# Redis入门

*Redis*是一个*Key-Value*存储系统。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*)*、*set(*集合*)*和*zset(*有序集合*)*。这些数据类型都支持*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优势：*

*性能极高– Redis能读的速度是110000次/s,写的速度是81000次/s。*

*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。*

## Redis数据类型及其底层实现方式

Redis支持5种数据类型，以K-V形式进行存储，K是String类型的，V支持5种不同的数据类型，分别是：string，list，hash，set，sorted set，每一种数据结构都有其特定的应用场景。Redis底层数据结构有以下数据类型：简单动态字符串（SDS），链表，字典，跳跃表，整数集合，压缩列表，对象。

### 简单动态字符串

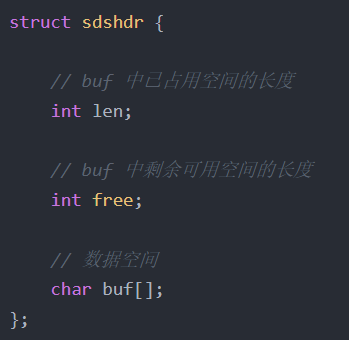
String的数据类型是由SDS实现的。Redis并没有采用C语言的字符串表示，而是自己构建了一种名为SDS的抽象类型，并将SDS作为Redis的默认字符串表示。



上边设置key=msg，value=hello world的键值对，它们的底层存储是：键（key）是字符串类型，其底层实现是一个保存着“msg”的SDS。值（value）是字符串类型，其底层实现是一个保存着“hello world”的SDS。

 注意：SDS除了用于实现字符串类型，还被用作AOF持久化时的缓冲区。

SDS的定义为



**为什么要使用SDS：**

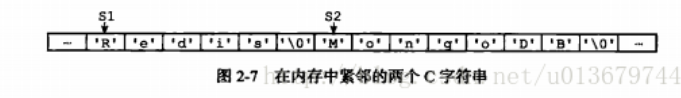
我们一定会思考，redis为什么不使用C语言的字符串而是费事搞一个SDS呢，这是因为C语言用N+1的字符数组来表示长度为N的字符串，这样做在获取字符串长度，字符串扩展等操作方面效率较低，并且无法满足redis对字符串在安全性、效率以及功能方面的要求。

**获取字符串长度（SDS O（1））**

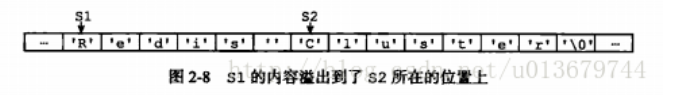
在C语言字符串中，为了获取一个字符串的长度，必须遍历整个字符串，时间复杂度为O(n)，而SDS中，有专门用于保存字符串长度的变量len，所以可以在O（1）时间内获得。

**防止缓冲区溢出**

C字符串，容易导致缓冲区溢出，假设在程序中存在内存紧邻的字符串s1和s2，s1保存redis，s2保存MongoDB，



如果我们现在将s1 的内容修改为redis cluster，但是又忘了重新为s1 分配足够的空间，这时候就会出现以下问题：



 因为s1和s2是紧邻的，所以原本s2 中的内容已经被S1的内容给占领了，s2 现在为 cluster，而不是“Mongodb”。而Redis中的SDS就杜绝了发生缓冲区溢出的可能性。 当我们需要对一个SDS 进行修改的时候，redis 会在执行拼接操作之前，**预先检查给定SDS 空间是否足够（free记录了剩余可用的数据长度）**，如果不够，会先拓展SDS 的空间，然后再执行拼接操作。

**减少扩展或收缩字符串带来的内存重分配次数**

当字符串进行扩展或收缩时，都会对内存空间进行重新分配。

1. 字符串拼接会产生字符串的内存空间的扩充，在拼接的过程中，原来的字符串的大小很可能小于拼接后的字符串的大小，那么这样的话，就会导致一旦忘记申请分配空间，就会导致内存的溢出。

