6  
7 # 默认的文件名是appendonly.aof，可以通过appendfilename参数修改  
8 appendfilename appendonly.aof
```

开课吧

Redis持久化-混合持久化

4.0 开始有该功能 5.0 默认开启

查询是否开启混合持久化可以使用

config get aof-use-rdb-preamble

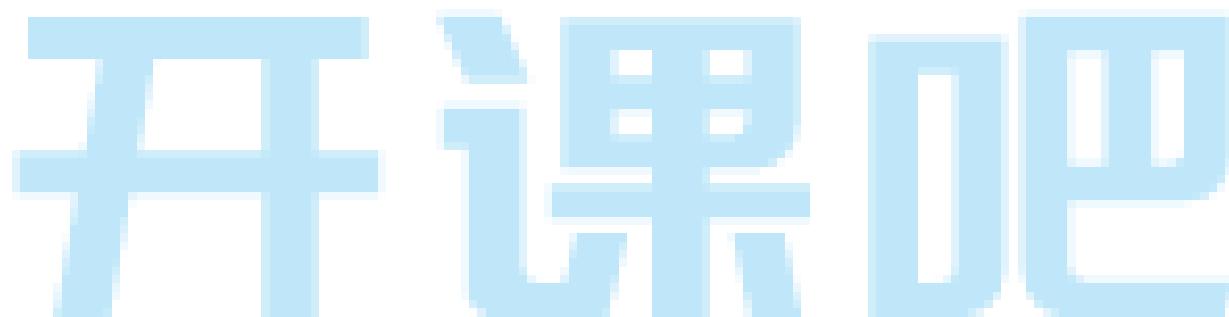
