

# 数据类型与底层数据结构

## 本章学习目标：

- 掌握Redis五种基本数据类型的用法和常见命令的使用
- 了解bitmap、geo、stream的使用
- 理解Redis底层数据结构（Hash、跳跃表、quicklist）
- 了解RedisDB和RedisObject
- 理解LRU算法
- 理解Redis缓存淘汰策略
- 能够较正确的应用Redis缓存淘汰策略

## Redis数据类型和应用场景

Redis是一个Key-Value的存储系统，使用ANSI C语言编写。

key的类型是字符串。

value的数据类型有：

常用的：string字符串类型、list列表类型、set集合类型、sortedset（zset）有序集合类型、hash类型。

不常见的：bitmap位图类型、geo地理位置类型。

Redis5.0新增一种：stream类型

注意：Redis中命令是忽略大小写，（set SET），key是不忽略大小写的（NAME name）

## Redis的Key的设计

1. 用:分割
2. 把表名转换为key前缀, 比如: user:
3. 第二段放置主键值
4. 第三段放置列名

比如：用户表user, 转换为redis的key-value存储

userid	username	password	email
9	zhangf	111111	<a href="mailto:zhangf@lagou.com">zhangf@lagou.com</a>

username 的 key：user:9:username

{userid:9,username:zhangf}

email的key user:9:email

表示明确：看key知道意思

不易被覆盖

# string字符串类型

Redis的String能表达3种值的类型：字符串、整数、浮点数 100.01 是个六位的串

常见操作命令如下表：

命令名称		命令描述
set	set key value	赋值
get	get key	取值
getset	getset key value	取值并赋值
setnx	setnx key value	当value不存在时采用赋值 set key value NX PX 3000 原子操作，px 设置毫秒数
append	append key value	向尾部追加值
strlen	strlen key	获取字符串长度
incr	incr key	递增数字
incrby	incrby key increment	增加指定的整数
decr	decr key	递减数字
decrby	decrby key decrement	减少指定的整数

应用场景：

- 1、key和命令是字符串
- 2、普通的赋值
- 3、incr用于乐观锁

incr：递增数字，可用于实现乐观锁 watch(事务)

- 4、setnx用于分布式锁

当value不存在时采用赋值，可用于实现分布式锁

举例：

setnx:

```
127.0.0.1:6379> setnx name zhangf      #如果name不存在赋值
(integer) 1
127.0.0.1:6379> setnx name zhaoyun    #再次赋值失败
(integer) 0
127.0.0.1:6379> get name
"zhangf"
```

set

```
127.0.0.1:6379> set age 18 NX PX 10000 #如果不存在赋值 有效期10秒
OK
127.0.0.1:6379> set age 20 NX #赋值失败
(nil)
127.0.0.1:6379> get age #age失效
(nil)
127.0.0.1:6379> set age 30 NX PX 10000 #赋值成功
OK
127.0.0.1:6379> get age
"30"
```

## list列表类型

list列表类型可以存储有序、可重复的元素

获取头部或尾部附近的记录是极快的

list的元素个数最多为 $2^{32}-1$ 个 ( 40亿 )

常见操作命令如下表：

命令名称	命令格式	描述
lpush	lpush key v1 v2 v3 ...	从左侧插入列表
lpop	lpop key	从列表左侧取出
rpush	rpush key v1 v2 v3 ...	从右侧插入列表
rpop	rpop key	从列表右侧取出
lpushx	lpushx key value	将值插入到列表头部
rpushx	rpushx key value	将值插入到列表尾部
blpop	blpop key timeout	从列表左侧取出，当列表为空时阻塞，可以设置最大阻塞时间，单位为秒
brpop	brpop key timeout	从列表右侧取出，当列表为空时阻塞，可以设置最大阻塞时间，单位为秒
llen	llen key	获得列表中元素个数
lindex	lindex key index	获得列表中下标为index的元素 index从0开始
lrange	lrange key start end	返回列表中指定区间的元素，区间通过start和end指定
lrem	lrem key count value	删除列表中与value相等的元素 当count>0时，lrem会从列表左边开始删除;当count<0时，lrem会从列表后边开始删除;当count=0时，lrem删除所有值为value的元素
lset	lset key index value	将列表index位置的元素设置成value的值
ltrim	ltrim key start end	对列表进行修剪，只保留start到end区间
rpoplpush	poplpush key1 key2	从key1列表右侧弹出并插入到key2列表左侧
brpoplpush	brpoplpush key1 key2	从key1列表右侧弹出并插入到key2列表左侧，会阻塞
linsert	linsert key BEFORE/AFTER pivot value	将value插入到列表，且位于值pivot之前或之后

应用场景：

1、作为栈或队列使用

列表有序可以作为栈和队列使用

2、可用于各种列表，比如用户列表、商品列表、评论列表等。

举例：

```
127.0.0.1:6379> lpush list:1 1 2 3 4 5 3
(integer) 5
127.0.0.1:6379> lrange list:1 0 -1
1) "5"
2) "4"
3) "3"
4) "2"
5) "1"
127.0.0.1:6379> lpop list:1 # 从0开始
"5"
127.0.0.1:6379> rpop list:1
"1"
127.0.0.1:6379> lindex list:1 1
"3"
127.0.0.1:6379> lrange list:1 0 -1
1) "4"
2) "3"
3) "2"
127.0.0.1:6379> lindex list:1 1
"3"
127.0.0.1:6379> rpoplpush list:1 list:2
"2"
127.0.0.1:6379> lrange list:2 0 -1
1) "2"
127.0.0.1:6379> lrange list:1 0 -1
1) "4"
2) "3"
```

## set集合类型

Set：无序、唯一元素

集合中最大的成员数为  $2^{32} - 1$

常见操作命令如下表：

命令名称	命令格式	描述
sadd	sadd key mem1 mem2 ....	为集合添加新成员
srem	srem key mem1 mem2 ....	删除集合中指定成员
smembers	smembers key	获得集合中所有元素
spop	spop key	返回集合中一个随机元素，并将该元素删除
randmember	randmember key	返回集合中一个随机元素，不会删除该元素
scard	scard key	获得集合中元素的数量
sismember	sismember key member	判断元素是否在集合内
sinter	sinter key1 key2 key3	求多集合的交集
sdiff	sdiff key1 key2 key3	求多集合的差集
sunion	sunion key1 key2 key3	求多集合的并集

应用场景：

适用于不能重复的且不需要顺序的数据结构

比如：关注的用户，还可以通过spop进行随机抽奖

举例：

```
127.0.0.1:6379> sadd set:1 a b c d
(integer) 4
127.0.0.1:6379> smembers set:1
1) "d"
2) "b"
3) "a"
4) "c"
127.0.0.1:6379> srandmember set:1
"c"
127.0.0.1:6379> srandmember set:1
"b"
127.0.0.1:6379> sadd set:2 b c r f
(integer) 4
127.0.0.1:6379> sinter set:1 set:2
1) "b"
2) "c"
127.0.0.1:6379> spop set:1
"d"
127.0.0.1:6379> smembers set:1
1) "b"
2) "a"
3) "c"
```

## sortedset有序集合类型

SortedSet(ZSet) 有序集合：元素本身是无序不重复的

每个元素关联一个分数(score)

可按分数排序，分数可重复

常见操作命令如下表：

命令名称	命令格式	描述
zadd	zadd key score1 member1 score2 member2 ...	为有序集合添加新成员
zrem	zrem key mem1 mem2 ....	删除有序集合中指定成员
zcard	zcard key	获得有序集合中的元素数量
zcount	zcount key min max	返回集合中score值在[min,max]区间的元素数量
zincrby	zincrby key increment member	在集合的member分值上加increment
zscore	zscore key member	获得集合中member的分值
zrank	zrank key member	获得集合中member的排名（按分值从小到大）
zrevrank	zrevrank key member	获得集合中member的排名（按分值从大到小）
zrange	zrange key start end	获得集合中指定区间成员，按分数递增排序
zrevrange	zrevrange key start end	获得集合中指定区间成员，按分数递减排序

应用场景：

由于可以按照分值排序，所以适用于各种排行榜。比如：点击排行榜、销量排行榜、关注排行榜等。

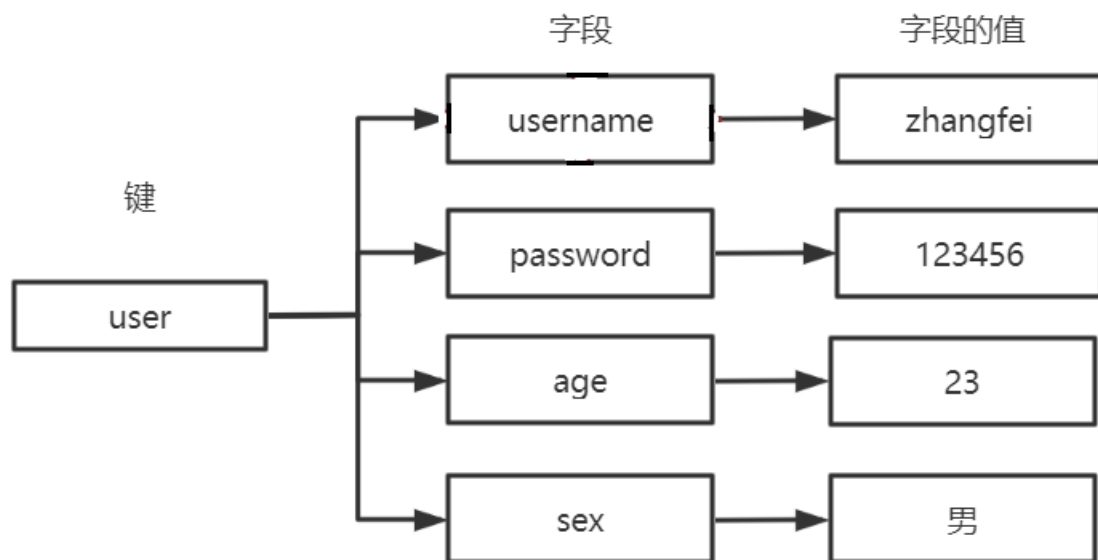
举例：

```
127.0.0.1:6379> zadd hit:1 100 item1 20 item2 45 item3
(integer) 3
127.0.0.1:6379> zcard hit:1
(integer) 3
127.0.0.1:6379> zscore hit:1 item3
"45"
127.0.0.1:6379> zrevrange hit:1 0 -1
1) "item1"
2) "item3"
3) "item2"
127.0.0.1:6379>
```