　2. 字符串在进行收缩的时候，内存空间会相应的收缩，而如果在进行字符串的切割的时候，没有对内存的空间进行一个重新分配，那么这部分多出来的空间就成为了内存泄露。

比如：字符串"redis"，当进行字符串拼接时，将redis+cluster=13，会将SDS的长度修改为13，同时将free也改为13，这意味着进行预分配了，将buffer大小变为了26。这是为了如果再次执行字符串拼接操作，如果拼接的字符串长度<13,就不需要重新进行内存分配了。

**通过这种预分配策略（将free空间与已用空间置为相等），SDS将连续增长N次字符串所需的内存重分配次数从必定N次降低为最多N次。通过惰性空间释放，SDS 避免了缩短字符串时所需的内存重分配操作，并未将来可能有的增长操作提供了优化。**

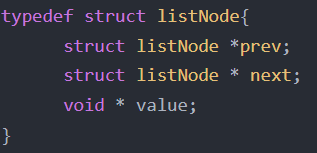
**二进制安全**

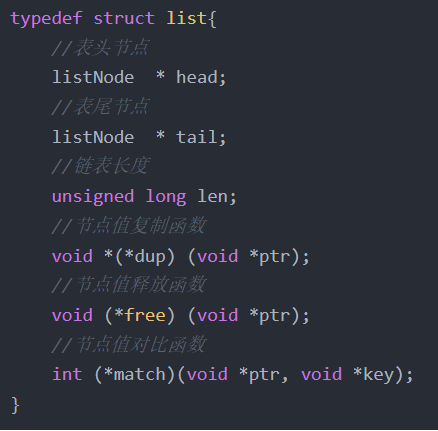
    C 字符串中的字符必须符合某种编码，并且除了字符串的末尾之外，字符串里面不能包含空字符，否则最先被程序读入的空字符将被误认为是字符串结尾，这些限制使得C字符串只能保存文本数据，而不能保存想图片，音频，视频，压缩文件这样的二进制数据。

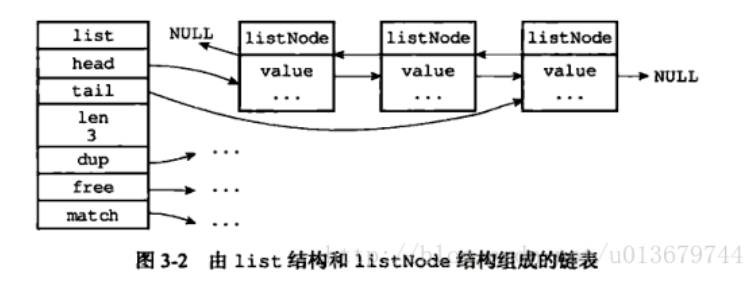
　但是在Redis中，不是靠空字符来判断字符串的结束的，而是通过len这个属性。那么，即便是中间出现了空字符对于SDS来说，读取该字符仍然是可以的。

### 链表

链表是list的实现方式之一。当list包含了数量较多的元素，或者列表中包含的元素都是比较长的字符串时，Redis会使用链表作为实现List的底层实现。此链表是双向链表：



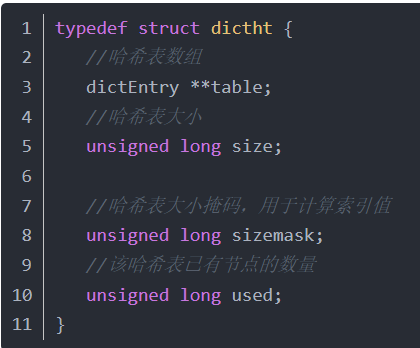




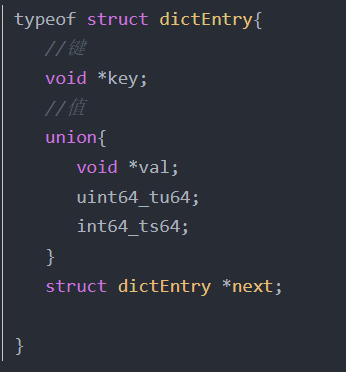
链表结构的特点是可以快速的在表头和表尾插入和删除元素，但查找复杂度高，是列表的底层实现之一，也因此列表没有提供判断某一元素是否在列表中的接口，因为在链表中查找复杂度高。