开课吧



05

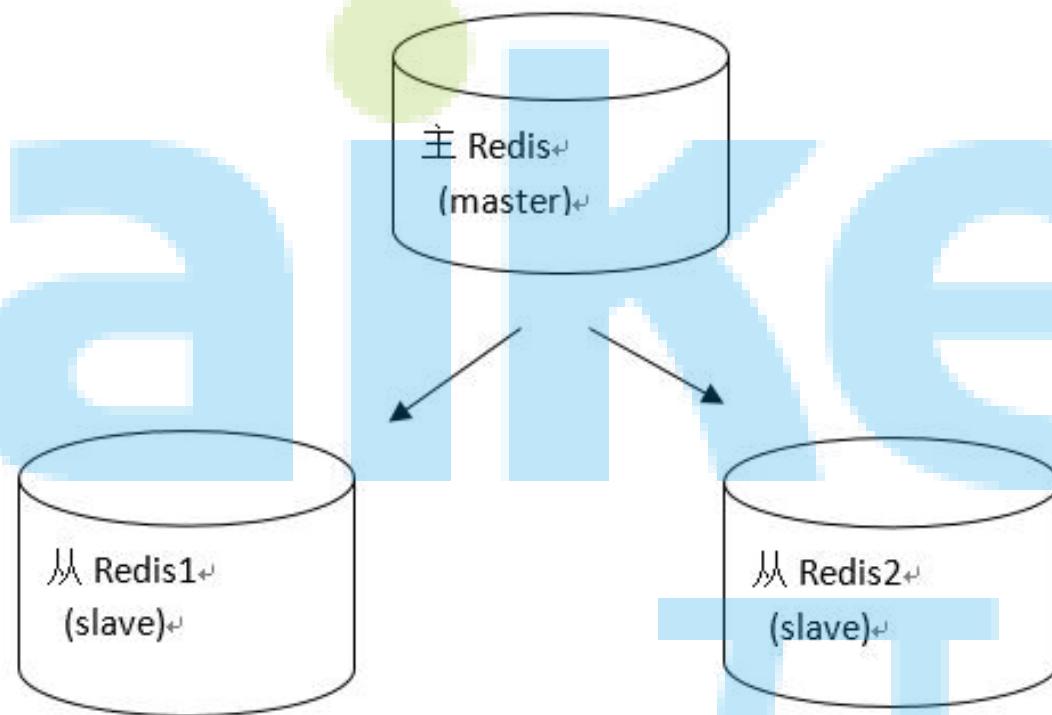


Redis 主从复制



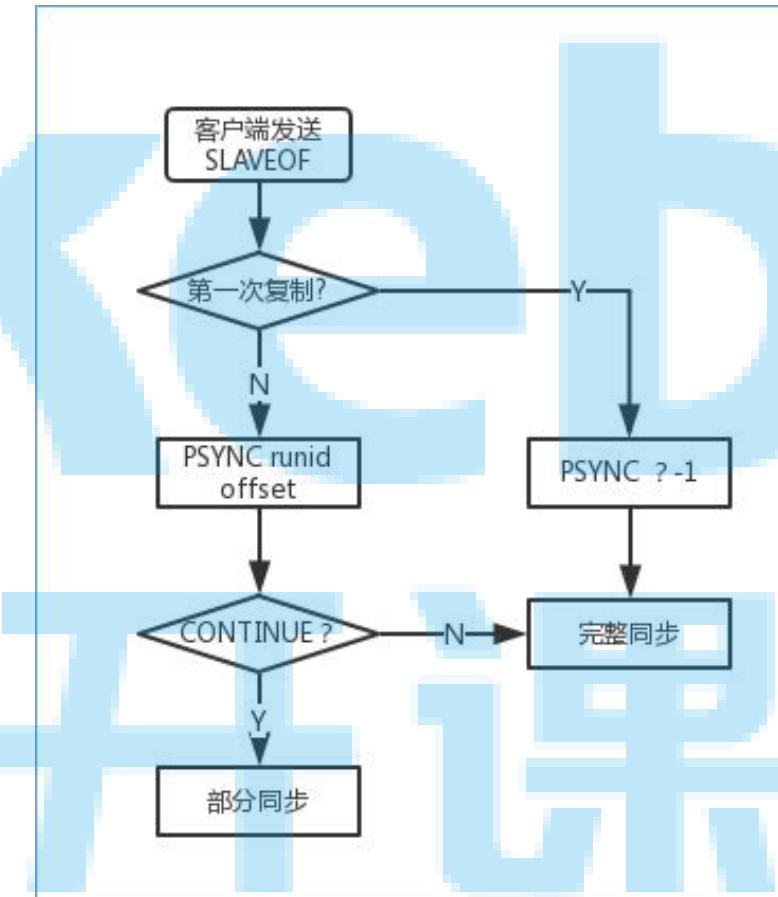
开课吧

Redis主从复制

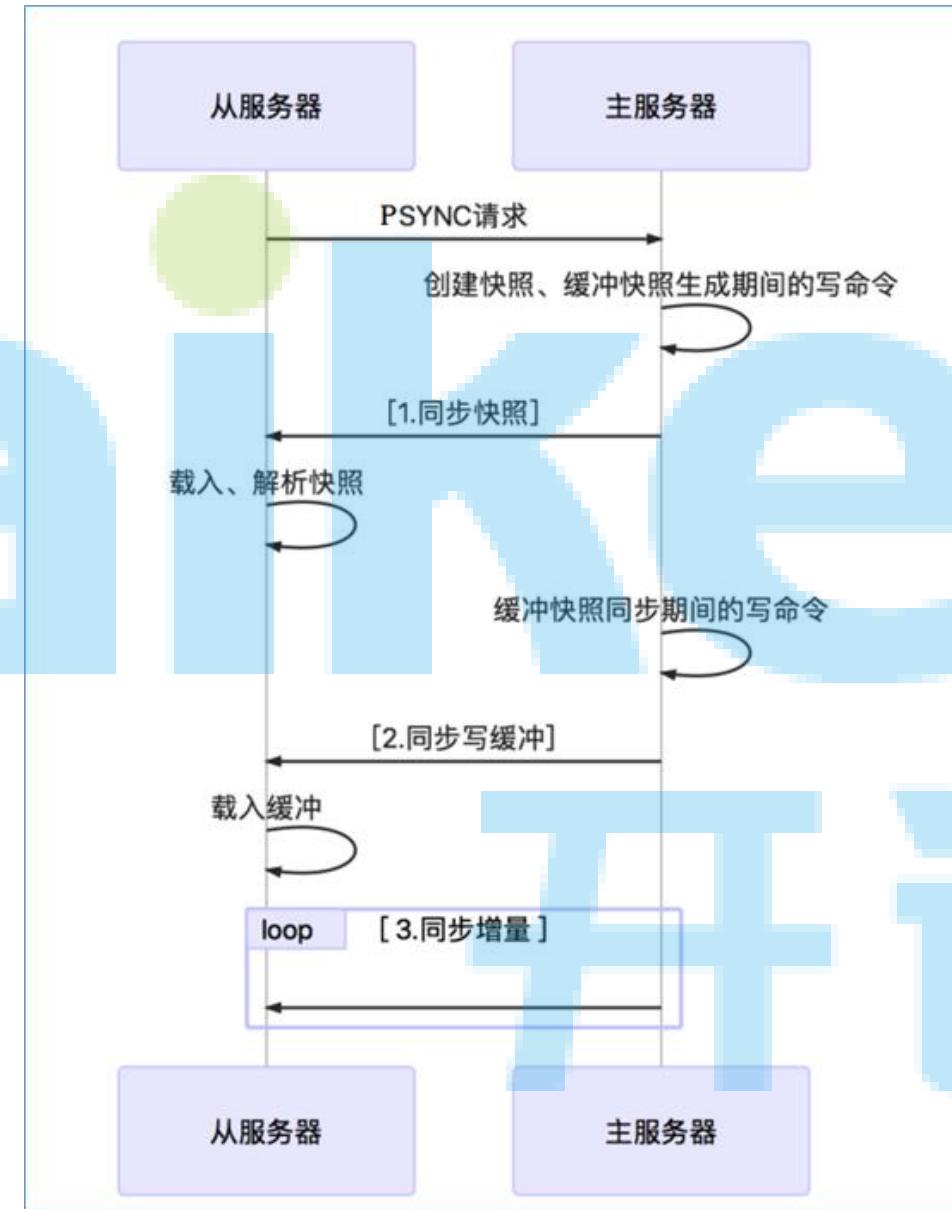


Redis主从复制原理

- Redis`的主从同步，分为**全量同步**和**增量同步**。
