课程目标

1、了解 Redis 的定位与基本特性

2、掌握 Redis 基本数据类型的操作命令、底层存储结构、应用场景

内容定位

1、没用过 Redis 的同学

2、项目里面在用 Redis，但是只知道可以做缓存，只知道 get set 方法

3、不知道 Redis 其他数据类型、各种数据类型的底层存储结构的同学

基于最新版本：5.0.5

参考资料：http://redisbook.com

https://github.com/antirez/redis

1. Redis 入门

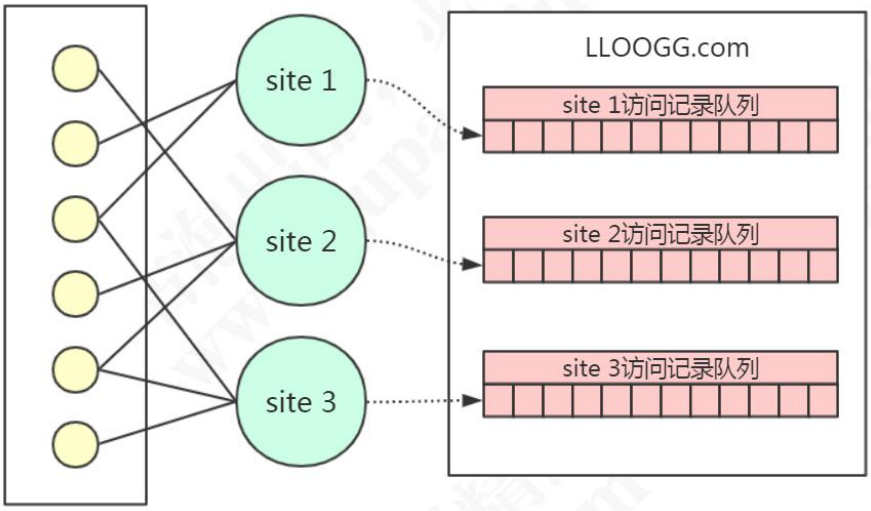
1.1. Redis 诞生历程

1.1.1. 从一个故事开始

08 年的时候有一个意大利西西里岛的小伙子，笔名 antirez（http://invece.org/），创建了一个访客信息网站 LLOOGG.COM。有的时候我们需要知道网站的访问情况，比如访

客的 IP、操作系统、浏览器、使用的搜索关键词、所在地区、访问的网页地址等等。在国内，有很多网站提供了这个功能，比如 CNZZ，百度统计，国外也有谷歌的 Google Analytics。我们不用自己写代码去实现这个功能，只需要在全局的 footer 里面嵌入一段 JS 代码就行了，当页面被访问的时候，就会自动把访客的信息发送到这些网站统计的服务器，然后我们登录后台就可以查看数据了。

LLOOGG.COM 提供的就是这种功能，它可以查看最多 10000 条的最新浏览记录。这样的话，它需要为每一个网站创建一个列表（List），不同网站的访问记录进入到不同的列表。如果列表的长度超过了用户指定的长度，它需要把最早的记录删除（先进先出）。



当 LLOOGG.COM 的用户越来越多的时候，它需要维护的列表数量也越来越多，这种记录最新的请求和删除最早的请求的操作也越来越多。LLOOGG.COM 最初使用的数据库是 MySQL，可想而知，因为每一次记录和删除都要读写磁盘，因为数据量和并发量太大，在这种情况下无论怎么去优化数据库都不管用了。

考虑到最终限制数据库性能的瓶颈在于磁盘，所以 antirez 打算放弃磁盘，自己去实现一个具有列表结构的数据库的原型，把数据放在内存而不是磁盘，这样可以大大地提升列表的 push 和 pop 的效率。antirez 发现这种思路确实能解决这个问题，所以用 C 语

言重写了这个内存数据库，并且加上了持久化的功能，09 年，Redis 横空出世了。从最开始只支持列表的数据库，到现在支持多种数据类型，并且提供了一系列的高级特性，Redis 已经成为一个在全世界被广泛使用的开源项目。

为什么叫 REDIS 呢？它的全称是 REmote DIctionary Service，直接翻译过来是远程字典服务。

从 Redis 的诞生历史我们看到了，在某些场景中，关系型数据库并不适合用来存储我们的 Web 应用的数据。那么，关系型数据库和非关系型数据库，或者说 SQL 和 NoSQL，到底有什么不一样呢？

1.2. Redis 定位与特性

1.2.1. SQL 与 NoSQL

在绝大部分时候，我们都会首先考虑用关系型数据库来存储我们的数据，比如 SQLServer，Oracle，MySQL 等等。

关系型数据库的特点：

1、它以表格的形式，基于行存储数据，是一个二维的模式。

2、它存储的是结构化的数据，数据存储有固定的模式（schema），数据需要适应表结构。

3、表与表之间存在关联（Relationship）。

4、大部分关系型数据库都支持 SQL（结构化查询语言）的操作，支持复杂的关联查询。

5、通过支持事务（ACID 酸）来提供严格或者实时的数据一致性。

但是使用关系型数据库也存在一些限制，比如：

1、要实现扩容的话，只能向上（垂直）扩展，比如磁盘限制了数据的存储，就要扩大磁盘容量，通过堆硬件的方式，不支持动态的扩缩容。水平扩容需要复杂的技术来实现，比如分库分表。

2、表结构修改困难，因此存储的数据格式也受到限制。

3、在高并发和高数据量的情况下，我们的关系型数据库通常会把数据持久化到磁盘，基于磁盘的读写压力比较大。

为了规避关系型数据库的一系列问题，我们就有了非关系型的数据库，我们一般把它叫做“non-relational”或者“Not Only SQL”。NoSQL 最开始是不提供 SQL 的数据库的意思，但是后来意思慢慢地发生了变化。

非关系型数据库的特点：

1、存储非结构化的数据，比如文本、图片、音频、视频。

2、表与表之间没有关联，可扩展性强。

3、保证数据的最终一致性。遵循 BASE（碱）理论。 Basically Available（基本可用）； Soft-state（软状态）； Eventually Consistent（最终一致性）。

4、支持海量数据的存储和高并发的高效读写。

5、支持分布式，能够对数据进行分片存储，扩缩容简单。

对于不同的存储类型，我们又有各种各样的非关系型数据库，比如有几种常见的类型：

1、KV 存储，用 Key Value 的形式来存储数据。比较常见的有 Redis 和 MemcacheDB。

2、文档存储，MongoDB。

3、列存储，HBase。

4、图存储，这个图（Graph）是数据结构，不是文件格式。Neo4j。

5、对象存储。

6、XML 存储等等等等。

这个网页列举了各种各样的 NoSQL 数据库 http://nosql-database.org/ 。

NewSQL 结合了 SQL 和 NoSQL 的特性（例如 PingCAP 的 TiDB）。

1.2.2. Redis 特性

官网介绍：https://redis.io/topics/introduction

中文网站：http://www.redis.cn

硬件层面有 CPU 的缓存；浏览器也有缓存；手机的应用也有缓存。我们把数据缓存起来的原因就是从原始位置取数据的代价太大了，放在一个临时位置存储起来，取回就可以快一些。

Redis 的特性：

1）更丰富的数据类型

2）进程内与跨进程；单机与分布式

3）功能丰富：持久化机制、过期策略

4）支持多种编程语言

5）高可用，集群

1.3. Redis 安装启动

1.5.1. 服务端安装

1、Linux 安装

参考：
CentOS7 安装 Redis 单实例 https://gper.club/articles/7e7e7f7ff7g5egc4g6b
Docker 安装 RabbitMQ 集群 https://gper.club/articles/7e7e7f7ff7g5egc5g6c
主要是注意配置文件几处关键内容（后台启动、绑定 IP、密码）的修改，配置别名