## hash类型（散列表）

Redis hash 是一个 string 类型的 field 和 value 的映射表，它提供了字段和字段值的映射。

每个 hash 可以存储  $2^{32} - 1$  键值对（40多亿）。



常见操作命令如下表：

命令名称	命令格式	描述
hset	hset key field value	赋值，不区别新增或修改
hmset	hmset key field1 value1 field2 value2	批量赋值
hsetnx	hsetnx key field value	赋值，如果field存在则不操作
hexists	hexists key field	查看某个field是否存在
hget	hget key field	获取一个字段值
hmget	hmget key field1 field2 ...	获取多个字段值
hgetall	hgetall key	
hdel	hdel key field1 field2...	删除指定字段
hincrby	hincrby key field increment	指定字段自增increment
hlen	hlen key	获得字段数量

应用场景：

对象的存储，表数据的映射

举例：

```
127.0.0.1:6379> hmset user:001 username zhangfei password 111 age 23 sex M
OK
127.0.0.1:6379> hgetall user:001
1) "username"
2) "zhangfei"
3) "password"
4) "111"
5) "age"
6) "23"
7) "sex"
```



```
8) "M"
127.0.0.1:6379> hget user:001 username
"zhangfei"
127.0.0.1:6379> hincrby user:001 age 1
(integer) 24
127.0.0.1:6379> hlen user:001
(integer) 4
```

## Jedis客户端操作，安装启动卸载

普通命令

## bitmap位图类型

bitmap是进行位操作的

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。

bitmap本身会极大的节省储存空间。

常见操作命令如下表：

命令名称	命令格式	描述
setbit	setbit key offset value	设置key在offset处的bit值(只能是0或者1)。
getbit	getbit key offset	获得key在offset处的bit值
bitcount	bitcount key	获得key的bit位为1的个数
bitpos	bitpos key value	返回第一个被设置为bit值的索引值
bitop	bitop and[or/xor/not] destkey key [key ...]	对多个key 进行逻辑运算后存入destkey 中

应用场景：

- 1、用户每月签到，用户id为key，日期作为偏移量 1表示签到
- 2、统计活跃用户,日期为key，用户id为偏移量 1表示活跃
- 3、查询用户在线状态，日期为key，用户id为偏移量 1表示在线

举例：

```
127.0.0.1:6379> setbit user:sign:1000 20200101 1 #id为1000的用户20200101签到
(integer) 0
127.0.0.1:6379> setbit user:sign:1000 20200103 1 #id为1000的用户20200103签到
(integer) 0
127.0.0.1:6379> getbit user:sign:1000 20200101 #获得id为1000的用户20200101签到状态
1 表示签到
(integer) 1
127.0.0.1:6379> getbit user:sign:1000 20200102 #获得id为1000的用户20200102签到状态
0表示未签到
(integer) 0
```

```
127.0.0.1:6379> bitcount user:sign:1000 # 获得id为1000的用户签到次数
(integer) 2
127.0.0.1:6379> bitpos user:sign:1000 1 #id为1000的用户第一次签到的日期
(integer) 20200101
127.0.0.1:6379> setbit 20200201 1000 1 #20200201的1000号用户上线
(integer) 0
127.0.0.1:6379> setbit 20200202 1001 1 #20200202的1000号用户上线
(integer) 0
127.0.0.1:6379> setbit 20200201 1002 1 #20200201的1002号用户上线
(integer) 0
127.0.0.1:6379> bitcount 20200201 #20200201的上线用户有2个
(integer) 2
127.0.0.1:6379> bitop or desk1 20200201 20200202 #合并20200201的用户和20200202上线了的用户
(integer) 126
127.0.0.1:6379> bitcount desk1 #统计20200201和20200202都上线的用户个数
(integer) 3
```

## geo地理位置类型

geo是Redis用来处理位置信息的。在Redis 3.2中正式使用。主要是利用了Z阶曲线、Base32编码和geohash算法

### Z阶曲线

在x轴和y轴上将十进制数转化为二进制数，采用x轴和y轴对应的二进制数依次交叉后得到一个六位数编码。把数字从小到大依次连起来的曲线称为Z阶曲线，Z阶曲线是把多维转换成一维的一种方法。

	x:	0	1	2	3	4	5	6	7
		000	001	010	011	100	101	110	111
y: 0	000	000000	000001	000100	000101	010000	010001	010100	010101
1	001	000010	000011	000110	000111	010010	010011	010110	010111
2	010	001000	001001	001100	001101	011000	011001	011100	011101
3	011	001010	001011	001110	001111	011010	011011	011110	011111
4	100	100000	100001	100100	100101	110000	110001	110100	110101
5	101	100010	100011	100110	100111	110010	110011	110110	110111
6	110	101000	101001	101100	101101	111000	111001	111100	111101
7	111	101010	101011	101110	101111	111010	111011	111110	111111

## Base32编码

Base32这种数据编码机制，主要用来把二进制数据编码成可见的字符串，其编码规则是：任意给定一个二进制数据，以5个位(bit)为一组进行切分(base64以6个位(bit)为一组)，对切分而成的每个组进行编码得到1个可见字符。Base32编码表字符集中的字符总数为32个（0-9、b-z去掉a、i、l、o），这也是Base32名字的由来。

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 32	0	1	2	3	4	5	6	7	8	9	b	c	d	e	f	g
Decimal	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Base 32	h	j	k	m	n	p	q	r	s	t	u	v	w	x	y	z

## geohash算法

Gustavo在2008年2月上线了geohash.org网站。Geohash是一种地理位置信息编码方法。经过geohash映射后，地球上任意位置的经纬度坐标可以表示成一个较短的字符串。可以方便的存储在数据库中，附在邮件上，以及方便的使用在其他服务中。以北京的坐标举例，[39.928167,116.389550]可以转换成 wx4g0s8q3jf9。

Redis中经纬度使用52位的整数进行编码，放进zset中，zset的value元素是key，score是GeoHash的52位整数值。在使用Redis进行Geo查询时，其内部对应的操作其实只是zset(skiplist)的操作。通过zset的score进行排序就可以得到坐标附近的其它元素，通过将score还原成坐标值就可以得到元素的原始坐标。

常见操作命令如下表：

命令名称	命令格式	描述
geoadd	geoadd key 经度 纬度 成员名称1 经度1 纬度1 成员名称2 经度2 纬度 2 ...	添加地理坐标
geohash	geohash key 成员名称1 成员名称2...	返回标准的geohash串
geopos	geopos key 成员名称1 成员名称2...	返回成员经纬度
geodist	geodist key 成员1 成员2 单位	计算成员间距离
georadiusbymember	georadiusbymember key 成员 值单位 count 数 asc[desc]	根据成员查找附近的成员

应用场景：

- 1、记录地理位置
- 2、计算距离
- 3、查找"附近的人"

举例：

```
127.0.0.1:6379> geoadd user:addr 116.31 40.05 zhangf 116.38 39.88 zhaoyun 116.47 40.00 diaochan #添加用户地址 zhangf、zhaoyun、diaochan的经纬度
(integer) 3
127.0.0.1:6379> geohash user:addr zhangf diaochan #获得zhangf和diaochan的geohash
码
1) "wx4eydyk5m0"
2) "wx4gd3fbgs0"
127.0.0.1:6379> geopos user:addr zhaoyun #获得zhaoyun的经纬度
1) 1) "116.38000041246414185"
2) "39.88000114172373145"
127.0.0.1:6379> geodist user:addr zhangf diaochan #计算zhangf到diaochan的距离,单位是m
"14718.6972"
127.0.0.1:6379> geodist user:addr zhangf diaochan km #计算zhangf到diaochan的距离,单位是km
"14.7187"
127.0.0.1:6379> geodist user:addr zhangf zhaoyun km
"19.8276"
127.0.0.1:6379> georadiusbymember user:addr zhangf 20 km withcoord withdist count 3 asc
# 获得距离zhangf20km以内的按由近到远的顺序排出前三名的成员名称、距离及经纬度
#withcoord : 获得经纬度 withdist: 获得距离 withhash: 获得geohash码
1) 1) "zhangf"
2) "0.0000"
3) 1) "116.31000012159347534"
2) "40.04999982043828055"
```

```
2) 1) "diaochan"
   2) "14.7187"
   3) 1) "116.46999925374984741"
      2) "39.99999991084916218"
3) 1) "zhaoyun"
   2) "19.8276"
   3) 1) "116.38000041246414185"
      2) "39.88000114172373145"
```

## stream数据流类型

stream是Redis5.0后新增的数据结构，用于可持久化的消息队列。

几乎满足了消息队列具备的全部内容，包括：

- 消息ID的序列化生成
- 消息遍历
- 消息的阻塞和非阻塞读取
- 消息的分组消费
- 未完成消息的处理
- 消息队列监控

每个Stream都有唯一的名称，它就是Redis的key，首次使用 xadd 指令追加消息时自动创建。

常见操作命令如下表：

命令名称	命令格式	描述
xadd	xadd key id <*> field1 value1....	将指定消息数据追加到指定队列(key)中，*表示最新生成的id（当前时间+序列号）
xread	xread [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]	从消息队列中读取，COUNT：读取条数，BLOCK：阻塞读（默认不阻塞）key：队列名称 id：消息id
xrange	xrange key start end [COUNT]	读取队列中给定ID范围的消息 COUNT：返回消息条数（消息id从小到大）
xrevrange	xrevrange key start end [COUNT]	读取队列中给定ID范围的消息 COUNT：返回消息条数（消息id从大到小）
xdel	xdel key id	删除队列的消息
xgroup	xgroup create key groupname id	创建一个新的消费组
xgroup	xgroup destory key groupname	删除指定消费组
xgroup	xgroup delconsumer key groupname cname	删除指定消费组中的某个消费者

命令名称	命令格式	描述
xgroup	xgroup setid key id	修改指定消息的最大id
xreadgroup	xreadgroup group groupname consumer COUNT streams key	从队列中的消费组中创建消费者并消费数据 ( consumer不存在则创建 )

应用场景：

消息队列的使用

```

127.0.0.1:6379> xadd topic:001 * name zhangfei age 23
"1591151905088-0"
127.0.0.1:6379> xadd topic:001 * name zhaoyun age 24 name diaochan age 16
"1591151912113-0"
127.0.0.1:6379> xrange topic:001 - +
1) 1) "1591151905088-0"
    2) 1) "name"
        2) "zhangfei"
        3) "age"
        4) "23"
    2) 1) "1591151912113-0"
        2) 1) "name"
            2) "zhaoyun"
            3) "age"
            4) "24"
            5) "name"
            6) "diaochan"
            7) "age"
            8) "16"
127.0.0.1:6379> xread COUNT 1 streams topic:001 0
1) 1) "topic:001"
    2) 1) 1) "1591151905088-0"
        2) 1) "name"
            2) "zhangfei"
            3) "age"
            4) "23"

#创建的group1
127.0.0.1:6379> xgroup create topic:001 group1 0
OK
# 创建cus1加入到group1 消费 没有被消费过的消息 消费第一条
127.0.0.1:6379> xreadgroup group group1 cus1 count 1 streams topic:001 >
1) 1) "topic:001"
    2) 1) 1) "1591151905088-0"
        2) 1) "name"
            2) "zhangfei"
            3) "age"
            4) "23"

#继续消费 第二条
127.0.0.1:6379> xreadgroup group group1 cus1 count 1 streams topic:001 >
1) 1) "topic:001"
    2) 1) 1) "1591151912113-0"
        2) 1) "name"
            2) "zhaoyun"
            3) "age"
            4) "24"
            5) "name"

