<http://blog.csdn.net/teaandnoodle/article/details/52270286>

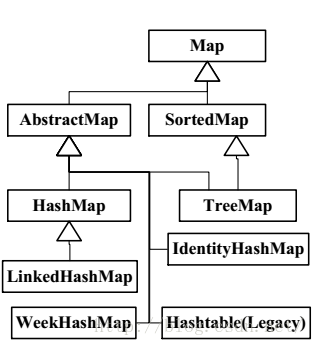
版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/teaandnoodle/article/details/52270286)

标准库中包含了几种Map的基本实现，包括：HashMap、TreeMap、LinkedHashMap、WeekHashMap、ConcurrentHashMap、IdentityHashMap。它们都有同样的基本接口Map，但是行为特性各不相同，这表现在效率，键值对的保存及呈现次序、对象的保存周期、映射表如何在多线程程序中工作和判定“键”等价的策略等方面。

Map可以将键映射到值。一个映射不能包含重复的键；每个键最多只能映射到一个值。Map 接口提供三种collection 视图，允许以键集、值集或键-值映射关系集的形式查看某个映射的内容。映射顺序定义为迭代器在映射的Collection视图上返回其元素的顺序。某些映射实现可明确保证其顺序，如TreeMap类；另一些映射实现则不保证顺序，如HashMap类。

这几种Map中HashMap是查询效率最高的Map，LinkedHashMap只比HashMap慢一点儿，但是它可以更快的遍历关键字，TreeMap中的关键字都是排序过的，所以可以按序输出。



|  |  |
| --- | --- |
| **HashMap\*** | Map基于散列表的实现（它取代了Hashtable）。插入和查询“键值对”的开销是固定的。可以通过构造器设置容量和负载因子，以调整容器的性能。 |
| **LinkedHashMap** | 类似于HashMap，但是迭代遍历它时，取得“键值对”的顺序是其插入次序，或者是最近最少使用（LRU）的次序。只比HashMap慢一点；而在迭代访问时反而更快，因为它使用链表维护内部次序。 |
| **TreeMap** | 基于红黑树的实现。查看“键”或“键值对”时，它们会被排序（次序由Comparable或Comparator决定）。TreeMap的特点在于，所得到的结果是经过排序的，TreeMap是唯一带有subMap方法的Map，它可以返回一个子树。 |
| **WeekHashMap** | 弱键（week key）映射，允许释放映射所指向的对象；这是为解决某些类特殊问题而设计的。如果映射之外没有引用指向某个“键”，则此“键”可以被垃圾收集器回收。 |
| **ConcurrentHashMap** | 一种线程安全的Map，它不涉及同步加锁。 |
| **IdentityHashMap** | 使用==代替equals对“键”进行比较的散列映射，专为解决特殊问题而设计。 |

### Map接口中的（部分主要）方法

containsKey(Object key)：如果此映射包含指定键的映射关系，则返回 true；  
containsValue(Object value)：如果此映射将一个或多个键映射到指定值，则返回 true；  
entrySet()：返回此映射中包含的映射关系的 Set 视图；  
get(Object key)：返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null；  
keySet()：返回此映射中包含的键的 Set 视图；  
put(K key, V value)：将指定的值与此映射中的指定键关联（可选操作）。

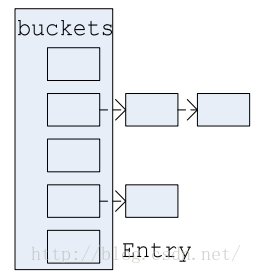
AbstractMap提供 Map 接口的骨干实现，以最大限度地减少实现Map接口所需的工作。要实现不可修改的映射，编程人员只需扩展此类并提供 entrySet 方法的实现即可，该方法将返回映射的映射关系Set视图。通常，返回的 set 将依次在AbstractSet上实现。此 set 不支持add或remove方法，其迭代器也不支持 remove 方法。要实现可修改的映射，编程人员必须另外重写此类的 put 方法（否则将抛出 UnsupportedOperationException），entrySet().iterator() 返回的迭代器也必须另外实现其remove方法。

### HashMap及其实现方式

HashMap是基于哈希表实现的，它实现了Map接口，同时允许使用null为key和null为value（除了非同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同）。HashMap不保证映射的顺序，特别是它不保证该顺序恒久不变。