2、Windows 服务端安装

Redis 作者没有为 Windows 编写 Redis 服务端 微软自行编写了一个 Redis 服务端，可用于基本的测试和学习。

https://github.com/MicrosoftArchive/redis/tags

1.5.2. 服务启动

src 目录下，直接启动

```
./redis-server
```

后台启动（指定配置文件）

1、redis.conf 修改两行配置

```
daemonize yes
bind 0.0.0.0
```

2、启动 Redis

```
redis-server /usr/local/soft/redis-5.0.5/redis.conf
```

总结：redis 的参数可以通过三种方式配置，一种是 redis.conf，一种是启动时--携

带的参数，一种是 config set。

1.5.3. 基本操作

默认有 16 个库（0-15），可以在配置文件中修改，默认使用第一个 db0。

```
databases 16
```

因为没有完全隔离，不像数据库的 database，不适合把不同的库分配给不同的业务使用。

切换数据库

```
select 0
```

清空当前数据库

```
flushdb
```

清空所有数据库

```
flushall
```

Redis 是字典结构的存储方式，采用 key-value 存储。key 和 value 的最大长度限制是 512M（来自官网 https://redis.io/topics/data-types-intro/ ）。

键的基本操作。

命令参考：http://redisdoc.com/index.html

存值

```
set qingshan 2673
```

取值

```
get qingshan
```

查看所有键

```
keys *
```

获取键总数

```
dbsize
```

查看键是否存在

```
exists qingshan
```

删除键

```
del qingshan jack
```

重命名键

```
rename qingshan pengyuyan
```

查看类型

```
type qingshan
```

Redis 一共有几种数据类型？（注意是数据类型不是数据结构）

官网：https://redis.io/topics/data-types-intro

String、Hash、Set、List、Zset、Hyperloglog、Geo、Streams

1.4. Redis 基本数据类型

最基本也是最常用的数据类型就是 String。set 和 get 命令就是 String 的操作命令。为什么叫 Binary-safe strings 呢？

1.6.1. String 字符串

存储类型

可以用来存储字符串、整数、浮点数。

操作命令

设置多个值（批量操作，原子性）

```
mset qingshan 2673 jack 666
```

设置值，如果 key 存在，则不成功

```
setnx qingshan
```

基于此可实现分布式锁。用 del key 释放锁。
但如果释放锁的操作失败了，导致其他节点永远获取不到锁，怎么办？
加过期时间。单独用 expire 加过期，也失败了，无法保证原子性，怎么办？多参数

```
set key value [expiration EX seconds|PX milliseconds][NX|XX]
```

使用参数的方式

```
set lock1 1 EX 10 NX
```

（整数）值递增

```
incr qingshan
incrby qingshan 100
```

（整数）值递减

```
decr qingshan
decrby qingshan 100
```

浮点数增量

```
set f 2.6
incrbyfloat f 7.3
```

获取多个值

```
mget qingshan jack
```

获取值长度

```
strlen qingshan
```

字符串追加内容

```
append qingshan good
```

获取指定范围的字符

```
getrange qingshan 0 8
```

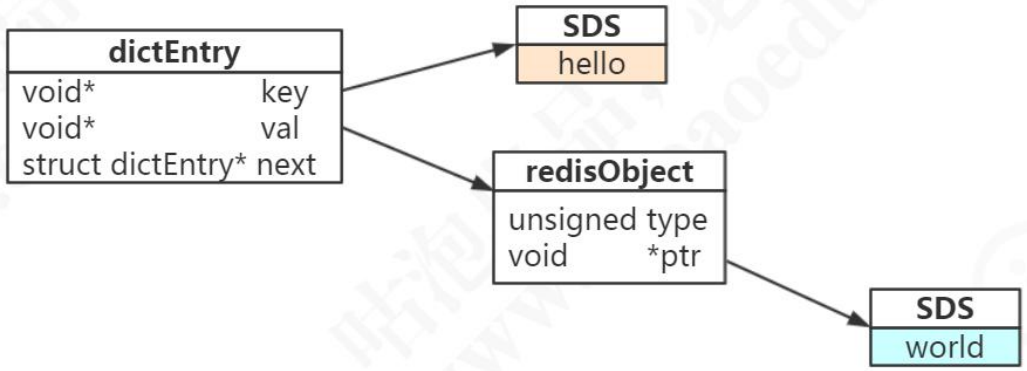
存储（实现）原理

数据模型

set hello word 为例，因为 Redis 是 KV 的数据库，它是通过 hashtable 实现的（我们把这个叫做外层的哈希）。所以每个键值对都会有一个 dictEntry（源码位置：dict.h），里面指向了 key 和 value 的指针。next 指向下一个 dictEntry。

```
typedef struct dictEntry {
  void *key; /* key 关键字定义 */

  union {
      void *val;       uint64_t u64; /* value 定义 */
      int64_t s64;       double d;
} v;
  struct dictEntry *next; /* 指向下一个键值对节点 */
} dictEntry;
```



key 是字符串，但是 Redis 没有直接使用 C 的字符数组，而是存储在自定义的 SDS 中。

value 既不是直接作为字符串存储，也不是直接存储在 SDS 中，而是存储在 redisObject 中。实际上五种常用的数据类型的任何一种，都是通过 redisObject 来存储的。

redisObject

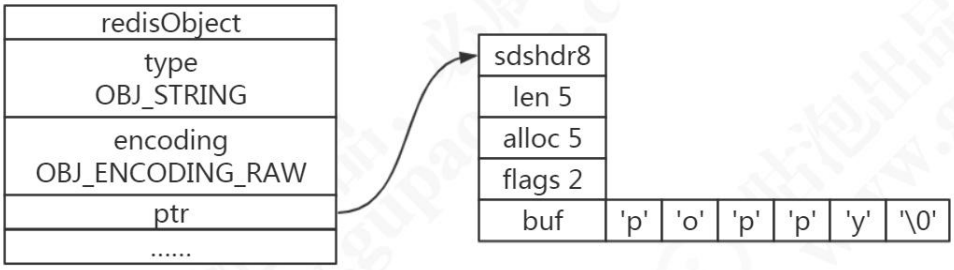
redisObject 定义在 src/server.h 文件中。

```
typedef struct redisObject {
  unsigned type:4; /* 对象的类型，包括：OBJ_STRING、OBJ_LIST、OBJ_HASH、OBJ_SET、OBJ_ZSET */
  unsigned encoding:4; /* 具体的数据结构 */
  unsigned lru:LRU_BITS; /* 24 位，对象最后一次被命令程序访问的时间，与内存回收有关 */
  int refcount; /* 引用计数。当 refcount 为 0 的时候，表示该对象已经不被任何对象引用，则可以进行垃圾回收了 */
  void *ptr; /* 指向对象实际的数据结构 */
} robj;
```

可以使用 type 命令来查看对外的类型。

```
127.0.0.1:6379> type qs
string
```

内部编码




```
127.0.0.1:6379> set number 1
OK
127.0.0.1:6379> set qs "is a good teacher in gupao, have crossed mountains and sea "
OK
127.0.0.1:6379> set jack bighead
OK
127.0.0.1:6379> object encoding number
"int"
127.0.0.1:6379> object encoding jack
"embstr"
127.0.0.1:6379> object encoding qs
"raw"
```

字符串类型的内部编码有三种：

1、int，存储 8 个字节的长整型（long，2^63-1）。

2、embstr，代表 embstr 格式的 SDS（Simple Dynamic String 简单动态字符串），存储小于 44 个字节的字符串。

3、raw，存储大于 44 个字节的字符串（3.2 版本之前是 39 字节）。为什么是 39？

```
/* object.c */
#define OBJ_ENCODING_EMBSTR_SIZE_LIMIT 44
```