- 只有从机第一次连接上主机是**全量同步**。
- 断线重连有可能触发**全量同步**也有可能是**增量同步**（master判断runid是否一致）。
- 除此之外的情况都是**增量同步**。



Redis主从复制-全量同步

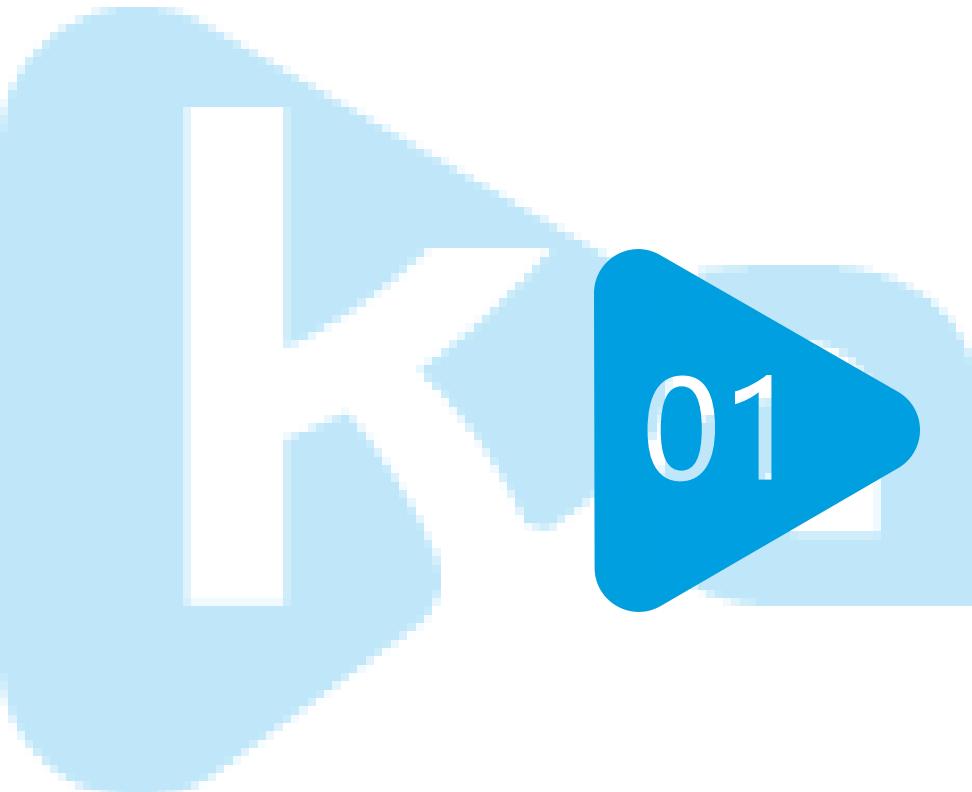


Redis主从复制-从Redis复制

修改从服务器上的 redis.conf 文件：

