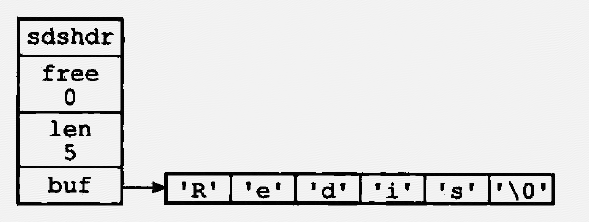
Redis设计与实现

## 第一部分：数据结构与对象

### 第二章：简单动态字符串（Simple Dynamic String）

Redis的默认字符串表示为SDS，即简单动态字符串。结构图为：



Len记录了SDS中记录的字符串的长度：查询长度的时间复杂度为O(1)。

buf记录了SDS中存放的数据；buf中的实际长度为len+free+1byte。

free表示了SDS中未使用的空间。实现了空间预分配和惰性空间释放两种策略。

1.空间预分配：

优化了SDS的字符串增长操作，减少了字符串增长带来的内存重分配次数。

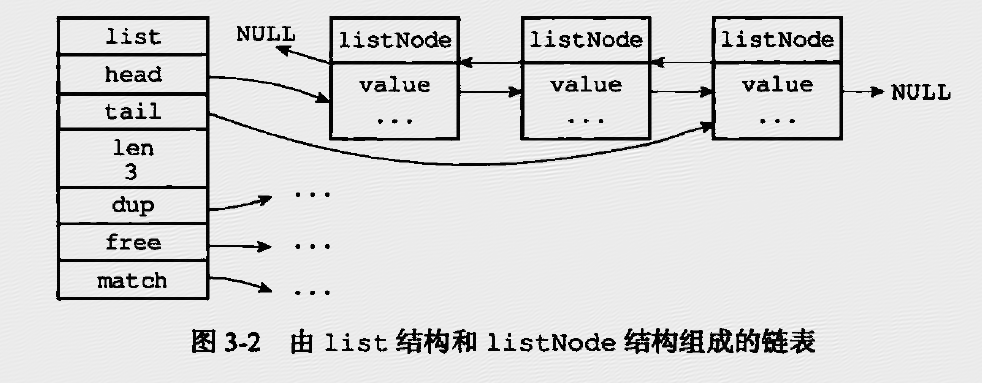
2.惰性空间释放：

优化了SDS的字符串缩短操作，减少了字符串缩减带来的内存重分配次数。

### 第三章：链表 list

链表提供了高效的节点重排能力，以及顺序性的节点访问方式，并且可以通过增删节点灵活的调整链表的长度。

链表在redis中应用举例：列表键的底层实现之一就是链表：当一个列表键包含了数量比较多的元素，或者列表中包含的元素都是比较长的字符串时，redis就会使用链表作为列表键的底层实现。发布与订阅，慢查询，监视器等功能也用到了链表，redis服务器使用链表保存多个客户端的状态信息，以及使用链表来构建客户端输出缓冲区。



Redis链表的实现特点：

1.双端：链表节点带有prev和next指针，获取某个节点的前置节点和后置节点复杂度都是O(1)

2.无环：表头节点的prev和表尾节点的next指针都指向null，所以对链表的访问都是nulll为终点

3.带表头指针和表尾指针：通过list结构的head和tail指针，获取链表的表头节点和表尾节点的复杂度都为O(1)

4.带链表长度计数器：程序可以使用list的len属性直接获取链表长度，复杂度O(1)

### 第四章：字典 map

字典，又称为符号表（symbol table），关联数组（associative array）或映射（map），是用于保存键值对的抽象数据结构。

字典在redis中应用广泛，举例：

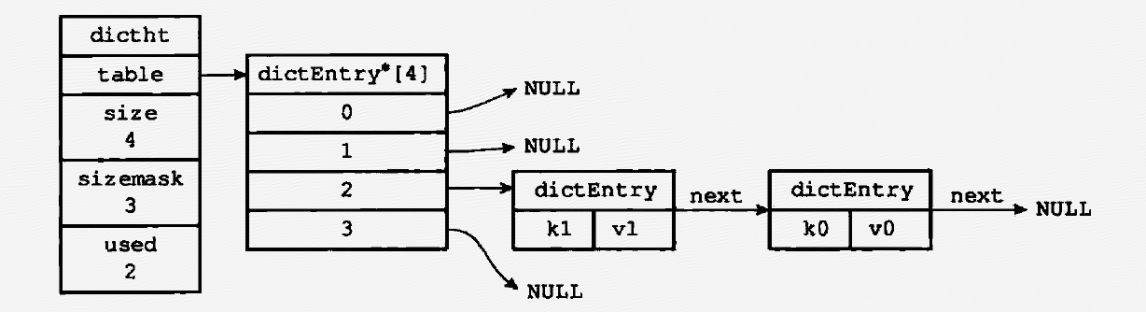
1).redis的数据库就是使用字典来作为底层实现的，对数据库的增删改查操作也是构建在对字典的操作之上