问题 1、什么是 SDS？

Redis 中字符串的实现。

在 3.2 以后的版本中，SDS 又有多种结构（sds.h）：sdshdr5、sdshdr8、sdshdr16、

sdshdr32、sdshdr64，用于存储不同的长度的字符串，分别代表 2^5=32byte，2^8=256byte，2^16=65536byte=64KB，2^32byte=4GB。

```
/* sds.h */
struct __attribute__ ((__packed__)) sdshdr8 {
 uint8_t len; /*  当前字符数组的长度  */
 uint8_t alloc; /*当前字符数组总共分配的内存大小  */
 unsigned char flags; /*  当前字符数组的属性、用来标识到底是 sdshdr8 还是 sdshdr16 等  */
 char buf[]; /*  字符串真正的值  */
};
```

问题 2、为什么 Redis 要用 SDS 实现字符串？

我们知道，C 语言本身没有字符串类型（只能用字符数组 char[]实现）。

1、使用字符数组必须先给目标变量分配足够的空间，否则可能会溢出。

2、如果要获取字符长度，必须遍历字符数组，时间复杂度是 O(n)。

3、C 字符串长度的变更会对字符数组做内存重分配。

4、通过从字符串开始到结尾碰到的第一个'\0'来标记字符串的结束，因此不能保存图片、音频、视频、压缩文件等二进制(bytes)保存的内容，二进制不安全。

SDS 的特点：

1、不用担心内存溢出问题，如果需要会对 SDS 进行扩容。

2、获取字符串长度时间复杂度为 O(1)，因为定义了 len 属性。

3、通过“空间预分配”（ sdsMakeRoomFor）和“惰性空间释放”，防止多次重分配内存。

4、判断是否结束的标志是 len 属性（它同样以'\0'结尾是因为这样就可以使用 C

语言中函数库操作字符串的函数了），可以包含'\0'。

存储二进制：BytesTest.java

C 字符串	SDS
获取字符串长度的复杂度为 O(N)	获取字符串长度的复杂度为 O(1)
API 是不安全的，可能会造成缓冲区溢出	API 是安全的，不会早晨个缓冲区溢出
修改字符串长度 N 次必然需要执行 N 次内存重分配	修改字符串长度 N 次最多需要执行 N 次内存重分配
只能保存文本数据	可以保存文本或者二进制数据
可以使用所有<string.h>库中的函数	可以使用一部分<string.h>库中的函数

问题 3、embstr 和 raw 的区别？

embstr 的使用只分配一次内存空间(因为 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。

问题 4：int 和 embstr 什么时候转化为 raw？

当 int 数据不再是整数，或大小超过了 long 的范围（2^63-1=9223372036854775807）时，自动转化为 embstr。

```
127.0.0.1:6379> set k1 1
OK
127.0.0.1:6379> append k1 a
(integer) 2
127.0.0.1:6379> object encoding k1
"raw"
```

问题 5：明明没有超过阈值，为什么变成 raw 了？

```
127.0.0.1:6379> set k2 a
OK
127.0.0.1:6379> object encoding k2
```

```
"embstr"
127.0.0.1:6379> append k2 b
(integer) 2
127.0.0.1:6379> object encoding k2
"raw"
```

对于 embstr，由于其实现是只读的，因此在对 embstr 对象进行修改时，都会先转化为 raw 再进行修改。

因此 ,只要是修改 embstr 对象 ,修改后的对象一定是 raw 的 ,无论是否达到了 44 个字节。

问题 6：当长度小于阈值时，会还原吗？

关于 Redis 内部编码的转换，都符合以下规律：编码转换在 Redis 写入数据时完成，且转换过程不可逆，只能从小内存编码向大内存编码转换（但是不包括重新 set）。

问题 7：为什么要对底层的数据结构进行一层包装呢？

通过封装，可以根据对象的类型动态地选择存储结构和可以使用的命令，实现节省空间和优化查询速度。

应用场景

缓存

String 类型

例如：热点数据缓存（例如报表，明星出轨），对象缓存，全页缓存。

可以提升热点数据的访问速度。

数据共享分布式

STRING 类型，因为 Redis 是分布式的独立服务，可以在多个应用之间共享

例如：分布式 Session

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

分布式锁

STRING 类型 setnx 方法，只有不存在时才能添加成功，返回 true。

http://redisdoc.com/string/set.html 建议用参数的形式

```
public Boolean getLock(Object lockObject){
   jedisUtil = getJedisConnetion();
   boolean flag = jedisUtil.setNX(lockObj, 1);
   if(flag){
       expire(locakObj,10);
  }
   return flag;
}

public void releaseLock(Object lockObject){
   del(lockObj);
}
```

全局 ID

INT 类型，INCRBY，利用原子性

```
incrby userid 1000
```

（分库分表的场景，一次性拿一段）

计数器

INT 类型，INCR 方法

例如：文章的阅读量，微博点赞数，允许一定的延迟，先写入 Redis 再定时同步到数据库。

限流

INT 类型，INCR 方法

以访问者的 IP 和其他信息作为 key ,访问一次增加一次计数 ,超过次数则返回 false。

位统计

String 类型的 BITCOUNT（1.6.6 的 bitmap 数据结构介绍）。

字符是以 8 位二进制存储的。

```
set k1 a
setbit k1 6 1
setbit k1 7 0
get k1
```

a 对应的 ASCII 码是 97，转换为二进制数据是 01100001

b 对应的 ASCII 码是 98，转换为二进制数据是 01100010

因为 bit 非常节省空间（1 MB=8388608 bit），可以用来做大数据量的统计。

例如：在线用户统计，留存用户统计

```
setbit onlineusers 0 1
setbit onlineusers 1 1
setbit onlineusers 2 0
```

支持按位与、按位或等等操作。

```
BITOP AND destkey key [key ...] ，对一个或多个 key 求逻辑并，并将结果保存到 destkey 。
BITOP OR destkey key [key ...] ，对一个或多个 key 求逻辑或，并将结果保存到 destkey 。
BITOP XOR destkey key [key ...] ，对一个或多个 key 求逻辑异或，并将结果保存到 destkey 。
BITOP NOT destkey key ，对给定 key 求逻辑非，并将结果保存到 destkey 。
```

计算出 7 天都在线的用户

```
BITOP "AND" "7_days_both_online_users" "day_1_online_users" "day_2_online_users" ... "day_7_online_users"
```

如果一个对象的 value 有多个值的时候，怎么存储？

例如用一个 key 存储一张表的数据。

id	sno	sname	company
1	GP16666	一个人的精彩	腾讯
2	GP17777	猫老公	百度
3	GP18888	菜鸟	阿里

序列化？例如 JSON/Protobuf/XML，会增加序列化和反序列化的开销，并且不能单独获取、修改一个值。

可以通过 key 分层的方式来实现，例如：

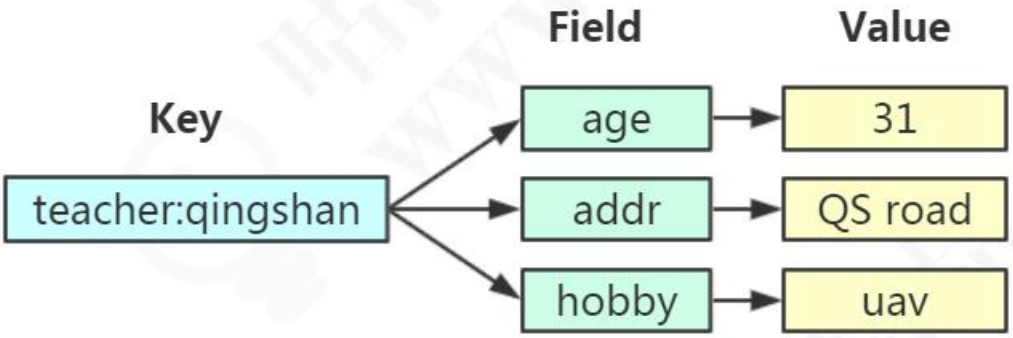
```
mset student:1:sno GP16666 student:1:sname 沐风 student:1:company 腾讯
```

