HashMap 与 HashTable 之间的区别

1.HashMap 线程不安全、HashTable 线程安全；
2.多线程的情况下使用 HashTable 能够保证数据安全性，是采用 synchronized 锁将整个 HashTable 中的数组锁住，在多个线程中只允许一个线程访问 Put 或者 Get，效率非常低。
3.多线程的情况下使用 HashMap 线程不安全，没有上锁，可能会发生一些数据冲突问题，但是效率比较高的。
4.HashMap 可以允许存放 key 值为 null，存放在数组第 0 个位置、HashTable 不允许存放的 key 为 null
补充概念“线程安全问题”多个线程同时访问一个全局共享变量 可能会发生线程安全问题。

为什么重写 equals 还要重写 hashcode 方法

1. HashCode 是 Object 类中的本地方法，底层是基于 C 语言实现的，方法直接返回对象的内存地址，让后再转换整数。
2. Equals 方法
规定：
1. 两个对象的 Hashcode 值相等，但是两个对象的内容值不一定相等；
2. 两个对象的值 Equals 比较相等的情况下，则两个对象的 Hashcode 值一定相等；
== 比较两个对象的内存地址是否相同、Equals 默认的情况下比较两个对象的内存地址，只是我们的 String 类重写了 Equals 方法比较两个对象的值是否相同。
根据阿里巴巴 Java 开发手册，对于 equals 与 HashCode 相关的说明：
1. 【强制】关于 hashCode 和 equals 的处理，遵循如下规则：
1） 只要覆写 equals，就必须覆写 hashCode。
2） 因为 Set 存储的是不重复的对象，依据 hashCode 和 equals 进行判断，所以 Set 存储的对象必须覆
写这两个方法。
3） 如果自定义对象作为 Map 的键，那么必须覆写 hashCode 和 equals。
说明：String 已覆写 hashCode 和 equals 方法，所以我们可以愉快地使用 String 对象作为 key 来使用。

HashMap 如何避免内存泄漏问题

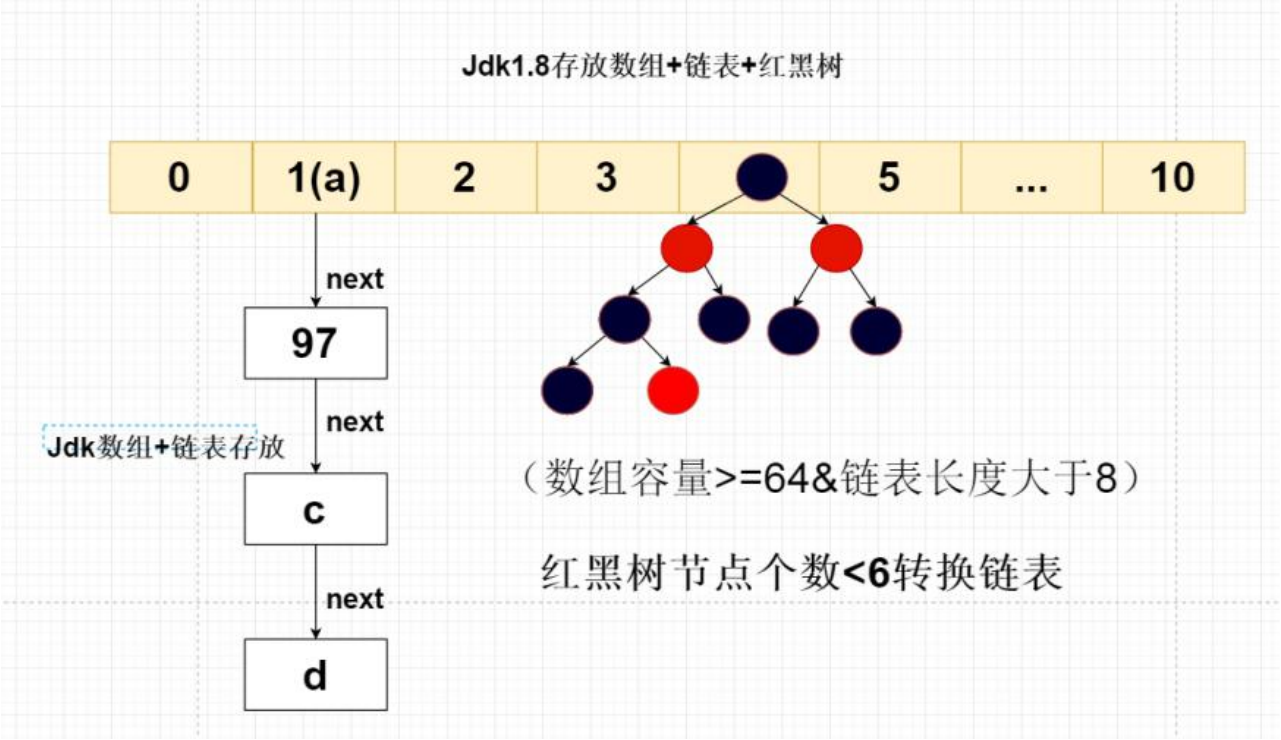
如果使用自定义对象作为 HashMap 集合的 key，注意需要重写 HashCode 和 Equals 方法、避免内存泄漏问题。

