# Redis

## 概述

redis是一个key-value[存储系统](http://baike.baidu.com/view/51839.htm)。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list([链表](http://baike.baidu.com/view/549479.htm))、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些[数据类型](http://baike.baidu.com/view/675645.htm)都 支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排 序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文 件，并且在此基础上实现了master-slave(主从)同步。

Redis 是一个高性能的key-value数据库。 redis的出现，很大程度补偿了[memcached](http://baike.baidu.com/view/794242.htm)这类key/value存储的不足，在部 分场合可以对关系数据库起到很好的补充作用。它提供了Python，Ruby，Erlang，PHP客户端，使用很方便。

Redis支持主从同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。这使得Redis可执行单层树复 制。从盘可以有意无意的对数据进行写操作。由于完全实现了发布/订阅机制，使得从数据库在任何地方同步树时，可订阅一个频道并接收主服务器完整的消息发布 记录。同步对读取操作的可扩展性和数据冗余很有帮助。

## 优点

* 性能极高 – Redis能支持超过 100K+ 每秒的读写频率。
* 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
* 原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。
* 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

## 简易教程

### String类型

Redis能存储二进制安全的字符串，最大长度为1GB

redis 127.0.0.1:6379> SET name "John Doe"

OK

redis 127.0.0.1:6379> GET name

"John Doe"

String类型还支持批量的读写操作

redis 127.0.0.1:6379> MSET age 30 sex "male"

OK

redis 127.0.0.1:6379> MGET age sex

1) "30"

2) "male"

String类型其实也可以用来存储数字，并支持对数字的加减操作。

redis 127.0.0.1:6379> INCR age

(integer) 31

redis 127.0.0.1:6379> INCRBY age 4

(integer) 35

redis 127.0.0.1:6379> GET age

"35"

redis 127.0.0.1:6379> DECR age

(integer) 34

redis 127.0.0.1:6379> DECRBY age 4

(integer) 30

redis 127.0.0.1:6379> GET age

"30"

String类型还支持对其部分的修改和获取操作

redis 127.0.0.1:6379> APPEND name " Mr."

(integer) 12

redis 127.0.0.1:6379> GET name

"John Doe Mr."

redis 127.0.0.1:6379> STRLEN name

(integer) 12

redis 127.0.0.1:6379> SUBSTR name 0 3

"John"

### List类型

Redis能够将数据存储成一个链表，并能对这个链表进行丰富的操作

redis 127.0.0.1:6379> LPUSH students "John Doe"

(integer) 1

redis 127.0.0.1:6379> LPUSH students "Captain Kirk"

(integer) 2

redis 127.0.0.1:6379> LPUSH students "Sheldon Cooper"

(integer) 3

redis 127.0.0.1:6379> LLEN students

(integer) 3

redis 127.0.0.1:6379> LRANGE students 0 2

1) "Sheldon Cooper"

2) "Captain Kirk"

3) "John Doe"

redis 127.0.0.1:6379> LPOP students

"Sheldon Cooper"

redis 127.0.0.1:6379> LLEN students

(integer) 2

redis 127.0.0.1:6379> LRANGE students 0 1

1) "Captain Kirk"

2) "John Doe"

redis 127.0.0.1:6379> LREM students 1 "John Doe"

(integer) 1

redis 127.0.0.1:6379> LLEN students

(integer) 1

redis 127.0.0.1:6379> LRANGE students 0 0

1) "Captain Kirk"

redis 127.0.0.1:6379> LINSERT students BEFORE "Captain Kirk" "Dexter Morgan"

(integer) 3

redis 127.0.0.1:6379> LRANGE students 0 2

1) "Dexter Morgan"

2) "Captain Kirk"

3) "John Doe"

redis 127.0.0.1:6379> LPUSH students "Peter Parker"

(integer) 4

redis 127.0.0.1:6379> LRANGE students 0 3

1) "Peter Parker"

2) "Dexter Morgan"

3) "Captain Kirk"

4) "John Doe"

redis 127.0.0.1:6379> LTRIM students 1 3

OK

redis 127.0.0.1:6379> LLEN students

(integer) 3

redis 127.0.0.1:6379> LRANGE students 0 2

1) "Dexter Morgan"

2) "Captain Kirk"

3) "John Doe"

redis 127.0.0.1:6379> LREM students 1 "John Doe"

(integer) 1

redis 127.0.0.1:6379> LLEN students

(integer) 1

redis 127.0.0.1:6379> LRANGE students 0 1

1) "Captain Kirk"

### 集合（Sets）类型

Redis能够将一系列不重复的值存储成一个集合

redis 127.0.0.1:6379> SADD birds crow

(integer) 1

redis 127.0.0.1:6379> SADD birds pigeon

(integer) 1

redis 127.0.0.1:6379> SADD birds bat

(integer) 1

redis 127.0.0.1:6379> SADD mammals dog

(integer) 1

redis 127.0.0.1:6379> SADD mammals cat

(integer) 1

redis 127.0.0.1:6379> SADD mammals bat

(integer) 1

redis 127.0.0.1:6379> SMEMBERS birds

1) "bat"

2) "crow"

3) "pigeon"

redis 127.0.0.1:6379> SMEMBERS mammals

1) "bat"

2) "cat"

3) "dog"

Sets结构也支持相应的修改操作

redis 127.0.0.1:6379> SREM mammals cat

(integer) 1

redis 127.0.0.1:6379> SMEMBERS mammals

1) "bat"

2) "dog"

redis 127.0.0.1:6379> SADD mammals human

(integer) 1

redis 127.0.0.1:6379> SMEMBERS mammals

1) "bat"

2) "human"

3) "dog"

Redis还支持对集合的子交并补等操作

redis 127.0.0.1:6379> SINTER birds mammals

1) "bat"

redis 127.0.0.1:6379> SUNION birds mammals

1) "crow"

2) "bat"

3) "human"

4) "pigeon"

5) "dog"

redis 127.0.0.1:6379> SDIFF birds mammals

1) "crow"

2) "pigeon"

### 有序集合（Sorted Sets）类型

Sorted Sets和Sets结构相似，不同的是存在Sorted Sets中的数据会有一个score属性，并会在写入时就按这个score排好序。

redis 127.0.0.1:6379> ZADD days 0 mon

(integer) 1

redis 127.0.0.1:6379> ZADD days 1 tue

(integer) 1

redis 127.0.0.1:6379> ZADD days 2 wed

(integer) 1

redis 127.0.0.1:6379> ZADD days 3 thu

(integer) 1

redis 127.0.0.1:6379> ZADD days 4 fri

(integer) 1

redis 127.0.0.1:6379> ZADD days 5 sat

(integer) 1

redis 127.0.0.1:6379> ZADD days 6 sun

(integer) 1

redis 127.0.0.1:6379> ZCARD days

(integer) 7

redis 127.0.0.1:6379> ZRANGE days 0 6

1) "mon"

2) "tue"

3) "wed"

4) "thu"

5) "fri"

6) "sat"

7) "sun"

redis 127.0.0.1:6379> ZSCORE days sat

"5"

redis 127.0.0.1:6379> ZCOUNT days 3 6

(integer) 4

redis 127.0.0.1:6379> ZRANGEBYSCORE days 3 6

1) "thu"

2) "fri"

3) "sat"

4) "sun"

### Hash类型

Redis能够存储key对多个属性的数据（比如user1.uname user1.passwd）

redis 127.0.0.1:6379> HKEYS student

1) "name"

2) "age"

3) "sex"

redis 127.0.0.1:6379> HVALS student

1) "Ganesh"

2) "30"

3) "Male"

redis 127.0.0.1:6379> HGETALL student

1) "name"

2) "Ganesh"

3) "age"

4) "30"

5) "sex"

6) "Male"

redis 127.0.0.1:6379> HDEL student sex

(integer) 1

redis 127.0.0.1:6379> HGETALL student

1) "name"

2) "Ganesh"

3) "age"

4) "30"

Hash数据结构能够批量修改和获取

redis 127.0.0.1:6379> HMSET kid name Akshi age 2 sex Female

OK

redis 127.0.0.1:6379> HMGET kid name age sex

1) "Akshi"

2) "2"

3) "Female"

### Publish/Subscribe

Redis支持这样一种特性，你可以将数据推到某个信息管道中，然后其它人可以通过订阅这些管道来获取推送过来的信息。

#### 订阅信息管道

用一个客户端订阅管道

redis 127.0.0.1:6379> SUBSCRIBE channelone

Reading messages... (press Ctrl-C to quit)

1) "subscribe"

2) "channelone"

3) (integer) 1

另一个客户端往这个管道推送信息

redis 127.0.0.1:6379> PUBLISH channelone hello

(integer) 1

redis 127.0.0.1:6379> PUBLISH channelone world

(integer) 1

然后第一个客户端就能获取到推送的信息

redis 127.0.0.1:6379> SUBSCRIBE channelone

Reading messages... (press Ctrl-C to quit)

1) "subscribe"

2) "channelone"

3) (integer) 1

1) "message"

2) "channelone"

3) "hello"

1) "message"

2) "channelone"

3) "world"

#### 按一定模式批量订阅

用下面的命令订阅所有channel开头的信息通道

redis 127.0.0.1:6379> PSUBSCRIBE channel\*

Reading messages... (press Ctrl-C to quit)

1) "psubscribe"

2) "channel\*"

3) (integer) 1

在另一个客户端对两个推送信息

redis 127.0.0.1:6379> PUBLISH channelone hello

(integer) 1

redis 127.0.0.1:6379> PUBLISH channeltwo world

(integer) 1

然后在第一个客户端就能收到推送的信息

redis 127.0.0.1:6379> PSUBSCRIBE channel\*

Reading messages... (press Ctrl-C to quit)

1) "psubscribe"

2) "channel\*"

3) (integer) 1

1) "pmessage"

2) "channel\*"

3) "channelone"

4) "hello"

1) "pmessage"

2) "channel\*"

3) "channeltwo"

4) "world"

### 数据过期设置

Redis支持按key设置过期时间，过期后值将被删除（在客户端看来是补删除了的）

用TTL命令可以获取某个key值的过期时间（-1表示永不过期）

redis 127.0.0.1:6379> SET name "John Doe"

OK

redis 127.0.0.1:6379> TTL name

(integer) -1

下面命令先用EXISTS命令查看key值是否存在，然后设置了5秒的过期时间

redis 127.0.0.1:6379> SET name "John Doe"

OK

redis 127.0.0.1:6379> EXISTS name

(integer) 1

redis 127.0.0.1:6379> EXPIRE name 5

(integer) 1

5秒后再查看

redis 127.0.0.1:6379> EXISTS name

(integer) 0

redis 127.0.0.1:6379> GET name

(nil)

这个值已经没有了。

上在是直接设置多少秒后过期，你也可以设置在某个时间点过期，下面例子是设置2011-09-24 00:40:00过期。

redis 127.0.0.1:6379> SET name "John Doe"

OK

redis 127.0.0.1:6379> EXPIREAT name 1316805000

(integer) 1

redis 127.0.0.1:6379> EXISTS name

(integer) 0

### 事务性

Redis本身支持一些简单的组合型的命令，比如以NX结尾命令都是判断在这个值没有时才进行某个命令。

redis 127.0.0.1:6379> SET name "John Doe"

OK

redis 127.0.0.1:6379> SETNX name "Dexter Morgan"

(integer) 0

redis 127.0.0.1:6379> GET name

"John Doe"

redis 127.0.0.1:6379> GETSET name "Dexter Morgan"

"John Doe"

redis 127.0.0.1:6379> GET name

"Dexter Morgan"

当然，Redis还支持自定义的命令组合，通过MULTI和EXEC，将几个命令组合起来执行

redis 127.0.0.1:6379> SET counter 0

OK

redis 127.0.0.1:6379> MULTI

OK

redis 127.0.0.1:6379> INCR counter

QUEUED

redis 127.0.0.1:6379> INCR counter

QUEUED

redis 127.0.0.1:6379> INCR counter

QUEUED

redis 127.0.0.1:6379> EXEC

1) (integer) 1

2) (integer) 2

3) (integer) 3

redis 127.0.0.1:6379> GET counter

"3"

