

## 第9讲 | 对比Hashtable、HashMap、TreeMap有什么不同？

2018-05-24 杨晓峰

Java核心技术36讲

[进入课程 >](#)



讲述：黄洲君

时长 12:16 大小 5.62M



Map 是广义 Java 集合框架中的另外一部分，HashMap 作为框架中使用频率最高的类型之一，它本身以及相关类型自然也是面试考察的热点。

今天我要问你的问题是，**对比 Hashtable、HashMap、TreeMap 有什么不同？**谈谈你对 HashMap 的掌握。

### 典型回答

Hashtable、HashMap、TreeMap 都是最常见的一些 Map 实现，是以**键值对**的形式存储和操作数据的容器类型。

Hashtable 是早期 Java 类库提供的一个[哈希表](#)实现，本身是同步的，不支持 null 键和值，由于同步导致的性能开销，所以已经很少被推荐使用。

HashMap 是应用更加广泛的哈希表实现，行为上大致上与 Hashtable 一致，主要区别在于 HashMap 不是同步的，支持 null 键和值等。通常情况下，HashMap 进行 put 或者 get 操作，可以达到常数时间的性能，所以**它是绝大部分利用键值对存取场景的首选**，比如，实现一个用户 ID 和用户信息对应的运行时存储结构。

TreeMap 则是基于红黑树的一种提供顺序访问的 Map，和 HashMap 不同，它的 get、put、remove 之类操作都是  $O(\log(n))$  的时间复杂度，具体顺序可以由指定的 Comparator 来决定，或者根据键的自然顺序来判断。

## 考点分析

上面的回答，只是对一些基本特征的简单总结，针对 Map 相关可以扩展的问题很多，从各种数据结构、典型应用场景，到程序设计实现的技术考量，尤其是在 Java 8 里，HashMap 本身发生了非常大的变化，这些都是经常考察的方面。

很多朋友向我反馈，面试官似乎钟爱考察 HashMap 的设计和实现细节，所以今天我会增加相应的源码解读，主要专注于下面几个方面：

理解 Map 相关类似整体结构，尤其是有序数据结构的一些要点。

从源码去分析 HashMap 的设计和实现要点，理解容量、负载因子等，为什么需要这些参数，如何影响 Map 的性能，实践中如何取舍等。

理解树化改造的相关原理和改进原因。

除了典型的代码分析，还有一些有意思的并发相关问题也经常会被提到，如 HashMap 在并发环境可能出现[无限循环占用 CPU](#)、size 不准确等诡异的问题。

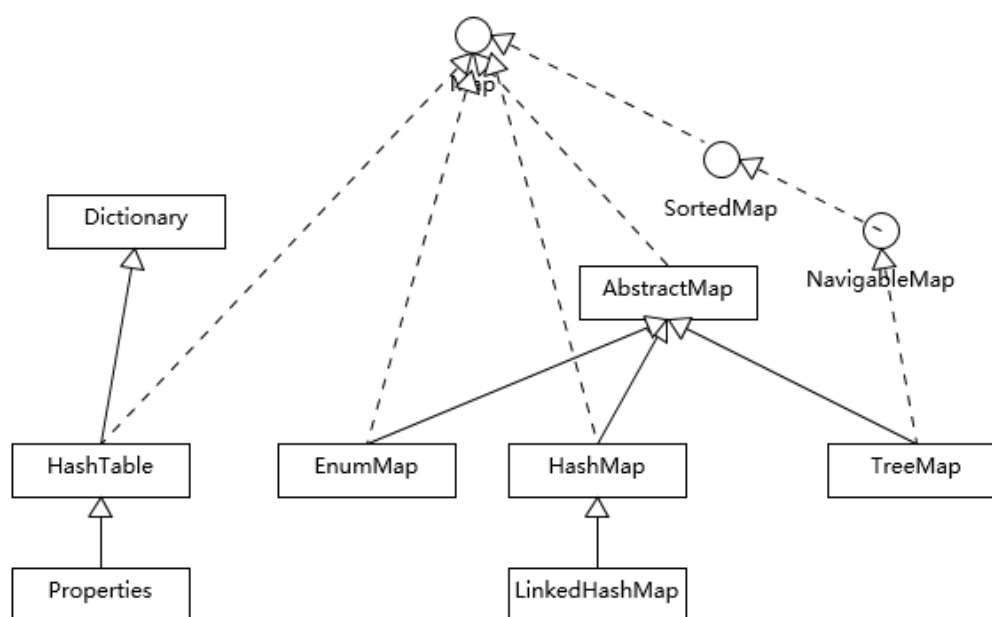
我认为这是一种典型的使用错误，因为 HashMap 明确声明不是线程安全的数据结构，如果忽略这一点，简单用在多线程场景里，难免会出现问题。