```
1 # replicaof <masterip> <masterport>
2 # 表示当前【从服务器】对应的【主服务器】的IP是192.168.10.135，端口是6379。
3 slaveof 192.168.10.135 6379
4 replicaof 192.168.10.135 6379
```

开课吧

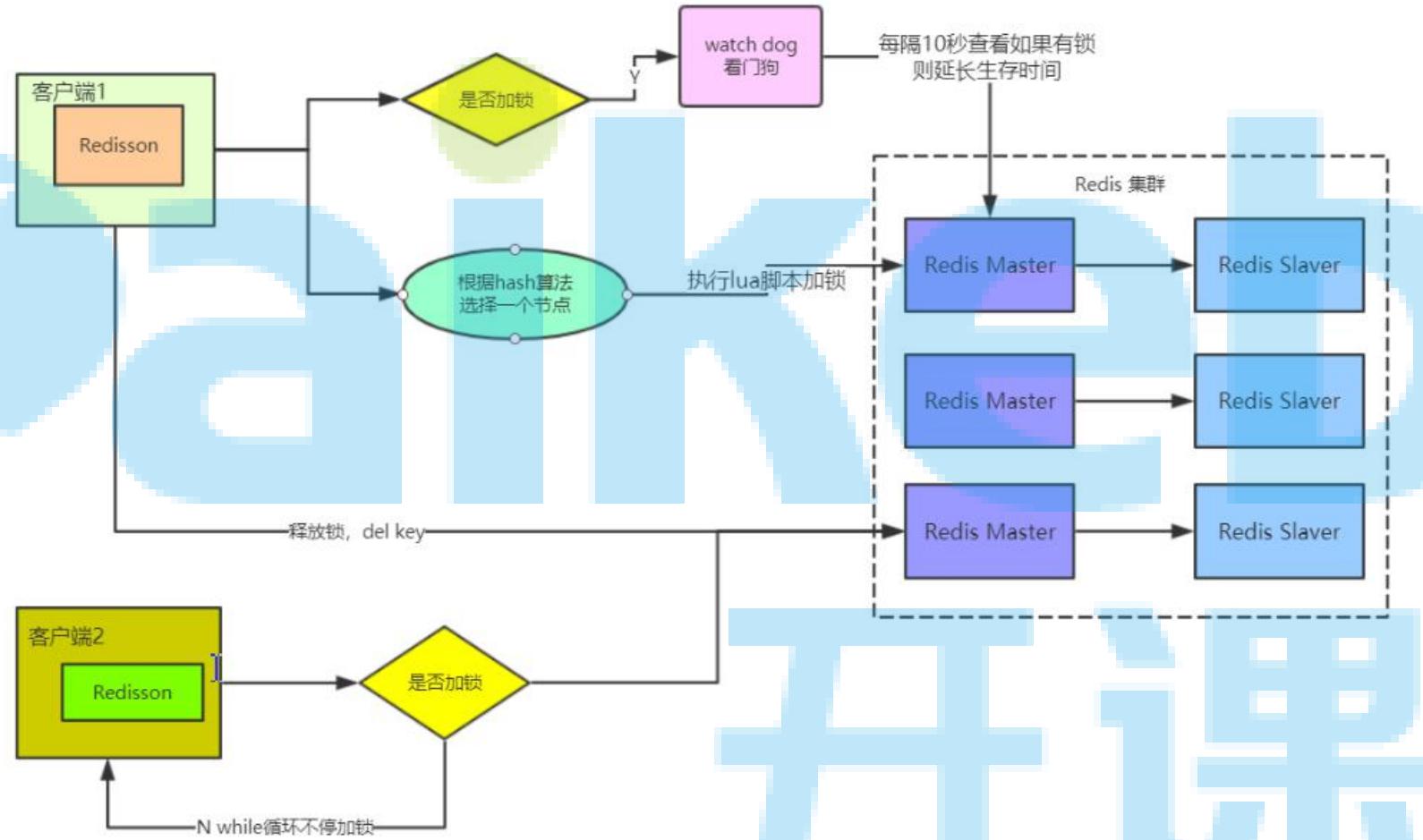


01

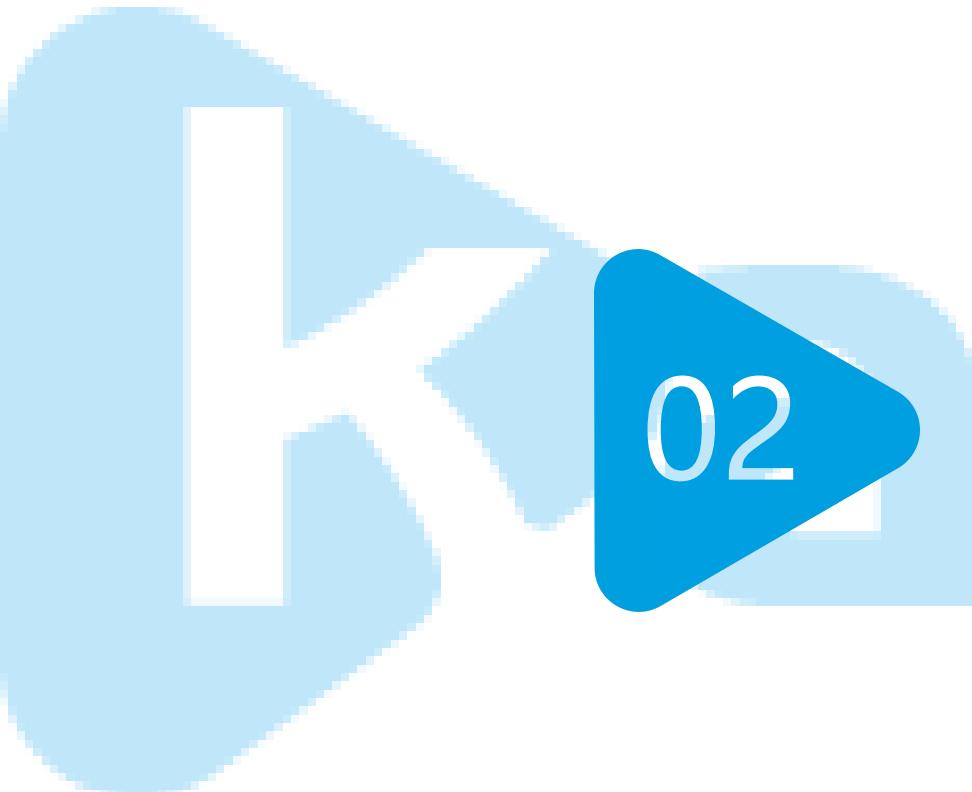
Redis分布式锁

开课吧

Redisson



开课吧



02

Redis性能优化

开课吧

| 禁用长耗时查询命令