假定哈希函数将元素适当地分布在各桶之间，可为基本操作（get 和 put）提供稳定的性能。迭代collection视图所需的时间与HashMap实例的“容量”（桶的数量）及其大小（键-值映射关系数）成比例。所以，如果迭代性能很重要，则不要将初始容量设置得太高（或将加载因子设置得太低）。

HashMap 的实例有两个参数影响其性能：初始容量和加载因子。容量是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部[**数据结构**](http://lib.csdn.net/base/datastructure)），从而哈希表将具有大约两倍的桶数。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少 rehash 操作次数。



**HashMap的数据结构**

在HashMap中比较重要的两个参数时初始容量和加载因子：

public HashMap(int initialCapacity, float loadFactor);

在构造完成之后，loadFactor会被记录下来，initialCapacity会变成2的最小次方数，并与loadFactor相乘得到threshold：

// Find a power of 2 >= initialCapacity  
int capacity = 1;  
while (capacity < initialCapacity)  
    capacity <<= 1;  
this.loadFactor = loadFactor;  
threshold = (int)Math.min(capacity \* loadFactor, MAXIMUM\_CAPACITY + 1);  
table = new Entry[capacity];

这样HashMap的所有元素都被放进了Entry构成的数组中，Entry相当于Hash表中的“桶”（这里只称HashMap内的Entry数组为桶，而不包含链表中的Entry），它的内部包含着key，value以及指向下一个和前一个Entry的指针。

这个“桶”就是HashMap中的表现为table数组：

transient Entry<K,V>[] table;

HashMap的get方法实际上就是为了找到某个Entry，这个Entry的key和给定对象相等：

**[java]** [view plain](http://blog.csdn.net/teaandnoodle/article/details/52270286) [copy](http://blog.csdn.net/teaandnoodle/article/details/52270286)

1. **public** V get(Object key) {
2. **if** (key == **null**)
3. **return** getForNullKey();
4. Entry<K,V> entry = getEntry(key);
6. **return** **null** == entry ? **null** : entry.getValue();
7. }
8. **final** Entry<K,V> getEntry(Object key) {//根据key查找Entry
9. **int** hash = (key == **null**) ? 0 : hash(key);//找到hash值，确定桶的位置
10. **for** (Entry<K,V> e = table[indexFor(hash, table.length)];//找到桶中第一个元素，判断是否在桶中
11. e != **null**;
12. e = e.next) {
13. Object k;
14. **if** (e.hash == hash &&//判断桶中的元素是否是要添加的元素
15. ((k = e.key) == key || (key != **null** && key.equals(k))))
16. **return** e;
17. }
18. **return** **null**;
19. }

Put方法就是要把数据放入特定的Entry中：

**[java]** [view plain](http://blog.csdn.net/teaandnoodle/article/details/52270286) [copy](http://blog.csdn.net/teaandnoodle/article/details/52270286)

1. **public** V put(K key, V value) {
2. **if** (key == **null**)
3. **return** putForNullKey(value);
4. **int** hash = hash(key);
5. **int** i = indexFor(hash, table.length);//计算位置
6. **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {//判断当前桶中是否已经包含该元素（判断key是否已经存在）
7. Object k;
8. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
9. V oldValue = e.value;
10. e.value = value;
11. e.recordAccess(**this**);
12. **return** oldValue;
13. }
14. }
16. modCount++;
17. addEntry(hash, key, value, i);
18. **return** **null**;
19. }
20. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
21. **if** ((size >= threshold) && (**null** != table[bucketIndex])) {//防止空间不足进行rehash
22. resize(2 \* table.length);
23. hash = (**null** != key) ? hash(key) : 0;
24. bucketIndex = indexFor(hash, table.length);
25. }
26. createEntry(hash, key, value, bucketIndex);//插入节点
27. }
28. **void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {
29. Entry<K,V> e = table[bucketIndex];//将当前bucketIndex元素存起来
30. table[bucketIndex] = **new** Entry<>(hash, key, value, e);//设置当前bucketIndex元素，并把原来的元素e，设置为新元素的后继（链表的头插法，新元素插入头部）
31. size++;
32. }

put时在使用indexFor找到下标后，需要注意当前桶中是否已经存在给定key对应的元素，这时需要遍历桶中的所有元素，然后在确定没有给定key对应的元素时，就可以将当前给定的元素插入这个桶的第一个位置（其他元素后移）。

做put操作时，可能会出现桶的空间不足（也就是size比threshold要大了，此时冲突的可能性会很大），就需要rehash一次，将空间变为当前空间的两倍（即，resize(2 \* table.length)），然后将所有的桶移入新的桶中：

