在Java 8 之前，**HashMap和其他基于map的类都是通过链地址法解决冲突**，它们使用单向链表来存储相同索引值的元素。在最坏的情况下，这种方式会将HashMap的get方法的性能从O(1)降低到O(n)。为了解决在频繁冲突时hashmap性能降低的问题，Java 8中使用平衡树来替代链表存储冲突的元素。这意味着我们可以将最坏情况下的性能从O(n)提高到O(logn)。  
在Java 8中使用常量TREEIFY\_THRESHOLD来控制是否切换到平衡树来存储。目前，这个常量值是8，这意味着当有超过8个元素的索引一样时，HashMap会使用树来存储它们。  
这一改变是为了继续优化常用类。大家可能还记得在Java 7中为了优化常用类对ArrayList和HashMap采用了延迟加载的机制，在有元素加入之前不会分配内存，这会减少空的链表和HashMap占用的内存。  
这一动态的特性使得HashMap一开始使用链表，并在冲突的元素数量超过指定值时用平衡二叉树替换链表。不过这一特性在所有基于hash table的类中并没有，例如Hashtable和WeakHashMap。  
目前，只有ConcurrentHashMap,LinkedHashMap和HashMap会在频繁冲突的情况下使用平衡树。

**什么时候会产生冲突**

HashMap中调用hashCode()方法来计算hashCode。  
由于在Java中两个不同的对象可能有一样的hashCode,所以不同的键可能有一样hashCode，从而导致冲突的产生。

**总结**

* HashMap在处理冲突时使用**链表存储相同索引的元素(链地址法)**。
* 从Java 8开始，HashMap，ConcurrentHashMap和LinkedHashMap在处理频繁冲突时将使用平衡树来代替链表，当同一hash桶中的元素数量超过特定的值便会由链表切换到平衡树，这会将get()方法的性能从O(n)提高到O(logn)。
* 当从链表切换到平衡树时，HashMap迭代的顺序将会改变。不过这并不会造成什么问题，因为HashMap并没有对迭代的顺序提供任何保证。
* 从Java 1中就存在的Hashtable类为了保证迭代顺序不变，即便在频繁冲突的情况下也不会使用平衡树。这一决定是为了不破坏某些较老的需要依赖于Hashtable迭代顺序的Java应用。
* 除了Hashtable之外，WeakHashMap和IdentityHashMap也不会在频繁冲突的情况下使用平衡树。
* 使用HashMap之所以会产生冲突是因为使用了键对象的hashCode()方法，而equals()和hashCode()方法不保证不同对象的hashCode是不同的。需要记住的是，**相同对象的hashCode一定是相同的，但相同的hashCode不一定是相同的对象。**
* 在HashTable和HashMap中，冲突的产生是由于不同对象的hashCode()方法返回了一样的值。

以上就是Java中HashMap如何处理冲突。这种方法被称为链地址法，因为使用链表存储同一桶内的元素。

通常情况HashMap，HashSet，LinkedHashSet，LinkedHashMap，ConcurrentHashMap，HashTable，IdentityHashMap和WeakHashMap均采用这种方法处理冲突。

从JDK 8开始，HashMap，LinkedHashMap和ConcurrentHashMap为了提升性能，在频繁冲突的时候使用平衡树来替代链表。因为HashSet内部使用了HashMap，LinkedHashSet内部使用了LinkedHashMap，所以他们的性能也会得到提升。

HashMap的快速高效，使其使用非常广泛。其原理是，调用hashCode（）和equals（）方法，并对hashcode进行一定的哈希运算（所以并不是一模一样的hash值，而只是哈希运算后，所得的结果一样）得到相应value的位置信息，将其分到不同的桶里。桶的数量一般会比所承载的实际键值对多。当通过key进行查找的时候，往往能够在常数时间内找到该value。

但是，当某种针对key的hashcode的哈希运算得到的位置信息重复了之后，就发生了哈希碰撞。这会对HashMap的性能产生灾难性的影响。

在Java 8 之前， 如果发生碰撞往往是将该value直接链接到该位置的其他所有value的末尾，即相互碰撞的所有value形成一个链表。

因此，在最坏情况下，HashMap的查找时间复杂度将退化到O（n）.

但是在Java 8 中，该碰撞后的处理进行了改进。当一个位置所在的冲突过多时，存储的value将形成一个排序二叉树，排序依据为key的hashcode。则，在最坏情况下，HashMap的查找时间复杂度将从O（1）退化到O（logn）。

虽然是一个小小的改进，但意义重大：

1、O（n）到O（logn）的时间开销。

2、如果恶意程序知道我们用的是Hash算法，在纯链表情况下，它能够发送大量请求导致哈希碰撞，然后不停访问这些key导致HashMap忙于进行线性查找（链表查找），最终陷入瘫痪，即形成了拒绝服务攻击（DoS）。