内存泄漏问题：申请堆内存 GC 一直无法释放引发内存泄漏问题。

HashMap 什么 HashCode 冲突

内容值不相同，但是 HashCode 值相同，则计算的 index 值会发生冲突。

HashMap1.7 与 1.8 底层如何实现（Put 方法底层实现）（重点）



1.7 底层实现
基于数组+链表实现(Key 和 value 封装成 Entry 对象)
1.根据 key 的 hash 值，计算该 key 存放在数组的 index 位置
2.如果发生 index 冲突，则会使用单向链表存放
同一个链表中存放的都是 hashCode 值相同，但是内容值却不同
如果 index 发生冲突，采用链表存放查询的时间复杂度是为 O（n），效率非常低，
所以在 JDK1.8 开始优化改为红黑树
3. 使用头插入法（并发下扩容可能会发生死循环问题）
1.8 底层实现

基于数组+链表+红黑树实现(Key 和 value 封装成 Entry 对象)
1.根据 key 的 hash 值，计算该 key 存放在数组的 index 位置
2.如果发生 index 冲突，则会使用单向链表存放
当数组的容量大于=64 且链表长度大于 8 则会将链表转化成红黑树。
红黑树查询的时间复杂度是为 O(logN)
当红黑树的节点个数<6 则将红黑树转换成链表
3.使用尾插法 解决了 HashMap1.7 版本并发扩容引发扩容死循环问题。

HashMap1.7 与 1.8 底层如何实现（Get 方法底层实现）（重点）

1.HashMap 集合 1.7 版本
会根据我们的 index 查找到在数组中的链表，会从头到尾部遍历我们链表做比较值是否相等，
如果 index 没有发生冲突，时间复杂度就是 O（1） 只需要查询一次；
如果 index 发生了冲突，时间复杂度就是为 O（N） 会从头查询到尾部 效率非常低。
2. HashMap 集合 1.8 版本
会根据我们的 index 查找到在数组中的链表或者是红黑树
如果是链表，则时间复杂度为 O(N);
如果是红黑树，则时间复杂度为 O(logN);

补充概念:O(1) 只需要查询一次 O（N） 从头查询到尾部

HashMapKey 为 null 存放在什么位置

HashMap 可以允许存放 key 值为 null，存放在数组第 0 个位置

为什么 HashMap1.8 需要引入红黑树（重点）

1.7HashMap 集合中，当我们发生了 Hash 冲突，则会存放在同一个链表中，当链表的查询长度过长，查询效率非常低，因为采用链表存放查询的时间复杂度是为 0（n），从头查询到尾部、在 JDK1.8 开始优化当数组容量>=64 且链表长度>8 则会将链表转化为改为红黑树，红黑树的时间复杂度为 0(logn)，性能有所提升。

HashMap1.8 链表在什么时候转化成红黑树

当数组的容量大于=64 且链表长度大于 8 则会将链表转化成红黑树。
红黑树查询的时间复杂度是为 O(logN)
当红黑树的节点个数<6 则将红黑树转换成链表

HashMap1.7 扩容死循环的问题有了解过吗

HashMap1.7 版本 使用头插入法（并发下扩容可能会发生死循环问题），在 HashMap1.8 版本采用尾插法解决了该问题。
注意的是：JDK 官方不承认该 1.7HashMapbug 问题，因为在多线程的情况下不推荐使用 HashMap 而是 HashTable 或者是 ConcurrentHashMap

HashMap 底层是有序存放的吗？

HashMap 是无序、散列存放 ，遍历的时候从数组 0 开始遍历每个链表，遍历结果存储顺序不保证一致，如果需要根据存储顺序保存，可以使用 LinkedHashMap 底层是基于双向链表存放，LinkedHashMap 基于双向链表实现 HashMap 基本单向链表实现

HashMap 如何解决 Hash 冲突问题

JDK1.7 版本的 HashMap
1.根据 key 的 hash 值，计算该 key 存放在数组的 index 位置
2.如果发生 index 冲突，则会使用单向链表存放
JDK1.8 版本的 HashMap
同一个链表中存放的都是 hashCode 值相同，但是内容值却不同

1.根据 key 的 hash 值，计算该 key 存放在数组的 index 位置

2.如果发生 index 冲突，则会使用单向链表存放,当数组的容量大于=64 且链表长度大于 8 则会将链表转化成红黑树。

HashMap 根据 key 查询的时间复杂度

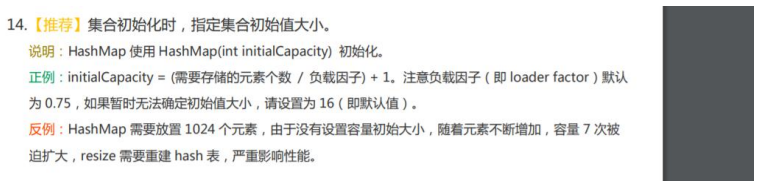
key 如果没有发生冲突：时间复杂度为 O（1）
Key 如果发生冲突：链表为 O（N），红黑树则是为 O(LogN)