你还可以用DICARD命令来中断执行中的命令序列

redis 127.0.0.1:6379> SET newcounter 0

OK

redis 127.0.0.1:6379> MULTI

OK

redis 127.0.0.1:6379> INCR newcounter

QUEUED

redis 127.0.0.1:6379> INCR newcounter

QUEUED

redis 127.0.0.1:6379> INCR newcounter

QUEUED

redis 127.0.0.1:6379> DISCARD

OK

redis 127.0.0.1:6379> GET newcounter

"0"

### 持久化

Redis的所有数据都存储在内存中，但是他也提供对这些数据的持久化。

#### 数据快照

数据快照的原理是将整个Redis中存的所有数据遍历一遍存到一个扩展名为rdb的数据文件中。通过SAVE命令可以调用这个过程。

redis 127.0.0.1:6379> SET name "John Doe"

OK

redis 127.0.0.1:6379> SAVE

OK

redis 127.0.0.1:6379> SET name "Sheldon Cooper"

OK

redis 127.0.0.1:6379> BGSAVE

Background saving started

如果你是使用的brew在Mac OSX上安全的Redis，那么rdb文件会存在如下路径

/usr/local/var/db/redis/dump.rdb

#### Append-Only File（追加式的操作日志记录）

Redis还支持一种追加式的操作日志记录，叫append only file，其日志文件以aof结局，我们一般各为aof文件。要开启aof日志的记录，你需要在配置文件中进行如下设置：

appendonly yes

这时候你所有的操作都会记录在aof日志文件中

### 管理命令

Redis支持多个DB，默认是16个，你可以设置将数据存在哪一个DB中，不同DB间的数据具有隔离性。也可以在多个DB间移动数据。

redis 127.0.0.1:6379> SELECT 0

OK

redis 127.0.0.1:6379> SET name "John Doe"

OK

redis 127.0.0.1:6379> SELECT 1

OK

redis 127.0.0.1:6379[1]> GET name

(nil)

redis 127.0.0.1:6379[1]> SELECT 0

OK

redis 127.0.0.1:6379> MOVE name 1

(integer) 1

redis 127.0.0.1:6379> SELECT 1

OK

redis 127.0.0.1:6379[1]> GET name

"John Doe"

Redis还能进行一些如下操作，获取一些运行信息

redis 127.0.0.1:6379[1]> DBSIZE

(integer) 1

redis 127.0.0.1:6379[1]> INFO

redis\_version:2.2.13

redis\_git\_sha1:00000000

redis\_git\_dirty:0

arch\_bits:64

multiplexing\_api:kqueue

......

Redis还支持对某个DB数据进行清除（当然清空所有数据的操作也是支持的）

redis 127.0.0.1:6379> SET name "John Doe"

OK

redis 127.0.0.1:6379> DBSIZE

(integer) 1

redis 127.0.0.1:6379> SELECT 1

OK

redis 127.0.0.1:6379[1]> SET name "Sheldon Cooper"

OK

redis 127.0.0.1:6379[1]> DBSIZE

(integer) 1

redis 127.0.0.1:6379[1]> SELECT 0

OK

redis 127.0.0.1:6379> FLUSHDB

OK

redis 127.0.0.1:6379> DBSIZE

(integer) 0

redis 127.0.0.1:6379> SELECT 1

OK

redis 127.0.0.1:6379[1]> DBSIZE

(integer) 1

redis 127.0.0.1:6379[1]> FLUSHALL

OK

redis 127.0.0.1:6379[1]> DBSIZE

(integer) 0

## 数据结构

redis提供五种数据类型：[string](http://baike.baidu.com/view/1005388.htm)，hash，list，set及zset(sorted set)。

### string（字符串）

string是最简单的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value，其上支持的操作与Memcached的操作类似。但它的功能更丰富。

redis采用结构sdshdr和sds封装了字符串，字符串相关的操作实现在[源文件](http://baike.baidu.com/view/385166.htm)sds.h/sds.c中。Sdshdr

[数据结构](http://baike.baidu.com/view/9900.htm)定义如下：

typedef char \*sds;

struct sdshdr {

long len;

long free;

char buf[];

};

## Windows下安装使用Redis

官方下载:

<http://redis.io/download>

### 启动Redis

Redis-server.exe redis.conf

这个窗口要保持开启  关闭时redis服务会自动关闭

redis会自动保存数据到硬盘 所以图中是我第二次开启时 多了一个 DB loaded from disk

### 连接Redis

redis-cli.exe -h 192.168.10.61 -p 6379

# Redis命令

## Set

SET key value

**加入版本 1.0.0。**

**时间复杂度：** O(1)

将key和value对应。如果key已经存在了，它会被覆盖，而不管它是什么类型。

### 返回值

[状态码](http://www.redis.cn/topics/protocol.html#status-reply)：总是OK，因为SET不会失败。

### 例子:

redis>  SET mykey "Hello"

OK

redis>  GET mykey

"Hello"

窗体顶端

redis>

窗体底端

## SETBIT

### SETBIT key offset value

**加入版本 2.2.0。**

**时间复杂度：** O(1)。

设置或者清空key的value(字符串)在offset处的bit值。

那个位置的bit要么被设置，要么被清空，这个由value（只能是0或者1）来决定。当key不存在的时候，就创建一个新的字符串value。要确保这个字符串大到在offset处有bit值。参数offset需要大于等于0，并且小于232(限制bitmap大小为512)。当key对应的字符串增大的时候，新增的部分bit值都是设置为0。

警告：当set最后一个bit(offset等于232-1)并且key还没有一个字符串value或者其value是个比较小的字符串时，Redis需要立即分配所有内存，这有可能会导致服务阻塞一会。在一台2010MacBook Pro上，offset为232-1（分配512MB）需要～300ms，offset为230-1(分配128MB)需要～80ms，offset为228-1（分配32MB）需要～30ms，offset为226-1（分配8MB）需要8ms。注意，一旦第一次内存分配完，后面对同一个key调用[SETBIT](http://www.redis.cn/commands/setbit.html)就不会预先得到内存分配。

### 返回值

[整型数字](http://www.redis.cn/topics/protocol.html#integer-reply)：在offset处原来的bit值

### 例子

redis>  SETBIT mykey 7 1

(integer) 0

redis>  SETBIT mykey 7 0

(integer) 1

redis>  GET mykey

"\x00"

窗体顶端

redis>

窗体底端

## SETEX

### SETEX key seconds value

**加入版本 2.0.0。**

**时间复杂度：** O(1)。

设置key对应字符串value，并且设置key在给定的seconds时间之后超时过期。这个命令等效于执行下面的命令：

SET mykey value

EXPIRE mykey seconds

[SETEX](http://www.redis.cn/commands/setex.html)是原子的，也可以通过把上面两个命令放到[MULTI](http://www.redis.cn/commands/multi.html)/[EXEC](http://www.redis.cn/commands/exec.html)块中执行的方式重现。相比连续执行上面两个命令，它更快，因为当Redis当做缓存使用时，这个操作更加常用。

### 返回值

### 例子

redis>  SETEX mykey 10 "Hello"

OK

redis>  TTL mykey

(integer) 10

redis>  GET mykey

"Hello"

窗体顶端

redis>

## SETNX

### SETNX key value

**加入版本 1.0.0。**

**时间复杂度：** O(1)

如果key不存在，就设置key对应字符串value。在这种情况下，该命令和[SET](http://www.redis.cn/commands/set.html)一样。当key已经存在时，就不做任何操作。[SETNX](http://www.redis.cn/commands/setnx.html)是"SET if Not eXists"。

### 返回值

[数字](http://www.redis.cn/topics/protocol.html#integer-reply)，只有以下两种值：

* 1 如果key被set
* 0 如果key没有被set

### 例子

redis>  SETNX mykey "Hello"

(integer) 1

redis>  SETNX mykey "World"

(integer) 0

redis>  GET mykey

"Hello"

窗体顶端

redis>

窗体底端

## GET

### GET key

**加入版本 1.0.0。**

**时间复杂度：** O(1)。

返回key的value。如果key不存在，返回特殊值nil。如果key的value不是string，就返回错误，因为GET只处理string类型的values

### 返回值

[Bulk reply](http://www.redis.cn/topics/protocol.html#bulk-reply):key对应的value，或者nil（key不存在时）

### 例子

redis>  GET nonexisting

(nil)

redis>  SET mykey "Hello"

OK

redis>  GET mykey

"Hello"

窗体顶端

redis>

## GETBIT

### GETBIT key offset

返回key对应的string在offset处的bit值 当offset超出了字符串长度的时候，这个字符串就被假定为由0比特填充的连续空间。当key不存在的时候，它就认为是一个空字符串，所以offset总是超出范围，然后value也被认为是由0比特填充的连续空间。到内存分配。

### 返回值

[整型数字](http://www.redis.cn/topics/protocol.html#integer-reply)：在offset处的bit值

### 例子

redis>  SETBIT mykey 7 1

(integer) 0

redis>  GETBIT mykey 0

(integer) 0

redis>  GETBIT mykey 7

(integer) 1

redis>  GETBIT mykey 100

(integer) 0

窗体顶端

redis>

## GETRANGE

### GETRANGE key start end

**加入版本 2.4.0。**

**时间复杂度：** O(N) ，这里的N是返回的string的长度。复杂度是由返回的字符串长度决定的，但是因为从一个已经存在的字符串创建一个子串是很容易的，所以对于较小的字符串，可以认为是O(1)的复杂度。

警告：这个命令是被改成GETRANGE的，在小于2.0的Redis版本中叫SUBSTR。 返回key对应的字符串value的子串，这个子串是由start和end位移决定的（两者都在string内）。可以用负的位移来表示从string尾部开始数的下标。所以-1就是最后一个字符，-2就是倒数第二个，以此类推。

这个函数处理超出范围的请求时，都把结果限制在string内。

### 返回值

[Bulk reply](http://www.redis.cn/topics/protocol.html#bulk-reply)（专有词汇，protocol有解释）。

### 例子

redis>  SET mykey "This is a string"

OK

redis>  GETRANGE mykey 0 3

"This"

redis>  GETRANGE mykey -3 -1

"ing"

redis>  GETRANGE mykey 0 -1

"This is a string"

redis>  GETRANGE mykey 10 100

"string"

窗体顶端

redis>

窗体底端

## GETSET

### GETSET key value

**加入版本 1.0.0。**

**时间复杂度：** O(1)。

自动将key对应到value并且返回原来key对应的value。如果key存在但是对应的value不是字符串，就返回错误。

### 设计模式

[GETSET](http://www.redis.cn/commands/getset.html)可以和[INCR](http://www.redis.cn/commands/incr.html)一起使用实现支持重置的计数功能。举个例子：每当有事件发生的时候，一段程序都会调用[INCR](http://www.redis.cn/commands/incr.html)给key mycounter加1，但是有时我们需要获取计数器的值，并且自动将其重置为0。这可以通过GETSET mycounter "0"来实现：

### 例子

redis>  INCR mycounter

(integer) 1

redis>  GETSET mycounter "0"

"1"

redis>  GET mycounter

"0"

窗体顶端

redis>

窗体底端

## HSET

### HSET key field value

**加入版本 2.0.0。**

**时间复杂度：** O(1)。

设置 key 指定的哈希集中指定字段的值。如果 key 指定的哈希集不存在，会创建一个新的哈希集并与 key 关联。如果字段在哈希集中存在，它将被重写。

### 返回值

[整数](http://www.redis.cn/topics/protocol.html#integer-reply)：含义如下

* 1如果field是一个新的字段
* 0如果field原来在map里面已经存在

### 例子

redis>  HSET myhash field1 "Hello"

(integer) 1

redis>  HGET myhash field1

"Hello"

窗体顶端

redis>

## HGET

### HGET key field

**加入版本 2.0.0。**

**时间复杂度：** O(1)。

返回 key 指定的哈希集中该字段所关联的值

### 返回值

[散值](http://www.redis.cn/topics/protocol.html#bulk-reply)：该字段所关联的值。当字段不存在或者 key 不存在时返回nil。

### 例子

redis>  HSET myhash field1 "foo"

(integer) 1

redis>  HGET myhash field1

"foo"

redis>  HGET myhash field2

(nil)