KEYS pattern

Available since 1.0.0.

Time complexity: $O(N)$ with N being the number of keys in the database, under the assumption that the key names in the database and the given pattern have limited length.

禁止 keys

避免一次所有 需使用 scan

hash set zset

排序, 交集, 并集 放在客户端进行

del 大数据 建议使用unlink

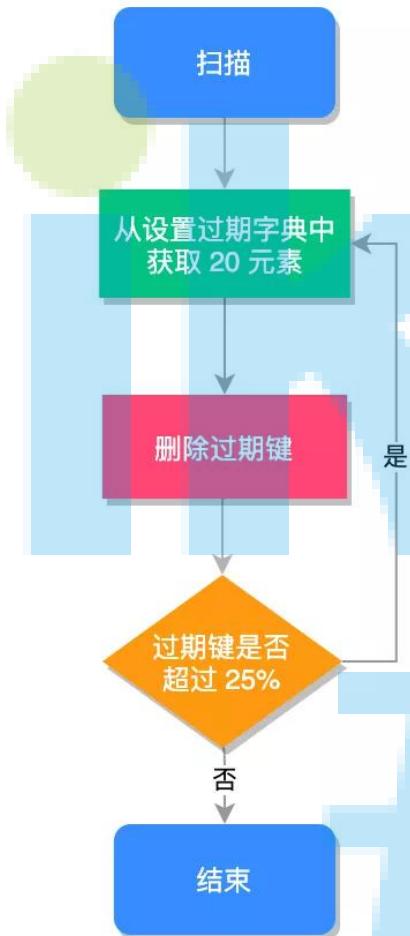