### 字典

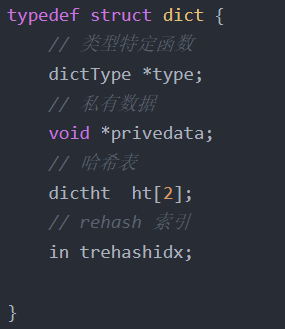
字典，又称为符号表（symbol table）、关联数组（associative array）或映射（map），是一种用于保存键值对的抽象数据结构。在字典中，一个键（key）可以和一个值（value）进行关联，字典中的每个键都是独一无二的。在C语言中，并没有这种数据结构，但是Redis 中构建了自己的字典实现。



我们可以想到对比Java hashMap的实现方式，在dictht中，table数组的类型是：

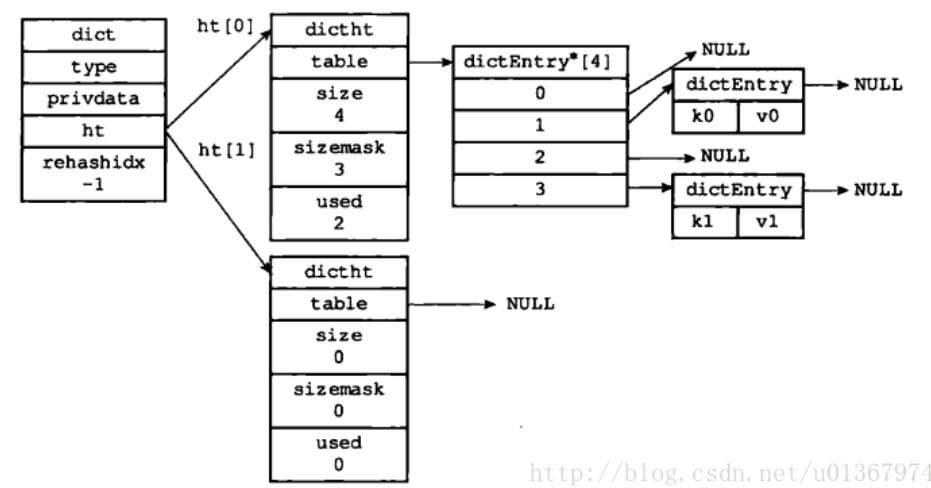


注意：Redis又在dictht的基础上，又抽象了一层字典dict，其定义为：



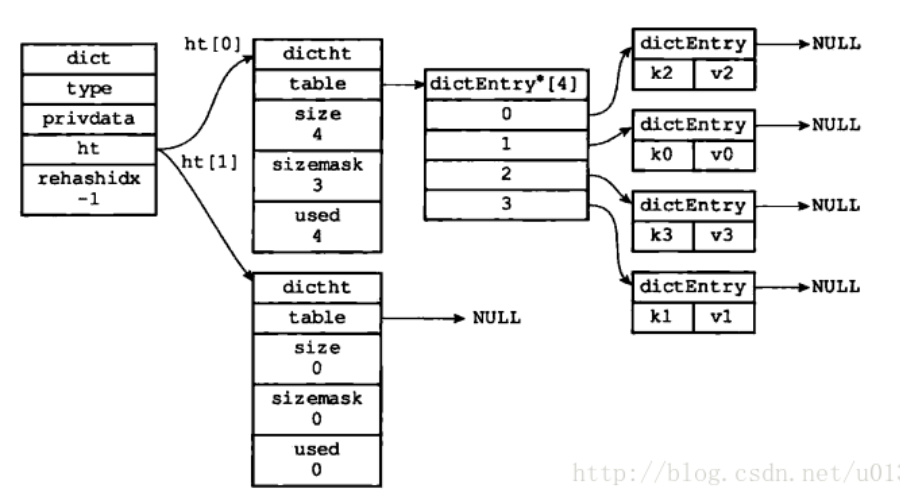
type 属性 和privdata 属性是针对不同类型的键值对，为创建多态字典而设置的。（同一个字典里会存在多种类型的键值对）