```

```
6) "diaochan"  
7) "age"  
8) "16"
```

#没有可消费

```
127.0.0.1:6379> xreadgroup group group1 cus1 count 1 streams topic:001 >  
(nil)
```

## Redis客户端访问

### Java程序访问Redis

采用jedis API进行访问即可

#### 1、关闭RedisServer端的防火墙

```
systemctl stop firewalld (默认)  
systemctl disable firewalld.service (设置开启不启动)
```

#### 2、新建maven项目后导入Jedis包

pom.xml

```
<dependency>  
    <groupId>redis.clients</groupId>  
    <artifactId>jedis</artifactId>  
    <version>2.9.0</version>  
</dependency>
```


#### 3、写程序

```
@Test  
public void testConn(){  
    //与Redis建立连接 IP+port  
    Jedis redis = new Jedis("192.168.127.128", 6379);  
    //在Redis中写字符串 key value  
    redis.set("jedis:name:1", "jd-zhangfei");  
    //获得Redis中字符串的值  
    System.out.println(redis.get("jedis:name:1"));  
    //在Redis中写list  
    redis.lpush("jedis:list:1", "1", "2", "3", "4", "5");  
    //获得list的长度  
    System.out.println(redis.llen("jedis:list:1"));  
}
```

## Spring访问Redis

#### 1. 新建Spring项目

新建maven项目

 New Project

Name:

Location:

▼ Artifact Coordinates

GroupId:   
The name of the artifact group, usually a company domain

ArtifactId:   
The name of the artifact within the group, usually a project name

Version:

添加Spring依赖

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>5.2.5.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.2.5.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.5.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.2.5.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

## 2. 添加redis依赖



```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-redis</artifactId>
  <version>1.0.3.RELEASE</version>
</dependency>
```

### 3. 添加Spring配置文件

添加redis.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="propertyConfigurer"

    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
      <list>
        <value>classpath:redis.properties</value>
      </list>
    </property>
  </bean>

  <!-- redis config -->
  <bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
    <property name="maxActive" value="${redis.pool.maxActive}" />
    <property name="maxIdle" value="${redis.pool.maxIdle}" />
    <property name="maxWait" value="${redis.pool.maxWait}" />
    <property name="testOnBorrow" value="${redis.pool.testOnBorrow}" />
  </bean>

  <bean id="jedisConnectionFactory"
    class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">
    <property name="hostname" value="${redis.server}"/>
    <property name="port" value="${redis.port}"/>
    <property name="timeout" value="${redis.timeout}" />
    <property name="poolConfig" ref="jedisPoolConfig" />
  </bean>

  <bean id="redisTemplate"
    class="org.springframework.data.redis.core.RedisTemplate">
    <property name="connectionFactory" ref="jedisConnectionFactory"/>
    <property name="keySerializer">
      <bean
        class="org.springframework.data.redis.serializer.StringRedisSerializer">
      </bean>
    </property>
    <property name="valueSerializer">
      <bean
        class="org.springframework.data.redis.serializer.StringRedisSerializer">
      </bean>
```

```
        </property>
    </bean>
</beans>
```

#### 4. 添加properties文件

添加redis.properties

```
redis.pool.maxActive=100
redis.pool.maxIdle=50
redis.pool.maxWait=1000
redis.pool.testOnBorrow=true

redis.timeout=50000
redis.server=192.168.72.128
redis.port=6379
```

#### 5. 编写测试用例

```
import org.junit.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.test.context.ContextConfiguration;
import
org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;

import java.io.Serializable;

@ContextConfiguration({ "classpath:redis.xml" })
public class RedisTest extends AbstractJUnit4SpringContextTests {


    @Autowired
    private RedisTemplate<Serializable, Serializable> rt;

    @Test
    public void testConn() {
        rt.opsForValue().set("name", "zhangfei");
        System.out.println(rt.opsForValue().get("name"));
    }

}
```

## SpringBoot访问Redis

### 1. 新建springboot项目

 New Project

**Project Metadata**

Group:

Artifact:

Type:  ▾

Language:  ▾

Packaging:  ▾


Java Version:  ▾

Version:

Name:

Description:

Package:

 New Project

Dependencies

Spring Boot  ▾

Developer Tools	<input checked="" type="checkbox"/> Spring Web
Web	<input type="checkbox"/> Spring Reactive Web
Template Engines	<input type="checkbox"/> Rest Repositories
Security	<input type="checkbox"/> Spring Session
SQL	<input type="checkbox"/> Rest Repositories HAL Explorer
NoSQL	<input type="checkbox"/> Rest Repositories HAL Browser
Messaging	

## 添加redis依赖包

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

## 2. 添加配置文件application.yml

```
spring:
  redis:
    host: 192.168.72.128
    port: 6379
    jedis:
      pool:
        min-idle: 0
        max-idle: 8
        max-active: 80
        max-wait: 30000
        timeout: 3000
```

## 3. 添加配置类RedisConfig

```
package com.lagou.sbr.cache;
```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
public class RedisConfig {
    @Autowired
    private RedisConnectionFactory factory;

    @Bean
    public RedisTemplate<String, Object> redisTemplate() {
        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        redisTemplate.setHashKeySerializer(new StringRedisSerializer());
        redisTemplate.setHashValueSerializer(new StringRedisSerializer());
        redisTemplate.setValueSerializer(new StringRedisSerializer());
        redisTemplate.setConnectionFactory(factory);
        return redisTemplate;
    }
}

```

#### 4. 添加RedisController

```

package com.lagou.sbr.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.TimeUnit;

@RestController
@RequestMapping(value = "/redis")
public class RedisController {
    @Autowired
    RedisTemplate redisTemplate;

    @GetMapping("/put")
    public String put(@RequestParam(required = true) String key,
        @RequestParam(required = true) String value) {
        //设置过期时间为20秒
        redisTemplate.opsForValue().set(key,value,20, TimeUnit.SECONDS);
        return "Success";
    }

    @GetMapping("/get")
    public String get(@RequestParam(required = true) String key){
        return (String) redisTemplate.opsForValue().get(key);
    }
}

```

```
}
```

## 5. 修改Application并运行

```
package com.lagou.sbr;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

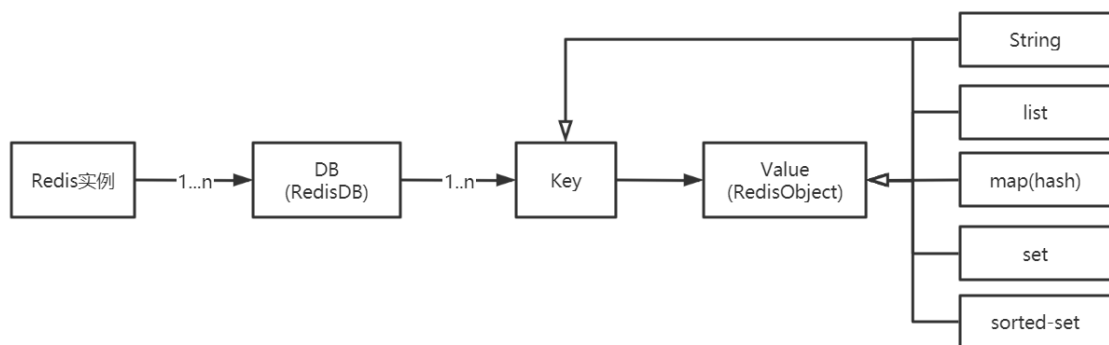
@SpringBootApplication
@EnableCaching
public class SpringbootRedisApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootRedisApplication.class, args);
    }

}
```

## 底层数据结构

Redis作为Key-Value存储系统，数据结构如下：



Redis没有表的概念，Redis实例所对应的db以编号区分，db本身就是key的命名空间。

比如：user:1000作为key值，表示在user这个命名空间下id为1000的元素，类似于user表的id=1000的行。

## RedisDB结构

Redis中存在“数据库”的概念，该结构由redis.h中的redisDb定义。

当redis 服务器初始化时，会预先分配 16 个数据库

所有数据库保存到结构 redisServer 的一个成员 redisServer.db 数组中

redisClient中存在一个名叫db的指针指向当前使用的数据库

RedisDB结构体源码：

```
typedef struct redisDb {
    int id;           //id是数据库序号，为0-15（默认Redis有16个数据库）
    long avg_ttl;     //存储的数据库对象的平均ttl（time to live），用于统计
    dict *dict;       //存储数据库所有的key-value
    dict *expires;    //存储key的过期时间
    dict *blocking_keys; //blpop 存储阻塞key和客户端对象
    dict *ready_keys; //阻塞后push 响应阻塞客户端 存储阻塞后push的key和客户端对象
    dict *watched_keys; //存储watch监控的key和客户端对象

} redisDb;
```

## id

id是数据库序号，为0-15（默认Redis有16个数据库）

## dict

存储数据库所有的key-value，后面要详细讲解

## expires

存储key的过期时间，后面要详细讲解

## RedisObject结构

Value是一个对象

包含字符串对象，列表对象，哈希对象，集合对象和有序集合对象

## 结构信息概览

```
typedef struct redisObject {
    unsigned type:4; //类型 五种对象类型
    unsigned encoding:4; //编码
    void *ptr; //指向底层实现数据结构的指针
    //...
    int refcount; //引用计数
    //...
    unsigned lru:LRU_BITS; //LRU_BITS为24bit 记录最后一次被命令程序访问的时间
    //...
} robj;
```

## 4位type

type 字段表示对象的类型，占 4 位；

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

```
127.0.0.1:6379> type a1
string
```

## 4位encoding

encoding 表示对象的内部编码，占 4 位

每个对象有不同的实现编码

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。

通过 object encoding 命令，可以查看对象采用的编码方式

```
127.0.0.1:6379> object encoding a1
"int"
```