2).字典还是哈希键的底层实现之一，当一个哈希键包含的键值对比较多，或者键值对中的元素都是比较长的字符串时，redis就会使用字典作为哈希键的底层实现。

4.1 字典的实现

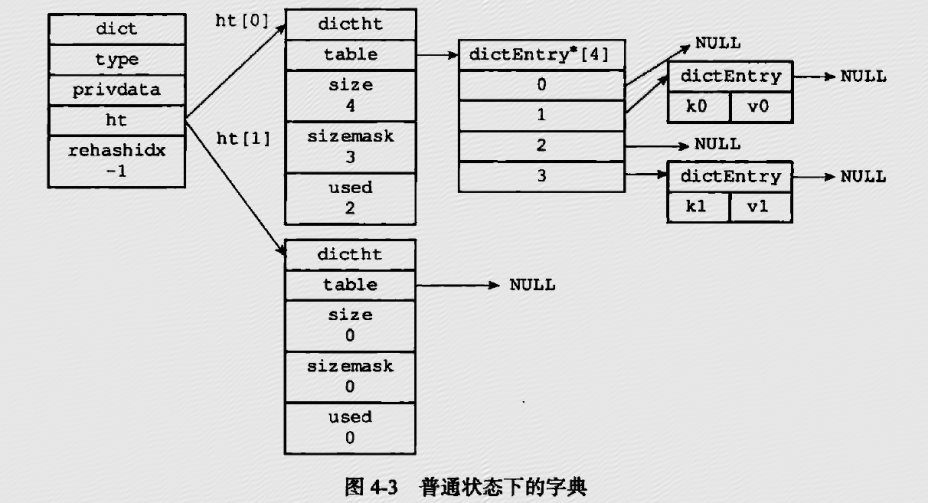
Redis的字典使用哈希表作为底层实现，一个哈希表里面可以有多个哈希表节点，每个哈希表节点就保存了字典中的一个键值对。

Redis字典所使用的哈希表由dict.h/dictht结构定义



上图中table是一个数组，由dictEntry结构组成，size表示哈希表大小，即table数组大小，used表示已有节点(键值对)的数量。dictEntry包含了k，v，还有一个next，next指向了下一个hash值相等的键值对，形成链表来解决键冲突的问题，注意，为了性能考虑，程序总是将新节点放在该链表的表头位置，排在其他节点的前面。

Redis中的字典由dict.h/dict结构表示：



Redis中的字典如上图表示，type属性和pirvdata属性是针对不同类型的键值对，为创建多态字典而设置的。ht属性是一个包含两个项的数组，每一项都是一个dictht哈希表，字典只使用ht[0]哈希表，ht[1]哈希表只会在对ht[0]进行rehash时使用。Rehashidx记录了rehash目前的进度，如果没有进行rehash，那它的值为-1.

在对哈希表进行扩展和收缩操作时，程序需要将现有hash表所有键值对rehash，并且这个rehash不是一次性完成的，渐进式的。

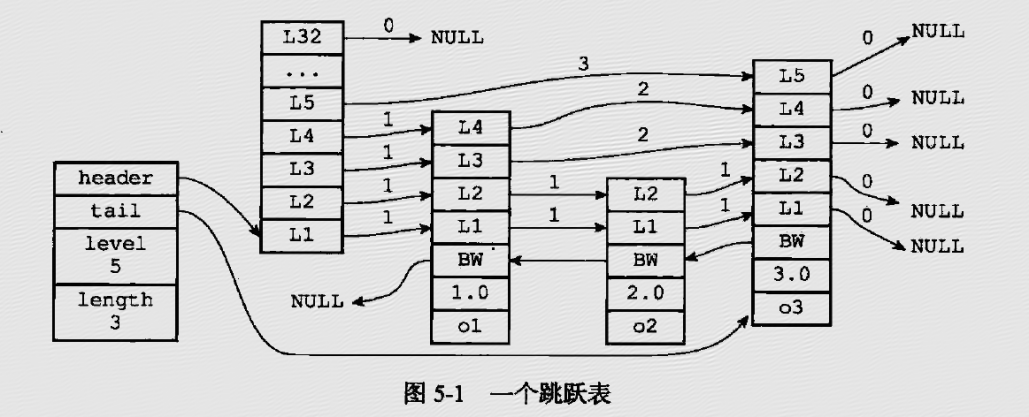
### 第五章：跳跃表

跳跃表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，达到快速访问节点的目的。

redis使用跳跃表作为有序集合键的底层实现之一，如果一个有序集合包含的元素数量比较多，又或者有序集合中元素的成员是比较长的字符串，redis就会使用跳跃表来作为有序集合键的底层实现。

Redis只在两个地方用到了跳跃表：一是实现有序集合，二是在集群节点中用作内部数据结构。

跳跃表由redis.h/zskiplistNode和redis.h/zskiplist两个结构定义，zskiplistNode表示跳跃表节点，zskiplist用于保存跳跃表节点的相关信息。跳跃表示例：



如上图所示，最左边是一个zskiplist结构，该结构包含属性：

1. Header：指向跳跃表的表头节点

2. Tail：指向跳跃表的表尾节点

3. Level：记录目前跳跃表中层数最大的那个节点的层数（表头节点的层数不计算在内）

4. Length：跳跃表的长度，即跳跃表目前包含的节点数量（表头节点不计算在内）

位于zskiplist结构右方的是四个zskiplistNode结构，该结构包含属性：

1. 层（level）：节点用L1、L2、L3等字样标记节点各个层。每个层都带有两个属性:前进指针和跨度。前进指针用于访问表尾方向的其他节点，跨度则记录了前进指针所指向节点与当前节点的举例

2. 后退指针（backward）：节点中用BW字样标记节点的后退指针，它指向位于当前节点的前一个节点。后退指针在程序从表尾向表头遍历时使用。

3. 分值（score）：各个节点中的1.0，2.0和3.0是节点锁保存的分值。在跳跃表中，节点按各自所保存的分值从小到大排列。

4. 成员对象（obj）：各个节点中的o1、o2和o3是节点所保存的成员对象。

注意表头节点和其他节点的构造是一样的，表头节点也有后退指针，分值和成员对象，不过都不会用到，上图省略了。

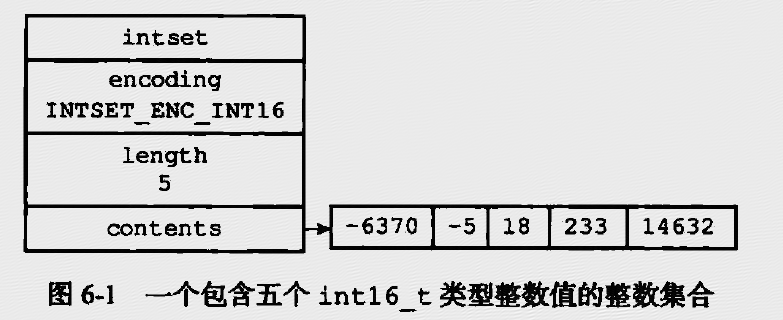
在同一个跳跃表中，多个节点可以包含相同的分值，但是每个节点的成员对象必须是唯一的。