窗体顶端

redis>

窗体底端

## LPUSH

### LPUSH key value [value ...]

**加入版本 1.0.0。**

**时间复杂度：** O(1)。

将所有指定的值插入到存于 key 的列表的头部。如果 key 不存在，那么在进行 push 操作前会创建一个空列表。 如果 key 对应的值不是一个 list 的话，那么会返回一个错误。

可以使用一个命令把多个元素 push 进入列表，只需在命令末尾加上多个指定的参数。元素是从最左端的到最右端的、一个接一个被插入到 list 的头部。 所以对于这个命令例子 LPUSH mylist a b c，返回的列表是 c 为第一个元素， b 为第二个元素， a 为第三个元素。

### 返回值

[整型回复](http://www.redis.cn/topics/protocol.html#integer-reply): 在 push 操作后的 list 长度。

### 历史

>= 2.4: 接受多个 value 参数。版本老于 2.4 的 Redis 只能每条命令 push 一个值。

### 例子

redis>  LPUSH mylist "world"

(integer) 1

redis>  LPUSH mylist "hello"

(integer) 2

redis>  LRANGE mylist 0 -1

1) "hello"

2) "world"

窗体顶端

redis>

窗体底端

## LPOP

## ZRANK key member

返回有序集key中成员member的排名。其中有序集成员按score值递增(从小到大)顺序排列。排名以0为底，也就是说，score值最小的成员排名为0。

使用[ZREVRANK](http://www.redis.cn/commands/zrevrank.html)命令可以获得成员按score值递减(从大到小)排列的排名。

### 返回值

* 如果member是有序集key的成员，返回member的排名的[整数](http://www.redis.cn/topics/protocol.html#integer-reply)。
* 如果member不是有序集key的成员，返回[Bulk reply](http://www.redis.cn/topics/protocol.html#bulk-reply): nil。

### 例子:

redis>  ZADD myzset 1 "one"

(integer) 1

redis>  ZADD myzset 2 "two"

(integer) 1

redis>  ZADD myzset 3 "three"

(integer) 1

redis>  ZRANK myzset "three"

(integer) 2

redis>  ZRANK myzset "four"

(nil)

窗体顶端

redis>

窗体底端

# Redis设计实现

## 内部数据结构

Redis 和其他很多 key-value 数据库的不同之处在于， Redis 不仅支持简单的字符串键值对， 它还提供了一系列数据结构类型值， 比如列表、哈希、集合和有序集， 并在这些数据结构类型上定义了一套强大的 API 。

通过对不同类型的值进行操作， Redis 可以很轻易地完成其他只支持字符串键值对的 key-value 数据库很难（或者无法）完成的任务

在 Redis 的内部， 数据结构类型值由高效的数据结构和算法进行支持， 并且在 Redis 自身的构建当中， 也大量用到了这些数据结构。

### 简单动态字符串

Sds （Simple Dynamic String，简单动态字符串）是 Redis 底层所使用的字符串表示， 几乎所有的 Redis 模块中都用了 sds。

本章将对 sds 的实现、性能和功能等方面进行介绍， 并说明 Redis 使用 sds 而不是传统 C 字符串的原因。

#### sds 的用途

Sds 在 Redis 中的主要作用有以下两个：

实现字符串对象（StringObject）；

在 Redis 程序内部用作 char\* 类型的替代品；

#### 实现字符串对象

Redis 是一个键值对数据库（key-value DB）， 数据库的值可以是字符串、集合、列表等多种类型的对象， 而数据库的键则总是字符串对象。

对于那些包含字符串值的字符串对象来说， 每个字符串对象都包含一个 sds 值。

举个例子， 以下命令创建了一个新的数据库键值对， 这个键值对的键和值都是字符串对象， 它们都包含一个 sds 值：

redis> SET book "Mastering C++ in 21 days"

OK

redis> GET book

"Mastering C++ in 21 days"

以下命令创建了另一个键值对， 它的键是字符串对象， 而值则是一个集合对象：

redis> SADD nosql "Redis" "MongoDB" "Neo4j"

(integer) 3

redis> SMEMBERS nosql

1) "Neo4j"

2) "Redis"

3) "MongoDB"

### Redis 中的字符串

在 C 语言中，字符串可以用一个 \0 结尾的 char 数组来表示。

比如说， hello world 在 C 语言中就可以表示为 "hello world\0" 。

这种简单的字符串表示，在大多数情况下都能满足要求，但是，它并不能高效地支持长度计算和追加（append）这两种操作：

* 每次计算字符串长度（strlen(s)）的复杂度为 *θ*(*N*) 。
* 对字符串进行 N 次追加，必定需要对字符串进行 N 次内存重分配（realloc）。

在 Redis 内部， 字符串的追加和长度计算很常见， 而 [APPEND](http://redis.readthedocs.org/en/latest/string/append.html#append) 和 [STRLEN](http://redis.readthedocs.org/en/latest/string/strlen.html#strlen) 更是这两种操作，在 Redis 命令中的直接映射， 这两个简单的操作不应该成为性能的瓶颈。

另外， Redis 除了处理 C 字符串之外， 还需要处理单纯的字节数组， 以及服务器协议等内容， 所以为了方便起见， Redis 的字符串表示还应该是[二进制安全的](http://en.wikipedia.org/wiki/Binary-safe)： 程序不应对字符串里面保存的数据做任何假设， 数据可以是以 \0 结尾的 C 字符串， 也可以是单纯的字节数组， 或者其他格式的数据。

考虑到这两个原因， Redis 使用 sds 类型替换了 C 语言的默认字符串表示： sds 既可高效地实现追加和长度计算， 同时是二进制安全的。

#### sds 的实现

在前面的内容中， 我们一直将 sds 作为一种抽象数据结构来说明， 实际上， 它的实现由以下两部分组成：

typedef char \*sds;

struct sdshdr {

// buf 已占用长度

int len;

// buf 剩余可用长度

int free;

// 实际保存字符串数据的地方

char buf[];

};

其中，类型 sds 是 char \* 的别名（alias），而结构 sdshdr 则保存了 len 、 free 和 buf 三个属性。

作为例子，以下是新创建的，同样保存 hello world 字符串的 sdshdr 结构：

struct sdshdr {

len = 11;

free = 0;

buf = "hello world\0"; // buf 的实际长度为 len + 1

};

通过 len 属性， sdshdr 可以实现复杂度为 *θ*(1) 的长度计算操作。

另一方面， 通过对 buf 分配一些额外的空间， 并使用 free 记录未使用空间的大小， sdshdr 可以让执行追加操作所需的内存重分配次数大大减少， 下一节我们就会来详细讨论这一点。

当然， sds 也对操作的正确实现提出了要求 —— 所有处理 sdshdr 的函数，都必须正确地更新 len 和 free 属性，否则就会造成 bug

### 优化追加操作

在前面说到过，利用 sdshdr 结构，除了可以用 *θ*(1) 复杂度获取字符串的长度之外，还可以减少追加（append）操作所需的内存重分配次数，以下就来详细解释这个优化的原理。

为了易于理解，我们用一个 Redis 执行实例作为例子，解释一下，当执行以下代码时， Redis 内部发生了什么：

redis> SET msg "hello world"

OK

redis> APPEND msg " again!"

(integer) 18

redis> GET msg

"hello world again!"

首先， SET 命令创建并保存 hello world 到一个 sdshdr 中，这个 sdshdr 的值如下：

struct sdshdr {

len = 11;

free = 0;

buf = "hello world\0";

}

当执行 [APPEND](http://redis.readthedocs.org/en/latest/string/append.html#append) 命令时，相应的 sdshdr 被更新，字符串 " again!" 会被追加到原来的 "hello world" 之后：

struct sdshdr {

len = 18;

free = 18;

buf = "hello world again!\0 "; // 空白的地方为预分配空间，共 18 + 18 + 1 个字节

}

注意， 当调用 SET 命令创建 sdshdr 时， sdshdr 的 free 属性为 0 ， Redis 也没有为 buf 创建额外的空间 —— 而在执行 [*APPEND*](http://redis.readthedocs.org/en/latest/string/append.html#append) 之后， Redis 为 buf 创建了多于所需空间一倍的大小。

在这个例子中， 保存 "hello world again!" 共需要 18 + 1 个字节， 但程序却为我们分配了 18 + 18 + 1 = 37 个字节 —— 这样一来， 如果将来再次对同一个 sdshdr 进行追加操作， 只要追加内容的长度不超过 free 属性的值， 那么就不需要对 buf 进行内存重分配。

比如说， 执行以下命令并不会引起 buf 的内存重分配， 因为新追加的字符串长度小于 18

redis> APPEND msg " again!"

(integer) 25

再次执行 [APPEND](http://redis.readthedocs.org/en/latest/string/append.html#append) 命令之后， msg 的值所对应的 sdshdr 结构可以表示如下

struct sdshdr {

len = 25;

free = 11;

buf = "hello world again! again!\0 "; // 空白的地方为预分配空间，共 18 + 18 + 1 个字节

}

sds.c/sdsMakeRoomFor 函数描述了 sdshdr 的这种内存预分配优化策略， 以下是这个函数的伪代码版本：

def sdsMakeRoomFor(sdshdr, required\_len):

# 预分配空间足够，无须再进行空间分配

if (sdshdr.free >= required\_len):

return sdshdr

# 计算新字符串的总长度

newlen = sdshdr.len + required\_len

# 如果新字符串的总长度小于 SDS\_MAX\_PREALLOC

# 那么为字符串分配 2 倍于所需长度的空间

# 否则就分配所需长度加上 SDS\_MAX\_PREALLOC 数量的空间

if newlen < SDS\_MAX\_PREALLOC:

newlen \*= 2

else:

newlen += SDS\_MAX\_PREALLOC

# 分配内存

newsh = zrelloc(sdshdr, sizeof(struct sdshdr)+newlen+1)

# 更新 free 属性

newsh.free = newlen - sdshdr.len

# 返回

return newsh

在目前版本的 Redis 中， SDS\_MAX\_PREALLOC 的值为 1024 \* 1024 ， 也就是说， 当大小小于 1MB 的字符串执行追加操作时， sdsMakeRoomFor 就为它们分配多于所需大小一倍的空间； 当字符串的大小大于 1MB ， 那么 sdsMakeRoomFor 就为它们额外多分配 1MB 的空间。

#### 小结

* Redis 的字符串表示为 sds ，而不是 C 字符串（以 \0 结尾的 char\*）。
* 对比 C 字符串， sds 有以下特性：
  + 可以高效地执行长度计算（strlen）；
  + 可以高效地执行追加操作（append）；
  + 二进制安全；
* sds 会为追加操作进行优化：加快追加操作的速度，并降低内存分配的次数，代价是多占用了一些内存，而且这些内存不会被主动释放。

### 双端链表

链表作为数组之外的一种常用序列抽象， 是大多数高级语言的基本数据类型， 因为 C 语言本身不支持链表类型， 大部分 C 程序都会自己实现一种链表类型， Redis 也不例外 —— 实现了一个双端链表结构。双端链表作为一种常见的数据结构， 在大部分的数据结构或者算法书里都有讲解， 因此， 这一章关注的是 Redis 双端链表的具体实现， 以及该实现的 API ， 而对于双端链表本身， 以及双端链表所对应的算法， 则不做任何解释。

#### 双端链表的应用

双端链表作为一种通用的数据结构， 在 Redis 内部使用得非常多： 既是 Redis 列表结构的底层实现之一， 同时为大量 Redis 模块所用， 用于构建 Redis 的其他功能。

##### 实现 Redis 的列表类型

双端链表还是 Redis 列表类型的底层实现之一， 当对列表类型的键进行操作 —— 比如执行 [RPUSH](http://redis.readthedocs.org/en/latest/list/rpush.html#rpush) 、 [LPOP](http://redis.readthedocs.org/en/latest/list/lpop.html#lpop) 或 [LLEN](http://redis.readthedocs.org/en/latest/list/llen.html#llen) 等命令时， 程序在底层操作的可能就是双端链表

redis> RPUSH brands Apple Microsoft Google

(integer) 3

redis> LPOP brands

"Apple"

redis> LLEN brands

(integer) 2

redis> LRANGE brands 0 -1

1) "Microsoft"

2) "Google"