## 24位LRU

lru 记录的是对象最后一次被命令程序访问的时间，（ 4.0 版本占 24 位，2.6 版本占 22 位）。

高16位存储一个分钟数级别的时间戳，低8位存储访问计数（lfu：最近访问次数）

lru----> 高16位: 最后被访问的时间

lfu----->低8位：最近访问次数

## refcount

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。

当对象的refcount>1时，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。

## ptr

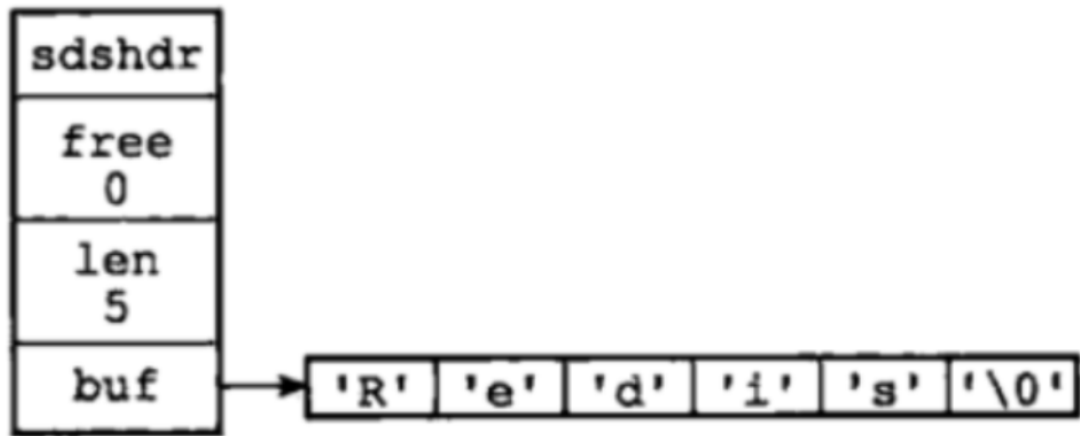
ptr 指针指向具体的数据，比如：set hello world，ptr 指向包含字符串 world 的 SDS。

## 7种type

### 字符串对象

C语言：字符数组 "\0"

Redis 使用了 SDS(Simple Dynamic String)。用于存储字符串和整型数据。



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

buf[] 的长度=len+free+1

SDS的优势：

1、SDS 在 C 字符串的基础上加入了 free 和 len 字段，获取字符串长度：SDS 是  $O(1)$ ，C 字符串是  $O(n)$ 。

buf数组的长度=free+len+1

2、SDS 由于记录了长度，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。

3、可以存取二进制数据，以字符串长度len来作为结束标识

C：\0 空字符串 二进制数据包括空字符串，所以没有办法存取二进制数据

SDS：非二进制 \0

二进制：字符串长度 可以存二进制数据

使用场景：

SDS的主要应用在：存储字符串和整型数据、存储key、AOF缓冲区和用户输入缓冲。

### 跳跃表（重点）

跳跃表是有序集合（sorted-set）的底层实现，效率高，实现简单。

跳跃表的基本思想：

将有序链表中的部分节点分层，每一层都是一个有序链表。

### 查找

在查找时优先从最高层开始向后查找，当到达某个节点时，如果next节点值大于要查找的值或next指针指向null，则从当前节点下降一层继续向后查找。

举例：

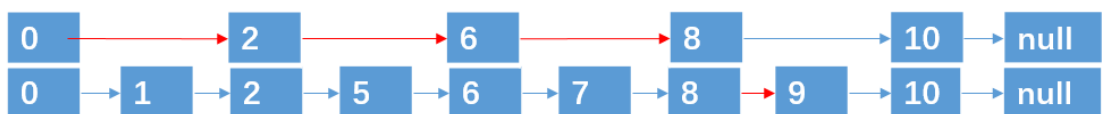




查找元素9，按道理我们需要从头结点开始遍历，一共遍历8个结点才能找到元素9。

第一次分层：

遍历5次找到元素9（红色的线为查找路径）



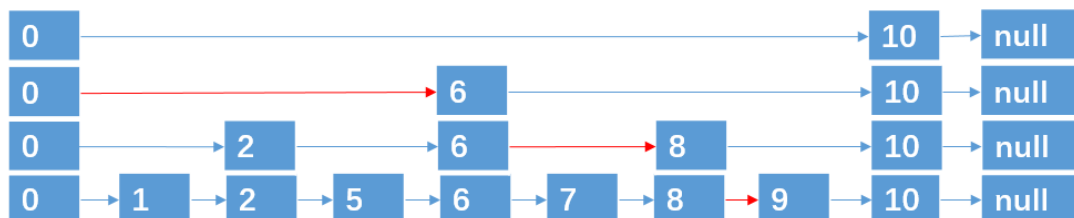
第二次分层：

遍历4次找到元素9



第三层分层:

遍历4次找到元素9



这种数据结构，就是跳跃表，它具有二分查找的功能。

插入与删除

上面例子中，9个结点，一共4层，是理想的跳跃表。

通过抛硬币（概率1/2）的方式来决定新插入结点跨越的层数：

正面:插入上层

背面：不插入

达到1/2概率（计算次数）

**删除**

找到指定元素并删除每层的该元素即可

跳跃表特点：

每层都是一个有序链表

查找次数近似于层数（1/2）

底层包含所有元素

空间复杂度  $O(n)$  扩充了一倍

**Redis跳跃表的实现**

```
//跳跃表节点
typedef struct zskiplistNode {
```

```

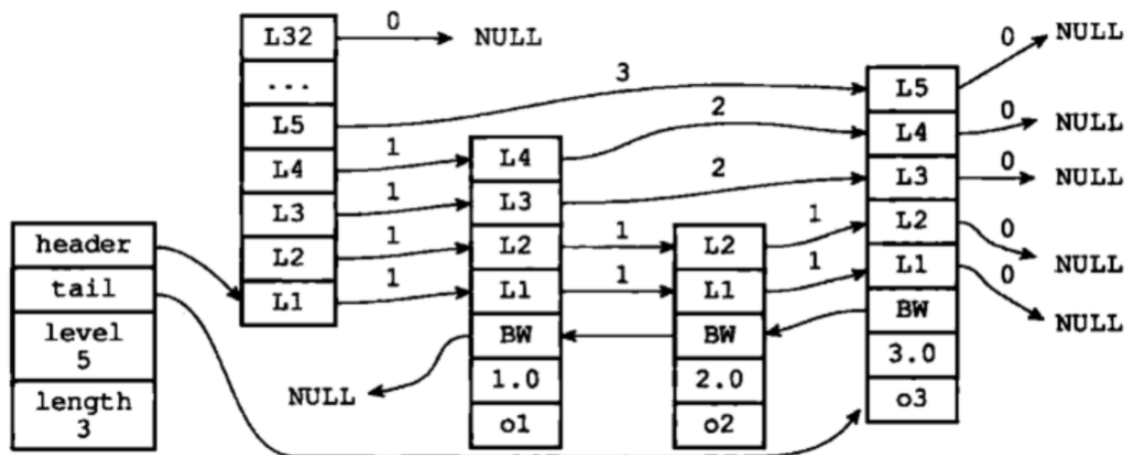
sds ele; /* 存储字符串类型数据 redis3.0版本中使用robj类型表示，
           但是在redis4.0.1中直接使用sds类型表示 */
double score; //存储排序的分值
struct zskiplistNode *backward; //后退指针，指向当前节点最底层的前一个节点
/*
   层，柔性数组，随机生成1-64的值
*/
struct zskiplistLevel {
    struct zskiplistNode *forward; //指向本层下一个节点

    unsigned int span; //本层下个节点到本节点的元素个数
} level[];
} zskiplistNode;

//链表
typedef struct zskiplist{
    //表头节点和表尾节点
    struct zskiplistNode *header, *tail;
    //表中节点的数量
    unsigned long length;
    //表中层数最大的节点的层数
    int level;
} zskiplist;

```

完整的跳跃表结构体：



跳跃表的优点：

- 1、可以快速查找到需要的节点  $O(\log n)$
- 2、可以在  $O(1)$  的时间复杂度下，快速获得跳跃表的头节点、尾节点、长度和高度。

应用场景：有序集合的实现

## 字典（重点+难点）

字典dict又称散列表（hash），是用来存储键值对的一种数据结构。

Redis整个数据库是用字典来存储的。（K-V结构）

对Redis进行CURD操作其实就是对字典中的数据进行CURD操作。

数组

数组：用来存储数据的容器，采用头指针+偏移量的方式能够以O(1)的时间复杂度定位到数据所在的内存地址。

Redis 海量存储 快

Hash函数

Hash（散列），作用是把任意长度的输入通过散列算法转换成固定类型、固定长度的散列值。

hash函数可以把Redis里的key：包括字符串、整数、浮点数统一转换成整数。

key=100.1 String “100.1” 5位长度的字符串

Redis-cli :times 33

Redis-Server : MurmurHash

数组下标=hash(key)%数组容量(hash值%数组容量得到的余数)

Hash冲突

不同的key经过计算后出现数组下标一致，称为Hash冲突。

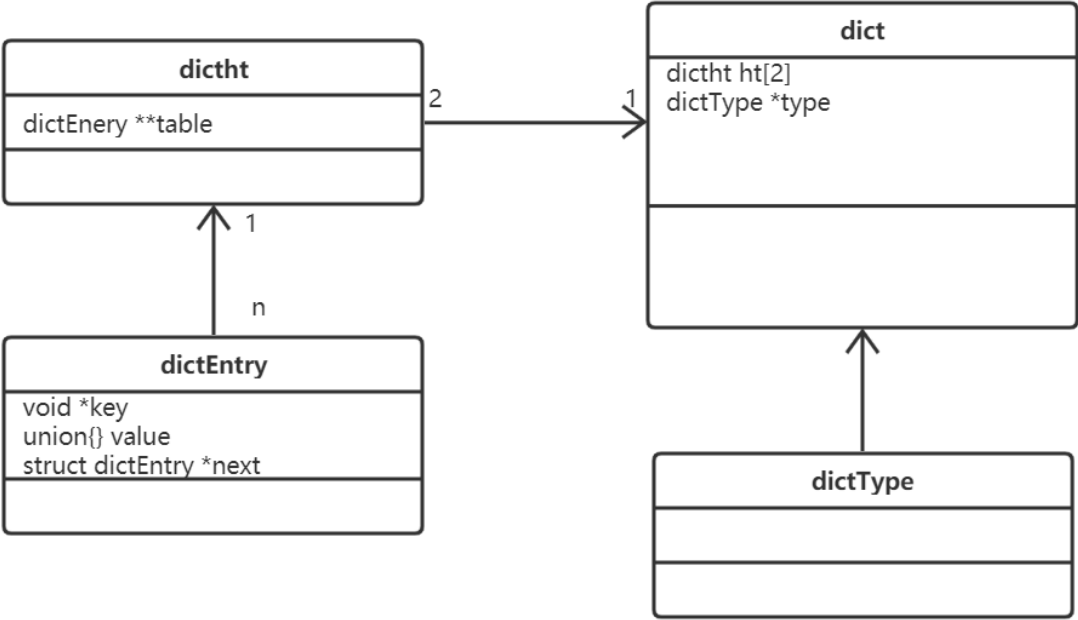
采用单链表在相同的下标位置处存储原始key和value

当根据key找Value时，找到数组下标，遍历单链表可以找出key相同的value

image-20200603161007186

Redis字典的实现

Redis字典实现包括：字典(dict)、Hash表(dictht)、Hash表节点(dictEntry)。



Hash表

```
typedef struct dictht {
    dictEntry **table;           // 哈希表数组
    unsigned long size;         // 哈希表数组的大小
    unsigned long sizemask;     // 用于映射位置的掩码，值永远等于(size-1)
    unsigned long used;         // 哈希表已有节点的数量，包含next单链表数据
} dictht;
```