跳跃表中的节点按照分值大小进行排序，当分值相同时，节点按照成员对象的大小进行排序。

### 第六章：整数集合

整数集合（intset）是集合键的底层实现之一，当一个集合只包含整数值元素，并且这个集合的元素数量不多时，redis就会使用整数集合作为集合键的底层实现。

每个intset.h/intset结构表示一个整数集合。



1. Contents数组是整数集合的底层实现：整数集合的每个元素都是contents数组的一个数据项（item），各个项在数组中从小到大有序排列，并且数组中不包含任何重复项。

2. length属性记录了整数集合包含的元素数量，即contents数组长度。

每当有新元素添加到整数集合里面，并且新元素的类型比整数集合现有所有元素的类型都要长时，整数集合需要先进行升级（upgrade），然后才能将新元素添加到整数集合里面。升级整数集合并添加新元素共分为三步进行：

1. 根据新元素类型，扩展整数集合底层数组的空间大小，并为新元素分配空间

2. 将底层数组现有的所有元素都转换成与新元素相同的类型，并将类型转换后的元素放置到正确的位上，而且在放置元素的过程中，需要继续维持底层数组的有序性质不变。

3. 将新元素添加到底层数组里面。

整数集合的升级策略带来的好处：

1. 提升整数集合的灵活性。

随意将不同类型（int16\_t,int32\_t,int64\_t）的整数添加到集合，而不必担心类型错误。

2. 尽可能的节约内存。

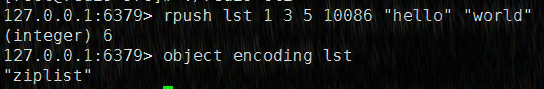
如果数组要保存int16\_t,int32\_t,int64\_t三种类型的值，最简单的做法是直接使用int64\_t作为整数集合的底层实现。不过这样的话会浪费内存。整数集合现在的做法是既可以让集合能同时保存三种不同类型的值，又可以确保升级操作只会在有需要的时候（要存入的整数比现有的类型都要长）进行，尽量节省内存。

整数集合只支持升级，不支持降级操作。

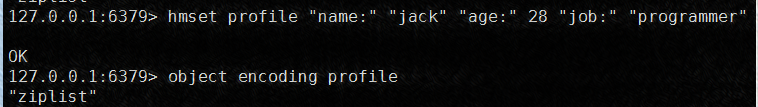
### 第七章：压缩列表

压缩列表（ziplist）是列表键和哈希键的底层实现之一。

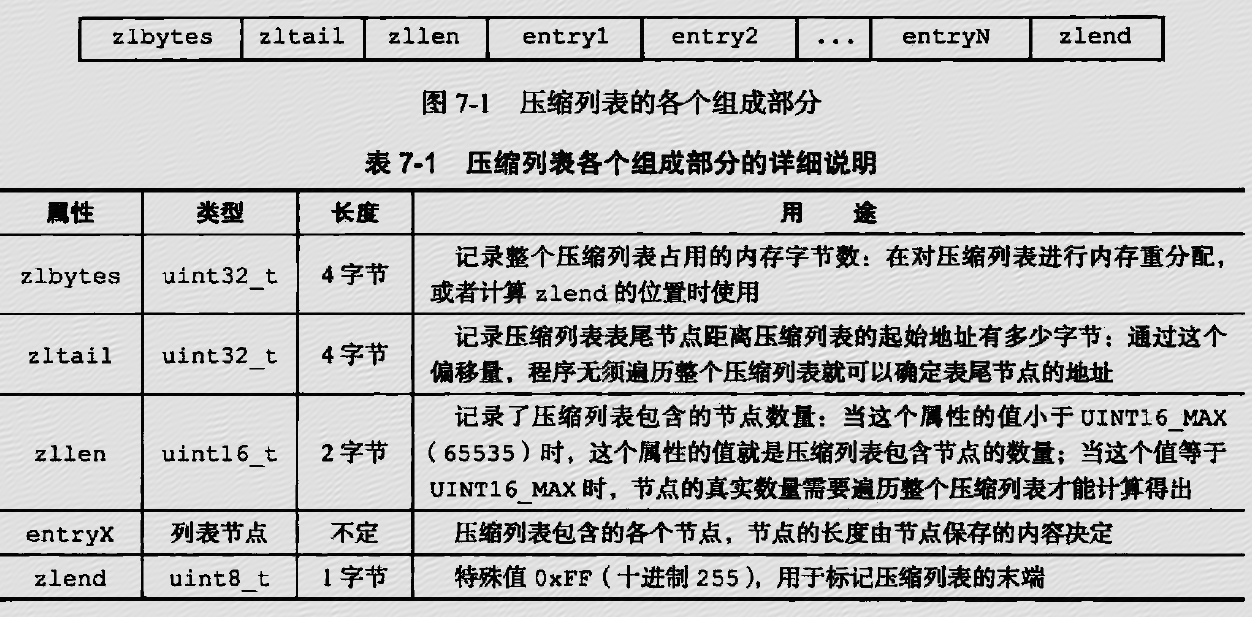
当一个列表键只包含少量列表项，并且每个列表项要么是小整数值，要么是长度比较短的字符串，那么redis就会使用压缩列表作为列表键的底层实现：举例



