Redis数据结构底层

Redis用C语言来实现的

String 字符串

C语言表示字符串是用char类型的数组来表示。
redis并没有直接使用C语言的方式表示字符串。\0

redis使用SDS（简单动态字符串）表示字符串。

特点：

1. **二进制安全的数据结构。**
   C语言的字符串不安全，只能保存文本数据，SDS可以保存文本和二进制数据。能保准数据不丢失。
2. **内存预分配，避免频繁分配内存**
   SDS结构：
   - length 8
   - free 剩余空间
   - char buff[] = "liujiang";

   当String修改的时候，比如liujiang改为liujiang123，会引发数组扩容。
   原始长度8，修改后11，增加了addlength=3;
   扩容的时候新数组长度为 (length + addlength) * 2 = 22
3. **能与C语言的函数库兼容**

数组存储的时候会在最末尾隐式的添加一个\0，兼容C语言（因为C语言中字符读取的时候是以 \0 结尾的，\0后面的字符会被丢弃）。

Map 字典表

K-V数据结构。数组+链表

底层通过数组存储，hash之后可能产生hash冲突，通过链表存储相同key，当table.length = size的时候扩容，1:1的时候扩容。新的数组长度为原数组长度的2倍。

转移元素的时候，通过**单线程渐进式转移**，访问到的时候转移，这里的单线程是指只有一个线程在扩容，而在扩容的同时其他的线程可以并发的进行读写。

> ht[0]，是存放数据的table，作为非扩容时容器。
> ht[1]，只有正在进行扩容时才会使用，它也是存放数据的table，**长度为ht[0]的两倍。**
> 扩容时，单线程A负责把数据从ht[0] copy到ht[1] 中。如果这时有其他线程
> 进行读操作：**会先去ht[0]中找，找不到再去ht[1]中找。**
> 进行写操作：**直接写在ht[1]中。**
> 进行删除操作：与读类似

List 列表

在版本3.2之前，Redis 列表list使用两种数据结构作为底层实现：

- 压缩列表ziplist
- 双向链表linkedlist

因为双向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，并且在有需要的时候， 才从压缩列表实现转换到双向链表实现。

压缩列表转化成双向链表条件

创建新列表时 redis 默认使用 redis_encoding_ziplist 编码， 当以下任意一个条件被满足时， 列表会被转换成 redis_encoding_linkedlist 编码：

- 试图往列表新添加一个字符串值，且这个字符串的长度超过 server.list_max_ziplist_value （默认值为 64 ）。
- ziplist 包含的节点超过 server.list_max_ziplist_entries （默认值为 512 ）。

注意：这两个条件是可以修改的，在 redis.conf 中：

```
Plain Text | 复制代码
1  list-max-ziplist-value 64
2  list-max-ziplist-entries 512
```

双向链表linkedlist

当链表entry数据超过512、或单个value 长度超过64，底层就会转化成linkedlist编码；

linkedlist是标准的双向链表，Node节点包含prev和next指针，可以进行双向遍历；

还保存了 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 (1) —— 这是高效实现 LPUSH 、 RPOP、 RPOPLPUSH 等命令的关键。

linkedlist比较简单，我们重点来分析ziplist。

压缩列表ziplist

压缩列表 ziplist 是为 Redis 节约内存而开发的。

ziplist 是由一系列特殊编码的内存块构成的列表(像内存连续的数组，但每个元素长度不同)， 一个 ziplist 可以包含多个节点（entry）。

ziplist 将表中每一项存放在前后连续的地址空间内，每一项因占用的空间不同，而采用变长编码。

当元素个数较少时，Redis 用 ziplist 来存储数据，当元素个数超过某个值时，链表键中会把 ziplist 转化为 linkedlist，字典键中会把 ziplist 转化为 hashtable。

由于内存是连续分配的，所以遍历速度很快。

> **在3.2之后，ziplist被quicklist替代。但是仍然是zset底层实现之一。**

ziplist 是一个特殊的双向链表

特殊之处在于：没有维护双向指针:prev next；而是存储上一个 entry的长度和 当前entry的长度，通过长度推算下一个元素在什么地方。

牺牲读取的性能，获得高效的存储空间，因为(简短字符串的情况)存储指针比存储entry长度 更费内存。这是典型的“时间换空间”。

ziplist使用局限性

字段、值比较小，才会用ziplist。

快速列表(quicklist)

一个由压缩列表ziplist组成的双向链表。但是一个快速列表quicklist可以有多个quicklist节点，它很像B树的存储方式。是在redis3.2版本中新加的数据结构，用在列表的底层实现。

> Redis中的列表list，在版本3.2之前，列表底层的编码是ziplist和linkedlist实现的，但是在版本3.2之后，重新引入 quicklist，列表的底层都由quicklist实现。