1、hash表的数组初始容量为4，随着k-v存储量的增加需要对hash表数组进行扩容，新扩容量为当前量的一倍，即4,8,16,32

2、索引值=Hash值&掩码值（Hash值与Hash表容量取余）

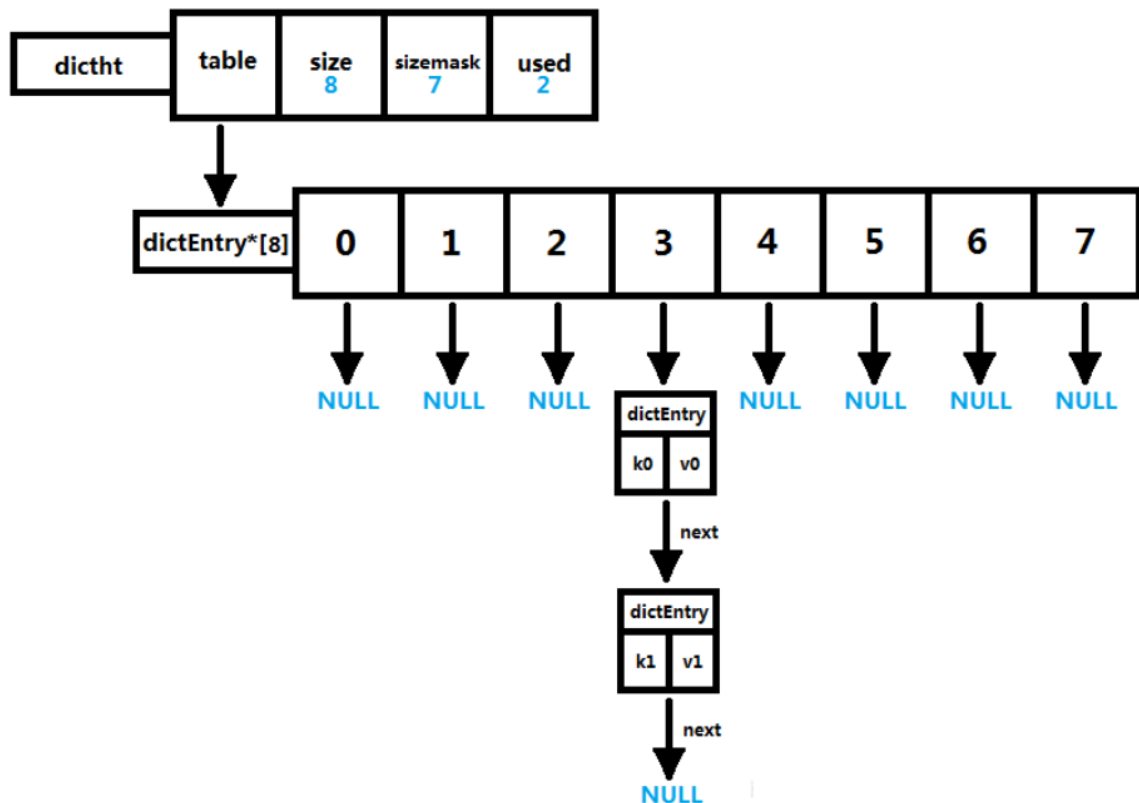
## Hash表节点

```
typedef struct dictEntry {
    void *key;                 // 键
    union {                   // 值v的类型可以是以下4种类型
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    struct dictEntry *next;    // 指向下一个哈希表节点，形成单向链表 解决hash冲突
} dictEntry;
```

key字段存储的是键值对中的键

v字段是个联合体，存储的是键值对中的值。

next指向下一个哈希表节点，用于解决hash冲突



dict字典

```

typedef struct dict {
    dictType *type;                // 该字典对应的特定操作函数
    void *privdata;                // 上述类型函数对应的可选参数
    dictht ht[2];                  /* 两张哈希表，存储键值对数据，ht[0]为原生
    哈希表，                                ht[1]为 rehash 哈希表 */
    long rehashidx;                /*rehash标识 当等于-1时表示没有在
    rehash，                                则表示正在进行rehash操作，存储的值表示
    hash表                                ht[0]的rehash进行到哪个索引值
    (数组下标)*/
    int iterators;                 // 当前运行的迭代器数量
} dict;

```

type字段，指向dictType结构体，里边包括了对该字典操作的函数指针

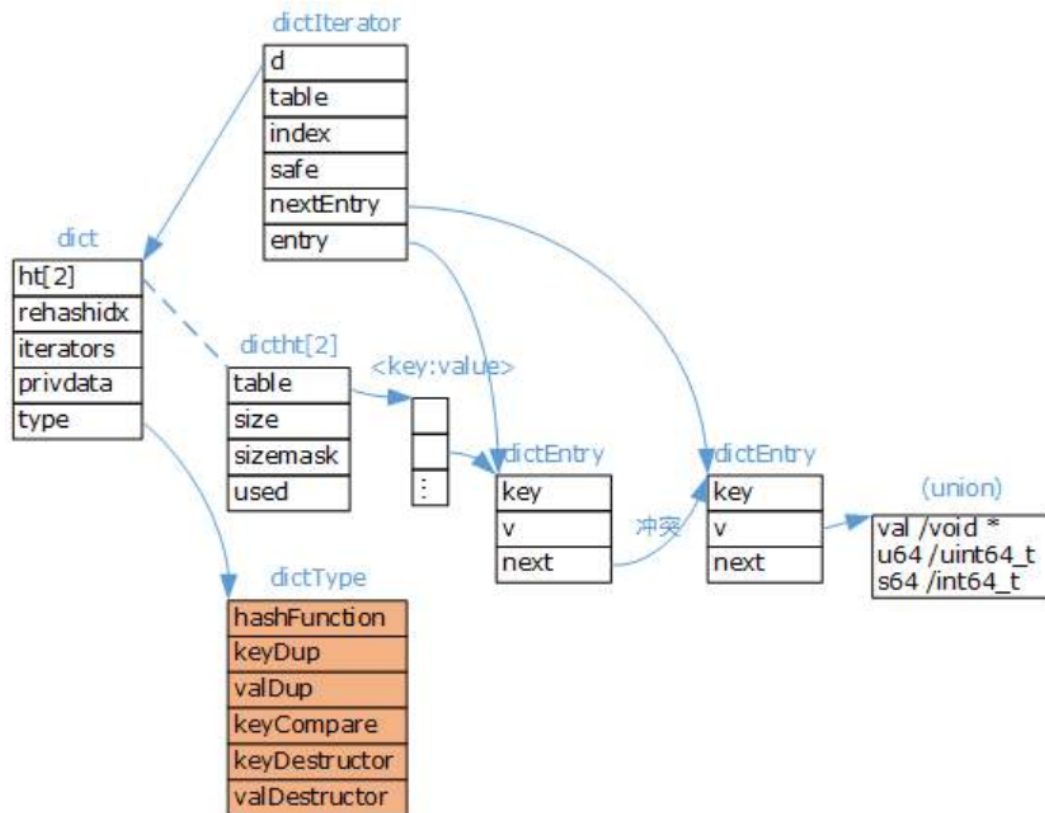
```

typedef struct dictType {
    // 计算哈希值的函数
    unsigned int (*hashFunction)(const void *key);
    // 复制键的函数
    void (*keyDup)(void *privdata, const void *key);
    // 复制值的函数
    void (*valDup)(void *privdata, const void *obj);
    // 比较键的函数
    int (*keyCompare)(void *privdata, const void *key1, const void *key2);
    // 销毁键的函数
    void (*keyDestructor)(void *privdata, void *key);
    // 销毁值的函数
    void (*valDestructor)(void *privdata, void *obj);
} dictType;

```

Redis字典除了主数据库的K-V数据存储以外，还可以用于：散列表对象、哨兵模式中的主从节点管理等。在不同的应用中，字典的形态都可能不同，dictType是为了实现各种形态的字典而抽象出来的操作函数（多态）。

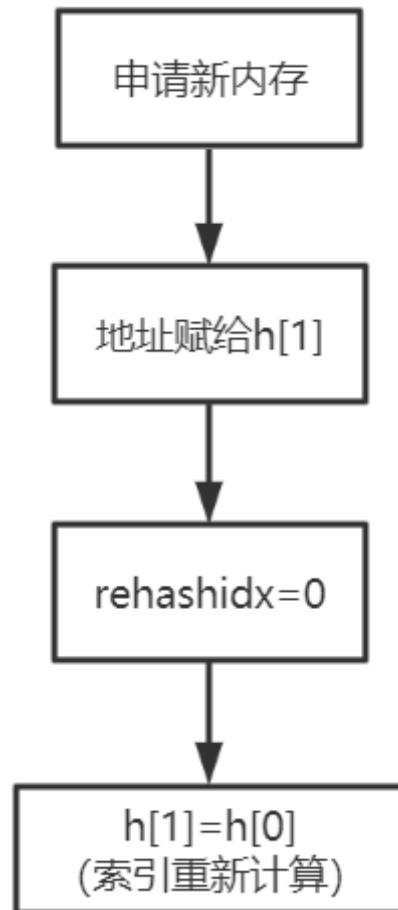
完整的Redis字典数据结构：



## 字典扩容

字典达到存储上限（阈值 0.75），需要rehash（扩容）

扩容流程：



说明：

1. 初次申请默认容量为4个dictEntry，非初次申请为当前hash表容量的一倍。
2. rehashidx=0表示要进行rehash操作。
3. 新增加的数据在新的hash表h[1]
4. 修改、删除、查询在老hash表h[0]、新hash表h[1]中（rehash中）
5. 将老的hash表h[0]的数据重新计算索引值后全部迁移到新的hash表h[1]中，这个过程称为rehash。

### 渐进式rehash

当数据量巨大时rehash的过程是非常缓慢的，所以需要进行优化。

服务器忙，则只对一个节点进行rehash

服务器闲，可批量rehash(100节点)