当一个哈希键只包含少量键值对，并且每个键值对的键和值要么是小整数值，要么就是长度比较短的字符串，那么redis就会使用压缩列表作为哈希键的底层实现。举例



压缩列表是redis为了节约内存而开发的，有一系列特殊编码的连续内存块组成的顺序型数据结构。一个压缩列表可以包含任意多个节点（entry），每个节点可以保存一个字节数组或者一个整数值。下图为压缩列表的组成。



### 第八章：对象

Redis没有直接使用上面的数据结构来实现键值对数据库，而是基于上述数据结构创建了对象：

字符串对象：<http://redisdoc.com/string/index.html>

列表对象：<http://redisdoc.com/list/index.html>

哈希对象：<http://redisdoc.com/hash/index.html>

集合对象：<http://redisdoc.com/set/index.html>

有序集合对象：<http://redisdoc.com/sorted_set/index.html>

字符串对象是redis五中类型的对象中唯一一个会被其他四种类型对象用到的对象。

每种对象都用到了至少一种前面所介绍的数据结构。Redis对象系统的好处：

1.在执行命令之前，根据对象类型判断一个对象是否可以执行给定的命令

2.针对不同的使用场景，为对象设置多种不同的数据结构实现，从而优化对象在不同场景下的使用效率

3.redis的对象系统实现了基于引用计数的内存回收机制，不使用了就会自动释放

4.redis通过引用计数技术实现了对象共享机制，在适当条件下，通过让多个数据库共享同一个对象来节约内存

5.redis的对象带有访问时间记录信息，该信息可以用于计算数据库键的空转时长，在服务器启用了maxmemory功能的情况下，空转时长较大的键会优先被服务器删除

#### 8.1 对象的类型和编码

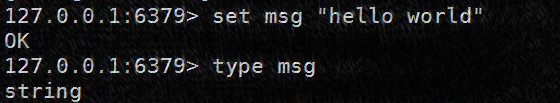
Redis使用对象来表示数据库中的键和值，键值对的键和值分别都是一个对象。Redis的每个对象都由一个redisObject结构表示，该结构和保存数据有关的三个属性分别是type属性，encoding属性和ptr属性：