获取值的时候一次获取多个值：

```
mget student:1:sno student:1:sname student:1:company
```

缺点：key 太长，占用的空间太多。有没有更好的方式？

1.6.2. Hash 哈希



存储类型

包含键值对的无序散列表。value 只能是字符串，不能嵌套其他类型。

同样是存储字符串，Hash 与 String 的主要区别？

1、把所有相关的值聚集到一个 key 中，节省内存空间

2、只使用一个 key，减少 key 冲突

3、当需要批量获取值的时候，只需要使用一个命令，减少内存/IO/CPU 的消耗

Hash 不适合的场景：

1、Field 不能单独设置过期时间

2、没有 bit 操作

3、需要考虑数据量分布的问题（value 值非常大的时候，无法分布到多个节点）

操作命令

```
hset h1 f 6
hset h1 e 5
hmset h1 a 1 b 2 c 3 d 4
```

```
hget h1 a
hmget h1 a b c d
hkeys h1
hvals h1
hgetall h1
```

key 操作

```
hget exists h1
hdel h1
hlen h1
```

存储（实现）原理

Redis 的 Hash 本身也是一个 KV 的结构，类似于 Java 中的 HashMap。

外层的哈希（Redis KV 的实现）只用到了 hashtable。当存储 hash 数据类型时，我们把它叫做内层的哈希。内层的哈希底层可以使用两种数据结构实现：

ziplist：OBJ_ENCODING_ZIPLIST（压缩列表）

hashtable：OBJ_ENCODING_HT（哈希表）

```
127.0.0.1:6379> hset h2 f aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
(integer) 1
127.0.0.1:6379> hset h3 f aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
(integer) 1
127.0.0.1:6379> object encoding h2
"ziplist"
127.0.0.1:6379> object encoding h3
"hashtable"
```

ziplist 压缩列表

ziplist 压缩列表是什么？

/* ziplist.c 源码头部注释 */

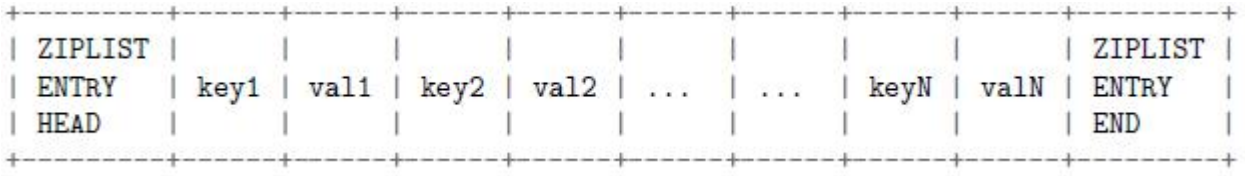
The ziplist is a specially encoded dually linked list that is designed to be very memory efficient. It stores both strings and integer values, where integers are encoded as actual integers instead of a series of characters. It allows push and pop operations on either side of the list in O(1) time. However, because every operation requires a reallocation of the memory used by the ziplist, the actual complexity is related to the amount of memory used by the ziplist.

ziplist 是一个经过特殊编码的双向链表，它不存储指向上一个链表节点和指向下一个链表节点的指针，而是存储上一个节点长度和当前节点长度，通过牺牲部分读写性能，来换取高效的内存空间利用率，是一种时间换空间的思想。只用在字段个数少，字段值小的场景里面。

ziplist 的内部结构？

ziplist.c 源码第 16 行的注释：

* <zlbytes> <zltail> <zllen> <entry> <entry> ... <entry> <zlend>



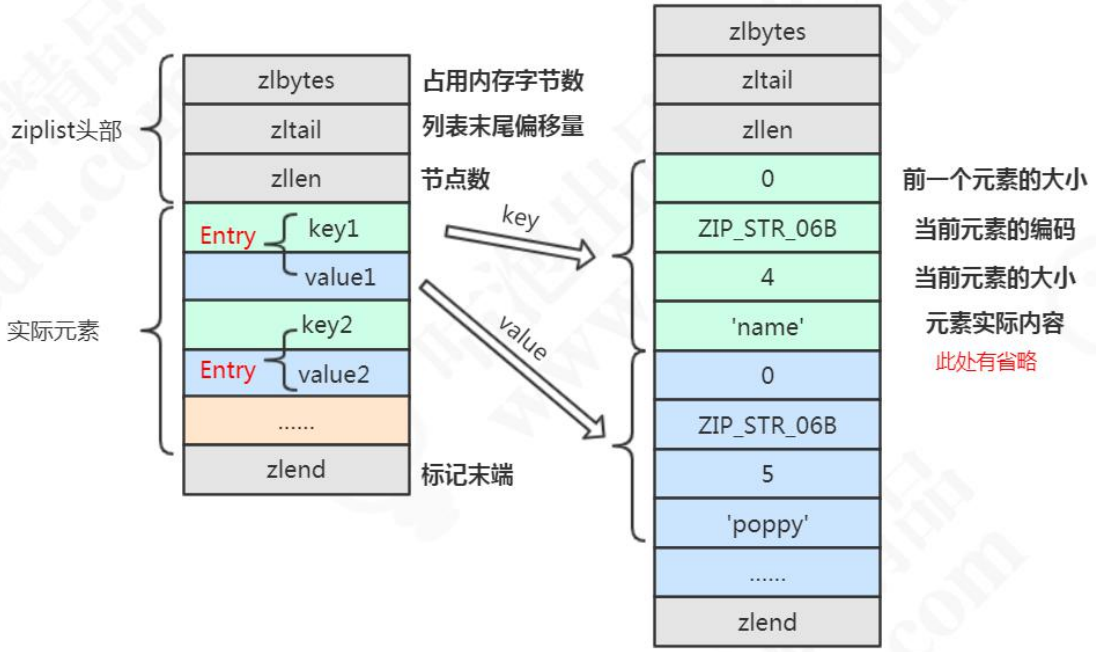

```
typedef struct zlentry {
    unsigned int prevrawlensize; /* 上一个链表节点占用的长度 */
    unsigned int prevrawlen;     /* 存储上一个链表节点的长度数值所需要的字节数 */
    unsigned int lensize;        /* 存储当前链表节点长度数值所需要的字节数 */
    unsigned int len;            /* 当前链表节点占用的长度 */
    unsigned int headersize;     /* 当前链表节点的头部大小（prevrawlensize + lensize），即非数据域的大小 */
    unsigned char encoding;      /* 编码方式 */
    unsigned char *p;            /* 压缩链表以字符串的形式保存，该指针指向当前节点起始位置 */
} zlentry;
```

编码 encoding（ziplist.c 源码第 204 行）

#define ZIP_STR_06B (0 << 6) //长度小于等于 63 字节

#define ZIP_STR_14B (1 << 6) //长度小于等于 16383 字节

#define ZIP_STR_32B (2 << 6) //长度小于等于 4294967295 字节



问题：什么时候使用 ziplist 存储？

当 hash 对象同时满足以下两个条件的时候，使用 ziplist 编码：

1）所有的键值对的健和值的字符串长度都小于等于 64byte（一个英文字母一个字节）；

2）哈希对象保存的键值对数量小于 512 个。

/* src/redis.conf 配置 */

```
hash-max-ziplist-value 64 // ziplist 中最大能存放的值长度
hash-max-ziplist-entries 512 // ziplist 中最多能存放的 entry 节点数量
```

```
/* 源码位置：t_hash.c ， 当达字段个数超过阈值，使用 HT 作为编码 */
    if (hashTypeLength(o) > server.hash_max_ziplist_entries)
        hashTypeConvert(o, OBJ_ENCODING_HT);
/*源码位置： t_hash.c，当字段值长度过大，转为 HT */
  for (i = start; i <= end; i++) {
    if (sdsEncodedObject(argv[i]) &&
        sdslen(argv[i]->ptr) > server.hash_max_ziplist_value)
    {
        hashTypeConvert(o, OBJ_ENCODING_HT);
        break;
```

```
    }
}
```