Redis 列表使用两种数据结构作为底层实现：

1. 双端链表
2. 压缩列表

因为双端链表占用的内存比压缩列表要多， 所以当创建新的列表键时， 列表会优先考虑使用压缩列表作为底层实现， 并且在有需要的时候， 才从压缩列表实现转换到双端链表实现。

后续章节会对压缩链表和 Redis 类型做更进一步的介绍。

##### Redis 自身功能的构建

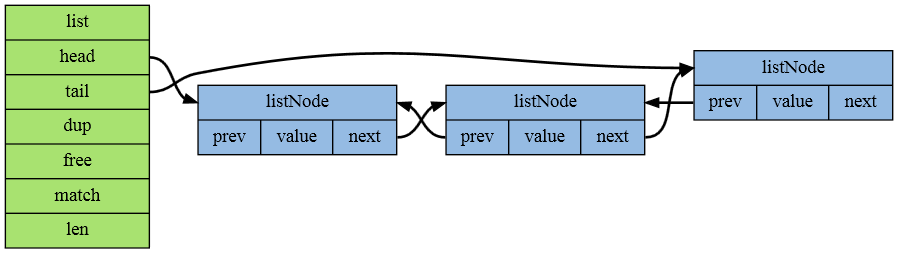
除了实现列表类型以外， 双端链表还被很多 Redis 内部模块所应用：

* 事务模块使用双端链表依序保存输入的命令；
* 服务器模块使用双端链表来保存多个客户端；
* 订阅/发送模块使用双端链表来保存订阅模式的多个客户端；
* 事件模块使用双端链表来保存时间事件（time event）；

类似的应用还有很多， 在后续的章节中我们将看到， 双端链表在 Redis 中发挥着重要的作用。

#### 双端链表的实现

双端链表的实现由 listNode 和 list 两个数据结构构成， 下图展示了由这两个结构组成的一个双端链表实例：



其中， listNode 是双端链表的节点：

typedef struct listNode {

// 前驱节点

struct listNode \*prev;

// 后继节点

struct listNode \*next;

// 值

void \*value;

} listNode;

而 list 则是双端链表本身：

typedef struct list {

// 表头指针

listNode \*head;

// 表尾指针

listNode \*tail;

// 节点数量

unsigned long len;

// 复制函数

void \*(\*dup)(void \*ptr);

// 释放函数

void (\*free)(void \*ptr);

// 比对函数

int (\*match)(void \*ptr, void \*key);

} list;

注意， listNode 的 value 属性的类型是 void \* ，说明这个双端链表对节点所保存的值的类型不做限制。

对于不同类型的值，有时候需要不同的函数来处理这些值，因此， list 类型保留了三个函数指针 —— dup 、 free 和 match ，分别用于处理值的复制、释放和对比匹配。在对节点的值进行处理时，如果有给定这些函数，就会调用这些函数。

举个例子：当删除一个 listNode 时，如果包含这个节点的 list 的 list->free 函数不为空，就会先调用删除函数 list->free(listNode->value) 来清空节点的值，再执行余下的删除操作（比如说，释放节点）。

另外，从这两个数据结构的定义上，也可以了解到一些行为和性能特征：

* listNode 带有 prev 和 next 两个指针，因此，遍历可以双向进行：从表头到表尾，表尾到表头。
* list 保存了 head 和 tail 两个指针，因此，对链表的表头和表尾进行插入的复杂度都为 *θ*(1) —— 这是高效实现 [*LPUSH*](http://redis.readthedocs.org/en/latest/list/lpush.html#lpush) 、 [*RPOP*](http://redis.readthedocs.org/en/latest/list/rpop.html#rpop) 、 [*RPOPLPUSH*](http://redis.readthedocs.org/en/latest/list/rpoplpush.html#rpoplpush) 等命令的关键。
* list 带有保存节点数量的 len 属性，所以计算链表长度的复杂度仅为 *θ*(1) ，这也保证了 [*LLEN*](http://redis.readthedocs.org/en/latest/list/llen.html#llen) 命令不会成为性能瓶颈。

#### 迭代器

Redis 为双端链表实现了一个[迭代器](http://en.wikipedia.org/wiki/Iterator) ， 这个迭代器可以从两个方向对双端链表进行迭代：

* 沿着节点的 next 指针前进，从表头向表尾迭代；
* 沿着节点的 prev 指针前进，从表尾向表头迭代；

以下是迭代器的数据结构定义：

typedef struct listIter {

// 下一节点

listNode \*next;

// 迭代方向

int direction;

} listIter;

direction 记录迭代应该从那里开始：

* 如果值为 adlist.h/AL\_START\_HEAD ，那么迭代器执行从表头到表尾的迭代；
* 如果值为 adlist.h/AL\_START\_TAIL ，那么迭代器执行从表尾到表头的迭代；

#### 小结

* Redis 实现了自己的双端链表结构。
* 双端链表主要有两个作用：
  + 作为 Redis 列表类型的底层实现之一；
  + 作为通用数据结构，被其他功能模块所使用；
* 双端链表及其节点的性能特性如下：
  + 节点带有前驱和后继指针，访问前驱节点和后继节点的复杂度为 *O*(1) ，并且对链表的迭代可以在从表头到表尾和从表尾到表头两个方向进行；
  + 链表带有指向表头和表尾的指针，因此对表头和表尾进行处理的复杂度为 *O*(1) ；
  + 链表带有记录节点数量的属性，所以可以在 *O*(1) 复杂度内返回链表的节点数量（长度）；

### 字典

[字典（dictionary）， 又名映射（map）或关联数组（associative array），](http://en.wikipedia.org/wiki/Associative_array) 是一种抽象数据结构， 由一集键值对（key-value pairs）组成， 各个键值对的键各不相同， 程序可以添加新的键值对到字典中， 或者基于键进行查找、更新或删除等操作。

本章先对字典在 Redis 中的应用进行介绍， 接着讲解字典的具体实现方式， 以及这个字典实现要解决的问题， 最后， 以对字典迭代器的介绍作为本章的结束。

#### 字典的应用

字典在 Redis 中的应用广泛， 使用频率可以说和 SDS 以及双端链表不相上下， 基本上各个功能模块都有用到字典的地方。

其中， 字典的主要用途有以下两个：

1. 实现数据库键空间（key space）；
2. 用作 Hash 类型键的底层实现之一；

以下两个小节分别介绍这两种用途。

##### 实现数据库键空间

Redis 是一个键值对数据库， 数据库中的键值对由字典保存： 每个数据库都有一个对应的字典， 这个字典被称之为键空间（key space）。

当用户添加一个键值对到数据库时（不论键值对是什么类型）， 程序就将该键值对添加到键空间； 当用户从数据库中删除键值对时， 程序就会将这个键值对从键空间中删除； 等等

举个例子，执行 [FLUSHDB](http://redis.readthedocs.org/en/latest/server/flushdb.html#flushdb) 可以清空键空间里的所有键值对数据：

redis> FLUSHDB

OK

执行 [DBSIZE](http://redis.readthedocs.org/en/latest/server/dbsize.html#dbsize) 则返回键空间里现有的键值对

redis> DBSIZE

(integer) 0

还可以用 [SET](http://redis.readthedocs.org/en/latest/string/set.html#set) 设置一个字符串键到键空间， 并用 [GET](http://redis.readthedocs.org/en/latest/string/get.html#get) 从键空间中取出该字符串键的值：

redis> SET number 10086

OK

redis> GET number

"10086"

redis> DBSIZE

(integer) 1

##### 用作 Hash 类型键的底层实现之一

Redis 的 Hash 类型键使用以下两种数据结构作为底层实现:

1. 字典；
2. [*压缩列表*](http://www.redisbook.com/en/latest/compress-datastruct/ziplist.html#ziplist-chapter)；

因为压缩列表比字典更节省内存， 所以程序在创建新 Hash 键时， 默认使用压缩列表作为底层实现， 当有需要时， 程序才会将底层实现从压缩列表转换到字典。

当用户操作一个 Hash 键时， 键值在底层就可能是一个哈希表：

redis> HSET book name "The design and implementation of Redis"

(integer) 1

redis> HSET book type "source code analysis"

(integer) 1

redis> HSET book release-date "2013.3.8"

(integer) 1

redis> HGETALL book

1) "name"

2) "The design and implementation of Redis"

3) "type"

4) "source code analysis"

5) "release-date"

6) "2013.3.8"

#### 字典的实现

实现字典的方法有很多种：

* 最简单的就是使用链表或数组，但是这种方式只适用于元素个数不多的情况下；
* 要兼顾高效和简单性，可以使用哈希表；
* 如果追求更为稳定的性能特征，并希望高效地实现排序操作的话，则可使用更为复杂的平衡树；

在众多可能的实现中， Redis 选择了高效、实现简单的哈希表，作为字典的底层实现。

dict.h/dict 给出了这个字典的定义：

/\*

\* 字典

\*

\* 每个字典使用两个哈希表，用于实现渐进式 rehash

\*/

typedef struct dict {

// 特定于类型的处理函数

dictType \*type;

// 类型处理函数的私有数据

void \*privdata;

// 哈希表（2 个）

dictht ht[2];

// 记录 rehash 进度的标志，值为 -1 表示 rehash 未进行

int rehashidx;

// 当前正在运作的安全迭代器数量

int iterators;

} dict;

注意 dict 类型使用了两个指针，分别指向两个哈希表。

其中， 0 号哈希表（ht[0]）是字典主要使用的哈希表， 而 1 号哈希表（ht[1]）则只有在程序对 0 号哈希表进行 rehash 时才使用。

接下来两个小节将对哈希表的实现，以及哈希表所使用的哈希算法进行介绍。

##### 哈希表实现

字典所使用的哈希表实现由 dict.h/dictht 类型定义：

/\*

\* 哈希表

\*/

typedef struct dictht {

// 哈希表节点指针数组（俗称桶，bucket）

dictEntry \*\*table;

// 指针数组的大小

unsigned long size;

// 指针数组的长度掩码，用于计算索引值

unsigned long sizemask;

// 哈希表现有的节点数量

unsigned long used;

} dictht;

table 属性是个数组， 数组的每个元素都是个指向 dictEntry 结构的指针。

每个 dictEntry 都保存着一个键值对， 以及一个指向另一个 dictEntry 结构的指针：

\*

\* 哈希表节点

\*/

typedef struct dictEntry {

// 键

void \*key;

// 值

union {

void \*val;

uint64\_t u64;

int64\_t s64;

} v;

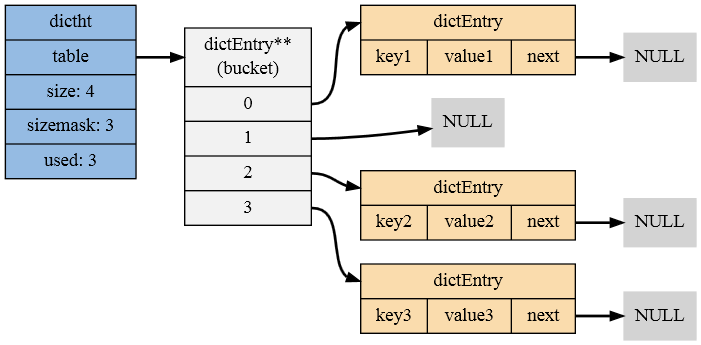
// 链往后继节点

struct dictEntry \*next;

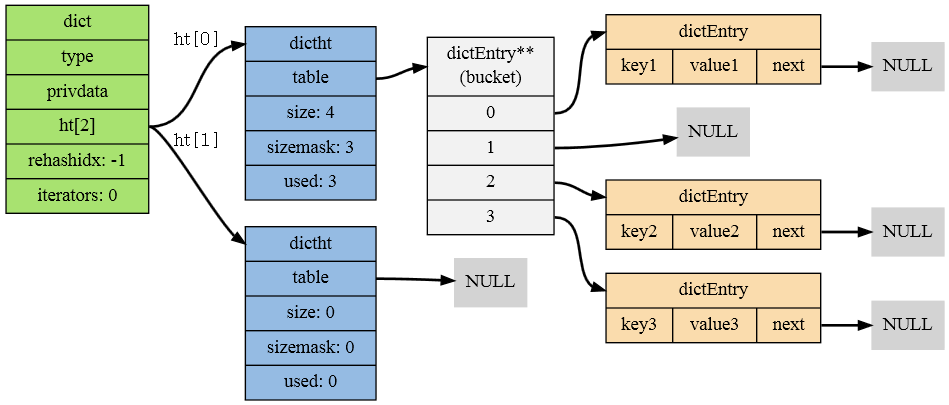
} dictEntry;

next 属性指向另一个 dictEntry 结构， 多个 dictEntry 可以通过 next 指针串连成链表， 从这里可以看出， dictht [使用链地址法来处理键碰撞](http://en.wikipedia.org/wiki/Hash_table#Separate_chaining)： 当多个不同的键拥有相同的哈希值时，哈希表用一个链表将这些键连接起来。