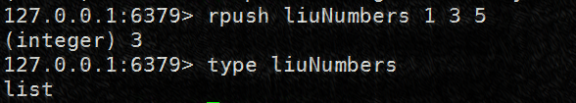
8.1.1 类型：

Redis数据库保存的键值对，键总是字符串对象，值可以是字符串对象，列表对象，哈希对象，集合对象或者有序集合对象。

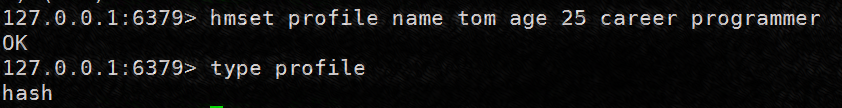
#键为字符串对象，值为字符串对象



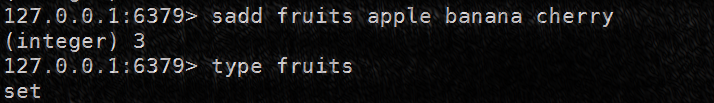
#键为字符串对象，值为列表对象



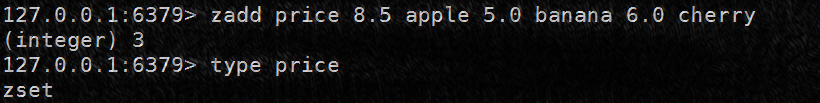
#键为字符串对象，值为哈希对象



#键为字符串对象，值为集合对象

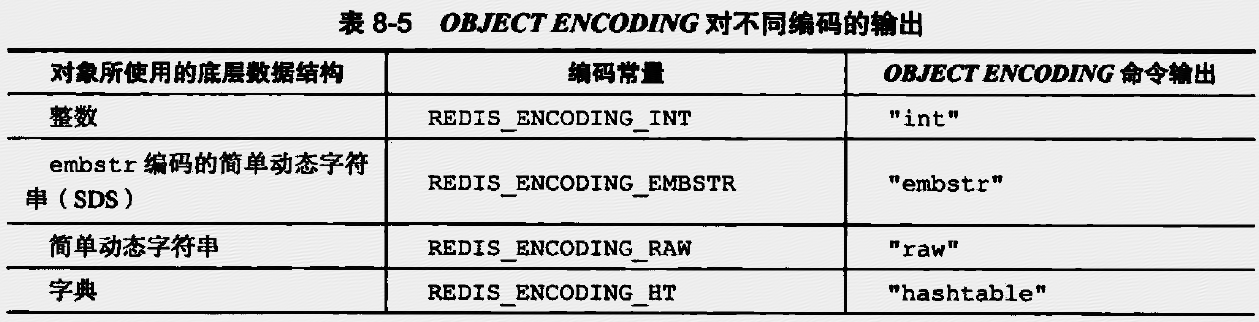


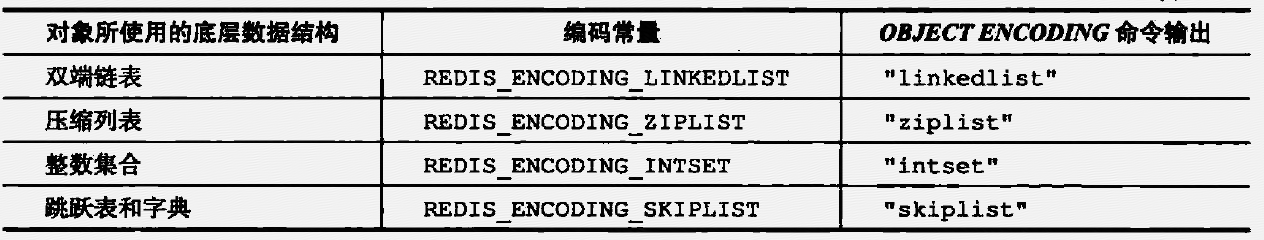
#键为字符串对象，值为有序集合对象



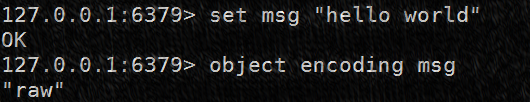
8.1.2 编码和底层实现

对象的ptr指针指向对象的底层实现数据结构，encoding属性记录了对象所使用的编码，即这个对象使用了什么数据结构作为对象的底层实现。

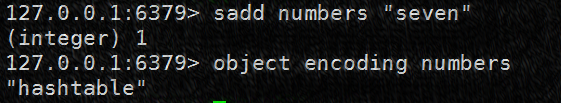




使用object encoding命令查看键对应值对象的编码





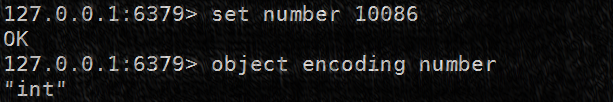
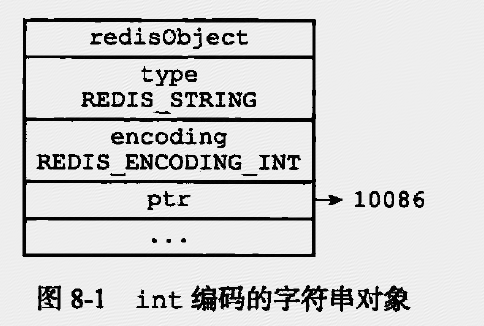


#### 8.2 字符串对象

字符串对象的编码可以是int，raw或者embstr。

1. int

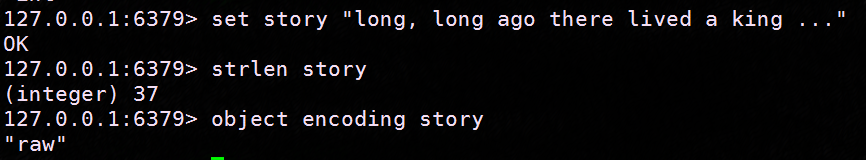
如果一个字符串保存的对象是整数值，并且这个整数值可以用long类型来表示，那么字符串对象会将整数值保存在字符串对象结构的ptr属性中（void 转换为long），并将字符串对象编码设置为int。



2. raw

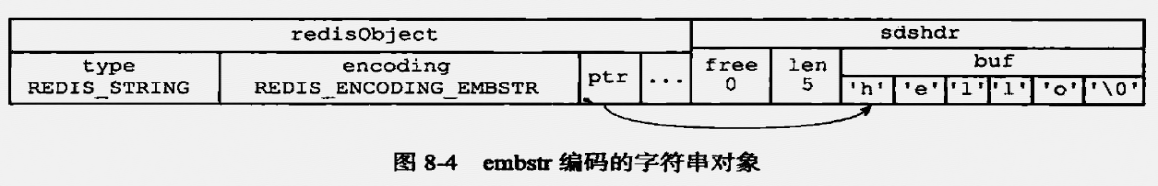
如果字符串对象保存的是一个字符串值，并且这个字符串值大于32个字节，那字符串对象将使用一个简单动态字符串（SDS）来保存这个字符串值，编码设置为raw。

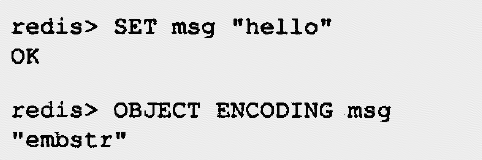




3. embstr

如果字符串保存的是一个字符串值，并且长度小于32个字节，那么对象将使用embstr编码来保存这个字符串值。（我在本地测试，没有成功）





embstr对比raw编码优点：

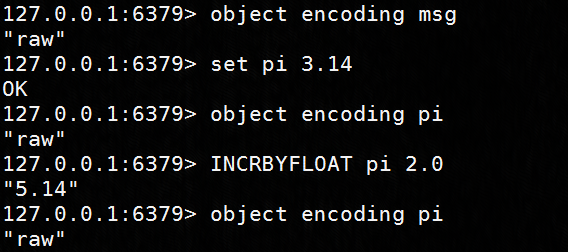
1.raw和embstr都是使用RedisObject结构和sdshdr结构来表示字符串对象，但是raw编码会进行两次内存分配分别创建redisobject结构和sdshdr结构来表示字符串，embstr只调用一次内存分配创建一块连续空间，空间就包含了redisobject和sdshdr两个结构。

2.释放embstr编码的字符串对象只需要一次内存释放函数，raw需要两次。

3.embstr编码所有数据都在一块连续空间中，能跟好的利用缓存带来的优势。

Int和embstr编码会在一定条件下转换成raw编码。（放入的不再是整数值，或者超过32个字节）

浮点类型long double：

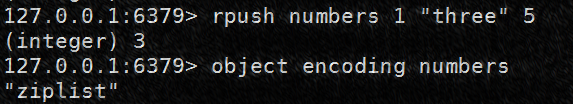


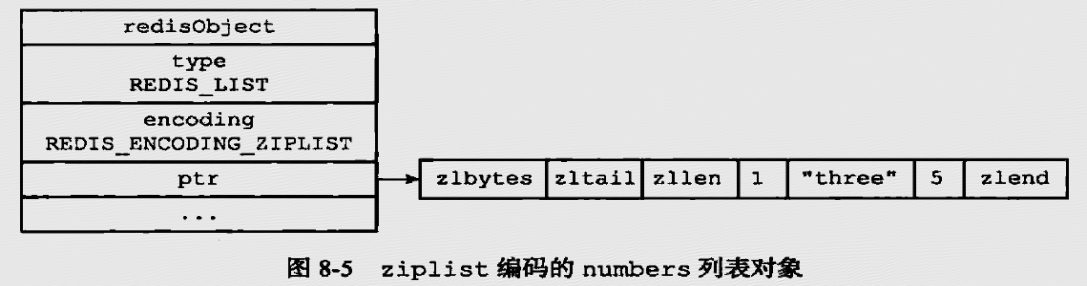
对于浮点类型，保存时是以字符串值来保存的，编码为embstr或者raw，当对该值进行计算的时候，会将字符串能转换成浮点类型，然后将计算结果再次转换成字符串保存