ht 属性是一个包含两个项（两个哈希表）的数组（扩容），如图：

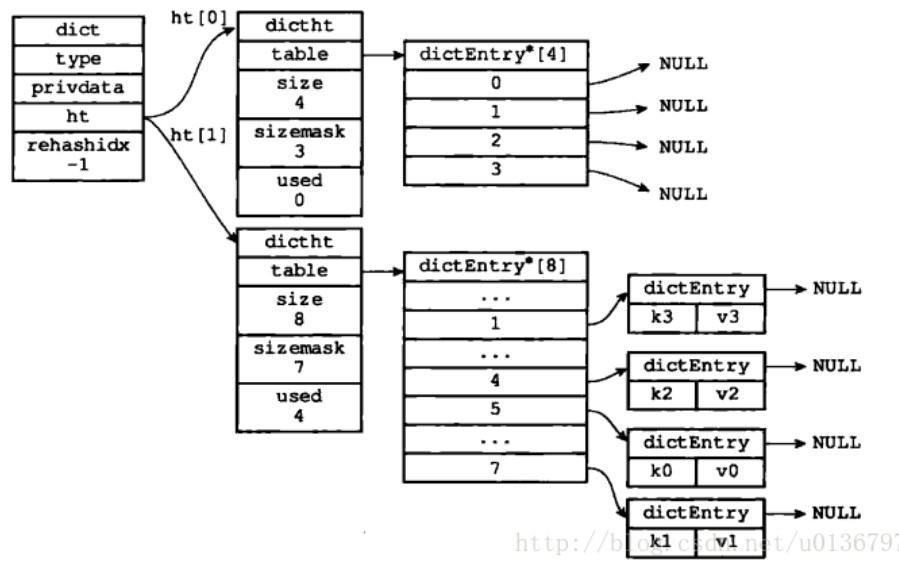


**解决hash冲突**：采用链地址法来实现。

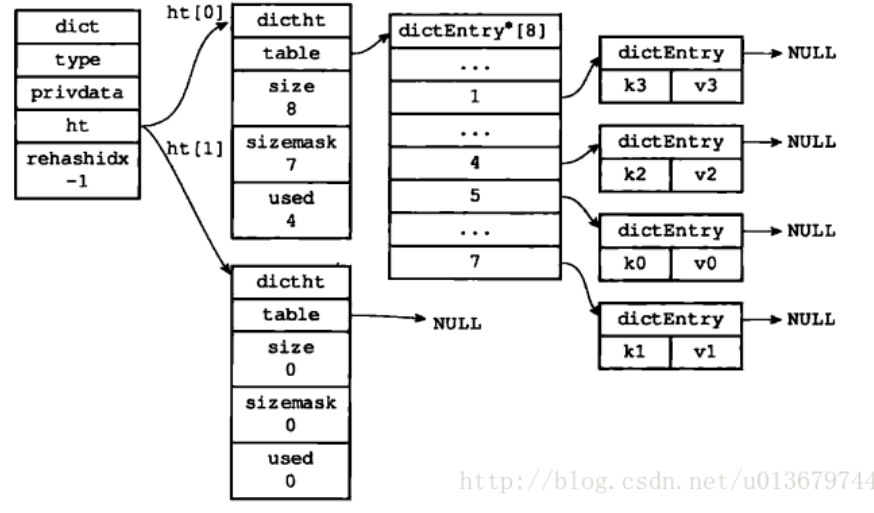
**扩充Rehash：**　随着对哈希表的不断操作，哈希表保存的键值对会逐渐的发生改变，为了让哈希表的负载因子维持在一个合理的范围之内，我们需要对哈希表的大小进行相应的扩展或者压缩，这时候，我们可以通过 rehash（重新散列）操作来完成。其实现方式和hashmap略有不同，因为dict有两个hash表dictht，所以它是通过这两个dictht互相进行转移的。（提高并发性）