下图展示了一个由 dictht 和数个 dictEntry 组成的哈希表例子：



如果再加上之前列出的 dict 类型，那么整个字典结构可以表示如下：



在上图的字典示例中， 字典虽然创建了两个哈希表， 但正在使用的只有 0 号哈希表， 这说明字典未进行 rehash 状态。

##### 哈希算法

Redis 目前使用两种不同的哈希算法：

1. MurmurHash2 32 bit 算法：这种算法的分布率和速度都非常好， 具体信息请参考 MurmurHash 的主页： <http://code.google.com/p/smhasher/> 。
2. 基于 djb 算法实现的一个大小写无关散列算法：具体信息请参考 [http://www.cse.yorku.ca/~oz/hash.html](http://www.cse.yorku.ca/%7Eoz/hash.html) 。

使用哪种算法取决于具体应用所处理的数据：

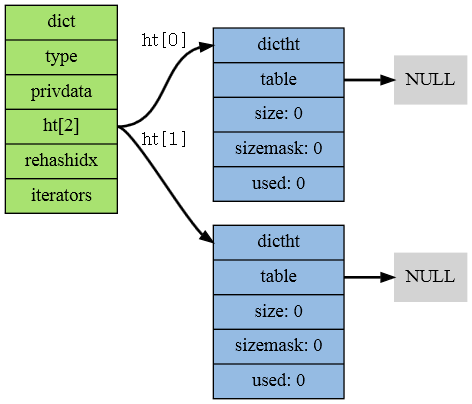
* 命令表以及 Lua 脚本缓存都用到了算法 2 。
* 算法 1 的应用则更加广泛：数据库、集群、哈希键、阻塞操作等功能都用到了这个算法。

#### 创建新字典

dictCreate 函数创建并返回一个新字典：

dict \*d = dictCreate(&hash\_type, NULL);

d 的值可以用图片表示如下：



新创建的两个哈希表都没有为 table 属性分配任何空间：

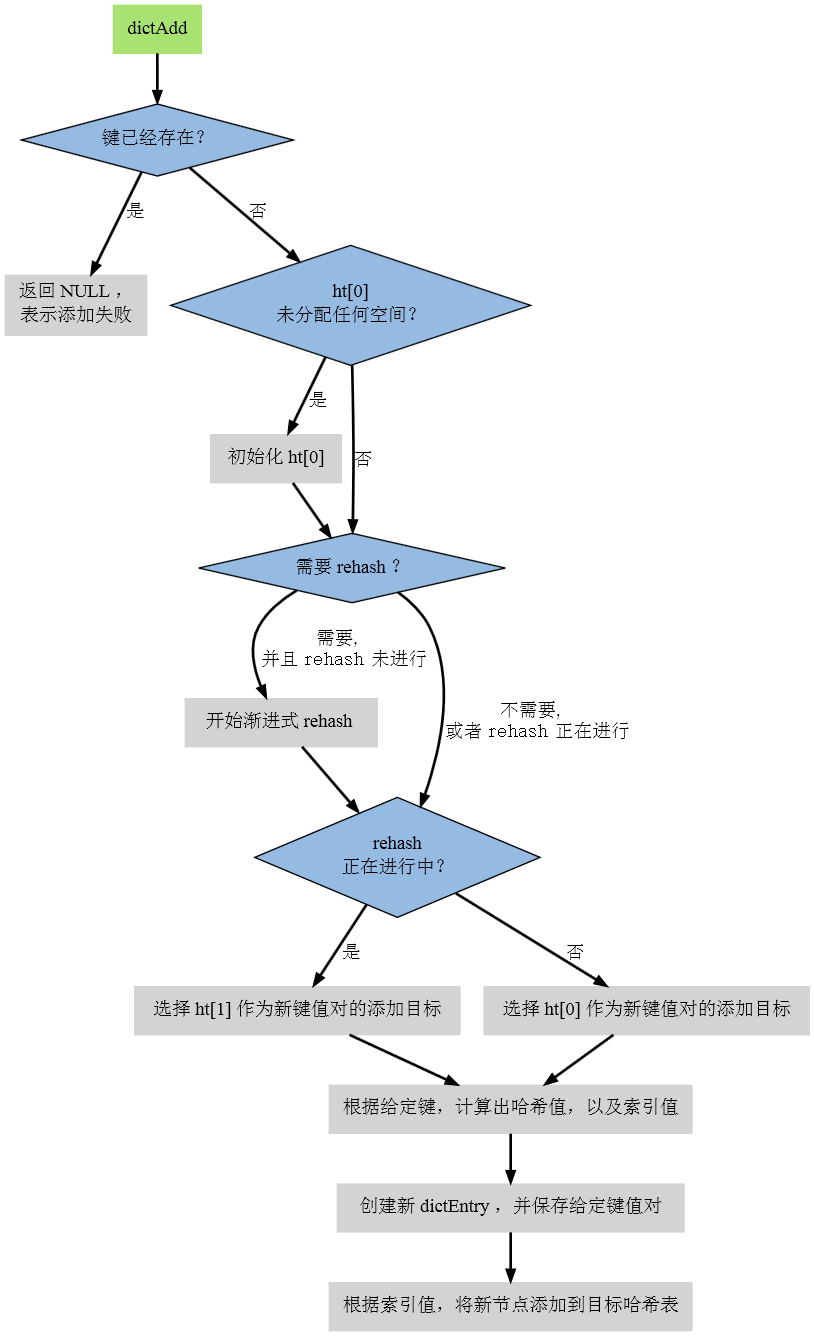
* ht[0]->table 的空间分配将在第一次往字典添加键值对时进行；
* ht[1]->table 的空间分配将在 rehash 开始时进行；

#### 添加键值对到字典

根据字典所处的状态， 将给定的键值对添加到字典可能会引起一系列复杂的操作：

* 如果字典为未初始化（即字典的 0 号哈希表的 table 属性为空），则程序需要对 0 号哈希表进行初始化；
* 如果在插入时发生了键碰撞，则程序需要处理碰撞；
* 如果插入新元素，使得字典满足了 rehash 条件，则需要启动相应的 rehash 程序；

当程序处理完以上三种情况之后，新的键值对才会被真正地添加到字典上。整个添加流程可以用下图表示



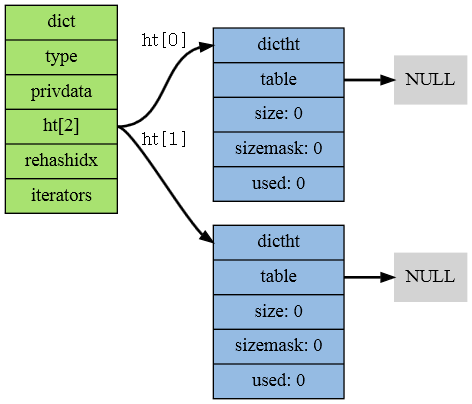
在接下来的三节中， 我们将分别看到，添加操作如何在以下三种情况中执行：

* 字典为空；
* 添加新键值对时发生碰撞处理；
* 添加新键值对时触发了 rehash 操作；

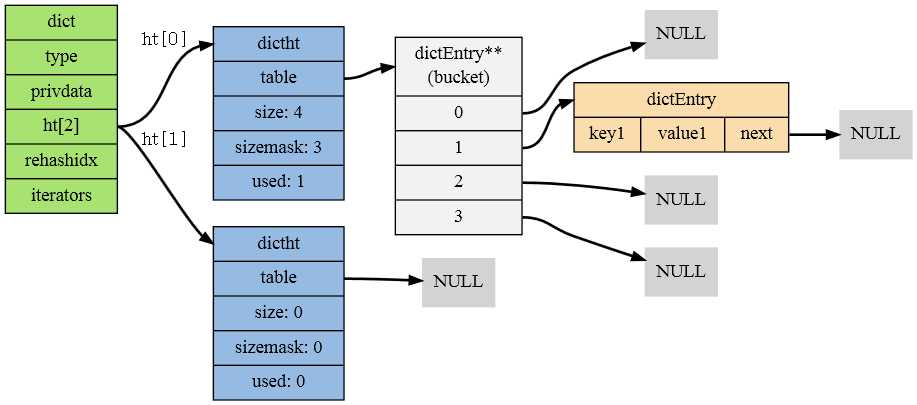
#### 添加新元素到空白字典

当第一次往空字典里添加键值对时， 程序会根据 dict.h/DICT\_HT\_INITIAL\_SIZE 里指定的大小为 d->ht[0]->table 分配空间 （在目前的版本中， DICT\_HT\_INITIAL\_SIZE 的值为 4 ）。

以下是字典空白时的样子：



以下是往空白字典添加了第一个键值对之后的样子：

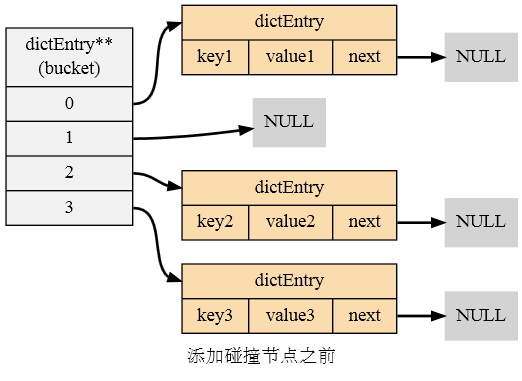


#### 添加新键值对时发生碰撞处理

在哈希表实现中， 当两个不同的键拥有相同的哈希值时， 称这两个键发生碰撞（collision）， 而哈希表实现必须想办法对碰撞进行处理。

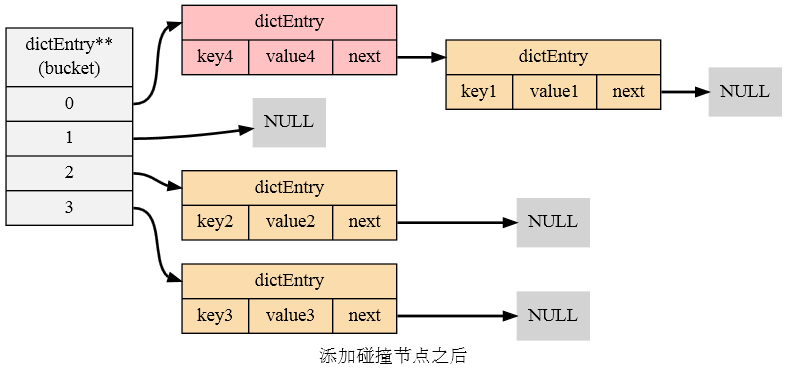
字典哈希表所使用的碰撞解决方法被称之为[链地址法](http://en.wikipedia.org/wiki/Hash_table#Separate_chaining)： 这种方法使用链表将多个哈希值相同的节点串连在一起， 从而解决冲突问题。