**[java]** [view plain](http://blog.csdn.net/teaandnoodle/article/details/52270286) [copy](http://blog.csdn.net/teaandnoodle/article/details/52270286)

1. **void** resize(**int** newCapacity) {
2. Entry[] oldTable = table;
3. **int** oldCapacity = oldTable.length;
4. **if** (oldCapacity == MAXIMUM\_CAPACITY) {
5. threshold = Integer.MAX\_VALUE;
6. **return**;
7. }
9. Entry[] newTable = **new** Entry[newCapacity];
10. **boolean** oldAltHashing = useAltHashing;
11. useAltHashing |= sun.misc.VM.isBooted() &&
12. (newCapacity >= Holder.ALTERNATIVE\_HASHING\_THRESHOLD);
13. **boolean** rehash = oldAltHashing ^ useAltHashing;
14. transfer(newTable, rehash);//这里进行了转移操作
15. table = newTable;
16. threshold = (**int**)Math.min(newCapacity \* loadFactor, MAXIMUM\_CAPACITY + 1);
17. }
18. **void** transfer(Entry[] newTable, **boolean** rehash) {
19. **int** newCapacity = newTable.length;
20. **for** (Entry<K,V> e : table) {//逐个移动桶中元素
21. **while**(**null** != e) {
22. Entry<K,V> next = e.next;
23. **if** (rehash) {
24. e.hash = **null** == e.key ? 0 : hash(e.key);
25. }
26. **int** i = indexFor(e.hash, newCapacity);
27. e.next = newTable[i];//依然采用头插法，会导致桶中元素逆序
28. newTable[i] = e;
29. e = next;
30. }
31. }
32. }

删除操作也是如此，找到对应的桶，然后遍历桶中元素，并在找到元素后删除它：

**[java]** [view plain](http://blog.csdn.net/teaandnoodle/article/details/52270286) [copy](http://blog.csdn.net/teaandnoodle/article/details/52270286)

1. **public** V remove(Object key) {
2. Entry<K,V> e = removeEntryForKey(key);
3. **return** (e == **null** ? **null** : e.value);
4. }
5. **final** Entry<K,V> removeEntryForKey(Object key) {
6. **int** hash = (key == **null**) ? 0 : hash(key);
7. **int** i = indexFor(hash, table.length);
8. Entry<K,V> prev = table[i];
9. Entry<K,V> e = prev;
11. **while** (e != **null**) {//遍历桶中元素
12. Entry<K,V> next = e.next;
13. Object k;
14. **if** (e.hash == hash &&//检查到目标元素
15. ((k = e.key) == key || (key != **null** && key.equals(k)))) {
16. modCount++;
17. size--;
18. **if** (prev == e)//第一个元素就是目标元素
19. table[i] = next;
20. **else**//第一个元素不是目标元素
21. prev.next = next;
22. e.recordRemoval(**this**);
23. **return** e;
24. }
25. prev = e;
26. e = next;
27. }
29. **return** e;
30. }

### TreeMap及其实现方式（TreeMap的实现要看红黑树的实现方式）

TreeMap是基于红黑树（Red-Black tree，具体请参照博客：[Red Black Tree](http://blog.csdn.net/teaandnoodle/article/details/52462902)）的 NavigableMap 实现。该映射根据其键的自然顺序进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。它能够保证containsKey、get、put 和 remove 操作的时间开销为 log(n)。这里的红黑树[**算法**](http://lib.csdn.net/base/datastructure)是依据算法导论中的红黑树算法实现的。

在TreeMap中保存着红黑树的树根：

private transient Entry<K,V> root = null;

当要使用put方法插入数据时，会依据红黑树的插入算法，将数据插入特定的位置，由于红黑树本身是二叉排序树，因此可以按照结点的大小找到目标位置，并插入当前位置，然后再维护红黑树的结构，使得它的结构符合红黑树的约束。

使用get方法获取数据时，会按照二叉排序树的规则比较从根到叶节点的元素，直到发现或找不到目标元素为止：

**[java]** [view plain](http://blog.csdn.net/teaandnoodle/article/details/52270286) [copy](http://blog.csdn.net/teaandnoodle/article/details/52270286)

