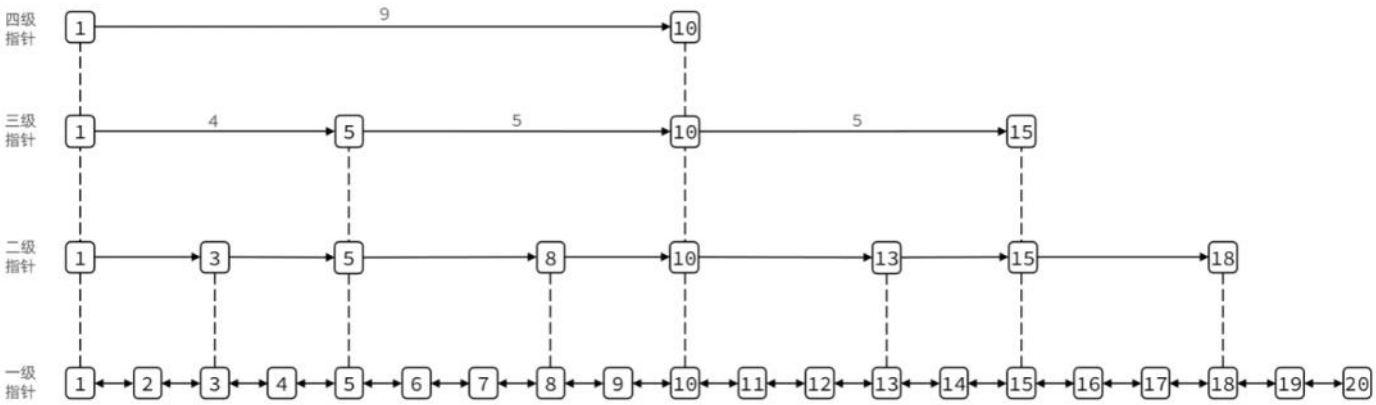
跳表

2024年5月18日 17:14

SkipList

SkipList（跳表）首先是链表，但与传统链表相比有几点差异：

- ◆ 元素按照升序排列存储
- ◆ 节点可能包含多个指针，指针跨度不同。



SkipList

SkipList（跳表）首先是链表，但与传统链表相比有几点差异：

- ◆ 元素按照升序排列存储
- ◆ 节点可能包含多个指针，指针跨度不同。

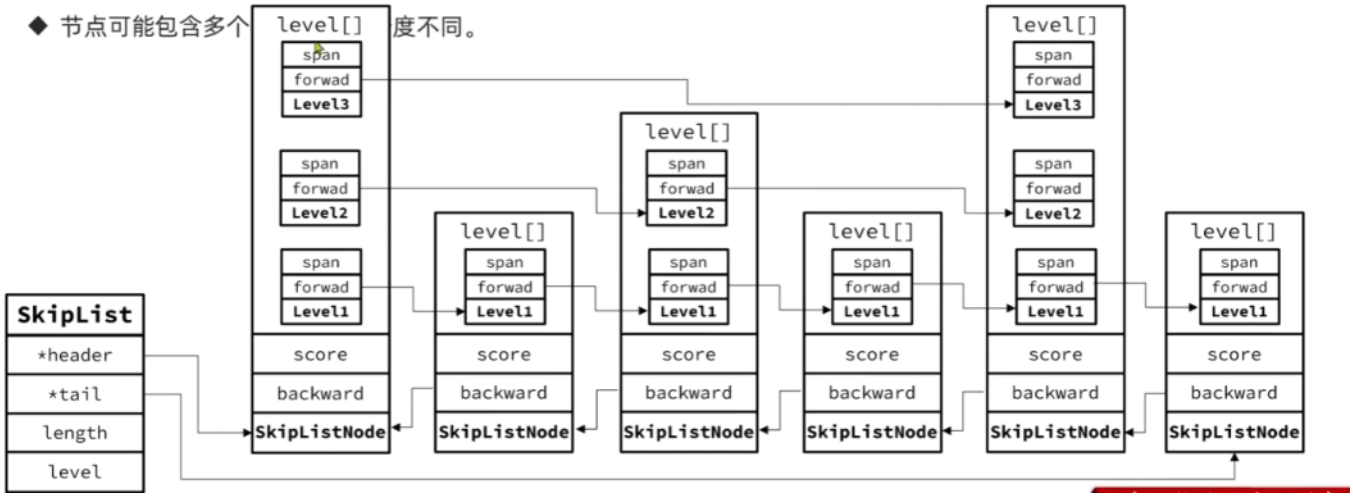
```c
// t_zset.c
typedef struct zskiplist {
    // 头尾节点指针
    struct zskiplistNode *header, *tail;
    // 节点数量
    unsigned long length;
    // 最大的索引层级，默认是1
    int level;
} zskiplist;
```

```c
// t_zset.c
typedef struct zskiplistNode {
    sds ele; // 节点存储的值
    double score;// 节点分数，排序、查找用
    struct zskiplistNode *backward; // 前一个节点指针
    struct zskiplistLevel {
        struct zskiplistNode *forward; // 下一个节点指针
        unsigned long span; // 索引跨度
    } level[]; // 多级索引数组
} zskiplistNode;
```