应用场景：

- 1、主数据库的K-V数据存储
- 2、散列表对象（hash）
- 3、哨兵模式中的主从节点管理

### 压缩列表

压缩列表（ziplist）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

节省内存

是一个字节数组，可以包含多个节点（entry）。每个节点可以保存一个字节数组或一个整数。

压缩列表的数据结构如下：

zlbytes	zltail	zllen	entry1	entry2	...	entryN	zlend
---------	--------	-------	--------	--------	-----	--------	-------

zlbytes：压缩列表的字节长度

zltail：压缩列表尾元素相对于压缩列表起始地址的偏移量

zllen：压缩列表的元素个数

entry1..entryX：压缩列表的各个节点

zlend：压缩列表的结尾，占一个字节，恒为0xFF（255）

entryX元素的编码结构：



previous\_entry\_length：前一个元素的字节长度

encoding:表示当前元素的编码

content:数据内容

ziplist结构体如下：

```
struct ziplist<T>{
    unsigned int zlbytes; // ziplist的长度字节数，包含头部、所有entry和zipend。
    unsigned int zloffset; // 从ziplist的头指针到指向最后一个entry的偏移量，用于快速反向查询
    unsigned short int zllength; // entry元素个数
    T[] entry; // 元素值
    unsigned char zlend; // ziplist结束符，值固定为0xFF
}

typedef struct zlentry {
    unsigned int prevrawlensize; //previous_entry_length字段的长度
    unsigned int prevrawlen; //previous_entry_length字段存储的内容

    unsigned int lensize; //encoding字段的长度
    unsigned int len; //数据内容长度

    unsigned int headersize; //当前元素的首部长度，即previous_entry_length字段长度与encoding字段长度之和。

    unsigned char encoding; //数据类型

    unsigned char *p; //当前元素首地址
} zlentry;
```

应用场景：

sorted-set和hash元素个数少且是小整数或短字符串（直接使用）

list用快速链表(quicklist)数据结构存储，而快速链表是双向列表与压缩列表的组合。（间接使用）



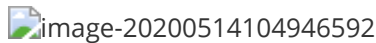
## 整数集合

整数集合(intset)是一个有序的（整数升序）、存储整数的连续存储结构。

当Redis集合类型的元素都是整数并且都处在64位有符号整数范围内（ $2^{64}$ ），使用该结构体存储。

```
127.0.0.1:6379> sadd set:001 1 3 5 6 2
(integer) 5
127.0.0.1:6379> object encoding set:001
"intset"
127.0.0.1:6379> sadd set:004 1 10000000000000000000000000000000 9999999999
(integer) 3
127.0.0.1:6379> object encoding set:004
"hashtable"
```

intset的结构图如下：

image-20200514104946592

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

应用场景：

可以保存类型为int16\_t、int32\_t 或者int64\_t 的整数值，并且保证集合中不会出现重复元素。

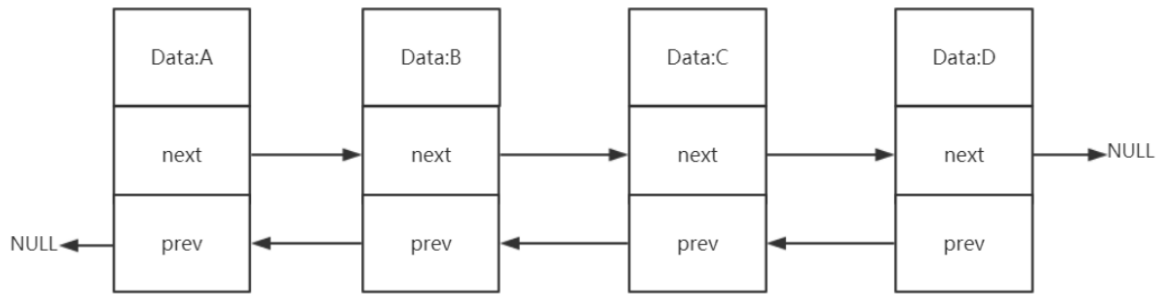
## 快速列表（重要）

快速列表（quicklist）是Redis底层重要的数据结构。是列表的底层实现。（在Redis3.2之前，Redis采用双向链表（adlist）和压缩列表（ziplist）实现。）在Redis3.2以后结合adlist和ziplist的优势Redis设计出了quicklist。

```
127.0.0.1:6379> lpush list:001 1 2 5 4 3
(integer) 5
127.0.0.1:6379> object encoding list:001
"quicklist"
```

## 双向列表（adlist）

双向链表:可以从两个方向遍历

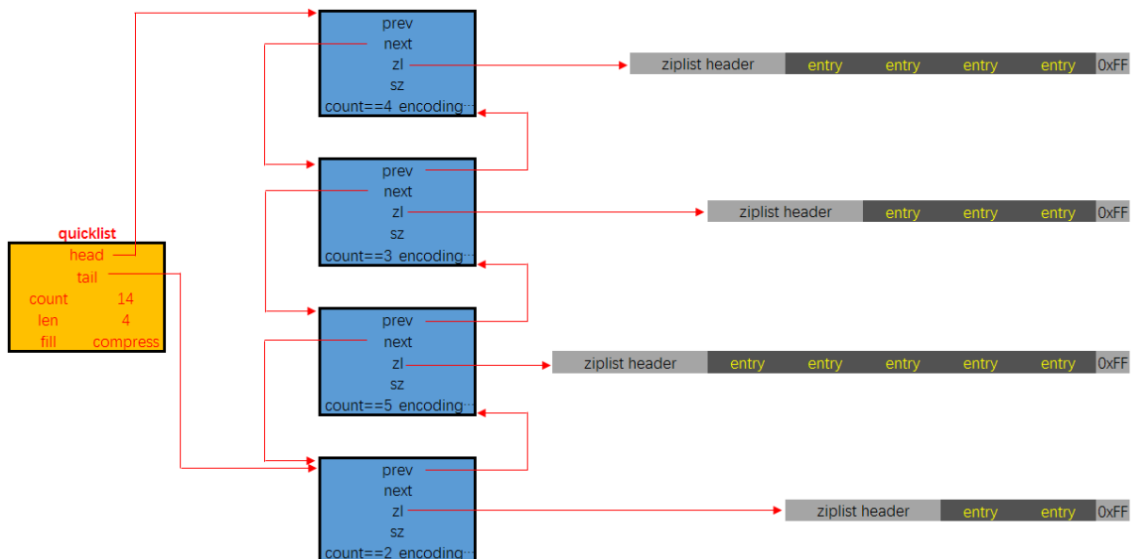


双向链表优势：

1. 双向：链表具有前置节点和后置节点的引用，获取这两个节点时间复杂度都为 $O(1)$ 。
2. 普通链表（单链表）：节点类保留下一节点的引用。链表类只保留头节点的引用，只能从头节点插入删除
3. 无环：表头节点的 prev 指针和表尾节点的 next 指针都指向 NULL,对链表的访问都是以 NULL 结束。  
环状：头的前一个节点指向尾节点
4. 带链表长度计数器：通过 len 属性获取链表长度的时间复杂度为  $O(1)$ 。
5. 多态：链表节点使用 void\* 指针来保存节点值，可以保存各种不同类型的值。

### 快速列表

quicklist是一个双向链表，链表中的每个节点是一个ziplist结构。quicklist中的每个节点ziplist都能够存储多个数据元素。



quicklist的结构定义如下：

```
typedef struct quicklist {
    quicklistNode *head;           // 指向quicklist的头部
    quicklistNode *tail;          // 指向quicklist的尾部
    unsigned long count;           // 列表中所有数据项的个数总和
    unsigned int len;              // quicklist节点的个数，即ziplist的个数
    int fill : 16;                 // ziplist大小限定，由list-max-ziplist-size给定
    (Redis设定)
    unsigned int compress : 16;    // 节点压缩深度设置，由list-compress-depth给定
    (Redis设定)
} quicklist;
```

quicklistNode的结构定义如下：

```
typedef struct quicklistNode {
    struct quicklistNode *prev;    // 指向上一个ziplist节点
    struct quicklistNode *next;    // 指向下一个ziplist节点
    unsigned char *zl;             // 数据指针，如果没有被压缩，就指向ziplist结构，反之
    指向                                quicklistLZF结构
    unsigned int sz;               // 表示指向ziplist结构的总长度(内存占用长度)
    unsigned int count : 16;       // 表示ziplist中的数据项个数
    unsigned int encoding : 2;     // 编码方式，1--ziplist, 2--quicklistLZF
    unsigned int container : 2;    // 预留字段，存放数据的方式，1--NONE, 2--ziplist
    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
    unsigned int attempted_compress : 1; // 测试相关
    unsigned int extra : 10;       // 扩展字段，暂时没用
} quicklistNode;
```

## 数据压缩

quicklist每个节点的实际数据存储结构为ziplist，这种结构的优势在于节省存储空间。为了进一步降低ziplist的存储空间，还可以对ziplist进行压缩。Redis采用的压缩算法是LZF。其基本思想是：数据与前面重复的记录重复位置及长度，不重复的记录原始数据。

压缩过后的数据可以分成多个片段，每个片段有两个部分：解释字段和数据字段。quicklistLZF的结构体如下：

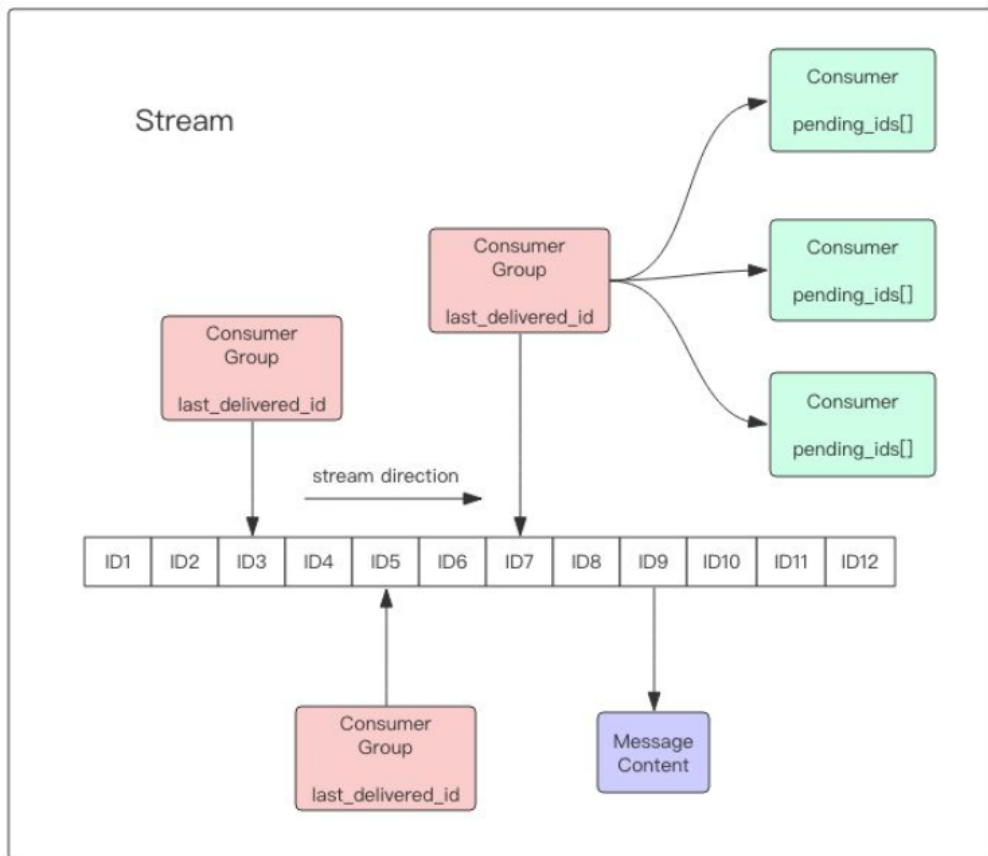
```
typedef struct quicklistLZF {
    unsigned int sz; // LZF压缩后占用的字节数
    char compressed[]; // 柔性数组，指向数据部分
} quicklistLZF;
```