一个哈希对象超过配置的阈值（键和值的长度有>64byte，键值对个数>512 个）时，会转换成哈希表（hashtable）。

hashtable（dict）

在 Redis 中，hashtable 被称为字典（dictionary），它是一个数组+链表的结构。

源码位置：dict.h

前面我们知道了，Redis 的 KV 结构是通过一个 dictEntry 来实现的。

Redis 又对 dictEntry 进行了多层的封装。

```
typedef struct dictEntry {
  void *key; /* key 关键字定义 */

  union {
      void *val;       uint64_t u64; /* value 定义 */
      int64_t s64;       double d;
} v;
  struct dictEntry *next; /* 指向下一个键值对节点 */
} dictEntry;
```

dictEntry 放到了 dictht（hashtable 里面）：

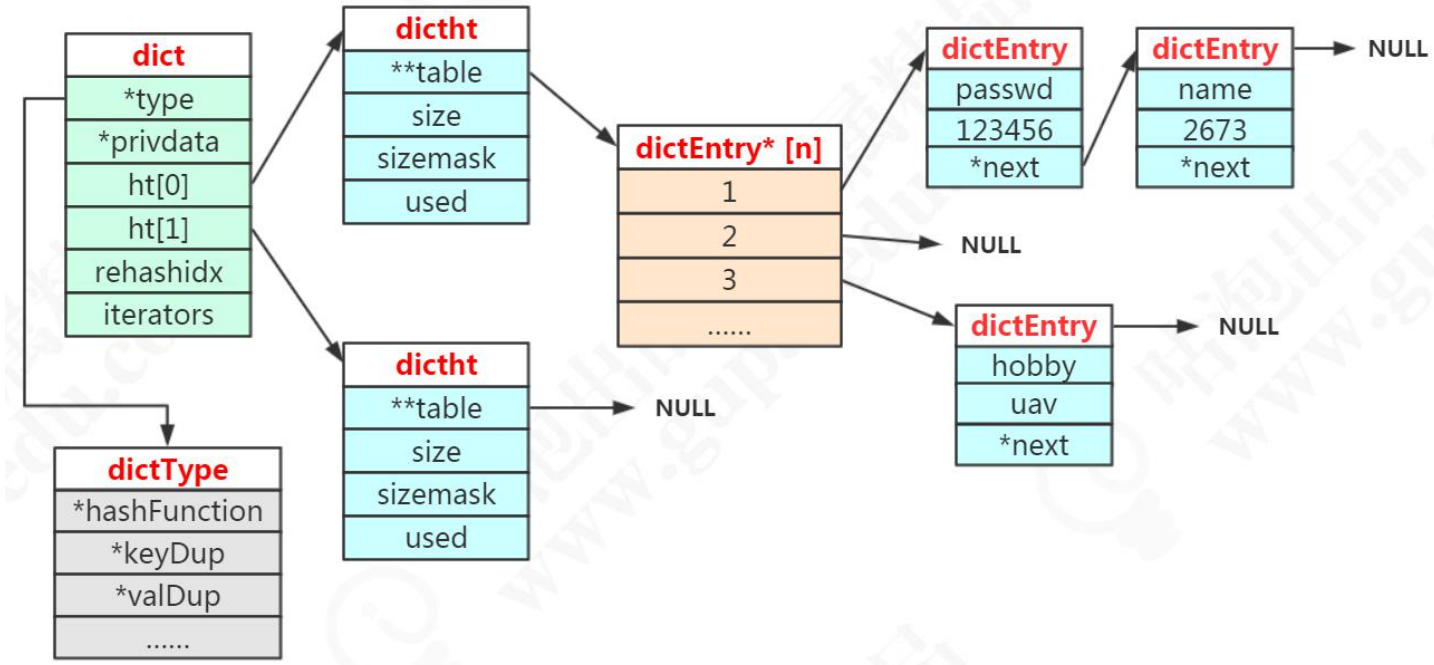
```
/* This is our hash table structure. Every dictionary has two of this as we
* implement incremental rehashing, for the old to the new table. */
typedef struct dictht {
  dictEntry **table; /* 哈希表数组 */
  unsigned long size; /* 哈希表大小 */
  unsigned long sizemask; /* 掩码大小，用于计算索引值。总是等于 size-1 */
  unsigned long used; /* 已有节点数 */
} dictht;
```

ht 放到了 dict 里面：

```
typedef struct dict {
  dictType *type; /* 字典类型 */
  void *privdata; /* 私有数据 */
  dictht ht[2]; /* 一个字典有两个哈希表 */
  long rehashidx; /* rehash 索引 */
  unsigned long iterators; /* 当前正在使用的迭代器数量 */
} dict;
```

从最底层到最高层 dictEntry——dictht——dict——OBJ_ENCODING_HT

总结：哈希的存储结构



注意：dictht 后面是 NULL 说明第二个 ht 还没用到。dictEntry*后面是 NULL 说明没有 hash 到这个地址。dictEntry 后面是 NULL 说明没有发生哈希冲突。

问题：为什么要定义两个哈希表呢？ht[2]

redis 的 hash 默认使用的是 ht[0]，ht[1]不会初始化和分配空间。

哈希表 dictht 是用链地址法来解决碰撞问题的。在这种情况下，哈希表的性能取决于它的大小（size 属性）和它所保存的节点的数量（used 属性）之间的比率：

- 比率在 1:1 时（一个哈希表 ht 只存储一个节点 entry），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（这个比例用 ratio 表示，5 表示平均

一个 ht 存储 5 个 entry），那么哈希表就会退化成多个链表，哈希表本身的性能优势就不再存在。

在这种情况下需要扩容。Redis 里面的这种操作叫做 rehash。

rehash 的步骤：

1、为字符 ht[1]哈希表分配空间，这个哈希表的空间大小取决于要执行的操作，以及 ht[0]当前包含的键值对的数量。

扩展：ht[1]的大小为第一个大于等于 ht[0].used*2。

2、将所有的 ht[0]上的节点 rehash 到 ht[1]上，重新计算 hash 值和索引，然后放入指定的位置。

3、当 ht[0]全部迁移到了 ht[1]之后，释放 ht[0]的空间，将 ht[1]设置为 ht[0]表，并创建新的 ht[1]，为下次 rehash 做准备。

问题：什么时候触发扩容？

负载因子（源码位置：dict.c）：

```
static int dict_can_resize = 1;
static unsigned int dict_force_resize_ratio = 5;
```

ratio = used / size，已使用节点与字典大小的比例

dict_can_resize 为 1 并且 dict_force_resize_ratio 已使用节点数和字典大小之间的比率超过 1：5，触发扩容

扩容判断 _dictExpandIfNeeded（源码 dict.c）

```
if (d->ht[0].used >= d->ht[0].size &&
    (dict_can_resize ||
```

```
        d->ht[0].used/d->ht[0].size > dict_force_resize_ratio))
    {
        return dictExpand(d, d->ht[0].used*2);
    }
    return DICT_OK;
```

扩容方法 dictExpand（源码 dict.c）

```
static int dictExpand(dict *ht, unsigned long size) {
    dict n; /* the new hashtable */
    unsigned long realsize = _dictNextPower(size), i;

    /* the size is invalid if it is smaller than the number of
     * elements already inside the hashtable */
    if (ht->used > size)
        return DICT_ERR;

    _dictInit(&n, ht->type, ht->privdata);
    n.size = realsize;
    n.sizemask = realsize-1;
    n.table = calloc(realsize,sizeof(dictEntry*));

    /* Copy all the elements from the old to the new table:
     * note that if the old hash table is empty ht->size is zero,
     * so dictExpand just creates an hash table. */
    n.used = ht->used;
    for (i = 0; i < ht->size && ht->used > 0; i++) {
        dictEntry *he, *nextHe;

        if (ht->table[i] == NULL) continue;

        /* For each hash entry on this slot... */
        he = ht->table[i];
        while(he) {
            unsigned int h;

            nextHe = he->next;
            /* Get the new element index */
            h = dictHashKey(ht, he->key) & n.sizemask;
            he->next = n.table[h];
            n.table[h] = he;
            ht->used--;
            /* Pass to the next element */
```

```
            he = nextHe;
        }
    }
    assert(ht->used == 0);
    free(ht->table);

    /* Remap the new hashtable in the old */
    *ht = n;
    return DICT_OK;
}
```