1. **public** V get(Object key) {
2. Entry<K,V> p = getEntry(key);
3. **return** (p==**null** ? **null** : p.value);
4. }
5. **final** Entry<K,V> getEntry(Object key) {
6. // Offload comparator-based version for sake of performance
7. **if** (comparator != **null**)
8. **return** getEntryUsingComparator(key);
9. **if** (key == **null**)
10. **throw** **new** NullPointerException();
11. Comparable<? **super** K> k = (Comparable<? **super** K>) key;
12. Entry<K,V> p = root;
13. **while** (p != **null**) {
14. **int** cmp = k.compareTo(p.key);
15. **if** (cmp < 0)
16. p = p.left;
17. **else** **if** (cmp > 0)
18. p = p.right;
19. **else**
20. **return** p;
21. }
22. **return** **null**;
23. }

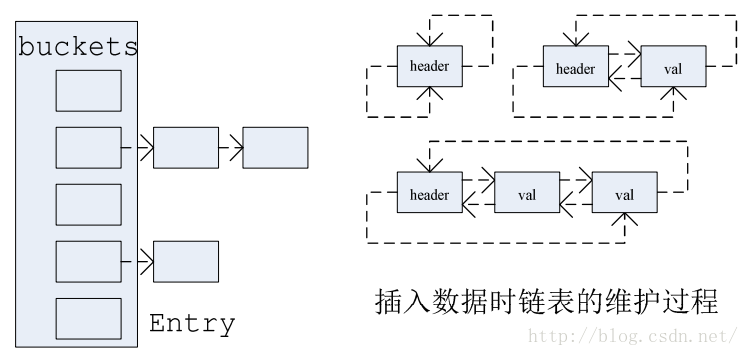
containsKey(Object)方法和get方法是一致的，也是使用getEntry来查找目标元素。

remove(Object)方法是利用查找到红黑树中的结点，然后删除该结点，并维护红黑树的特征来实现的。

对于TreeMap，重要的是遍历的方法，其遍历的方法是由EntryIterator来实现的，它继承自PrivateEntryIterator。在PrivateEntryIterator中可以发现，它使用了红黑树中求当前结点的前驱（比当前元素小的最大元素）和后继（比当前元素大且最小的元素）的算法来进行遍历。

### LinkedHashMap及其实现方式

LinkedHashMap是基于哈希表和链表对Map接口的实现，它可以保存数据插入链表的顺序（使用额外于HashMap的链表实现）。此实现与 HashMap 的不同之处在于，后者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序通常就是将键插入到映射中的顺序（插入顺序）。注意，如果在映射中重新插入键，则插入顺序不受影响。（如果在调用m.put(k, v)前m.containsKey(k)返回了true，则调用时会将键 k 重新插入到映射 m 中。）



注：这里虽然把链表和桶的图分开画了，但是实际上它们中的结点（除了header）都是共用的

LinkedHashMap继承自HashMap，所以它比HashMap的性能略差，但是可以维护元素间的插入顺序（使用一个双向链表来保存顺序）：

**[java]** [view plain](http://blog.csdn.net/teaandnoodle/article/details/52270286) [copy](http://blog.csdn.net/teaandnoodle/article/details/52270286)

1. **private** **transient** Entry<K,V> header;
2. **private** **static** **class** Entry<K,V> **extends** HashMap.Entry<K,V> {
3. // These fields comprise the doubly linked list used for iteration.
4. Entry<K,V> before, after;
5. …….//省略
6. }

当要调用put方法插入元素时，会调用HashMap的put方法，这个方法会调用addEntry()方法，这个方法在LinkedHashMap中被重定义了：

**[java]** [view plain](http://blog.csdn.net/teaandnoodle/article/details/52270286) [copy](http://blog.csdn.net/teaandnoodle/article/details/52270286)

1. //LinkedHashMap的addEntry方法
2. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
3. **super**.addEntry(hash, key, value, bucketIndex);//调用HashMap中的addEntry方法，会创建结点，同时会维护新创建的结点到双向链表中
4. // Remove eldest entry if instructed
5. Entry<K,V> eldest = header.after;
6. **if** (removeEldestEntry(eldest)) {
7. removeEntryForKey(eldest.key);
8. }
9. }
10. //HashMap中的addEntry方法
11. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
12. **if** ((size >= threshold) && (**null** != table[bucketIndex])) {
13. resize(2 \* table.length);
14. hash = (**null** != key) ? hash(key) : 0;
15. bucketIndex = indexFor(hash, table.length);
16. }
18. createEntry(hash, key, value, bucketIndex);
19. }
20. //LinkedHashMap中的createEntry，覆盖HashMap中的createEntry
21. **void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {
22. HashMap.Entry<K,V> old = table[bucketIndex];
23. Entry<K,V> e = **new** Entry<>(hash, key, value, old);
24. table[bucketIndex] = e;
25. e.addBefore(header);
26. size++;
27. }

