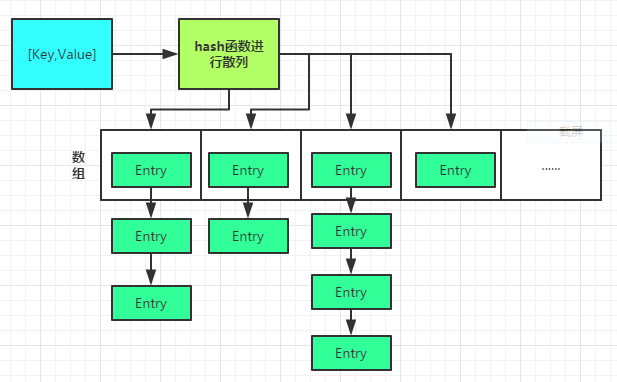
# HashMap底层实现

HashMap底层数据结构如下图，HashMap由“hash函数+数组+单链表”3个要素构成



通过写一个迷你版的HashMap来深刻理解

MyMap接口，定义一个接口，对外暴露快速存取的方法，并定义了一个内部接口Entry。

**public** **interface** MyMap<K,V> {

**public** V put(K k, V v);

**public** V get(K k);

**public** **interface** Entry<K, V>{

**public** K getKey();

**public** V getValue();

}

}

MyHashMap接口实现

**public** **class** MyHashMap<K, V> **implements** MyMap<K, V> {

/\* HashMap的要素之一就是数组，自然在这里，我们要定义数组，数组的初始化大小，还要考虑扩容的阀值\*/

**private** **static** **int** *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4;

**private** **static** **final** **float** ***DEFAULT\_LOAD\_FACTOR*** = 0.75f;

**private** **int** defaultInitSize = 1 << 2;

**private** **double** defaultLoadFactor;

**private** Entry<K, V>[] table;

**private** **int** entryUserSize;

/\*MyHashMap构造函数使用到了“门面模式”。这里的2个构造方法其实指向的是同一个，但是对外却暴露了2个“门面”！\*/

**public** MyHashMap() {

**this**(*DEFAULT\_INITIAL\_CAPACITY*, ***DEFAULT\_LOAD\_FACTOR***);

}

**public** MyHashMap(**int** defaultInitsize, **double** defaultLoadFactor) {

**this**.defaultInitSize = defaultInitsize;

**this**.defaultLoadFactor = defaultLoadFactor;

table = **new** Entry[**this**.defaultInitSize];

}

/\*Put函数，根据Key计算出在Entry[]中的位置index，如果Entry[index]中的元素为null，那么可以放入其中，如果不为空，那么得遍历单链表，key相同时更新value，不相同时形成一个新的Entry“挤压”单链表。

put函数要考虑扩容和冲突问题，对于扩容：HashMap中的Entry的数量（数组以及单链表中的所有Entry）达到阀值，意味着新生成一个Entry数组并重新散列。关于冲突：利用位运算，让hash函数得到的数据散列开来，从而减低了碰撞的概率，当发生冲突时采用单链表解决冲突。\*/

@Override

**public** V put(K k, V v) {

V oldValue = **null**;

**if** (entryUserSize > defaultInitSize \* defaultLoadFactor) {

reSize();

}

**int** index = hash(k) & (defaultInitSize - 1);

**if** (table[index] == **null**) {

table[index] = **new** Entry<K, V>(k, v, **null**);

entryUserSize++;

} **else** {

Entry<K, V> entry = table[index];

Entry<K, V> e = entry;

**while** (e != **null**) {

**if** (k == e.getKey() || k.equals(e.getKey())) {

oldValue = e.getValue();

e.value = v;

**break**;

}

e = e.next;

}

;

table[index] = **new** Entry<K, V>(k, v, entry);

entryUserSize++;

}

**return** oldValue;

}

/\*reSize函数,可以看出，对于HashMap而言，如果频繁进行resize操作，是会影响性能的，这个过程中数组变大，原来数组中的entry元素一个个的put到新数组的过程，需要注意的是一些状态变量的改变。\*/

**private** **void** reSize() {

List<Entry<K, V>> entryList = **new** ArrayList<Entry<K, V>>();

**for** (Entry<K, V> entry : table) {

**while** (entry != **null**) {

entryList.add(entry);

entry = entry.next;

}

}

**if**(entryList.size() > 0){

entryUserSize = 0;

defaultInitSize \*= 2;

table = **new** Entry[defaultInitSize];

**for**(Entry<K,V> entry : entryList){

put(entry.getKey(),entry.getValue());

}

}

}

/\*Has函数,要想散列均匀，就得对hash值进行二进制的位运算！\*/

**private** **int** hash(K k) {

**int** h = k.hashCode();

h ^= (h >>> 20) ^ (h >>> 12);

**return** h ^ (h >>> 7) ^ (h >>> 4);

}

/\*get函数，需要注意在遍历单链表过程中使用==或者equals来判断下即可。\*/

@Override

**public** V get(K k) {

**int** index = hash(k) & (defaultInitSize - 1);

Entry<K, V> entry = table[index];

**if** (entry == **null**) {

**return** **null**;

}

**do** {

**if** (k == entry.getKey() || k.equals(entry.getKey())) {

**return** entry.getValue();

}

entry = entry.next;

} **while** (entry != **null**);

**return** **null**;

}

/\*Entry接口实现\*/

**public** **class** Entry<K, V> **implements** MyMap.Entry<K, V> {

**private** K key;

**private** V value;

**private** Entry<K, V> next;

**public** Entry() {

}

**public** Entry(K key, V value, Entry<K, V> next) {

**this**.key = key;

**this**.value = value;

**this**.next = next;

}

@Override

**public** K getKey() {

**return** key;

}

@Override

**public** V getValue() {

**return** value;

}

}

}

单元测试

**public** **class** MyHashMapTest {

@Test

**public** **void** test() {

MyMap<String, String> map = **new** MyHashMap<String, String>();

**for**(**int** i =0; i < 30; i++)

map.put("key" + i, "value" + i);

**for**(**int** i =0; i < 30; i++){

System.***out***.println("key" + i + ":" + map.get("key" + i));

}

}

}

测试结果

**key0:value0 key1:value1 key2:value2 key3:value3**

**key4:value4 key5:value5 key6:value6 key7:value7**

**key8:value8 key9:value9 key10:value10 key11:value11**

**key12:value12 key13:value13 key14:value14 key15:value15**

**key16:value16 key17:value17 key18:value18 key19:value19**

**key20:value20 key21:value21 key22:value22 key23:value23**

**key24:value24 key25:value25 key26:value26 key27:value27**

**key28:value28 key29:value29**