HashMap 底层如何降低 Hash 冲突概率

1.高 16 位做异或运算
Hash 值 异或运算：(h = key.hashCode()) ^ (h >>> 16)
2.计算 Index 值与运算 会实现 i = (n - 1) & hash
最终可以实现减少 index 冲突的概率。
3.Hashmap 用 2 的 n 次幂作为容量

HashMap 如何存放 1 万条 key 效率最高

参考阿里巴巴官方手册：



(需要存储的元素个数 / 负载因子) + 1
10000/0.75+1=13334

正常如果存放 1 万个 key 的情况下 大概扩容 10 次=16384

为什么加载因子是 0.75 而不是 1

产生背景：减少 Hash 冲突 index 的概率；
查询效率与空间问题；

简单推断的情况下，提前做扩容：
1. 如果加载因子越大，空间利用率比较高，有可能冲突概率越大；
2. 如果加载因子越小，有可能冲突概率比较小，空间利用率不高；
空间和时间上平衡点：0.75
统计学概率：泊松分布是统计学和概率学常见的离散概率分布

为何 Hashmap 用 2 的 n 次幂作为容量

目的是为了减少 hash 冲突的概率，实现 hashkey 均匀存放。

算法的目的是为了得到小于 length 的更加均匀的数，如果不均匀容易产生 hash 碰撞，换句话说只有全是 1，进行按位与才是最均匀的，因为 1 与上任何数都等于任何数本身

为什么是 length-1 不是 length 了
16 是 10000 15 是 01111。16 与任何数只能是 0 或者 16。15 与任何数等于小于 16 的任何数本身。

为什么容量是 2 的 n 次方呢
2 的 n 次方一定是最高位 1 其它低位是 0，
这样减 1 的时候才能得到 01111 这样都是 1 的二进制。

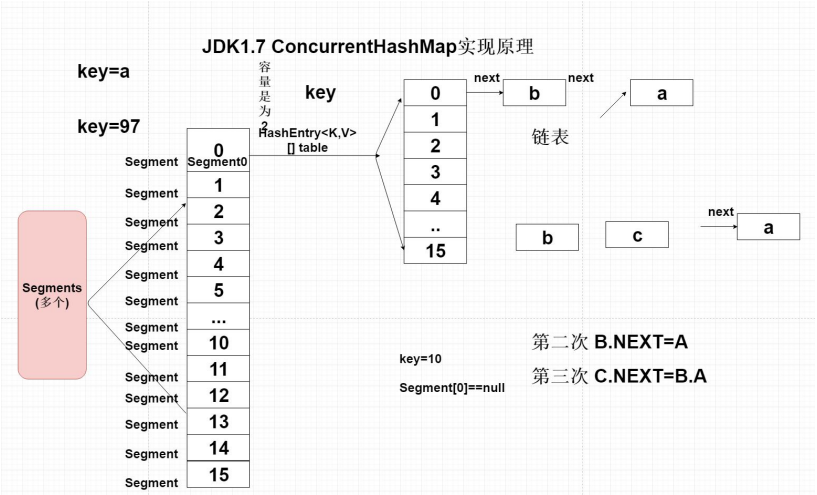
如何在高并发的情况下使用 HashMap

不建议使用 HashTable 效率偏低，建议使用 ConcurrentHashMap。

ConcurrentHashMap 底层实现的原理

1.使用传统 HashTable 保证线程问题，是采用 synchronized 锁将整个 HashTable 中的数组锁住，在多个线程中只允许一个线程访问 Put 或者 Get，效率非常低，但是能够保证线程安全问题。
2.在多线程的情况下推荐使用 ConcurrentHashMap

ConcurrentHashMap1.7 实现原理



1. ConcurrentHashMap 采用分段锁设计、将一个大的 HashMap 集合拆分成 n 多个不同的小的 HashTable（Segment），默认的情况下是分成 16 个不同的 Segment，每个 Segment 中都有自己独立的 HashEntry<K,V>[] table；
2. 数组+Segments 分段锁+HashEntry 链表实现
3. 使用 Lock 锁+CAS 乐观锁+UNSAFE 类

4.PUT 方法流程

A. 第一次需要计算出：key 出存放在那个 Segment 对象中
B. 还需要计算 key 存放在 Segment 对象中具体 index 位置。

ConcurrentHashMap1.8 实现原理

Put 原理 锁的粒度非常小，对每个数组 index 位置上锁 对 1.7ConcurrentHashMap 实现优化

1. 取消 segment 分段设计，使用 synchronized 锁
2. synchronized 在 JDK1.6 开始做了优化 默认实现锁的升级过程