缩容：server.c

```
int htNeedsResize(dict *dict) {
    long long size, used;

    size = dictSlots(dict);
    used = dictSize(dict);
    return (size > DICT_HT_INITIAL_SIZE &&
            (used*100/size < HASHTABLE_MIN_FILL));
}
```

应用场景

String

String 可以做的事情，Hash 都可以做。

存储对象类型的数据

比如对象或者一张表的数据，比 String 节省了更多 key 的空间，也更加便于集中管理。

购物车

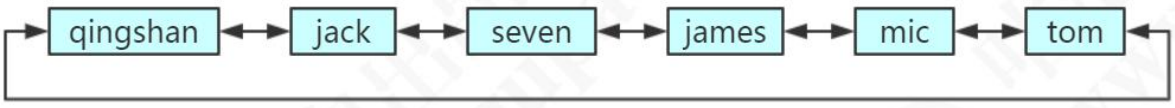
key：用户 id；field：商品 id；value：商品数量。

+1：hincr。-1：hdecr。删除：hdel。全选：hgetall。商品数：hlen。

1.6.3. List 列表

存储类型

存储有序的字符串（从左到右），元素可以重复。可以充当队列和栈的角色。



操作命令

元素增减：

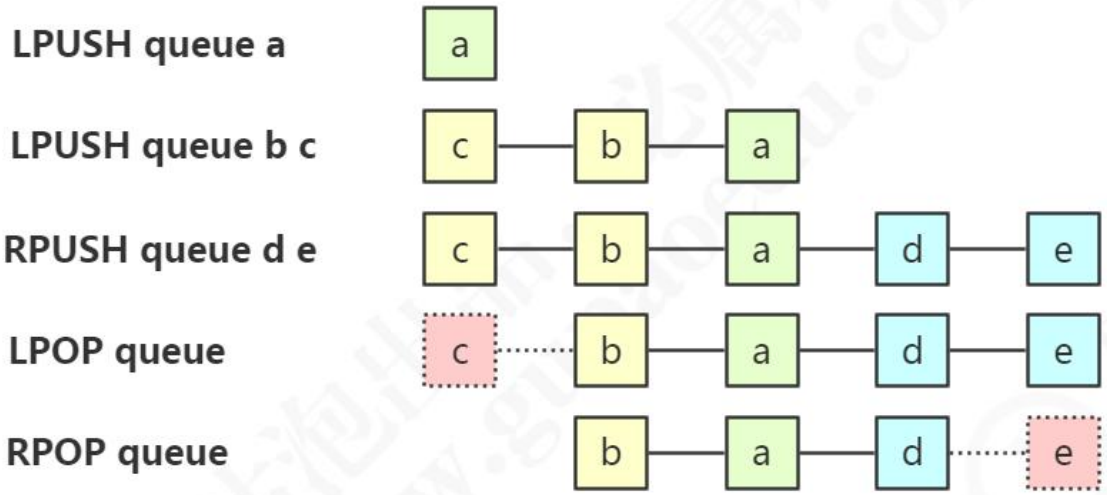
```
lpush queue a
lpush queue b c
```

```
rpush queue d e
lpop queue
rpop queue

blpop queue
brpop queue
```

取值

```
lindex queue 0
lrange queue 0 -1
```



存储（实现）原理

在早期的版本中，数据量较小时用 ziplist 存储，达到临界值时转换为 linkedlist 进行存储，分别对应 OBJ_ENCODING_ZIPLIST 和 OBJ_ENCODING_LINKEDLIST 。

3.2 版本之后，统一用 quicklist 来存储。quicklist 存储了一个双向链表，每个节点都是一个 ziplist。

```
127.0.0.1:6379> object encoding queue
"quicklist"
```

quicklist

quicklist（快速列表）是 ziplist 和 linkedlist 的结合体。

quicklist.h，head 和 tail 指向双向列表的表头和表尾

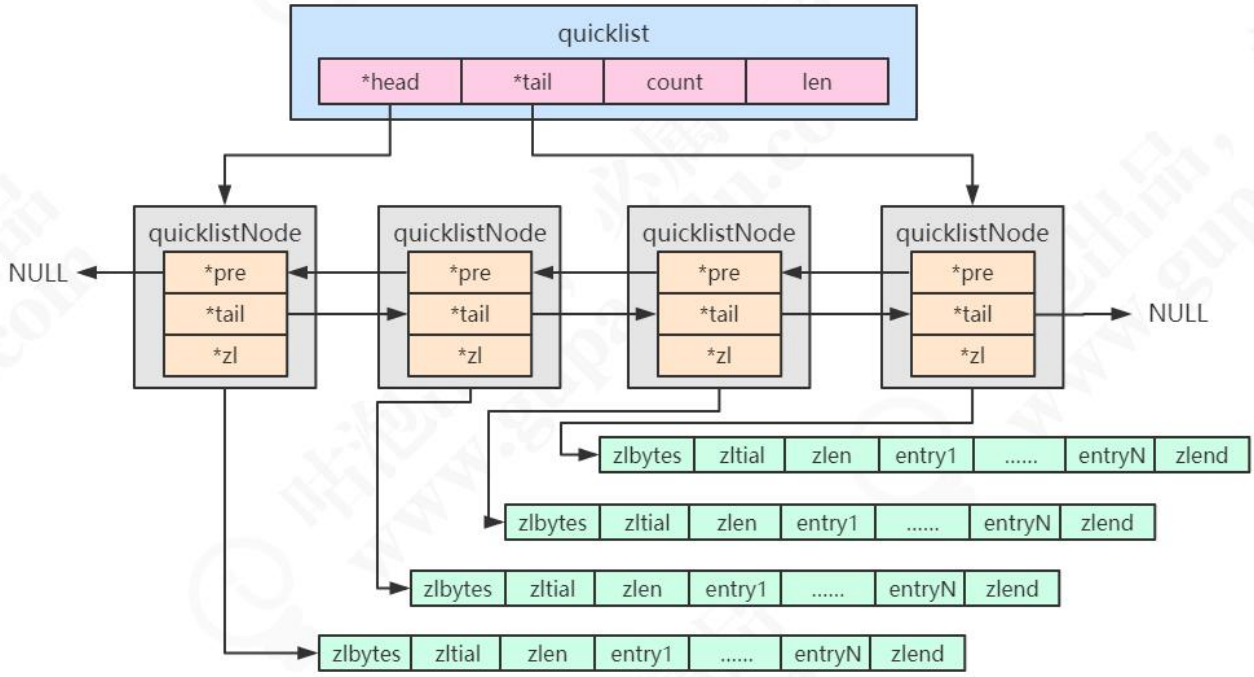
```
typedef struct quicklist {
   quicklistNode *head; /* 指向双向列表的表头 */
   quicklistNode *tail; /* 指向双向列表的表尾 */
   unsigned long count;        /* 所有的 ziplist 中一共存了多少个元素 */
   unsigned long len;          /* 双向链表的长度，node 的数量 */
   int fill : 16;              /* fill factor for individual nodes */
   unsigned int compress : 16; /* 压缩深度，0：不压缩； */
} quicklist;
```

redis.conf 相关参数：

参数	含义
list-max-ziplist-size（fill）	正数表示单个 ziplist 最多所包含的 entry 个数。 负数代表单个 ziplist 的大小，默认 8k。 -1：4KB；-2：8KB；-3：16KB；-4：32KB；-5：64KB
list-compress-depth（compress）	压缩深度，默认是 0。 1：首尾的 ziplist 不压缩；2：首尾第一第二个 ziplist 不压缩，以此类推