#### 8.3 列表对象

列表对象的编码可以是ziplist或者linkedlist。

Ziplist编码的列表对象使用压缩列表作为底层实现，每个压缩列表节点（entry）保存了一个列表元素。





Linkedlist使用双端链表作为底层实现，每个双端链表节点（node）都保存了一个字符串对象，每个字符串对象都保存了一个列表元素。



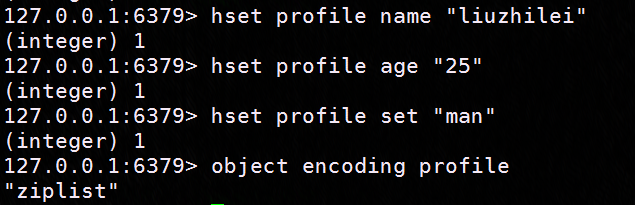
编码转换：

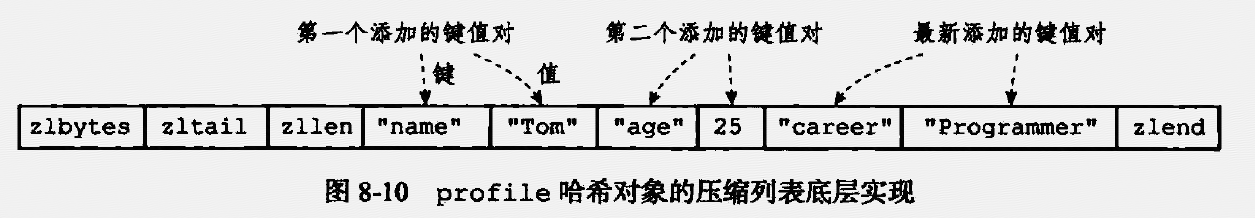
当列表对象保存的所有字符串元素的长度都小于64个字节，并且列表对象保存的元素个数小于512个，那么列表对象就会使用ziplist编码，否则就使用linkedlist编码。

#### 8.4 哈希对象

哈希对象的编码可以是ziplist或者hashtable

Ziplist编码的哈希对象使用压缩列表作为底层实现，每当有新的键值对要加入到哈希对象时，程序会先将保存了键的压缩列表节点推入到压缩列表表尾，然后再将保存了值的压缩列表节点推入到压缩列表表尾。





Hashtable编码的哈希对象使用字段作为底层实现，哈希对象的每个键值对都使用一个字典键值对来保存：

1.字典的每个键都是一个字符串对象，对象中保存了键值对的键

2.字典的每个值都是一个字符串对象，对象中保存了键值对的值



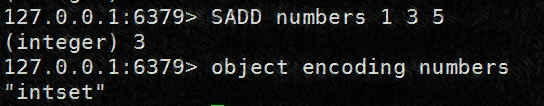
编码转换：

当哈希对象保存的所有键值对的键和值字符串长度都小于64个字节，并且键值对数量小于512个，就是用ziplist编码来保存哈希对象；否则就使用hashtable编码来保存。

#### 8.5 集合对象

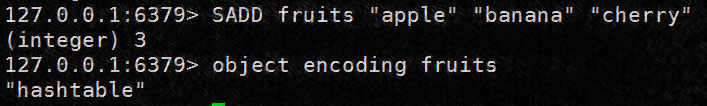
集合对象的编码可以是intset或者hashtable。

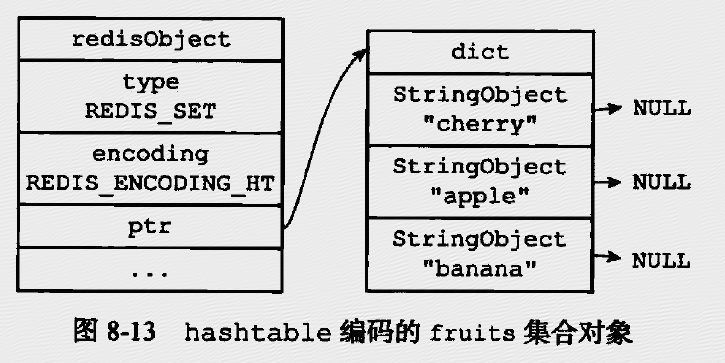
Intset编码的集合对象使用整数集合作为底层实现，集合对象包含的所有元素都被保存在整数集合里面。