理解导致这种错误的原因，也是深入理解并发程序运行的好办法。对于具体发生了什么，你可以参考这篇很久以前的[分析](#)，里面甚至提供了示意图，我就不再重复别人写好的内容了。

## 知识扩展

### 1.Map 整体结构

首先，我们先对 Map 相关类型有个整体了解，Map 虽然通常被包括在 Java 集合框架里，但是其本身并不是狭义上的集合类型（Collection），具体你可以参考下面这个简单类图。



Hashtable 比较特别，作为类似 Vector、Stack 的早期集合相关类型，它是扩展了 Dictionary 类的，类结构上与 HashMap 之类明显不同。

HashMap 等其他 Map 实现则是都扩展了 AbstractMap，里面包含了通用方法抽象。不同 Map 的用途，从类图结构就能体现出来，设计目的已经体现在不同接口上。

大部分使用 Map 的场景，通常就是放入、访问或者删除，而对顺序没有特别要求，HashMap 在这种情况下基本是最好的选择。**HashMap 的性能表现非常依赖于哈希码的有效性，请务必掌握 hashCode 和 equals 的一些基本约定，比如：**

equals 相等，hashCode 一定要相等。

重写了 hashCode 也要重写 equals。

hashCode 需要保持一致性，状态改变返回的哈希值仍然要一致。


equals 的对称、反射、传递等特性。

这方面内容网上有很多资料，我就不在这里详细展开了。

针对有序 Map 的分析内容比较有限，我再补充一些，虽然 LinkedHashMap 和 TreeMap 都可以保证某种顺序，但二者还是非常不同的。

LinkedHashMap 通常提供的是遍历顺序符合插入顺序，它的实现是通过为条目（键值对）维护一个双向链表。注意，通过特定构造函数，我们可以创建反映访问顺序的实例，所谓的 put、get、compute 等，都算作“访问”。

这种行为适用于一些特定应用场景，例如，我们构建一个空间占用敏感的资源池，希望可以自动将最不常被访问的对象释放掉，这就可以利用 LinkedHashMap 提供的机制来实现，参考下面的示例：

 复制代码

```
1 import java.util.LinkedHashMap;
2 import java.util.Map;
3 public class LinkedHashMapSample {
4     public static void main(String[] args) {
5         LinkedHashMap<String, String> accessOrderedMap = new LinkedHashMap<String, String>();
6         @Override
7         protected boolean removeEldestEntry(Map.Entry<String, String> eldest) { // 当大小超过3时，删除最老的元素
8             return size() > 3;
9         }
10    };
11    accessOrderedMap.put("Project1", "Valhalla");
12    accessOrderedMap.put("Project2", "Panama");
13    accessOrderedMap.put("Project3", "Loom");
14    accessOrderedMap.forEach( (k,v) -> {
15        System.out.println(k + ":" + v);
16    });
17    // 模拟访问
18    accessOrderedMap.get("Project2");
19    accessOrderedMap.get("Project2");
20    accessOrderedMap.get("Project3");
21    System.out.println("Iterate over should be not affected:");
22    accessOrderedMap.forEach( (k,v) -> {
23        System.out.println(k + ":" + v);
24    });
25    // 触发删除
26    accessOrderedMap.put("Project4", "Mission Control");
```

```

27         System.out.println("Oldest entry should be removed:");
28         accessOrderedMap.forEach( (k,v) -> { // 遍历顺序不变
29             System.out.println(k + ":" + v);
30         });
31     }
32 }
33