从以上代码中我们可以看到LinkedHashMap的put方法的过程，首先LinkedHashMap中没有put方法，所以会调用HashMap中的put方法，这个put方法会检查数据是否在Map中，如果不在就会调用addEntry方法，由于LinkedHashMap覆盖了父类的addEntry方法，所以会直接调用LinkedHashMap的addEntry方法，这个方法中又调用了HashMap的addEntry方法，addEntry又调用了createEntry方法，这个方法也是LinkedHashMap覆盖了HashMap的，它会创建结点到table中，同时会维护Entry（继承自HashMap.Entry的LinkedHashMap.Entry）的前后元素。

**[java]** [view plain](http://blog.csdn.net/teaandnoodle/article/details/52270286) [copy](http://blog.csdn.net/teaandnoodle/article/details/52270286)

1. //HashMap中的createEntry方法，对比以上LinkedHashMap中的createEntry方法发现，除了将Entry放入桶中之外，LinkedHashMap还维护了Entry指向之前元素和之后元素的指针
2. **void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {
3. Entry<K,V> e = table[bucketIndex];
4. table[bucketIndex] = **new** Entry<>(hash, key, value, e);
5. size++;
6. }

简单来讲，LinkedHashMap中的Entry是带有指向在它自己插入Map之前和之后的元素引用的对象，在put元素时，首先检查数据是否已经在Map中，如果不在就创建这个Entry，同时还要把这个Entry记录插入到之前元素构成的链表中（并没有真的简单的创建了个链表结点，而是这个链表本身就是这些Entry元素构成的）。这些Entry本身不但是Map中table的元素，还是链表元素。

在进行遍历时，它使用的是KeyIterator，而KeyIterator继承自LinkedHashIterator，在LinkedHashIterator内部有链表的头指针指向的下一个元素：

Entry<K,V> nextEntry = header.after;

由于这些Entry本身是链表元素，也是table中元素，故直接找到其后继就可以得到所有元素。剩下的遍历过程就是对一个链表的遍历了，每遍历到一个Entry就可以获得它的key和value。

此外，LinkedHashMap还能维护一个最近最少访问的序列，其本质还是维护Entry指针，每次使用get访问元素时，都会将这个元素插入Map尾部，这样链表头部就是最近访问次数最少的元素了，整个链表就是从近期访问最少到近期访问最多的顺序。

其实现方式是，在get中找到要get的元素后调用元素的recordAccess方法，这个方法就把这个Entry的前后指针进行了调整。

**[java]** [view plain](http://blog.csdn.net/teaandnoodle/article/details/52270286) [copy](http://blog.csdn.net/teaandnoodle/article/details/52270286)

1. //LinkedHashMap的get方法
2. **public** V get(Object key) {
3. Entry<K,V> e = (Entry<K,V>)getEntry(key);
4. **if** (e == **null**)
5. **return** **null**;
6. e.recordAccess(**this**);//调整指针
7. **return** e.value;
8. }
9. //Entry的recordAccess方法，参数m就是一个LinkedHashMap
10. **void** recordAccess(HashMap<K,V> m) {
11. LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
12. **if** (lm.accessOrder) {//是否按照最近最少访问排列
13. lm.modCount++;
14. remove();//从当前链中删除自己
15. addBefore(lm.header);//加入到链表尾部
16. }
17. }

总的来说，对于所有的集合类来说，对于List，如果随机存取多于修改首尾元素的可能，则应该选择ArrayList，如果要实现类似队列或者栈的功能或者首尾添加的功能较多，则应该选择LinkedList；对于Set，HashSet是常用的Set，毕竟通常对Set操作都是插入和查询，但是如果希望产生带有排序的Set则可以使用TreeSet，希望记录插入顺序则要使用LinkedHashSet；而Map和Set类似，如果需要快速的查询和添加，则可以用HashMap，如果需要Map中的元素按照一定的规则排序，则可以用TreeMap，如果需要记录数据加入Map的顺序，或者需要使用最近最少使用的规则，则可以用LinkedHashMap。