假设现在有一个带有三个节点的哈希表，如下图：



对于一个新的键值对 key4 和 value4 ， 如果 key4 的哈希值和 key1 的哈希值相同， 那么它们将在哈希表的 0 号索引上发生碰撞。

通过将 key4-value4 和 key1-value1 两个键值对用链表连接起来， 就可以解决碰撞的问题：

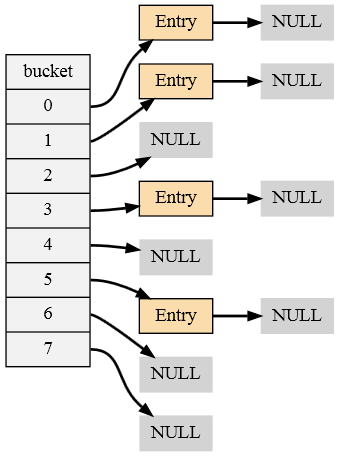


#### 添加新键值对时触发了 rehash 操作

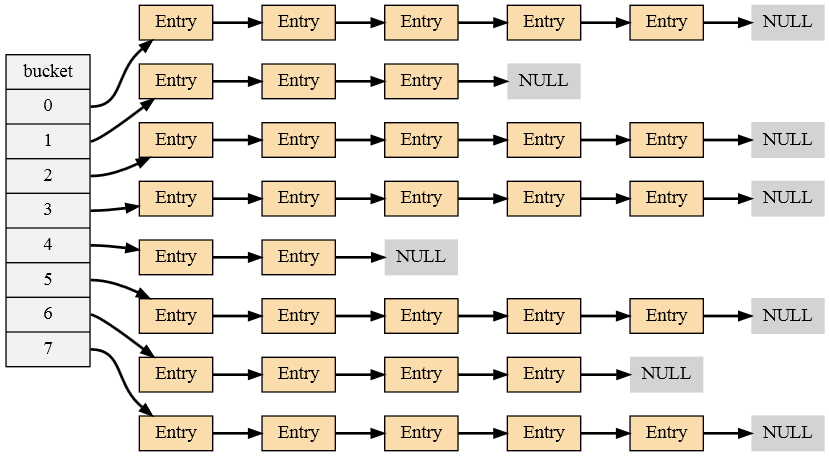
对于使用链地址法来解决碰撞问题的哈希表 dictht 来说， 哈希表的性能取决于大小（size属性）与保存节点数量（used属性）之间的比率：

* 哈希表的大小与节点数量，比率在 1:1 时，哈希表的性能最好；
* 如果节点数量比哈希表的大小要大很多的话，那么哈希表就会退化成多个链表，哈希表本身的性能优势便不复存在；

举个例子， 下面这个哈希表， 平均每次失败查找只需要访问 1 个节点（非空节点访问 2 次，空节点访问 1 次）：



而下面这个哈希表， 平均每次失败查找需要访问 5 个节点：



为了在字典的键值对不断增多的情况下保持良好的性能， 字典需要对所使用的哈希表（ht[0]）进行 rehash 操作： 在不修改任何键值对的情况下，对哈希表进行扩容， 尽量将比率维持在 1:1 左右。

dictAdd 在每次向字典添加新键值对之前， 都会对哈希表 ht[0] 进行检查， 对于 ht[0] 的 size 和 used 属性， 如果它们之间的比率 ratio = used / size 满足以下任何一个条件的话，rehash 过程就会被激活：

* 自然 rehash ： ratio >= 1 ，且变量 dict\_can\_resize 为真。
* 强制 rehash ： ratio 大于变量 dict\_force\_resize\_ratio （目前版本中， dict\_force\_resize\_ratio 的值为 5 ）。

什么时候 dict\_can\_resize 会为假？

在前面介绍字典的应用时也说到过， 数据库就是字典， 数据库里的哈希类型键也是字典， 当 Redis 使用子进程对数据库执行后台持久化任务时（比如执行 BGSAVE 或 BGREWRITEAOF 时）， 为了最大化地利用系统的 [copy on write](http://en.wikipedia.org/wiki/Copy-on-write) 机制， 程序会暂时将 dict\_can\_resize 设为假， 避免执行自然 rehash ， 从而减少程序对内存的触碰（touch）。

当持久化任务完成之后， dict\_can\_resize 会重新被设为真。

另一方面， 当字典满足了强制 rehash 的条件时， 即使 dict\_can\_resize 不为真（有 BGSAVE 或 BGREWRITEAOF 正在执行）， 这个字典一样会被 rehash 。

#### Rehash 执行过程

字典的 rehash 操作实际上就是执行以下任务：

*  创建一个比 ht[0]->table 更大的 ht[1]->table ；
* 将 ht[0]->table 中的所有键值对迁移到 ht[1]->table ；
*  将原有 ht[0] 的数据清空，并将 ht[1] 替换为新的 ht[0] ；

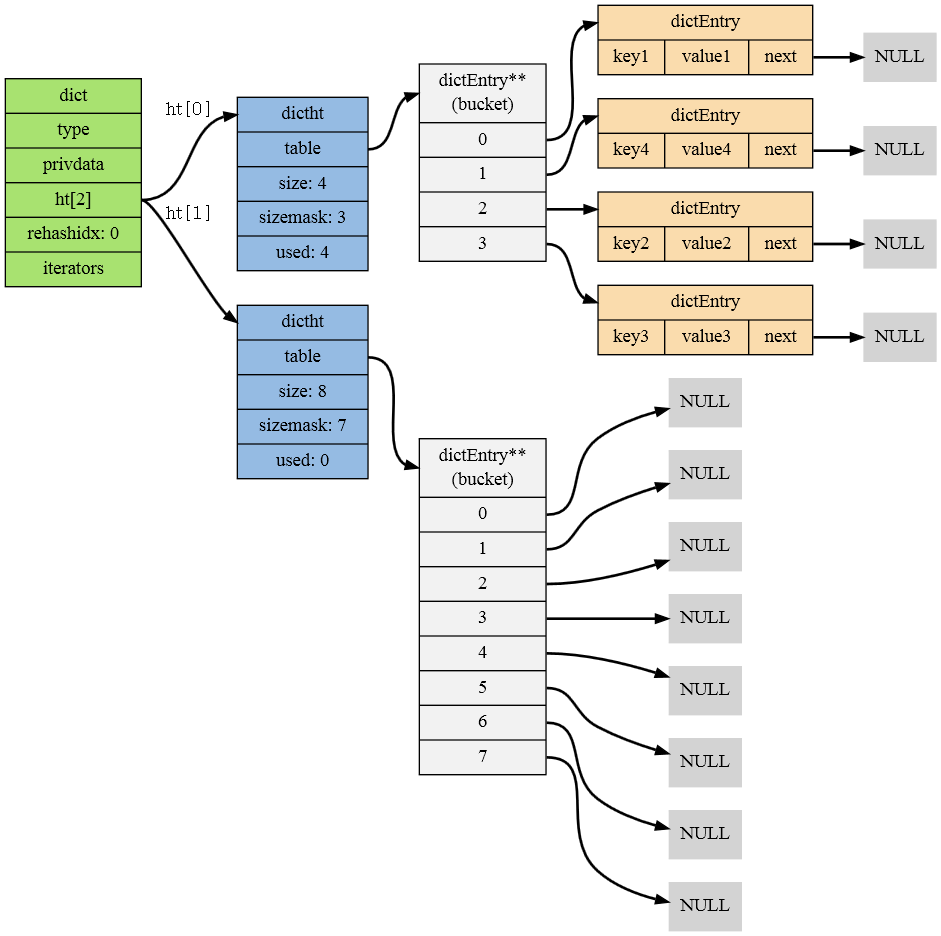
经过以上步骤之后， 程序就在不改变原有键值对数据的基础上， 增大了哈希表的大小。作为例子， 以下四个小节展示了一次对哈希表进行 rehash 的完整过程。

##### 开始 rehash

这个阶段有两个事情要做：

*  设置字典的 rehashidx 为 0 ，标识着 rehash 的开始；
* 为 ht[1]->table 分配空间，大小至少为 ht[0]->used 的两倍；

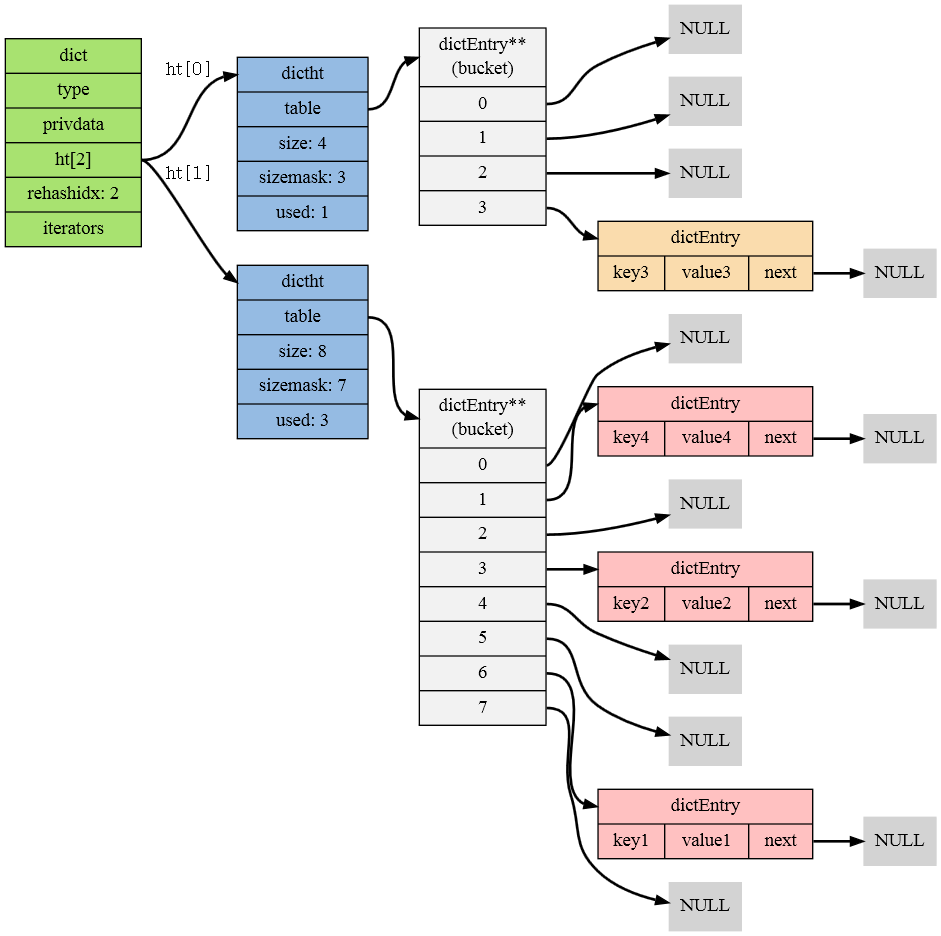
这时的字典是这个样子：



##### Rehash 进行中

在这个阶段， ht[0]->table 的节点会被逐渐迁移到 ht[1]->table ， 因为 rehash 是分多次进行的（细节在下一节解释）， 字典的 rehashidx 变量会记录 rehash 进行到 ht[0] 的哪个索引位置上。