```

对于 TreeMap，它的整体顺序是由键的顺序关系决定的，通过 Comparator 或 Comparable（自然顺序）来决定。

我在上一讲留给你的思考题提到了，构建一个具有优先级的调度系统的问题，其本质就是个典型的优先队列场景，Java 标准库提供了基于二叉堆实现的 PriorityQueue，它们都是依赖于同一种排序机制，当然也包括 TreeMap 的马甲 TreeSet。

类似 hashCode 和 equals 的约定，为了避免模棱两可的情况，自然顺序同样需要符合一个约定，就是 compareTo 的返回值需要和 equals 一致，否则就会出现模棱两可情况。

我们可以分析 TreeMap 的 put 方法实现：

 复制代码

```

1 public V put(K key, V value) {
2     Entry<K,V> t = ...
3     cmp = k.compareTo(t.key);
4     if (cmp < 0)
5         t = t.left;
6     else if (cmp > 0)
7         t = t.right;
8     else
9         return t.setValue(value);
10    // ...
11 }

```

从代码里，你可以看出什么呢？当我不遵守约定时，两个不符合唯一性（equals）要求的对象被当作是同一个（因为，compareTo 返回 0），这会导致歧义的行为表现。

## 2.HashMap 源码分析

前面提到，HashMap 设计与实现是个非常高频的面试题，所以我会在这一进行相对详细的源码解读，主要围绕：

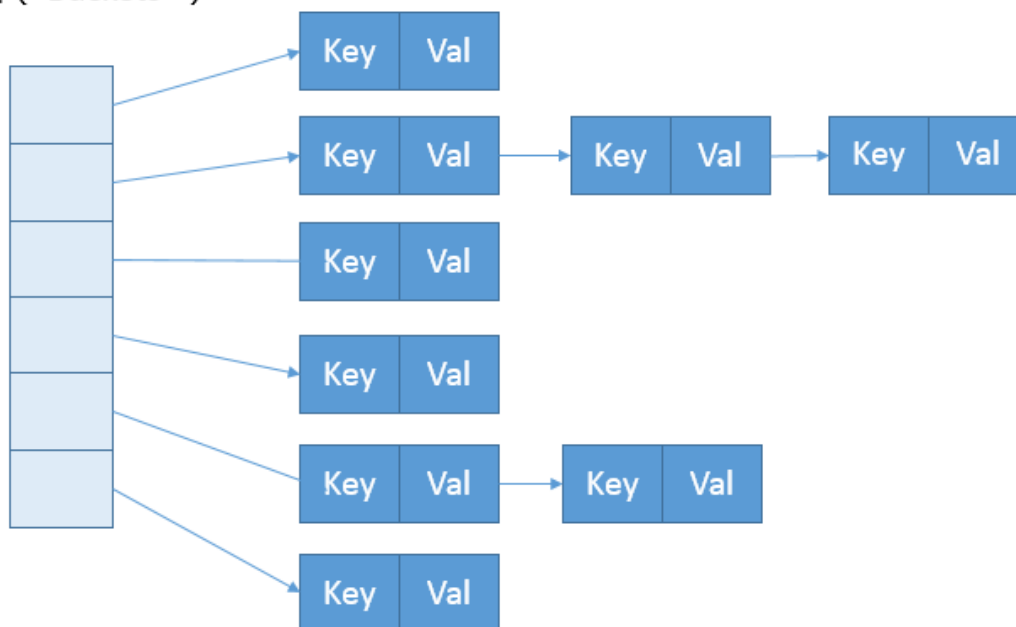
HashMap 内部实现基本点分析。

容量（capacity）和负载系数（load factor）。


树化。

首先，我们来一起看看 HashMap 内部的结构，它可以看作是数组（Node<K,V>[] table）和链表结合组成的复合结构，数组被分为一个个桶（bucket），通过哈希值决定了键值对在这个数组的寻址；哈希值相同的键值对，则以链表形式存储，你可以参考下面的示意图。这里需要注意的是，如果链表大小超过阈值（TREEIFY\_THRESHOLD, 8），图中的链表就会被改造为树形结构。

桶数组（Buckets）




从非拷贝构造函数的实现来看，这个表格（数组）似乎并没有在最初就初始化好，仅仅设置了一些初始值而已。

 复制代码


```
1 public HashMap(int initialCapacity, float loadFactor){
2     // ...
3     this.loadFactor = loadFactor;
4     this.threshold = tableSizeFor(initialCapacity);
5 }
```