上图这种情况，代表要进行扩容了，所以就要将ht[0]的数据转移到ht[1]中。ht[1]创建为2\*ht[0].size大小的



将ht[0]释放，然后将ht[1]设置成ht[0]，最后为ht[1]分配一个空白哈希表。



**渐进式rehash：**在实际开发过程中，这个rehash 操作并不是一次性、集中式完成的，而是分多次、渐进式地完成的。采用渐进式rehash 的好处在于它采取分而治之的方式，避免了集中式rehash 带来的庞大计算量。

渐进式rehash 的详细步骤：

　　1、为ht[1] 分配空间，让字典同时持有ht[0]和ht[1]两个哈希表

　　2、在几点钟维持一个索引计数器变量rehashidx，并将它的值设置为0，表示rehash 开始

 　 3、在rehash 进行期间，每次对字典执行CRUD操作时，程序除了执行指定的操作以外，还会将ht[0]中的数据rehash 到ht[1]表中，并且将rehashidx加一

  4、当ht[0]中所有数据转移到ht[1]中时，将rehashidx 设置成-1，表示rehash 结束

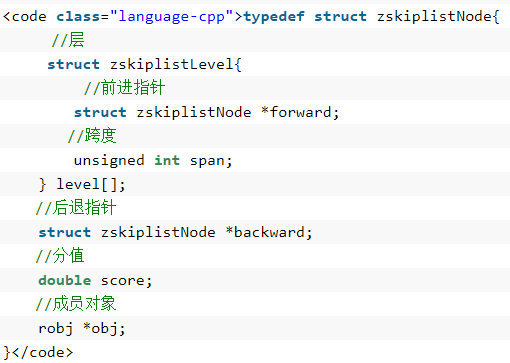
### 跳跃表

Redis 只在两个地方用到了跳跃表，一个是实现有序集合键（sorted Sets），另外一个是在集群节点中用作内部数据结构。

其实跳表主要是来替代平衡二叉树的，比起平衡树来说，跳表的实现要简单直观的多。

跳跃表（skiplist）是一种有序数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速查找访问节点的目的。跳跃表是一种随机化的数据,跳跃表以有序的方式在层次化的链表中保存元素，效率和平衡树媲美 ——查找、删除、添加等操作都可以在O（logn）期望时间下完成。

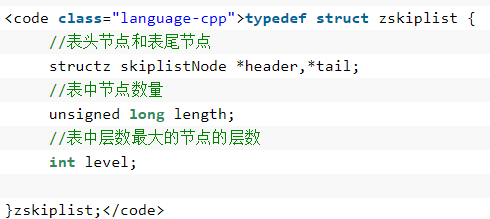
Redis 的跳跃表 主要由两部分组成：zskiplist（链表）和zskiplistNode （节点）：



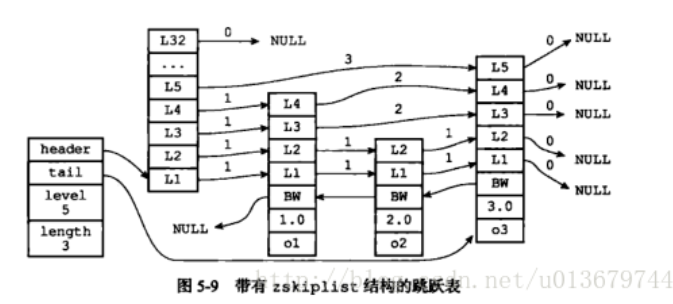
1、层：level 数组可以包含多个元素，每个元素都包含一个指向其他节点的指针。level数组的每个元素都包含：前进指针：用于指向表尾方向的前进指针，跨度：用于记录两个节点之间的距离

2、后退指针：用于从表尾向表头方向访问节点

3、分值和成员：跳跃表中的所有节点都按分值从小到大排序（按照这个进行排序的，也就是平衡二叉树（搜索树的）的节点大小）。成员对象指向一个字符串，这个字符串对象保存着一个SDS值（实际存储的值）