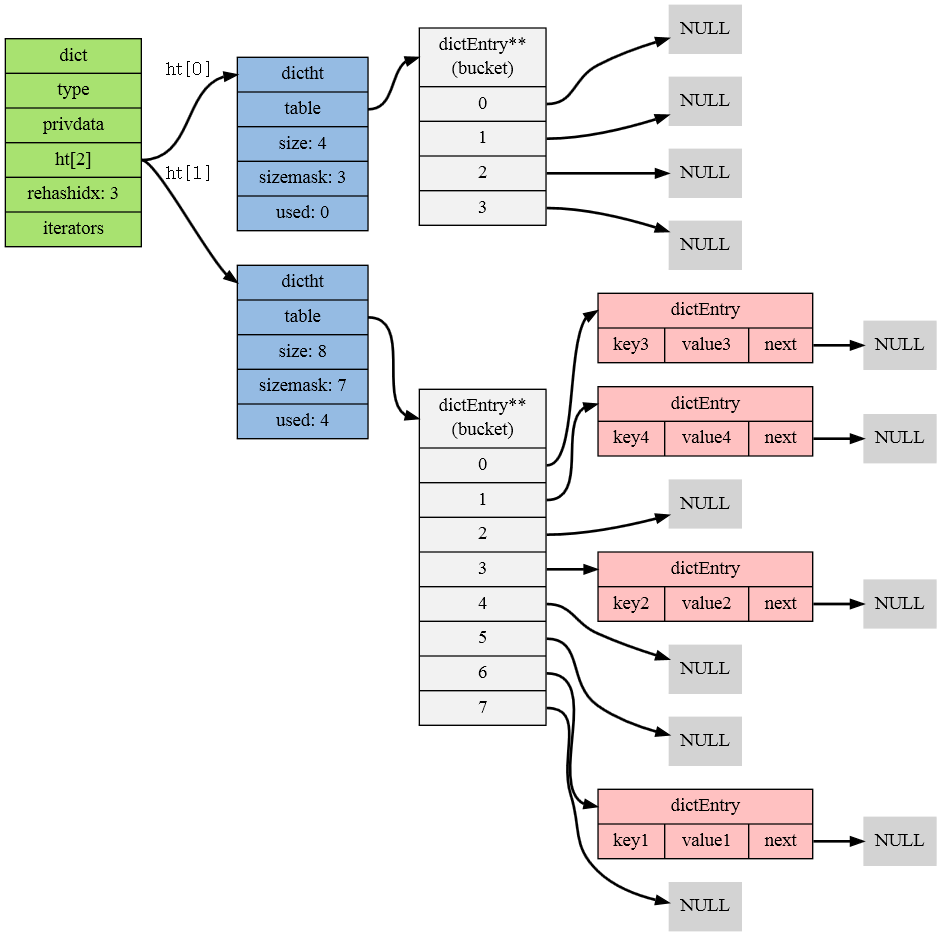
以下是 rehashidx 值为 2 时，字典的样子：



注意除了节点的移动外， 字典的 rehashidx 、 ht[0]->used 和 ht[1]->used 三个属性也产生了变化。

##### 节点迁移完毕

到了这个阶段，所有的节点都已经从 ht[0] 迁移到 ht[1] 了：

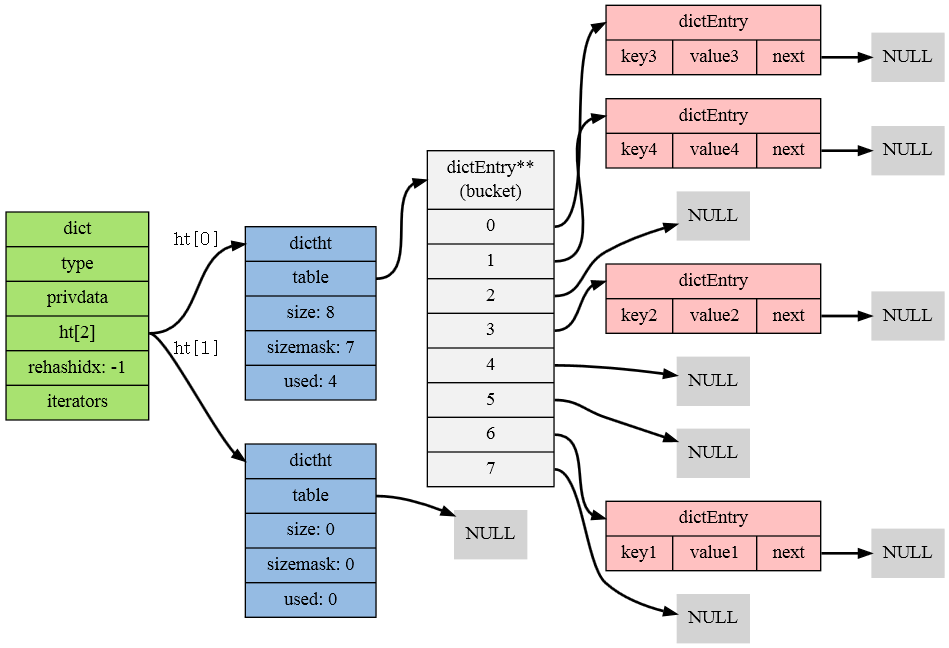


##### Rehash 完毕

在 rehash 的最后阶段，程序会执行以下工作：

1. 释放 ht[0] 的空间；
2. 用 ht[1] 来代替 ht[0] ，使原来的 ht[1] 成为新的 ht[0] ；
3. 创建一个新的空哈希表，并将它设置为 ht[1] ；
4. 将字典的 rehashidx 属性设置为 -1 ，标识 rehash 已停止；

以下是字典 rehash 完毕之后的样子：



对比字典 rehash 前后， 新的 ht[0] 空间更大， 并且字典原有的键值对也没有被修改或者删除。

#### 渐进式 rehash

在上一节，我们了解了字典的 rehash 过程， 需要特别指出的是， rehash 程序并不是在激活之后，就马上执行直到完成的， 而是分多次、渐进式地完成的。

假设这样一个场景：在一个有很多键值对的字典里， 某个用户在添加新键值对时触发了 rehash 过程， 如果这个 rehash 过程必须将所有键值对迁移完毕之后才将结果返回给用户， 这样的处理方式将是非常不友好的。

另一方面， 要求服务器必须阻塞直到 rehash 完成， 这对于 Redis 服务器本身也是不能接受的。

为了解决这个问题， Redis 使用了渐进式（incremental）的 rehash 方式： 通过将 rehash 分散到多个步骤中进行， 从而避免了集中式的计算。

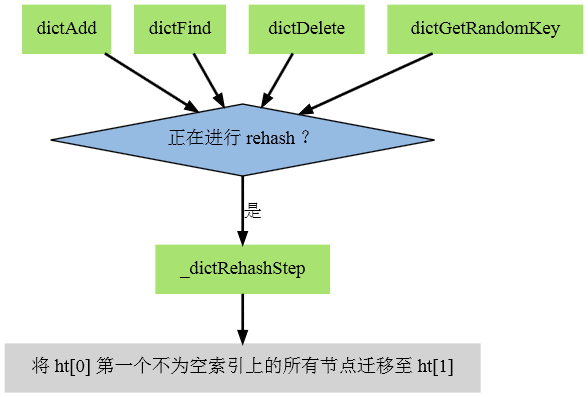
渐进式 rehash 主要由 \_dictRehashStep 和 dictRehashMilliseconds 两个函数进行：

* \_dictRehashStep 用于对数据库字典、以及哈希键的字典进行被动 rehash ；
* dictRehashMilliseconds 则由 Redis 服务器常规任务程序（server cron job）执行，用于对数据库字典进行主动 rehash ；

##### \_dictRehashStep

每次执行 \_dictRehashStep ， ht[0]->table 哈希表第一个不为空的索引上的所有节点就会全部迁移到 ht[1]->table 。

在 rehash 开始进行之后（d->rehashidx 不为 -1）， 每次执行一次添加、查找、删除操作， \_dictRehashStep 都会被执行一次



因为字典会保持哈希表大小和节点数的比率在一个很小的范围内， 所以每个索引上的节点数量不会很多（从目前版本的 rehash 条件来看，平均只有一个，最多通常也不会超过五个）， 所以在执行操作的同时，对单个索引上的节点进行迁移， 几乎不会对响应时间造成影响。

##### dictRehashMilliseconds

dictRehashMilliseconds 可以在指定的毫秒数内， 对字典进行 rehash 。

当 Redis 的服务器常规任务执行时， dictRehashMilliseconds 会被执行， 在规定的时间内， 尽可能地对数据库字典中那些需要 rehash 的字典进行 rehash ， 从而加速数据库字典的 rehash 进程（progress）。

##### 其他措施

在哈希表进行 rehash 时， 字典还会采取一些特别的措施， 确保 rehash 顺利、正确地进行：

*  因为在 rehash 时，字典会同时使用两个哈希表，所以在这期间的所有查找、删除等操作，除了在 ht[0] 上进行，还需要在 ht[1] 上进行。
* 在执行添加操作时，新的节点会直接添加到 ht[1] 而不是 ht[0] ，这样保证 ht[0] 的节点数量在整个 rehash 过程中都只减不增。

#### 字典的收缩

上面关于 rehash 的章节描述了通过 rehash 对字典进行扩展（expand）的情况， 如果哈希表的可用节点数比已用节点数大很多的话， 那么也可以通过对哈希表进行 rehash 来收缩（shrink）字典。

收缩 rehash 和上面展示的扩展 rehash 的操作几乎一样，执行以下步骤：

1. 创建一个比 ht[0]->table 小的 ht[1]->table ；
2. 将 ht[0]->table 中的所有键值对迁移到 ht[1]->table ；
3. 将原有 ht[0] 的数据清空，并将 ht[1] 替换为新的 ht[0] ；

扩展 rehash 和收缩 rehash 执行完全相同的过程， 一个 rehash 是扩展还是收缩字典， 关键在于新分配的 ht[1]->table 的大小：

* 如果 rehash 是扩展操作，那么 ht[1]->table 比 ht[0]->table 要大；
* 如果 rehash 是收缩操作，那么 ht[1]->table 比 ht[0]->table 要小；

#### 小结

*  字典是由键值对构成的抽象数据结构。
* Redis 中的数据库和哈希键都基于字典来实现。
* Redis 字典的底层实现为哈希表，每个字典使用两个哈希表，一般情况下只使用 0 号哈希表，只有在 rehash 进行时，才会同时使用 0 号和 1 号哈希表。
* 哈希表使用链地址法来解决键冲突的问题。
* Rehash 可以用于扩展或收缩哈希表。
* 对哈希表的 rehash 是分多次、渐进式地进行的。

## Redis 数据类型

### 对象处理机制

在 Redis 的命令中，用于对键（key）进行处理的命令占了很大一部分， 而对于键所保存的值的类型（后简称“键的类型”），键能执行的命令又各不相同。

比如说， [LPUSH](http://redis.readthedocs.org/en/latest/list/lpush.html#lpush) 和 [LLEN](http://redis.readthedocs.org/en/latest/list/llen.html#llen) 只能用于列表键， 而 [SADD](http://redis.readthedocs.org/en/latest/set/sadd.html#sadd) 和 [SRANDMEMBER](http://redis.readthedocs.org/en/latest/set/srandmember.html#srandmember) 只能用于集合键， 等等。

另外一些命令， 比如 [DEL](http://redis.readthedocs.org/en/latest/key/del.html#del) 、 [TTL](http://redis.readthedocs.org/en/latest/key/ttl.html#ttl) 和 [TYPE](http://redis.readthedocs.org/en/latest/key/type.html#type) ， 可以用于任何类型的键， 但是， 要正确实现这些命令， 必须为不同类型的键设置不同的处理方式： 比如说， 删除一个列表键和删除一个字符串键的操作过程就不太一样。

以上的描述说明，Redis 必须让每个键都带有类型信息， 使得程序可以检查键的类型， 并为它选择合适的处理方式。

另外，在前面介绍各个底层数据结构时有提到， Redis 的每一种数据类型，比如字符串、列表、有序集， 它们都拥有不只一种底层实现（Redis 内部称之为编码，encoding）， 这说明， 每当对某种数据类型的键进行操作时， 程序都必须根据键所采取的编码， 进行不同的操作。

比如说， 集合类型就可以由字典和整数集合两种不同的数据结构实现， 但是， 当用户执行 [ZADD](http://redis.readthedocs.org/en/latest/sorted_set/zadd.html#zadd) 命令时， 他/她应该不必关心集合使用的是什么编码， 只要 Redis 能按照 [ZADD](http://redis.readthedocs.org/en/latest/sorted_set/zadd.html#zadd) 命令的指示， 将新元素添加到集合就可以了。

这说明，操作数据类型的命令除了要对键的类型进行检查之外， 还需要根据数据类型的不同编码进行多态处理。

为了解决以上问题， Redis 构建了自己的类型系统， 这个系统的主要功能包括：

* redisObject 对象。

 基于 redisObject 对象的类型检查。

 基于 redisObject 对象的显式多态函数。

*  对 redisObject 进行分配、共享和销毁的机制。

#### redisObject 数据结构，以及 Redis 的数据类型

redisObject 是 Redis 类型系统的核心， 数据库中的每个键、值，以及 Redis 本身处理的参数， 都表示为这种数据类型。

redisObject 的定义位于 redis.h

/\*

\* Redis 对象

\*/

typedef struct redisObject {

// 类型

unsigned type:4;

// 对齐位

unsigned notused:2;

// 编码方式

unsigned encoding:4;

// LRU 时间（相对于 server.lruclock）

unsigned lru:22;

// 引用计数

int refcount;

// 指向对象的值

void \*ptr;