SkipList

SkipList（跳表）首先是链表，但与传统链表相比有几点差异：

- ◆ 元素按照升序排列存储
- ◆ 节点可能包含多个 level[] 度不同。



SkipList的特点：

- 跳跃表是一个双向链表，每个节点都包含score和ele值
- 节点按照score值排序，score值一样则按照ele字典排序
- 每个节点都可以包含多层指针，层数是1到32之间的随机数
- 不同层指针到下一个节点的跨度不同，层级越高，跨度越大
- 增删改查效率与红黑树基本一致，实现却更简单

ZSet

ZSet也就是SortedSet，其中每一个元素都需要指定一个score值和member值：

- 可以根据score值排序后
- member必须唯一
- 可以根据member查询分数

```
127.0.0.1:6379> ZADD z1 10 m1 20 m2 30 m3
(integer) 3
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379> ZSCORE z1 m1
"10"
```

score

member

根据member查询score

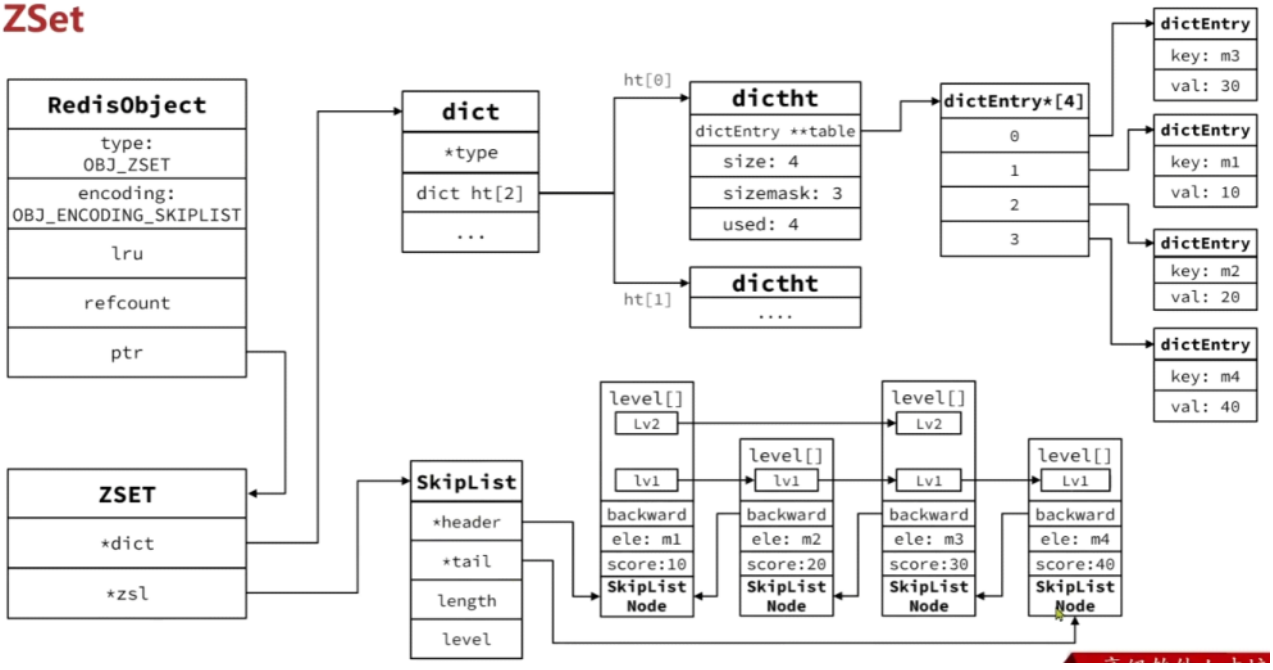
因此，zset底层数据结构必须满足**键值存储、键必须唯一、可排序**这几个需求。之前学习的哪种编码结构可以满足？