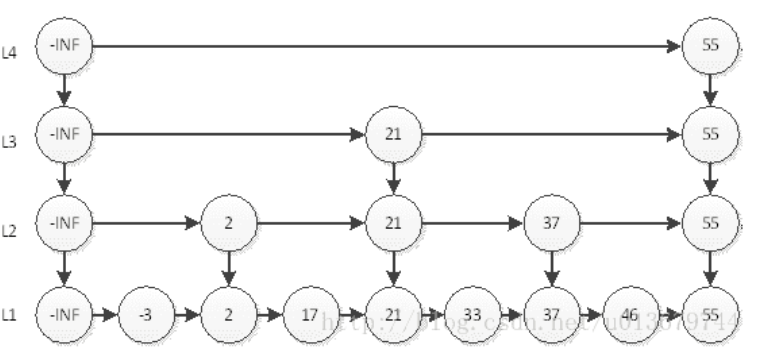


header，tail分别指向跳跃表的头结点和尾节点。level 用于记录最大的层数，length 用于记录我们的节点数量。



* 跳跃表是有序集合的底层实现之一
* 主要有zskiplist 和zskiplistNode两个结构组成
* 每个跳跃表节点的层高都是1至32之间的随机数
* 在同一个跳跃表中，多个节点可以包含相同的分值，但每个节点的对象必须是唯一的
* 节点按照分值的大小从大到小排序，如果分值相同，则按成员对象大小排序

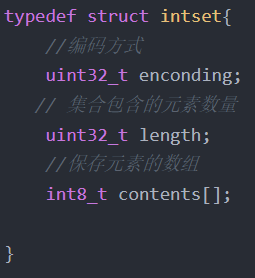
#### **怎么使用跳表来实现O（logn）的增删改查**



如果要查询46，从最高层开始，先和55比，比55小。向下一层，和21比，比21大，向前，和55比，比55小，向下。和37比，比37大，向前，和55比，比55小，向下，和46比，相等。最终比较6次。

### 整数集合（Intset）

一个特殊的集合，里面存储的数据只能够是整数，并且数据量不能过大。

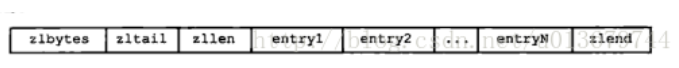


整数集合是集合建的底层实现之一.

整数集合的底层实现为数组，这个数组以有序，无重复的范式保存集合元素，在有需要时，程序会根据新添加的元素类型改变这个数组的类型.

### 压缩列表

压缩列表是列表键（list）和哈希键（hash）的底层实现之一。当一个列表键只有少量列表项，并且每个列表项要么就是小整数，要么就是长度比较短的字符串，那么Redis 就会使用压缩列表来做列表键的底层实现。