开课吧

| 使用Slowlog 优化耗时命令

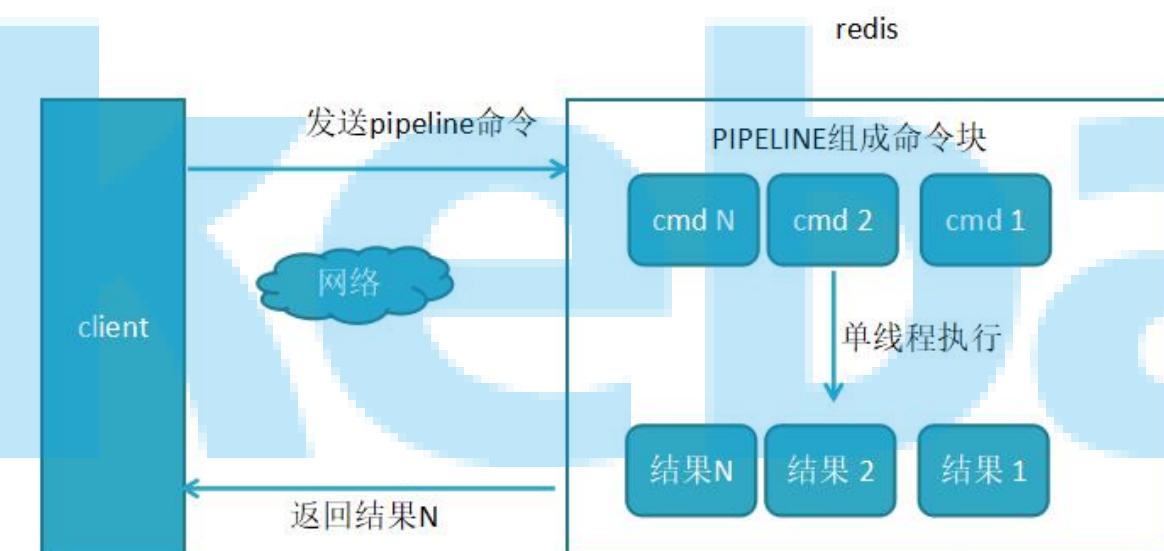
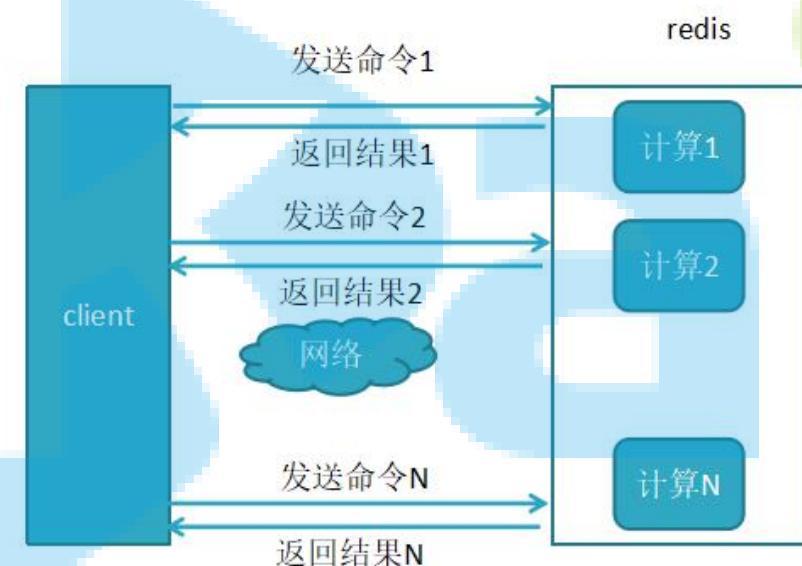
slowlog-log-slower-than : 用于设置慢查询的评定时间，也就是说超过此配置项的命令，将会被当成慢操作记录在慢查询日志中，它执行单位是微秒 (1 秒等于 1000000 微秒);
slowlog-max-len : 用来配置慢查询日志的最大记录数。

开课吧

避免大量数据同时失效



9 使用 Pipeline 批量操作数据



| 客户端使用优化