在版本3.2之前，当列表对象中元素的长度比较小或者数量比较少的时候，采用ziplist来存储，当列表对象中元素的长度比较大或者数量比较多的时候，则会转而使用双向列表linkedlist来存储。

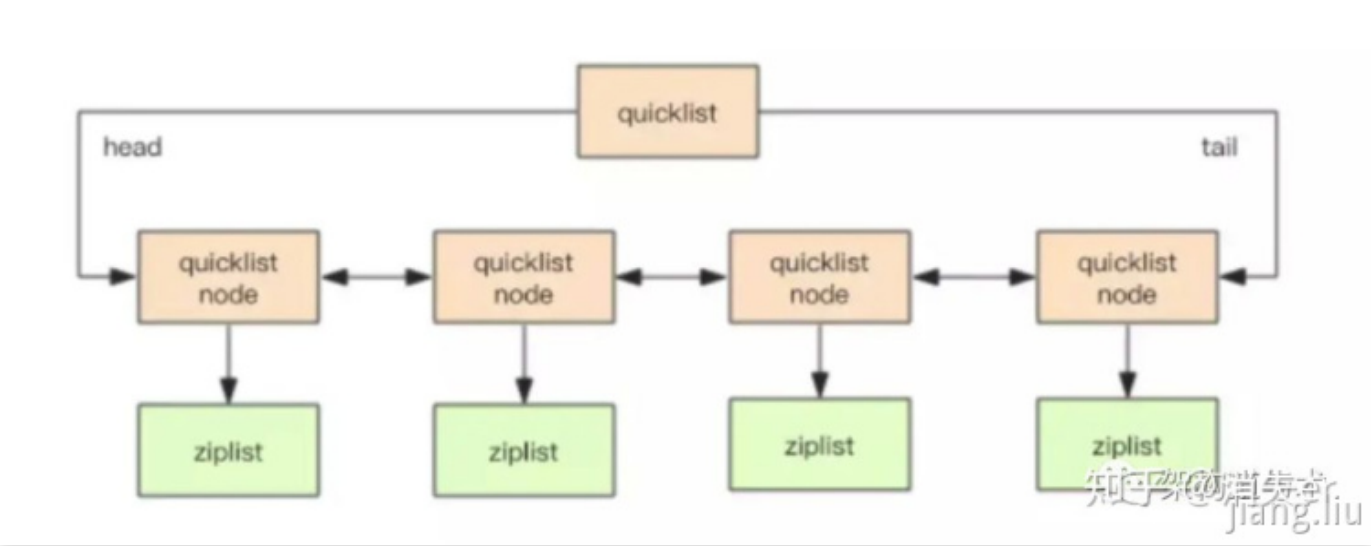
这两种存储方式的优缺点

- 双向链表linkedlist便于在表的两端进行push和pop操作，在插入节点上复杂度很低，但是它的内存开销比较大。首先，它在每个节点上除了要保存数据之外，还要额外保存两个指针；其次，双向链表的各个节点是单独的内存块，地址不连续，节点多了容易产生内存碎片。
- ziplist存储在一段连续的内存上，所以存储效率很高。但是，它不利于修改操作，插入和删除操作需要频繁的申请和释放内存。特别是当ziplist长度很长的时候，一次realloc可能会导致大批量的数据拷贝。

quickList

可以认为quickList，是ziplist和linkedlist二者的结合；quickList将二者的优点结合起来。

quickList是一个ziplist组成的双向链表。每个节点使用ziplist来保存数据。
本质上来说，quicklist里面保存着一个一个小的ziplist。结构如下：



quickList就是一个标准的双向链表的配置，有head 有tail；每一个节点是一个quicklistNode，包含prev和next指针。每一个quicklistNode 包含 一个ziplist，*zp 压缩链表里存储键值。所以quicklist是对ziplist进行一次封装，使用小块的ziplist来既保证了少使用内存，也保证了性能。

Set 集合

redis的集合对象set的底层存储结构特别神奇，我估计一般人想象不到，底层使用了 **intset** 和 **hashtable** 两种数据结构存储的，intset我们可以理解为数组，hashtable就是普通的哈希表（key为set的值，value为null）。

整数集合(intset)

Reids对整数存储专门作了优化，intset就是redis用于保存整数值的集合数据结构。当一个结合中只包含整数元素，redis就会用这个来存储。

intset数据结构：

Java 复制代码

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

根据contents字段来判断用哪个int类型更好，也就是对int存储作了优化。如果我们有个Int16类型的整数集合，现在要将65535(int32)加进这个集合，int16是存储不下的，所以就要对整数集合进行升级。并且不支持降级。

encoding升级的好处是什么呢?

1. 提高了整数集合的灵活性。
2. 尽可能节约内存(能用小的就不用大的)。