1、zlbytes:用于记录整个压缩列表占用的内存字节数

　　2、zltail：记录列表尾节点距离压缩列表的起始地址有多少字节

　　3、zllen：记录了压缩列表包含的节点数量。

　　4、entryX：要说列表包含的各个节点

　　5、zlend：用于标记压缩列表的末端

压缩列表是一种为了节约内存而开发的顺序型数据结构

压缩列表被用作列表键和哈希键的底层实现之一

压缩列表可以包含多个节点，每个节点可以保存一个字节数组或者整数值

添加新节点到压缩列表，可能会引发连锁更新操作。

## [数据类型的应用](https://www.cnblogs.com/xiaoxi/p/7007695.html)

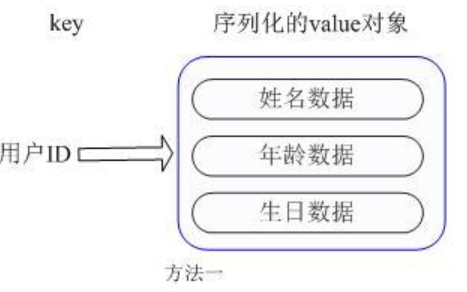
### **String**

**使用场景：**常规key-value缓存应用。常规计数: 微博数, 粉丝数。

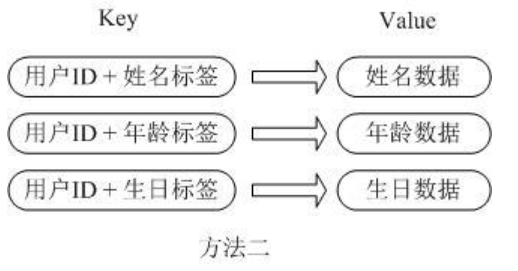
### **Hash**

我们简单举个实例来描述下Hash的应用场景，比如我们要存储一个用户信息对象数据，包含以下信息：

用户ID为查找的key，存储的value用户对象包含姓名，年龄，生日等信息，如果用普通的key/value结构来存储，主要有以下2种存储方式：

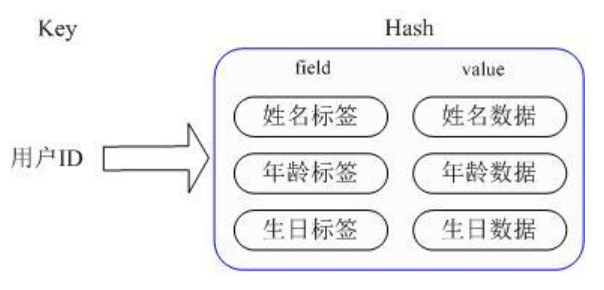


第一种方式将用户ID作为查找key,把其他信息封装成一个对象以序列化的方式存储，这种方式的缺点是，增加了序列化/反序列化的开销，并且在需要修改其中一项信息时，需要把整个对象取回，并且修改操作需要对并发进行保护，引入CAS等复杂问题。



第二种方法是这个用户信息对象有多少成员就存成多少个key-value对儿，用用户ID+对应属性的名称作为唯一标识来取得对应属性的值，虽然省去了序列化开销和并发问题，但是用户ID为重复存储，如果存在大量这样的数据，内存浪费还是非常可观的。

那么Redis提供的Hash很好的解决了这个问题，Redis的Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口，



也就是说，Key仍然是用户ID, value是一个Map，这个Map的key是成员的属性名，value是属性值，这样对数据的修改和存取都可以直接通过其内部Map的Key(Redis里称内部Map的key为field), 也就是通过 key(用户ID) + field(属性标签) 就可以操作对应属性数据了，既不需要重复存储数据，也不会带来序列化和并发修改控制的问题。很好的解决了问题。

**使用场景：**存储部分变更数据，如用户信息等。

### **List**

Redis list的应用场景非常多，也是Redis最重要的数据结构之一，比如twitter的关注列表，粉丝列表等都可以用Redis的list结构来实现。

List 就是链表，相信略有数据结构知识的人都应该能理解其结构。使用List结构，我们可以轻松地实现最新消息排行等功能。List的另一个应用就是消息队列，  
可以利用List的PUSH操作，将任务存在List中，然后工作线程再用POP操作将任务取出进行执行。Redis还提供了操作List中某一段的api，你可以直接查询，删除List中某一段的元素。

**使用场景**：消息[队列](http://blog.nosqlfan.com/tags/%E9%98%9F%E5%88%97)系统，使用list可以构建队列系统，使用sorted set甚至可以构建有优先级的队列系统；取最新N个数据的操作

### **Set**

**使用场景**：交集，并集，差集：(Set)；

在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis还为集合提供了求交集、并集、差集等操作，可以非常方便的实现如共同关注、共同喜好、二度好友等功能，对上面的所有集合操作，你还可以使用不同的[命令](http://blog.nosqlfan.com/tags/%E5%91%BD%E4%BB%A4)选择将结果返回给客户端还是存集到一个新的集合中。

### **Sorted Set**

Redis sorted set的使用场景与set类似，区别是set不是自动有序的，而sorted set可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择sorted set数据结构，比如twitter 的public timeline可以以发表时间作为score来存储，这样获取时就是自动按时间排好序的。

和Set相比，Sorted Set增加了一个权重参数score，使得集合中的元素能够按score进行有序排列，比如一个存储全班同学成绩的Sorted Set，其集合value可以是同学的学号，而score就可以是其考试得分，这样在数据插入集合的时候，就已经进行了天然的排序。另外还可以用Sorted Set来做带权重的队列，比如普通消息的score为1，重要消息的score为2，然后工作线程可以选择按score的倒序来获取工作任务。让重要的任务优先执行。