所以，我们深刻怀疑，HashMap 也许是按照 lazy-load 原则，在首次使用时被初始化（拷贝构造函数除外，我这里仅介绍最通用的场景）。既然如此，我们去看看 put 方法实现，似乎只有一个 putVal 的调用：

 复制代码

```
1 public V put(K key, V value) {
2     return putVal(hash(key), key, value, false, true);
3 }
```

看来主要的密码似乎藏在 putVal 里面，到底有什么秘密呢？为了节省空间，我这里只截取了 putVal 比较关键的几部分。

 复制代码


```
1 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
2               boolean evit) {
3     Node<K,V>[] tab; Node<K,V> p; int , i;
4     if ((tab = table) == null || (n = tab.length) == 0)
5         n = (tab = resize()).length;
6     if ((p = tab[i = (n - 1) & hash]) == null)
7         tab[i] = newNode(hash, key, value, null);
8     else {
9         // ...
10        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for first
11            treeifyBin(tab, hash);
12        // ...
13    }
14 }
15
```

从 putVal 方法最初的几行，我们就可以发现几个有意思的地方：

如果表格是 null，resize 方法会负责初始化它，这从 tab = resize() 可以看出。


resize 方法兼顾两个职责，创建初始存储表格，或者在容量不满足需求的时候，进行扩容 (resize)。

在放置新的键值对的过程中，如果发生下面条件，就会发生扩容。

 复制代码


```
1 if (++size > threshold)
2     resize();
```

具体键值对在哈希表中的位置（数组 index）取决于下面的位运算：

 复制代码

```
1 i = (n - 1) & hash
```

仔细观察哈希值的源头，我们会发现，它并不是 key 本身的 hashCode，而是来自于 HashMap 内部的另外一个 hash 方法。注意，为什么这里需要将高位数据移位到低位进行异或运算呢？**这是因为有些数据计算出的哈希值差异主要在高位，而 HashMap 里的哈希寻址是忽略容量以上的高位的，那么这种处理就可以有效避免类似情况下的哈希碰撞。**


 复制代码

```
1 static final int hash(Object key) {
2     int h;
3     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>>16);
4 }
```

我前面提到的链表结构（这里叫 bin），会在达到一定门限值时，发生树化，我稍后会分析为什么 HashMap 需要对 bin 进行处理。

可以看到，putVal 方法本身逻辑非常集中，从初始化、扩容到树化，全部都和它有关，推荐你阅读源码的时候，可以参考上面的主要逻辑。

我进一步分析一下身兼多职的 resize 方法，很多朋友都反馈经常被面试官追问它的源码设计。

 复制代码



```

1 final Node<K,V>[] resize() {
2     // ...
3     else if ((newCap = oldCap << 1) < MAXIMUM_CAPACIY &&
4             oldCap >= DEFAULT_INITIAL_CAPAITY)
5         newThr = oldThr << 1; // double there
6         // ...
7     else if (oldThr > 0) // initial capacity was placed in threshold
8         newCap = oldThr;
9     else {
10        // zero initial threshold signifies using defaultsfults
11        newCap = DEFAULT_INITIAL_CAPAITY;
12        newThr = (int)(DEFAULT_LOAD_ATOM* DEFAULT_INITIAL_CAPACITY;
13    }
14    if (newThr ==0) {
15        float ft = (float)newCap * loadFator;
16        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?(int)ft : :
17    }
18    threshold = neThr;
19    Node<K,V>[] newTab = (Node<K,V>[])new Node[newap];
20    table = n;
21    // 移动到新的数组结构 e 数组结构
22 }
23

```

依据 `resize` 源码，不考虑极端情况（容量理论最大极限由 `MAXIMUM_CAPACITY` 指定，数值为  $1 << 30$ ，也就是 2 的 30 次方），我们可以归纳为：

门限值等于（负载因子） $\times$ （容量），如果构建 `HashMap` 的时候没有指定它们，那么就是依据相应的默认常量值。

门限通常是以倍数进行调整（`newThr = oldThr << 1`），我前面提到，根据 `putVal` 中的逻辑，当元素个数超过门限大小时，则调整 `Map` 大小。