quicklistNode 中的*zl 指向一个 ziplist，一个 ziplist 可以存放多个元素。

```
typedef struct quicklistNode {
    struct quicklistNode *prev; /* 前一个节点 */
    struct quicklistNode *next; /* 后一个节点 */
    unsigned char *zl; /* 指向实际的 ziplist */
    unsigned int sz; /* 当前 ziplist 占用多少字节 */
    unsigned int count : 16; /*  当前 ziplist 中存储了多少个元素，占 16bit（下同），最大 65536 个 */
    unsigned int encoding : 2; /* 是否采用了 LZF 压缩算法压缩节点，1：RAW 2：LZF */
    unsigned int container : 2; /* 2：ziplist，未来可能支持其他结构存储 */
    unsigned int recompress : 1; /* 当前 ziplist 是不是已经被解压出来作临时使用 */
    unsigned int attempted_compress : 1;  /* 测试用 */
    unsigned int extra : 10;  /* 预留给未来使用 */
} quicklistNode;
```



ziplist 的结构前面已经说过了，不再重复。

应用场景

用户消息时间线 timeline

因为 List 是有序的，可以用来做用户时间线

消息队列

List 提供了两个阻塞的弹出操作：BLPOP/BRPOP，可以设置超时时间。

BLPOP：BLPOP key1 timeout 移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

BRPOP：BRPOP key1 timeout 移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

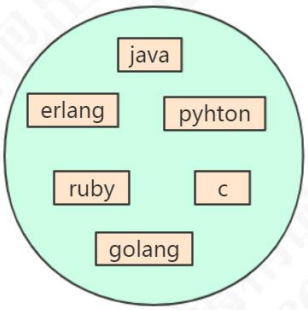
队列：先进先出：rpush blpop，左头右尾，右边进入队列，左边出队列。

栈：先进后出：rpush brpop

1.6.4. Set 集合

存储类型

String 类型的无序集合，最大存储数量 2^32-1（40 亿左右）。



操作命令

添加一个或者多个元素

```
sadd myset a b c d e f g
```

获取所有元素

```
smembers myset
```

统计元素个数

```
scard myset
```

随机获取一个元素

```
srandmember key
```

随机弹出一个元素

```
spop myset
```

移除一个或者多个元素

```
srem myset d e f
```

查看元素是否存在

```
sismember myset a
```

存储（实现）原理

Redis 用 intset 或 hashtable 存储 set。如果元素都是整数类型，就用 inset 存储。如果不是整数类型，就用 hashtable（数组+链表的存来储结构）。

问题：KV 怎么存储 set 的元素？key 就是元素的值，value 为 null。

如果元素个数超过 512 个，也会用 hashtable 存储。

配置文件 redis.conf
set-max-intset-entries 512

```
127.0.0.1:6379> sadd iset 1 2 3 4 5 6
(integer) 6
127.0.0.1:6379> object encoding iset
"intset"
127.0.0.1:6379> sadd myset a b c d e f
(integer) 6
```

127.0.0.1:6379> object encoding myset
"hashtable"

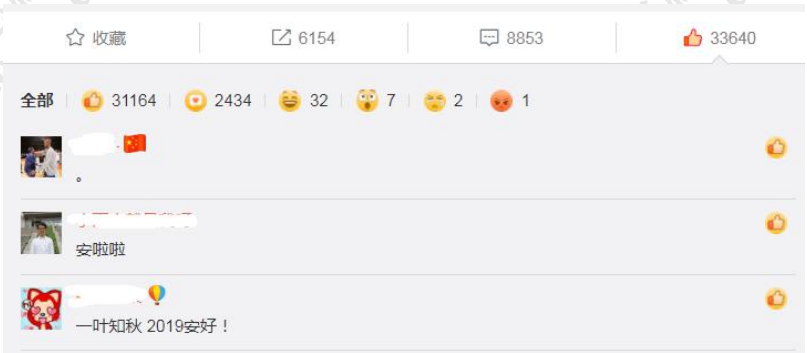
应用场景

抽奖

随机获取元素

spop myset

点赞、签到、打卡



这条微博的 ID 是 t1001，用户 ID 是 u3001。

用 like:t1001 来维护 t1001 这条微博的所有点赞用户。

点赞了这条微博：sadd like:t1001 u3001

取消点赞：srem like:t1001 u3001

是否点赞：sismember like:t1001 u3001

点赞的所有用户：smembers like:t1001

点赞数：scard like:t1001

比关系型数据库简单许多。

商品标签

用 tags:i5001 来维护商品所有的标签。

sadd tags:i5001 画面清晰细腻

sadd tags:i5001 真彩清晰显示屏

sadd tags:i5001 流畅至极

商品筛选

获取差集

```
sdiff set1 set2
```

获取交集（intersection ）

```
sinter set1 set2
```

获取并集

```
sunion set1 set2
```

iPhone11 上市了。

sadd brand:apple iPhone11

sadd brand:ios iPhone11

sad screensize:6.0-6.24 iPhone11

sad screentype:lcd iPhone11

筛选商品，苹果的，iOS 的，屏幕在 6.0-6.24 之间的，屏幕材质是 LCD 屏幕

sinter brand:apple brand:ios screensize:6.0-6.24 screentype:lcd

用户关注、推荐模型

思考

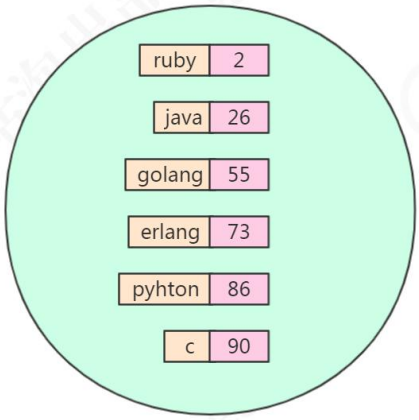
1）相互关注？

2）我关注的人也关注了他？

3）可能认识的人？

1.6.5. ZSet 有序集合

存储类型



sorted set，有序的 set，每个元素有个 score。

score 相同时，按照 key 的 ASCII 码排序。

数据结构对比：

数据结构	是否允许重复元素	是否有序	有序实现方式
列表 list	是	是	索引下标
集合 set	否	否	无
有序集合 zset	否	是	分值 score

操作命令

添加元素

```
zadd myzset 10 java 20 php 30 ruby 40 cpp 50 python
```

获取全部元素

```
zrange myzset 0 -1 withscores
zrevrange myzset 0 -1 withscores
```

根据分值区间获取元素

```
zrangebyscore myzset 20 30
```

移除元素
也可以根据 score rank 删除

```
zrem myzset php cpp
```

统计元素个数

```
zcard myzset
```

分值递增

```
zincrby myzset 5 python
```

根据分值统计个数

```
zcount myzset 20 60
```

获取元素 rank

```
zrank myzset java
```

获取元素 score

```
zsocre myzset java
```

也有倒序的 rev 操作（reverse）

存储（实现）原理

同时满足以下条件时使用 ziplist 编码：

- 元素数量小于 128 个
- 所有 member 的长度都小于 64 字节

在 ziplist 的内部，按照 score 排序递增来存储。插入的时候要移动之后的数据。

对应 redis.conf 参数：
zset-max-ziplist-entries 128
zset-max-ziplist-value 64

超过阈值之后，使用 skiplist+dict 存储。

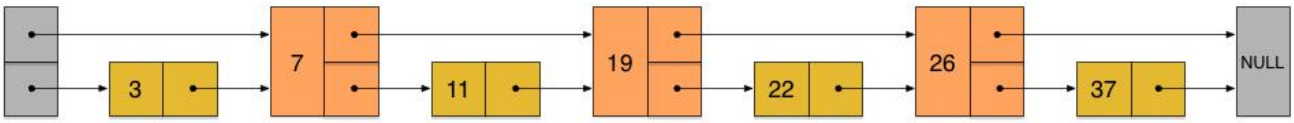
问题：什么是 skiplist？

我们先来看一下有序链表：

在这样一个链表中，如果我们要查找某个数据，那么需要从头开始逐个进行比较，直到找到包含数据的那个节点，或者找到第一个比给定数据大的节点为止（没找到）。也就是说，时间复杂度为 O(n)。同样，当我们要插入新数据的时候，也要经历同样的查找过程，从而确定插入位置。