```
import redis.clients.jedis.JedisPool;  
import redis.clients.jedis.JedisPoolConfig;
```

开课吧

使用分布式架构来增加读写速度

使用主从同步功能我们可以把写入放到主库上执行，把读功能转移到从服务上，因此就可以在单位时间内处理更多的请求，从而提升的 Redis 整体的运行速度。

而哨兵模式是对于主从功能的升级，但当主节点奔溃之后，无需人工干预就能自动恢复 Redis 的正常使用。

Redis Cluster 是 Redis 3.0 正式推出的，Redis 集群是通过将数据库分散存储到多个节点上来平衡各个节点的负载压力。

开课吧

其他优化

12 使用物理机而非虚拟机

通过 `./redis-cli --intrinsic-latency 100` 命令查看延迟时间

13 禁用 THP 特性

```
> echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

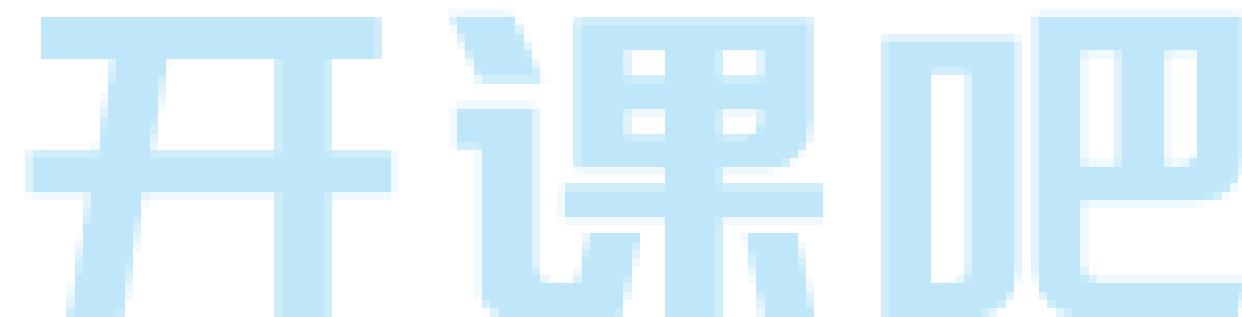
为了使机器重启后 THP 配置依然生效，可以在 `/etc/rc.local` 中追加
`echo never > /sys/kernel/mm/transparent_hugepage/enabled.`