## 应用场景

列表(List)的底层实现、发布与订阅、慢查询、监视器等功能。

## 流对象

stream主要由：消息、生产者、消费者和消费组构成。

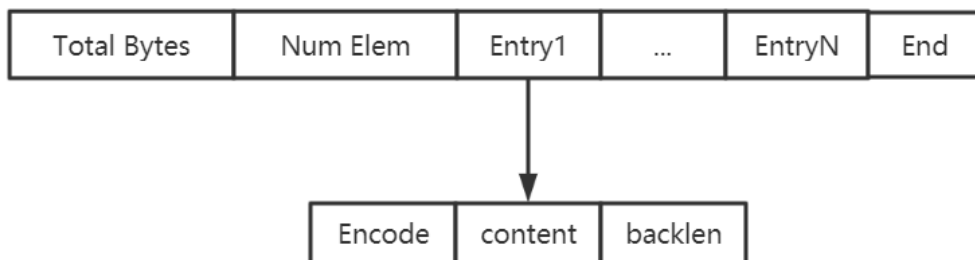


Redis Stream的底层主要使用了listpack（紧凑列表）和Rax树（基数树）。

### listpack

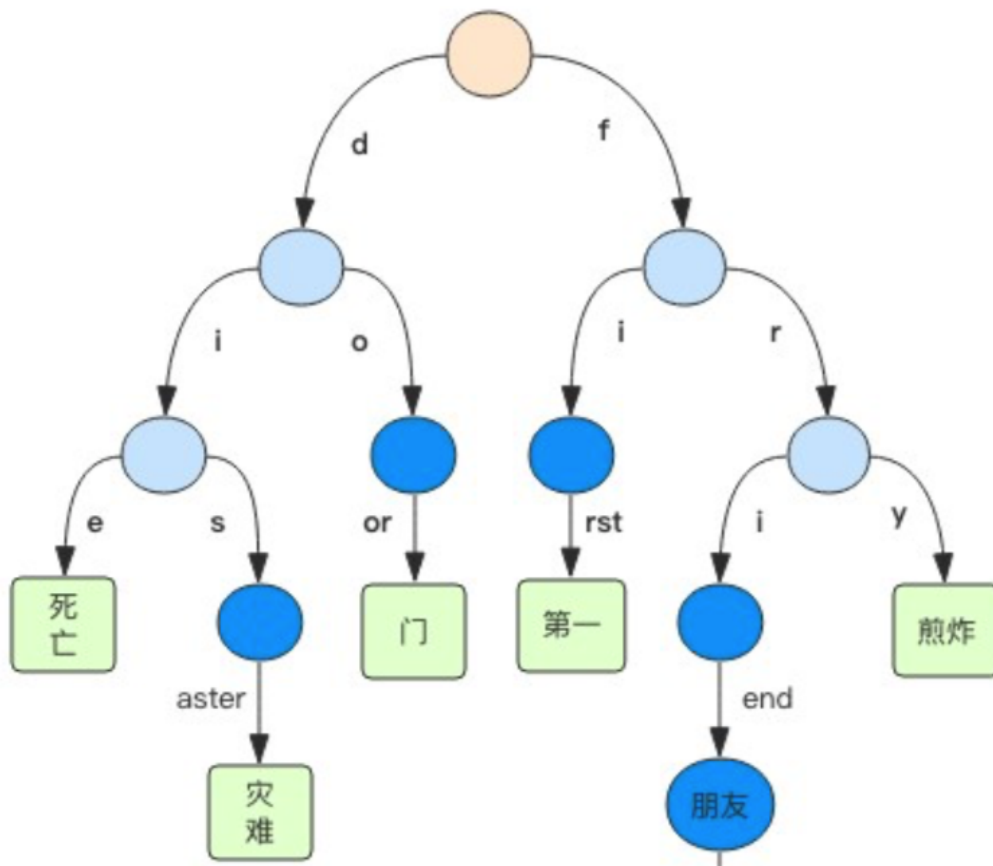
listpack表示一个字符串列表的序列化，listpack可用于存储字符串或整数。用于存储stream的消息内容。

结构如下图：



### Rax树

Rax 是一个有序字典树(基数树 Radix Tree)，按照 key 的字典序排列，支持快速地定位、插入和删除操作。



Rax 被用在 Redis Stream 结构里面用于存储消息队列，在 Stream 里面消息 ID 的前缀是时间戳 + 序号，这样的消息可以理解为时间序列消息。使用 Rax 结构 进行存储就可以快速地根据消息 ID 定位到具体的消息，然后继续遍历指定消息 之后的所有消息。

image-20200603175628621

应用场景：

stream的底层实现

## 10种encoding

encoding 表示对象的内部编码，占 4 位。

Redis通过 encoding 属性为对象设置不同的编码

对于少的和小的数据，Redis采用小的和压缩的存储方式，体现Redis的灵活性

大大提高了 Redis 的存储量和执行效率

比如Set对象：

intset ：元素是64位以内的整数

hashtable ：元素是64位以外的整数

如下所示：

```
127.0.0.1:6379> sadd set:001 1 3 5 6 2
(integer) 5
127.0.0.1:6379> object encoding set:001
"intset"
127.0.0.1:6379> sadd set:004 1 10000000000000000000000000000 9999999999
(integer) 3
127.0.0.1:6379> object encoding set:004
"hashtable"
```

## String

int、raw、embstr

**int**

REDIS\_ENCODING\_INT ( int类型的整数 )

```
127.0.0.1:6379> set n1 123
OK
127.0.0.1:6379> object encoding n1
"int"
```

## embstr

### REDIS\_ENCODING\_EMBSTR(编码的简单动态字符串)

小字符串 长度小于44个字节

```
127.0.0.1:6379> set name:001 zhangfei
OK
127.0.0.1:6379> object encoding name:001
"embstr"
```

## raw

## REDIS\_ENCODING\_RAW (简单动态字符串)

大字符串 长度大于44个字节

[illegible]**list**

列表的编码是quicklist。

## REDIS\_ENCODING\_QUICKLIST (快速列表)

```
127.0.0.1:6379> lpush list:001 1 2 5 4 3
(integer) 5
127.0.0.1:6379> object encoding list:001
"quicklist"
```

## hash

## 散列的编码是字典和压缩列表

## dict

## REDIS\_ENCODING\_HT (字典)

当散列表元素的个数比较多或元素不是小整数或短字符串时。

[illegible]

**ziplist**

## REDIS\_ENCODING\_ZIPLIST (压缩列表)

当散列表元素的个数比较少，且元素都是小整数或短字符串时。

```
127.0.0.1:6379> hmset user:001 username zhangfei password 111 age 23 sex M
OK
127.0.0.1:6379> object encoding user:001
"ziplist"
```

## set

## 集合的编码是整形集合和字典

**intset**

REDIS\_ENCODING\_INTSET ( 整数集合 )

当Redis集合类型的元素都是整数并且都处在64位有符号整数范围内（<18446744073709551616）

```
127.0.0.1:6379> sadd set:001 1 3 5 6 2
(integer) 5
127.0.0.1:6379> object encoding set:001
"intset"
```

## dict

## REDIS\_ENCODING\_HT (字典)

当Redis集合类型的元素都是整数并且都处在64位有符号整数范围外 (>18446744073709551616)

```
127.0.0.1:6379> sadd set:004 1 1000000000000000000000000000 9999999999
(integer) 3
127.0.0.1:6379> object encoding set:004
"hashtable"
```

**zset**

## 有序集合的编码是压缩列表和跳跃表+字典

## ziplist

## REDIS\_ENCODING\_ZIPLIST (压缩列表)

当元素的个数比较少，且元素都是小整数或短字符串时。

```
127.0.0.1:6379> zadd hit:1 100 item1 20 item2 45 item3
(integer) 3
127.0.0.1:6379> object encoding hit:1
"ziplist"
```

## skiplist + dict

### REDIS\_ENCODING\_SKIPLIST ( 跳跃表+字典 )

当元素的个数比较多或元素不是小整数或短字符串时。

[illegible]

## 缓存过期和淘汰策略

### Redis性能高：

官方数据

读：110000次/s

写：81000次/s

长期使用，key会不断增加，Redis作为缓存使用，物理内存也会满

内存与硬盘交换（swap）虚拟内存，频繁IO性能急剧下降

## maxmemory

## 不设置的场景

Redis的key是固定的，不会增加

Redis作为DB使用，保证数据的完整性，不能淘汰，可以做集群，横向扩展

缓存淘汰策略：禁止驱逐（默认）



## 设置的场景

Redis是作为缓存使用，不断增加Key

maxmemory：默认为0 不限制

问题：达到物理内存后性能急剧下架，甚至崩溃

内存与硬盘交换（swap）虚拟内存，频繁IO 性能急剧下降

设置多少？

与业务有关

1个Redis实例，保证系统运行 1 G，剩下的就都可以设置Redis

物理内存的3/4

slaver：留出一定的内存

在redis.conf中

```
maxmemory 1024mb
```

命令：获得maxmemory数

```
CONFIG GET maxmemory
```

设置maxmemory后，当趋近maxmemory时，通过缓存淘汰策略，从内存中删除对象

不设置maxmemory 无最大内存限制 maxmemory-policy noeviction（禁止驱逐）不淘汰

设置maxmemory maxmemory-policy 要配置

## expire数据结构

在Redis中可以使用expire命令设置一个键的存活时间(ttl: time to live)，过了这段时间，该键就会自动被删除。

## expire的使用

expire命令的使用方法如下：

expire key ttl(单位秒)

```
127.0.0.1:6379> expire name 2 #2秒失效
(integer) 1
127.0.0.1:6379> get name
(nil)
127.0.0.1:6379> set name zhangfei
OK
127.0.0.1:6379> ttl name #永久有效
(integer) -1
127.0.0.1:6379> expire name 30 #30秒失效
(integer) 1
127.0.0.1:6379> ttl name #还有24秒失效
```

```
(integer) 24
127.0.0.1:6379> ttl name    #失效
(integer) -2
```