而二分查找法只适用于有序数组，不适用于链表。

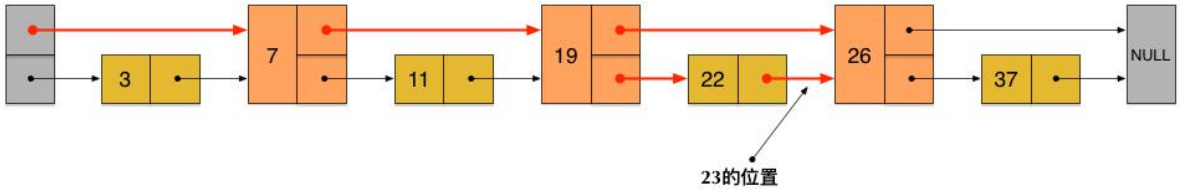
假如我们每相邻两个节点增加一个指针（或者理解为有三个元素进入了第二层），让指针指向下下个节点。



这样所有新增加的指针连成了一个新的链表，但它包含的节点个数只有原来的一半（上图中是 7, 19, 26）。在插入一个数据的时候，决定要放到那一层，取决于一个算法（在 redis 中 t_zset.c 有一个 zslRandomLevel 这个方法）。

现在当我们想查找数据的时候，可以先沿着这个新链表进行查找。当碰到比待查数据大的节点时，再回到原来的链表中的下一层进行查找。比如，我们想查找 23，查找的

路径是沿着下图中标红的指针所指向的方向进行的：



1. 23 首先和 7 比较，再和 19 比较，比它们都大，继续向后比较。
2. 但 23 和 26 比较的时候，比 26 要小，因此回到下面的链表（原链表），与 22 比较。
3. 23 比 22 要大，沿下面的指针继续向后和 26 比较。23 比 26 小，说明待查数据 23 在原链表中不存在

在这个查找过程中，由于新增加的指针，我们不再需要与链表中每个节点逐个进行比较了。需要比较的节点数大概只有原来的一半。这就是跳跃表。

为什么不用 AVL 树或者红黑树？因为 skiplist 更加简洁。

源码：server.h

```
typedef struct zskiplistNode {
    sds ele; /* zset 的元素 */
    double score; /* 分值 */
    struct zskiplistNode *backward; /* 后退指针 */
    struct zskiplistLevel {
        struct zskiplistNode *forward; /* 前进指针，对应 level 的下一个节点 */
        unsigned long span; /* 从当前节点到下一个节点的跨度（跨越的节点数） */
    } level[]; /* 层 */
} zskiplistNode;

typedef struct zskiplist {
    struct zskiplistNode *header, *tail; /* 指向跳跃表的头结点和尾节点 */
    unsigned long length;  /* 跳跃表的节点数 */
    int level; /* 最大的层数 */
} zskiplist;

typedef struct zset {
```

```
    dict *dict;
    zskiplist *zsl;
} zset;
```

随机获取层数的函数：

源码：t_zset.c

```
int zslRandomLevel(void) {
   int level = 1;
   while ((random()&0xFFFF) < (ZSKIPLIST_P * 0xFFFF))
       level += 1;
   return (level<ZSKIPLIST_MAXLEVEL) ? level : ZSKIPLIST_MAXLEVEL;
}
```

应用场景

排行榜

id 为 6001 的新闻点击数加 1：zincrby hotNews:20190926 1 n6001

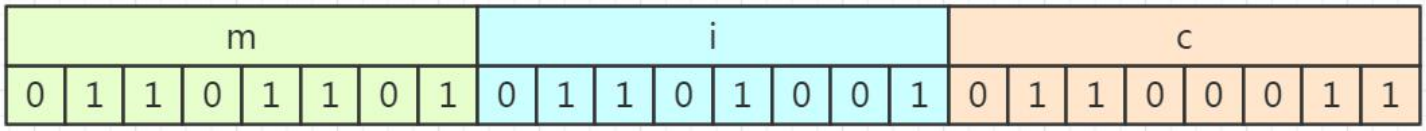
获取今天点击最多的 15 条：zrevrange hotNews:20190926 0 15 withscores

1.6.6. 其他数据结构简介

https://redis.io/topics/data-types-intro

BitMaps

Bitmaps 是在字符串类型上面定义的位操作。一个字节由 8 个二进制位组成。




```
set k1 a
```

获取 value 在 offset 处的值（a 对应的 ASCII 码是 97，转换为二进制数据是 01100001）

```
getbit k1 0
```

修改二进制数据（b 对应的 ASCII 码是 98，转换为二进制数据是 01100010）

```
setbit k1 6 1
setbit k1 7 0
get k1
```

统计二进制位中 1 的个数

```
bitcount k1
```

获取第一个 1 或者 0 的位置

```
bitpos k1 1
bitpos k1 0
```

BITOP 命令支持 AND 、 OR 、 NOT 、 XOR 这四种操作中的任意一种参数：
BITOP AND destkey srckey1 … srckeyN ，对一个或多个 key 求逻辑与，并将结果保存到 destkey
BITOP OR destkey srckey1 … srckeyN，对一个或多个 key 求逻辑或，并将结果保存到 destkey
BITOP XOR destkey srckey1 … srckeyN，对一个或多个 key 求逻辑异或，并将结果保存到 destkey
BITOP NOT destkey srckey，对给定 key 求逻辑非，并将结果保存到 destkey

应用场景：

用户访问统计

在线用户统计

Hyperloglogs

Hyperloglogs：提供了一种不太准确的基数统计方法，比如统计网站的 UV，存在一定的误差。HyperLogLogTest.java

Streams

5.0 推出的数据类型。支持多播的可持久化的消息队列，用于实现发布订阅功能，借鉴了 kafka 的设计。

1.6.7. 总结

数据结构总结

对象	对象 type 属性值	type 命令输出	底层可能的存储结构	object encoding
字符串对象	OBJ_STRING	"string"	OBJ_ENCODING_INT OBJ_ENCODING_EMBSTR OBJ_ENCODING_RAW	int embstr raw
列表对象	OBJ_LIST	"list"	OBJ_ENCODING_QUICKLIST	quicklist
哈希对象	OBJ_HASH	"hash"	OBJ_ENCODING_ZIPLIST OBJ_ENCODING_HT	ziplist hashtable
集合对象	OBJ_SET	"set"	OBJ_ENCODING_INTSET OBJ_ENCODING_HT	intset hashtable
有序集合对象	OBJ_ZSET	"zset"	OBJ_ENCODING_ZIPLIST OBJ_ENCODING_SKIPLIST	ziplist skiplist（包含 ht）

编码转换总结

对象	原始编码	升级编码	
字符串对象	INT	embstr	raw
	整数并且小于 long　2^63-1	超过 44 字节，被修改	
哈希对象	ziplist	hashtable	
	键和值的长度小于 64byte，键值对个数不超过 512 个，同时满足		
列表对象	quicklist		
集合对象	intset	hashtable	
	元素都是整数类型，元素个数小于 512 个，同时满足		
有序集合对象	ziplist	skiplist	
	元素数量不超过 128 个，任何一个 member 的长度小于 64 字节，同时满足。		

应用场景总结

缓存——提升热点数据的访问速度
共享数据——数据的存储和共享的问题
全局 ID —— 分布式全局 ID 的生成方案（分库分表）
分布式锁——进程间共享数据的原子操作保证
在线用户统计和计数
队列、栈——跨进程的队列/栈
消息队列——异步解耦的消息机制
服务注册与发现 —— RPC 通信机制的服务协调中心（Dubbo 支持 Redis）
购物车
新浪/Twitter 用户消息时间线
抽奖逻辑（礼物、转发）
点赞、签到、打卡
商品标签
用户（商品）关注（推荐）模型
电商产品筛选
排行榜

作者：咕泡学院-青山 2673