} robj;

type 、 encoding 和 ptr 是最重要的三个属性。

type 记录了对象所保存的值的类型，它的值可能是以下常量的其中一个（定义位于 redis.h）：

/\*

\* 对象类型

\*/

#define REDIS\_STRING 0 // 字符串

#define REDIS\_LIST 1 // 列表

#define REDIS\_SET 2 // 集合

#define REDIS\_ZSET 3 // 有序集

#define REDIS\_HASH 4 // 哈希表

encoding 记录了对象所保存的值的编码，它的值可能是以下常量的其中一个（定义位于 redis.h）：

/\*

\* 对象编码

\*/

#define REDIS\_ENCODING\_RAW 0 // 编码为字符串

#define REDIS\_ENCODING\_INT 1 // 编码为整数

#define REDIS\_ENCODING\_HT 2 // 编码为哈希表

#define REDIS\_ENCODING\_ZIPMAP 3 // 编码为 zipmap

#define REDIS\_ENCODING\_LINKEDLIST 4 // 编码为双端链表

#define REDIS\_ENCODING\_ZIPLIST 5 // 编码为压缩列表

#define REDIS\_ENCODING\_INTSET 6 // 编码为整数集合

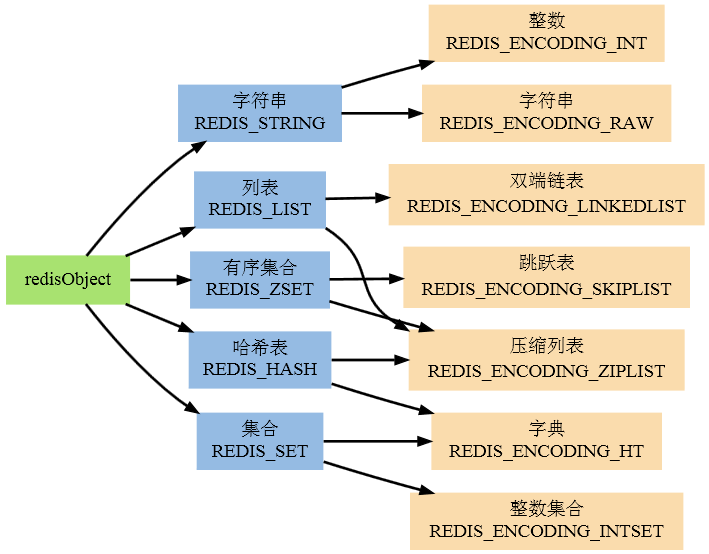
#define REDIS\_ENCODING\_SKIPLIST 7 // 编码为跳跃表

ptr 是一个指针，指向实际保存值的数据结构，这个数据结构由 type 属性和 encoding 属性决定。

举个例子，如果一个 redisObject 的 type 属性为 REDIS\_LIST ， encoding 属性为 REDIS\_ENCODING\_LINKEDLIST ，那么这个对象就是一个 Redis 列表，它的值保存在一个双端链表内，而 ptr 指针就指向这个双端链表；

另一方面，如果一个 redisObject 的 type 属性为 REDIS\_HASH ， encoding 属性为 REDIS\_ENCODING\_ZIPMAP ，那么这个对象就是一个 Redis 哈希表，它的值保存在一个 zipmap 里，而 ptr 指针就指向这个 zipmap ；诸如此类。

下图展示了 redisObject 、Redis 所有数据类型、以及 Redis 所有编码方式（底层实现）三者之间的关系：



#### 命令的类型检查和多态

有了 redisObject 结构的存在， 在执行处理数据类型的命令时， 进行类型检查和对编码进行多态操作就简单得多了。

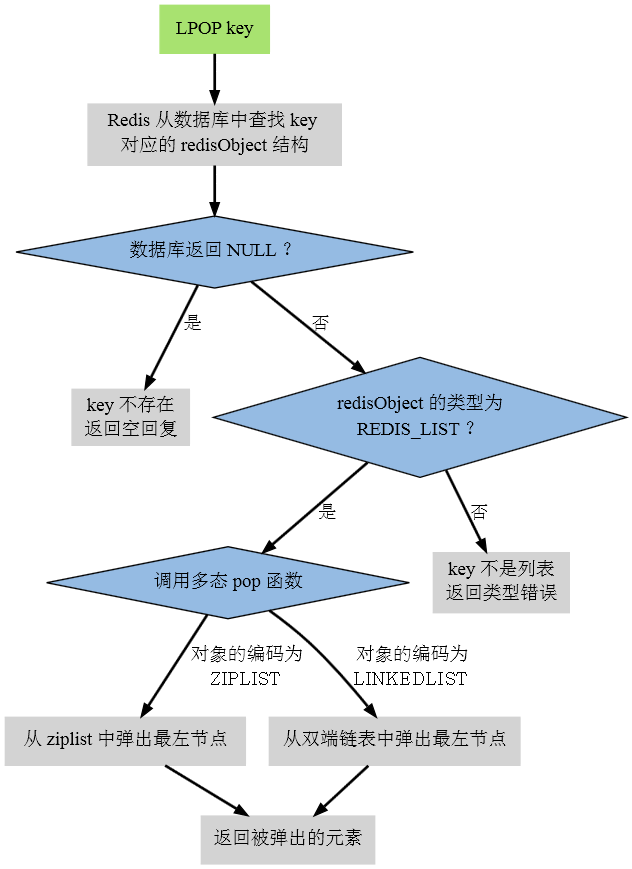
当执行一个处理数据类型的命令时， Redis 执行以下步骤：

有了 redisObject 结构的存在， 在执行处理数据类型的命令时， 进行类型检查和对编码进行多态操作就简单得多了。

当执行一个处理数据类型的命令时， Redis 执行以下步骤：

*  根据给定 key ，在数据库字典中查找和它相对应的 redisObject ，如果没找到，就返回 NULL 。
*  检查 redisObject 的 type 属性和执行命令所需的类型是否相符，如果不相符，返回类型错误。
*  根据 redisObject 的 encoding 属性所指定的编码，选择合适的操作函数来处理底层的数据结构。
* 返回数据结构的操作结果作为命令的返回值。

作为例子，以下展示了对键 key 执行 LPOP 命令的完整过程：



#### 对象共享

有一些对象在 Redis 中非常常见， 比如命令的返回值 OK 、 ERROR 、 WRONGTYPE 等字符， 另外，一些小范围的整数，比如个位、十位、百位的整数都非常常见

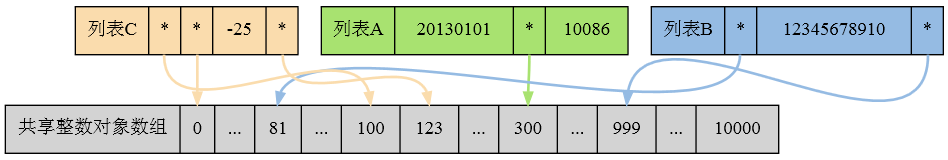
为了利用这种常见情况， Redis 在内部使用了一个 [Flyweight 模式](http://en.wikipedia.org/wiki/Flyweight_pattern) ： 通过预分配一些常见的值对象， 并在多个数据结构之间共享这些对象， 程序避免了重复分配的麻烦， 也节约了一些 CPU 时间。

Redis 预分配的值对象有如下这些：

* 各种命令的返回值，比如执行成功时返回的 OK ，执行错误时返回的 ERROR ，类型错误时返回的 WRONGTYPE ，命令入队事务时返回的 QUEUED ，等等。
* 包括 0 在内，小于 redis.h/REDIS\_SHARED\_INTEGERS 的所有整数（REDIS\_SHARED\_INTEGERS 的默认值为 10000）

因为命令的回复值直接返回给客户端， 所以它们的值无须进行共享； 另一方面， 如果某个命令的输入值是一个小于 REDIS\_SHARED\_INTEGERS 的整数对象， 那么当这个对象要被保存进数据库时， Redis 就会释放原来的值， 并将值的指针指向共享对象。

作为例子，下图展示了三个列表，它们都带有指向共享对象数组中某个值对象的指针：



三个列表的值分别为：

* 列表 A ： [20130101, 300, 10086] ，
* 列表 B ： [81, 12345678910, 999] ，
* 列表 C ： [100, 0, -25, 123] 。

#### 引用计数以及对象的销毁

当将 redisObject 用作数据库的键或者值， 而不是用来储存参数时， 对象的生命期是非常长的， 因为 C 语言本身没有自动释放内存的相关机制， 如果只依靠程序员的记忆来对对象进行追踪和销毁， 基本是不太可能的。

另一方面，正如前面提到的，一个共享对象可能被多个数据结构所引用， 这时像是“这个对象被引用了多少次？”之类的问题就会出现。

为了解决以上两个问题， Redis 的对象系统使用了[引用计数](http://en.wikipedia.org/wiki/Reference_counting)技术来负责维持和销毁对象， 它的运作机制如下：

* 每个 redisObject 结构都带有一个 refcount 属性，指示这个对象被引用了多少次。
* 当新创建一个对象时，它的 refcount 属性被设置为 1 。
* 当对一个对象进行共享时，Redis 将这个对象的 refcount 增一。
* 当使用完一个对象之后，或者取消对共享对象的引用之后，程序将对象的 refcount 减一。
* 当对象的 refcount 降至 0 时，这个 redisObject 结构，以及它所引用的数据结构的内存，都会被释放。

#### 小结

* Redis 使用自己实现的对象机制来实现类型判断、命令多态和基于引用计数的垃圾回收。
* 一种 Redis 类型的键可以有多种底层实现。
* Redis 会预分配一些常用的数据对象，并通过共享这些对象来减少内存占用，和避免频繁地为小对象分配内存。

### 字符串

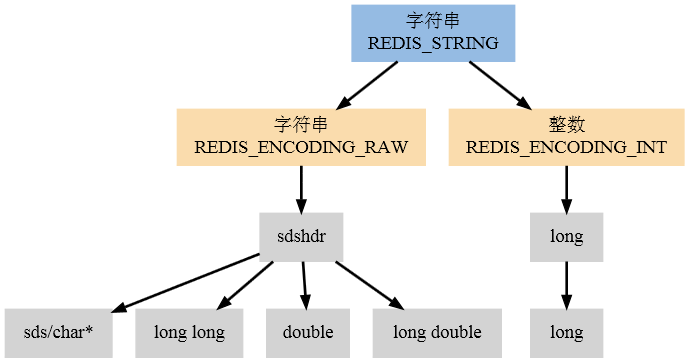
REDIS\_STRING （字符串）是 Redis 使用得最为广泛的数据类型， 它除了是 SET 、 GET 等命令的操作对象之外， 数据库中的所有键， 以及执行命令时提供给 Redis 的参数， 都是用这种类型保存的。

#### 字符串编码

字符串类型分别使用 REDIS\_ENCODING\_INT 和 REDIS\_ENCODING\_RAW 两种编码：

*  REDIS\_ENCODING\_INT 使用 long 类型来保存 long 类型值。
* REDIS\_ENCODING\_RAW 则使用 sdshdr 结构来保存 sds （也即是 char\* )、 long long 、 double 和 long double 类型值。

换句话来说， 在 Redis 中， 只有能表示为 long 类型的值， 才会以整数的形式保存， 其他类型的整数、小数和字符串， 都是用 sdshdr 结构来保存。’



#### 编码的选择

新创建的字符串默认使用 REDIS\_ENCODING\_RAW 编码， 在将字符串作为键或者值保存进数据库时， 程序会尝试将字符串转为 REDIS\_ENCODING\_INT 编码。

#### 字符串命令的实现

Redis 的字符串类型命令， 基本上是通过包装 sds 数据结构的操作函数来实现的， 没有什么需要说明的地方。

### 哈希表

REDIS\_HASH （哈希表）是 [HSET](http://redis.readthedocs.org/en/latest/hash/hset.html#hset) 、 [HLEN](http://redis.readthedocs.org/en/latest/hash/hlen.html#hlen) 等命令的操作对象， 它使用 REDIS\_ENCODING\_ZIPLIST 和 REDIS\_ENCODING\_HT 两种编码方式：

窗体底端

窗体底端

窗体底端

窗体底端

# QA

## Redis集群如何搭建,如何高可用

## Redis的java客户端如何编写

## Redis的key如何设置可以提高命中率

## Redis如何监控?