03



Redis常见生产问题



开课吧

| 缓存常见生产问题

缓存雪崩

缓存击穿

缓存穿透

DB-KV一致性



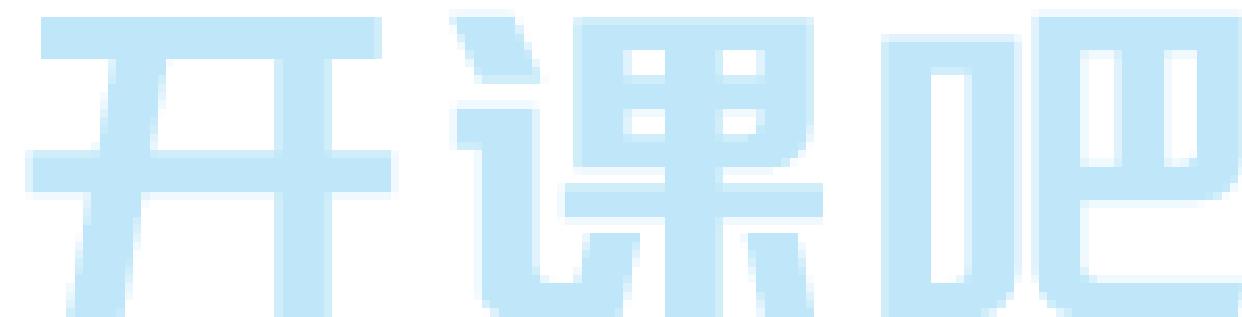
开课吧



04



Redis常见面试问题



开课吧

缓存常见生产问题一

1. Redis 有哪些数据结构
2. 能说一下他们的特性，还有分别的使用场景么？
3. 单机会有瓶颈，怎么解决这个瓶颈？
4. 哪他们之间是如何进行数据交互的？Redis 是如何进行持久化的？Redis 数据都在内存中，一断电或重启不就没有了吗？
5. 哪你是如何选择持久方式的？
6. Redis 还有其他保证集群高可用的方式吗？
7. 数据传输的时候网络断了或者服务器断了，怎么办？
8. 能说一下 Redis 的内存淘汰机制么？
9. 如果的如果，定期没删，我也没查询，那可咋整
10. 哨兵机制的原理是什么？
11. 哨兵组件的主要功能是什么？
12. Redis 的事务原理是什么？
13. Redis 事务为什么是“弱”事务？
14. Redis 为什么没有回滚操作？
15. 在 Redis 中整合 lua 有几种方式，你认为哪种更好，为什么？
16. lua 如何解决 Redis 并发问题？
17. 介绍 Redis 下的分布式锁实现原理、优势劣势和使用场景
18. Redis-Cluster 和 Redis 主从 + 哨兵的区别，你认为哪个更好，为什么。

缓存常见生产问题二

1. 什么情况下会造成缓存穿透，如何解决？
2. 什么情况下会造成缓存雪崩，如何解决？
3. 什么是缓存击穿，如何解决？
4. 什么情况下会造成数据库与Redis缓存数据不一致，如何解决？
5. 那你了解的最经典的KV，DB读写模式什么样？
6. 为什么是删除缓存，而不是更新缓存？

开课吧