- **SkipList**：可以排序，并且可以同时存储score和ele值（member）
- **HT（Dict）**：可以键值存储，并且可以根据key找value

```
// zset结构
typedef struct zset {
    // Dict指针
    dict *dict;
    // SkipList指针
    zskiplist *zsl;
} zset;
```

```
robj *createZsetObject(void) {
    zset *zs = zmalloc(sizeof(*zs));
    robj *o;
    // 创建Dict
    zs->dict = dictCreate(&zsetDictType,NULL);
    // 创建SkipList
    zs->zsl = zslCreate();
    o = createObject(OBJ_ZSET,zs);
    o->encoding = OBJ_ENCODING_SKIPLIST;
    return o;
}
```

ZSet



ZSet

当元素数量不多时，HT和SkipList的优势不明显，而且更耗内存。因此zset还会采用ZipList结构来节省内存，不过需要同时满足两个条件：

① 元素数量小于zset_max_ziplist_entries，默认值128

② 每个元素都小于zset_max_ziplist_value字节，默认值64

```c
// zadd添加元素时，先根据key找到zset，不存在则创建新的zset
zobj = lookupKeyWrite(c->db,key);
if (checkType(c,zobj,OBJ_ZSET)) goto cleanup;
// 判断是否存在
if (zobj == NULL) { // zset不存在
    if (server.zset_max_ziplist_entries == 0 ||
        server.zset_max_ziplist_value < sdslen(c->argv[scoreidx+1]->ptr))
    { // zset_max_ziplist_entries设置为0就是禁用了ZipList，
        // 或者value大小超过了zset_max_ziplist_value，采用HT + SkipList
        zobj = createZsetObject();
    } else { // 否则，采用 ZipList
        zobj = createZsetZiplistObject();
    }
    dbAdd(c->db,key,zobj);
}
// ....
zsetAdd(zobj, score, ele, flags, &retflags, &newscore);
```

```c
robj *createZsetObject(void) {
    // 申请内存
    zset *zs = zmalloc(sizeof(*zs));
    robj *o;
    // 创建Dict
    zs->dict = dictCreate(&zsetDictType,NULL);
    // 创建SkipList
    zs->zsl = zslCreate();
    o = createObject(OBJ_ZSET,zs);
    o->encoding = OBJ_ENCODING_SKIPLIST;
    return o;
}
```

```c
robj *createZsetZiplistObject(void) {
    // 创建ZipList
    unsigned char *zl = ziplistNew();
    robj *o = createObject(OBJ_ZSET,zl);
    o->encoding = OBJ_ENCODING_ZIPLIST;
    return o;
}
```


ZSet

当元素数量不多时，HT和SkipList的优势不明显，而且更耗内存。因此zset还会采用ZipList结构来节省内存，不过需要同时满足两个条件：

① 元素数量小于zset_max_ziplist_entries，默认值128

② 每个元素都小于zset_max_ziplist_value字节，默认值64

ziplist本身没有排序功能，而且没有键值对的概念，因此需要有zset通过编码实现：

- ZipList是连续内存，因此score和element是紧挨在一起的两个entry， element在前，score在后
- score越小越接近队首，score越大越接近队尾，按照score值升序排列