## expire原理

```
typedef struct redisDb {
    dict *dict;    -- key Value
    dict *expires; -- key ttl
    dict *blocking_keys;
    dict *ready_keys;
    dict *watched_keys;
    int id;
} redisDb;
```

上面的代码是Redis 中关于数据库的结构体定义，这个结构体定义中除了 id 以外都是指向字典的指针，其中我们只看 dict 和 expires。

dict 用来维护一个 Redis 数据库中包含的所有 Key-Value 键值对，expires则用于维护一个 Redis 数据库中设置了失效时间的键(即key与失效时间的映射)。

当我们使用 expire命令设置一个key的失效时间时，Redis 首先到 dict 这个字典表中查找要设置的key是否存在，如果存在就将这个key和失效时间添加到 expires 这个字典表。

当我们使用 setex命令向系统插入数据时，Redis 首先将 Key 和 Value 添加到 dict 这个字典表中，然后将 Key 和失效时间添加到 expires 这个字典表中。

简单地总结来说就是，设置了失效时间的key和具体的失效时间全部都维护在 expires 这个字典表中。

## 删除策略

Redis的数据删除有定时删除、惰性删除和主动删除三种方式。

Redis目前采用惰性删除+主动删除的方式。

### 定时删除

在设置键的过期时间的同时，创建一个定时器，让定时器在键的过期时间来临时，立即执行对键的删除操作。

需要创建定时器，而且消耗CPU，一般不推荐使用。

### 惰性删除

在key被访问时如果发现它已经失效，那么就删除它。

调用expireIfNeeded函数，该函数的意义是：读取数据之前先检查一下它有没有失效，如果失效了就删除它。

```
int expireIfNeeded(redisDb *db, robj *key) {
    //获取主键的失效时间    get当前时间-创建时间>ttl
    long long when = getExpire(db,key);
    //假如失效时间为负数，说明该主键未设置失效时间（失效时间默认为-1），直接返回0
    if (when < 0) return 0;
    //假如Redis服务器正在从RDB文件中加载数据，暂时不进行失效主键的删除，直接返回0
    if (server.loading) return 0;
    ...
}
```

```
//如果以上条件都不满足，就将主键的失效时间与当前时间进行对比，如果发现指定的主键
//还未失效就直接返回0
if (mstime() <= when) return 0;
//如果发现主键确实已经失效了，那么首先更新关于失效主键的统计个数，然后将该主键失
//效的信息进行广播，最后将该主键从数据库中删除
server.stat_expiredkeys++;
propagateExpire(db,key);
return dbDelete(db,key);
}
```

## 主动删除

在redis.conf文件中可以配置主动删除策略,默认是no-eviction (不删除)

```
maxmemory-policy allkeys-lru
```

## LRU

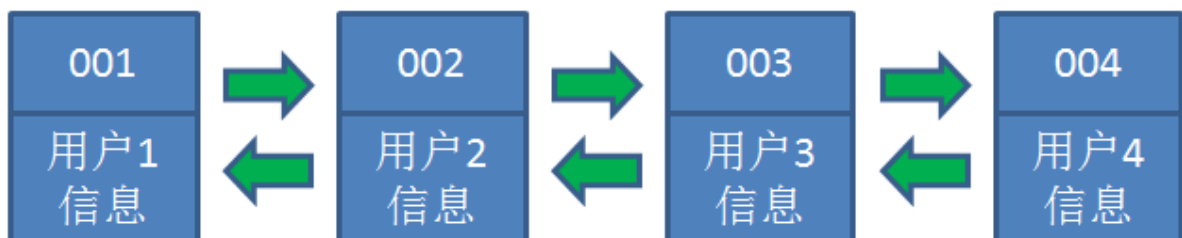
LRU (Least recently used) 最近最少使用，算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

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。
4. 在Java中可以使用LinkHashMap（哈希链表）去实现LRU

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：

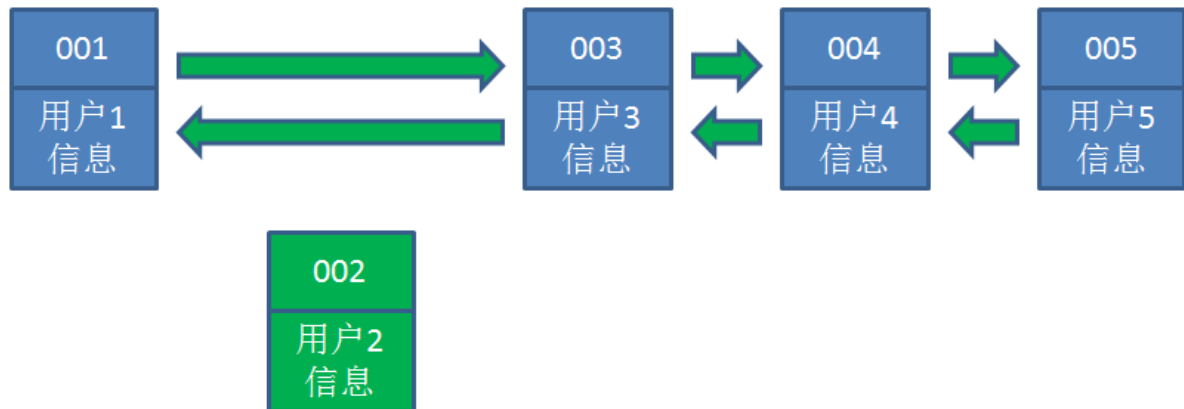
1.假设我们使用哈希链表来缓存用户信息，目前缓存了4个用户，这4个用户是按照时间顺序依次从链表右端插入的。



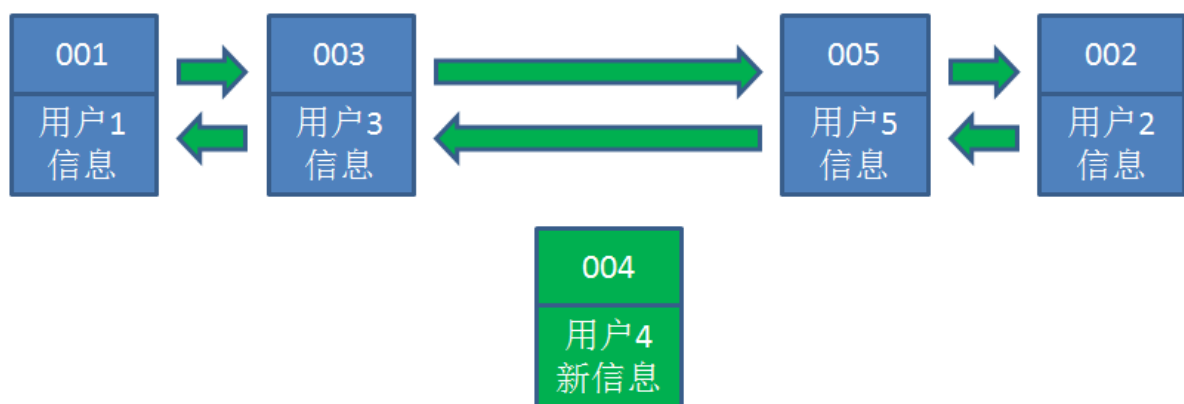
2.此时，业务方访问用户5，由于哈希链表中没有用户5的数据，我们从数据库中读取出来，插入到缓存当中。这时候，链表中最右端是最新访问到的用户5，最左端是最近最少访问的用户1。



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。





5.业务访问用户6，用户6在缓存里没有，需要插入到哈希链表。假设这时候缓存容量已经达到上限，必须先删除最近最少访问的数据，那么位于哈希链表最左端的用户1就会被删除掉，然后再把用户6插入到最右端。



### Redis的LRU 数据淘汰机制

在服务器配置中保存了 lru 计数器 `server.lrulock`，会定时（redis 定时程序 `serverCron()`）更新，`server.lrulock` 的值是根据 `server.unixtime` 计算出来的。

另外，从 `struct redisObject` 中可以发现，每一个 redis 对象都会设置相应的 `lru`。可以想象的是，每一次访问数据的时候，会更新 `redisObject.lru`。

LRU 数据淘汰机制是这样的：在数据集中随机挑选几个键值对，取出其中 `lru` 最大的键值对淘汰。

不可能遍历key 用当前时间-最近访问 越大 说明 访问间隔时间越长

volatile-lru

从已设置过期时间的数据集（`server.db[i].expires`）中挑选最近最少使用的数据淘汰

allkeys-lru

从数据集（`server.db[i].dict`）中挑选最近最少使用的数据淘汰

### LFU

LFU (Least frequently used) 最不经常用，如果一个数据在最近一段时间内使用次数很少，那么在将来一段时间内被使用的可能性也很小。

volatile-lfu

allkeys-lfu

### random

随机

volatile-random

从已设置过期时间的数据集（`server.db[i].expires`）中任意选择数据淘汰

allkeys-random

从数据集 ( server.db[i].dict ) 中任意选择数据淘汰

## **ttl**

volatile-ttl

从已设置过期时间的数据集 ( server.db[i].expires ) 中挑选将要过期的数据淘汰

redis 数据集数据结构中保存了键值对过期时间的表，即 redisDb.expires。

TTL 数据淘汰机制：从过期时间的表中随机挑选几个键值对，取出其中 ttl 最小的键值对淘汰。

## **noeviction**

禁止驱逐数据，不删除 默认

## **缓存淘汰策略的选择**

- allkeys-lru ： 在不确定时一般采用策略。冷热数据交换
- volatile-lru ： 比allkeys-lru性能差 存：过期时间
- allkeys-random ： 希望请求符合平均分布(每个元素以相同的概率被访问)
- 自己控制：volatile-ttl 缓存穿透

## **案例分享：字典库失效**

key-Value 业务表存 code 显示 文字

拉勾早期将字典库，设置了maxmemory，并设置缓存淘汰策略为allkeys-lru

结果造成字典库某些字段失效，缓存击穿，DB压力剧增，差点宕机。

分析：

字典库：Redis做DB使用，要保证数据的完整性

maxmemory设置较小，采用allkeys-lru，会对没有经常访问的字典库随机淘汰

当再次访问时会缓存击穿，请求会打到DB上。

解决方案：

- 1、不设置maxmemory
- 2、使用noeviction策略

Redis是作为DB使用的，要保证数据的完整性，所以不能删除数据。

可以将原始数据源 ( XML ) 在系统启动时一次性加载到Redis中。

Redis做主从+哨兵 保证高可用