Hashtable编码的集合对象使用字典作为底层实现，字典的每个键都是一个字符串对象，每个字符串对象包含了一个集合元素，而字典的值全部被设置为null。





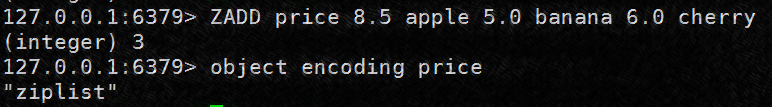
编码转换：

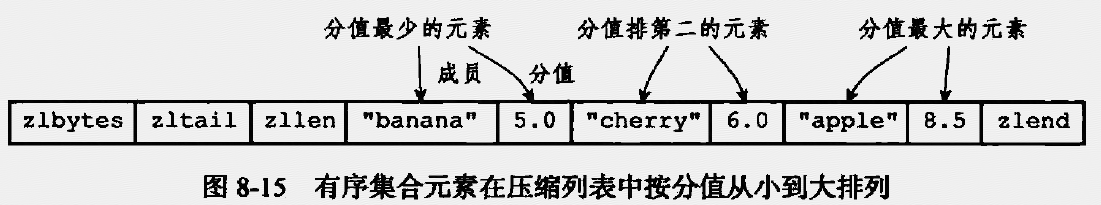
当集合对象保存的所有元素都是整数值，并且集合元素保存的数量不超过512个时，集合对象就是使用的intset编码；否则使用hastable编码。

#### 8.6 有序集合对象

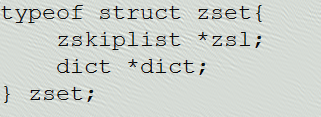
有序集合的编码可以是ziplist或者skiplist

Ziplist编码的有序集合对象使用压缩列表作为底层实现，每个集合氧元素使用两个紧挨在一起的压缩列表节点来保存，第一个节点保存元素的成员(member)，第二个元素保存元素的分值(score)。压缩列表内的集合元素按分值从小到大进行排序，分值较小的放在表头方向，分值较大的放置在靠近表尾方向。





Skiplist编码的有序集合对象使用zset结构作为底层实现，一个zset结构同时包含一个字典和一个跳跃表。



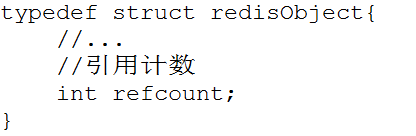
其实有序集合可以单独使用字典或者跳跃表的其中一种数据结构实现，但是两者结合起来性能是最好的。使用跳跃表，利于范围查找（zrank，zrange）；使用字典，方便精确查找（复杂度由跳跃表的O(logN)将为O(1)。

编码转换：

当有序集合保存的元素数量小于128，并且有序集合所有元素的长度都小于64个字节，就使用ziplist编码；否则使用skiplist编码

#### 8.8 内存回收

因为c语言不具备内存回收功能，所以redis在自己的对象系统中构建了一个由引用计数技术实现的内存回收机制。每个对象的引用计数信息由redisobject结构的refcount属性记录：



#### 8.9 对象共享

出了用于实现引用计数内存回收机制之外，对象的引用计数还带有对象共享的作用。Redis中实现多个键共享同一个值对象需要执行两个步骤：

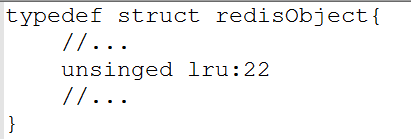
1.）将数据库键的值指针指向一个现有的值对象。

2.）将被共享的值对象的引用计数增加1。

注意，redis只能对包含整数值的字符串对象进行共享。Redis的共享值为0-9999的字符串对象。

#### 8.10 对象的空转时长

除了前面介绍的type，encoding，ptr和refcount四个属性之外，redisobject结构包含的最后一个属性为lru属性，记录了对象最后一次被命令程序访问的时间。

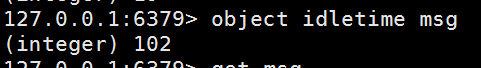




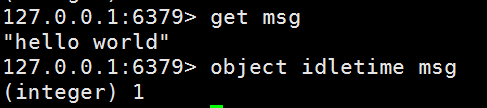
#等待一段时间



#等待很长时间



#获取一次，再查询



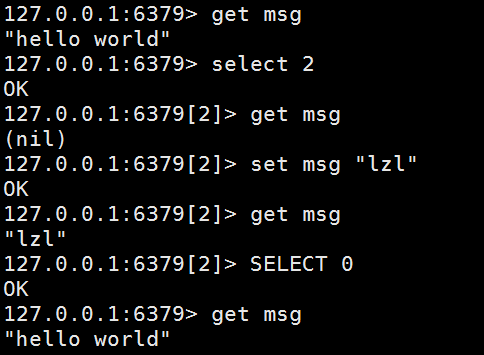
Object idletime 命令只对键有效，对值无效。

## 第二部分：单机数据库的实现

### 第九章：数据库

Redis服务器将所有数据库都保存在redis.h/redisServer结构的db数组中，数组中每一项都是一个redis.h/redisDb结构，每个redisDb代表一个数据库，redis服务器默认创建16个数据库。