扩容后，需要将老的数组中的元素重新放置到新的数组，这是扩容的一个主要开销来源。

### 3. 容量、负载因子和树化


前面我们快速梳理了一下 `HashMap` 从创建到放入键值对的相关逻辑，现在思考一下，为什么我们需要在乎容量和负载因子呢？

这是因为容量和负载系数决定了可用的桶的数量，空桶太多会浪费空间，如果使用的太满则会严重影响操作的性能。极端情况下，假设只有一个桶，那么它就退化成了链表，完全不能

提供所谓常数时间存的性能。

既然容量和负载因子这么重要，我们在实践中应该如何选择呢？

如果能够知道 HashMap 要存取的键值对数量，可以考虑预先设置合适的容量大小。具体数值我们可以根据扩容发生的条件来做简单预估，根据前面的代码分析，我们知道它需要符合计算条件：

 复制代码

```
1  负载因子 * 容量 > 元素数量
2
```

所以，预先设置的容量需要满足，大于“预估元素数量 / 负载因子”，同时它是 2 的幂数，结论已经非常清晰了。


而对于负载因子，我建议：

如果没有特别需求，不要轻易进行更改，因为 JDK 自身的默认负载因子是非常符合通用场景的需求的。

如果确实需要调整，建议不要设置超过 0.75 的数值，因为会显著增加冲突，降低 HashMap 的性能。

如果使用太小的负载因子，按照上面的公式，预设容量值也进行调整，否则可能会导致更加频繁的扩容，增加无谓的开销，本身访问性能也会受影响。

我们前面提到了树化改造，对应逻辑主要在 putVal 和 treeifyBin 中。

 复制代码

```
1  final void treeifyBin(Node<K,V>[] tab, int hash) {
2      int n, index; Node<K,V> e;
3      if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
4          resize();
5      else if ((e = tab[index = (n - 1) & hash]) != null) {
6          // 树化改造逻辑
7      }
8  }
9
```

上面是精简过的 treeifyBin 示意，综合这两个方法，树化改造的逻辑就非常清晰了，可以理解为，当 bin 的数量大于 TREEIFY\_THRESHOLD 时：

如果容量小于 MIN\_TREEIFY\_CAPACITY，只会进行简单的扩容。

如果容量大于 MIN\_TREEIFY\_CAPACITY，则会进行树化改造。

那么，为什么 HashMap 要树化呢？

**本质上这是个安全问题。**因为在元素放置过程中，如果一个对象哈希冲突，都被放置到同一个桶里，则会形成一个链表，我们知道链表查询是线性的，会严重影响存取的性能。

而在现实世界，构造哈希冲突的数据并不是非常复杂的事情，恶意代码就可以利用这些数据大量与服务器端交互，导致服务器端 CPU 大量占用，这就构成了哈希碰撞拒绝服务攻击，国内一线互联网公司就发生过类似攻击事件。

今天我从 Map 相关的几种实现对比，对各种 Map 进行了分析，讲解了有序集合类型容易混淆的地方，并从源码级别分析了 HashMap 的基本结构，希望你有所帮助。

## 一课一练

关于今天我们讨论的题目你做到心中有数了吗？留一道思考题给你，解决哈希冲突有哪些典型方法呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。

# Java 核心技术36讲

—— 前 Oracle 首席工程师  
带你修炼 Java 内功 ——

杨晓峰 前 Oracle 首席工程师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 第8讲 | 对比Vector、ArrayList、LinkedList有何区别？

下一篇 第10讲 | 如何保证集合是线程安全的？ConcurrentHashMap如何实现高效地线程安全？

## 精选留言 (72)

写留言



天凉好个秋 置顶

2018-05-24

89

解决哈希冲突的常用方法有：

开放定址法

基本思想是：当关键字key的哈希地址 $p = H(\text{key})$ 出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，...，直到找出...

展开



三口先生 置顶

2018-05-24

10

最常用的方法就是线性再散列。即插入元素时，没有发生冲突放在原有的规则下的空槽

下，发生冲突时，简单遍历hash表，找到表中下一个空槽，进行元素插入。查找元素时，找到相应的位置的元素，如果不匹配则进行遍历hash表。  
然后就是我们非线性再散列，就是冲突时，再hash，核心思想是，如果产生冲突，产生一个新的hash值进行寻址，如果还是冲突，则继续。...

展开 ▾



公号-代码...

2018-05-24

👍 156

### Hashtable、HashMap、TreeMap心得

三者均实现了Map接口，存储的内容是基于key-value的键值对映射，一个映射不能有重复的键，一个键最多只能映射一个值。

...

展开 ▾



清风

2018-07-05

👍 133

感觉每个知识点都很重要，但又点到为止，感觉读完不痛不痒，好像学到什么，但细想又没掌握什么，希望能够深入一点！

展开 ▾



amourling

2018-07-11

👍 68

提个意见，文章中请不要出现太多似乎，怀疑之类的必须，该是什么就是什么，不确定的不要拿出来。



小飞哥 ...

2018-07-20

👍 32

总觉得还是不太深，只是每个map的区别。我觉得每个map的实现都能讲出很多问题来，在面试时也经常碰壁，看完但也没觉得学到什么深入的地方

展开 ▾



鲤鱼

2018-05-29

👍 16

读到最后链表树化刚准备开始飙车，结果突然跳车。树化讲细点更好

作者回复: 感谢反馈，最近几章篇幅都超标了.....只能照顾大多数需求，抱歉



**Jerry银银**

2018-11-27

👍 13

为什么HashMap要树化？

文章说『本质是个安全问题』，但是导致安全问题的本质其实是性能问题。哈希碰撞频繁，导致链表过长，查询时间陡升，黑客则会利用这个『漏洞』来攻击服务器，让服务器CPU被大量占用，从而引起了安全问题。而树化(使用红黑树)能将时间复杂度降到...

展开 ▾

作者回复: 竟然无法反驳，哈哈



**j.c.**

2018-05-24

👍 10

这是面试必问题。什么时候也能讲讲红黑树的树化具体过程，那个旋转一直没搞懂。另外treeifyBin这个单词的词面意思是什么？

展开 ▾



**xinfangke**

2018-05-29

👍 9

老师 如果hashmap中不存在hash冲突 是不是就相当于一个数组结构呢 就不存在链表了呢

作者回复: 我理解是



**陈大麦**

2018-07-28

👍 8

老师我想问一下，hashmap明明继承了abstractmap，而abstractmap已经实现了map接口，为什么hashmap还要实现map接口呢？

展开 ▾



**睡骨**

2018-08-31

👍 6

希望作者分享源码的时候，最好备注是基于哪个版本的 毕竟有些地方不同版本差异较大



**Darcy**

2018-07-28

👍 5

equals 的对称、反射、传递等特性。

这里的反射是不是写错了，应该是自反性，对称性，传递性，一致性等等

展开 ▾



**kevin**

2018-09-27

👍 4

看不太懂，讲的还不是不太浅显，既然是面试题，最好不要太浅，但也不要太深，你这个度掌握的不是很好

作者回复: 嗯，谢谢指出



**zjh**

2018-05-28

👍 4

受教了，把java集合的源代码掌握了，对java和数据结构的了解都会有很大的提升



**代码狂徒**

2018-05-24

👍 4

针对负载因子，您所指的存太满会影响性能是指什么？毕竟已经开辟了相应内存空间的，没什么不用呢？

作者回复: 冲突可能会增加,影响查询之类性能，当然看具体的需求





Lh

2019-02-20

👍 3

## Hashtable、HashMap、TreeMap心得

三者均实现了Map接口，存储的内容是基于key-value的键值对映射，一个映射不能有重复的键，一个键最多只能映射一个值。

...

展开 ∨

---



coolboy

2018-06-24

👍 3

removeEldestEntry这个方法是不是指移除最旧的对象，也就是按照最先被put进来的顺序，而不是指不常访问的对象。

展开 ∨

---



灰飞灰猪不...

2018-05-24

👍 3

这是1.7的hashmap吧？

展开 ∨

---



Jerry银银

2018-05-24

👍 3

我一直认为：JAVA集合类是非常好的学习材料。

如果敢说精通JAVA集合类，计算机功底肯定不会太差

展开 ∨