Redis默认保存的数据库为0号库，可以手动切换：



Redis是一个键值对数据库服务器。每个键都是一个字符串对象，值可以是字符串对象、列表对象、哈希对象、集合对象或有序集合对象。

RDB模式对于过期键，不会保存。AOF模式，当键过期，会在AOF文件中追加一条DEL

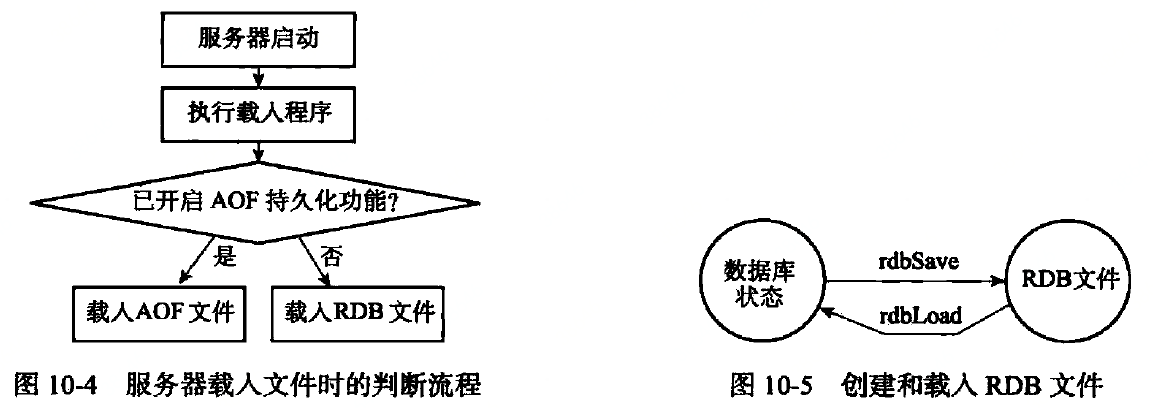
命令。

对于过期的键，如果服务器运行在复制模式下，从服务器的过期键删除动作由主服务器控制：主服务器删除一个过期键之后，会显示的向所有从服务器发送一个del命令。从服务器执行客户端发送的读命令时，不会删除过期键，只有接到主服务器发来的del命令，才会删除过期键。

### ：RDB持久化

Redis是一个键值对服务器。Save和bgsave命令用于生成RDB文件。Save命令会阻塞redis进程直到RDB文件创建完毕，阻塞期间redis不能处理任何命令。Bgsave会派生一个子进程负责创建RDB文件，不会阻塞服务器进程（父进程）。

RDB文件的载入是在服务器启动时自动执行，没有专门的redis命令。注意AOF的优先级比RDB的高。



服务器在载入RDB文件期间，会一直处于阻塞状态。

对于bgsave命令，可以通过save选项设置多个保存条件：例如（下面是redis的默认配置）

Save 900 1 服务器在900秒之内对数据库进行了至少1次修改

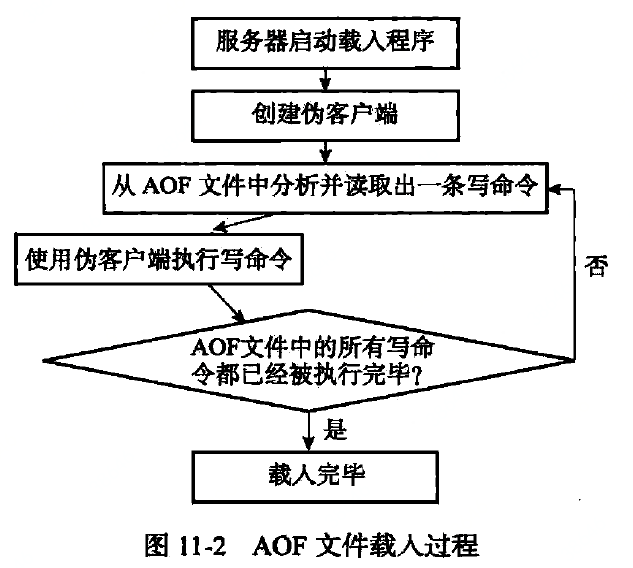
Save 300 10 服务器在300秒之内对数据库进行了至少10次修改

Save 60 10000 服务器在60秒之内对数据库进行了至少10000次修改

以上条件满足其一，就会触发bgsave执行。

### ：AOF持久化

RDB持久化是通过保存数据库中的键值对来记录数据库状态。AOF持久化是通过保存redis执行的写命令来记录数据库状态。下图为AOF文件的载入流程，创建不带网络连接的伪客户端是因为redis命令只能在客户端上下文执行，而载入AOF不需要网络连接，直接来源AOF文件。



对于AOF文件，redis提供了AOF文件重写功能。该功能可以创建一个新的AOF文件代替旧的，而且新的不包含冗余命令。

AOF重写是在一个子进程中，不影响服务器进程（父进程）继续处理命令请求。因为重写期间还有新的命令进来，为防止新旧AOF文件数据不一致，redis服务器维护一个AOF重写缓冲区，记录重写期间的新命令，当子进程完成新AOF文件之后，将重写缓冲区的内容追加到新AOF文件末尾使新旧AOF数据库状态一致。

（可以参考基础知识整理9的180知识点）

### ：事件

Redis服务器是一个事件驱动程序，服务器需要处理以下两类事件

#### 12.1文件事件

Redis服务器通过套接字与客户端进行连接，而文件事件就是服务器对套接字操作的抽象。服务器与客户端的通信会产生相应的文件事件，服务器通过监听和处理这些事件来完成网络通信操作。

Redis基于reactor模式开发了网络事件处理器，即文件事件处理器。虽然它是单线程方式运行（redis本身是单线程的），但是通过IO多路复用程序监听多个套接字实现了高性能的网络通信模型。

#### 12.2时间事件

Redis服务器中的一些操作需要在给定的时间点执行，而时间事件就是服务器对这类时间定时操作的抽象。主要分为定时事件和周期性事件。

### ：客户端

客户端创建：

使用connect函数

关闭客户端的情况：

1. 客户端进程退出或被杀死
2. 客户端向服务器发送了带有不符合协议格式的命令请求（没验证成功）
3. 客户端成为client kill命令的目标
4. 如果服务器设置了timeout配置，客户端超时会被关闭
5. 客户端发送的命令请求大小超过输入缓冲区限制
6. 要发送给客户端的命令回复的大小超过了输出缓冲区限制

### ：服务器

Redis服务器是典型的一对多服务器：一个服务器可以与多个客户端建立网络连接，通过IO多路复用器技术实现的文件事件处理器，使得redis服务器使用单线程单进程处理命令并与多个客户端进行网络通信。

服务器从启动到能够处理客户端的命令请求需要执行以下步骤

1. 初始化服务器状态
2. 载入服务器配置
3. 初始化服务器数据结构
4. 还原数据库状态
5. 执行事件循环

## 第三部分：多机数据库的实现