1. hashmap的数据结构

哈希表结构（数组+链表）实现的，当链表节点数超过8时转换为红黑树

1. hashmap的工作原理

hashmap底层是hash数组和单链表实现的，数组中的每个元素都是链表，有内部类node（实现Map.Entry）实现,hashmap通过put和get方法来存储和获取数据

存储数据：将键值传个put方法

1. 使用hash方法（参数为k）计算k的hash值，然后结合数组长度计算时数组下标位置
2. 调整数组大小（当容器中的元素个数大于capacity\*localfactor时，容器进行扩容，扩大为原来的2倍）
3. 当计算出来的hash值不存在时，执行插入，若存在，则发生hash碰撞
4. 当hash值存在，并且两者的equals方法返回true，那么就更新键值对
5. 当hash值存在，但是equal方法返回false，则插入你链表的尾部（尾插法），或者红黑树中（jdk1.7使用的是头插法，jdk1.8使用的是尾插法，当链表大于8数组长度大于64时，就把链表换成红黑树）

获取数据

将k传给get()方法

1. 调用hash(k)方法（计算的hash值）从而获取改键值所在链表的数组下标
2. 遍历链表，使用equals方法查找相同node连接中k值对应的v值，

hashCode 是定位的，存储位置；equals是定性的，比较两者是否相等。

1. 当两个hashcode相同会发生什么

因为 hashCode 相同，值不一定就是相等的（equals方法比较），所以两个对象所在数组的下标相同，"碰撞"就此发生。又因为 HashMap 使用链表存储对象，这个 Node 会存储到链表中。为什么要重写 hashcode 和 equals 方法？

1. Hash的实现是什么，为什么要这样实现

JDK 1.8 中，是通过 hashCode() 的高 16 位异或低 16 位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度，功效和质量来考虑的，减少系统的开销，也不会造成因为高位没有参与下标的计算，从而引起的碰撞。

1. 为什么要用异或运算

保证了对象的 hashCode 的 32 位值只要有一位发生改变，整个 hash() 返回值就会改变。尽可能的减少碰撞。

1. hashMap的table容量是如何确定的，该容量如何变化，这种变化带来什么问题；localFactor是什么，

①、table 数组大小是由 capacity 这个参数确定的，默认是16，也可以构造时传入，最大限制是1<<30；

②、loadFactor 是装载因子，主要目的是用来确认table 数组是否需要动态扩展，默认值是0.75，比如table 数组大小为 16，装载因子为 0.75 时，threshold 就是12，当 table 的实际大小超过 12 时，table就需要动态扩容；

③、扩容时，调用 resize() 方法，将 table 长度变为原来的两倍（注意是 table 长度，而不是 threshold）

④、如果数据很大的情况下，扩展时将会带来性能的损失，在性能要求很高的地方，这种损失很可能很致命。

1. hashmap put方法的执行过程

调用哈希函数获取Key对应的hash值，再计算其数组下标；

如果没有出现哈希冲突，则直接放入数组；如果出现哈希冲突，则以链表的方式放在链表后面；

如果链表长度超过阀值( TREEIFY THRESHOLD==8)，就把链表转成红黑树，链表长度低于6，就把红黑树转回链表;

如果结点的key已经存在，则替换其value即可；

如果集合中的键值对大于12，调用resize方法进行数组扩容。

1. 数组扩容的过程

创建一个新的数组，其容量为旧数组的两倍，并重新计算旧数组中结点的存储位置。结点在新数组中的位置只有两种，原下标位置或原下标+旧数组的大小。

1. 拉链法导致的链表过深问题为什么不使用二叉查找树代替，而选择红红黑树，为什么不一致使用红黑树

之所以选择红黑树是为了解决二叉查找树的缺陷，二叉查找树在特殊情况下会变成一条线性结构（这就跟原来使用链表结构一样了，造成很深的问题），遍历查找会非常慢。

而红黑树在插入新数据后可能需要**通过左旋，右旋、变色这些操作来保持平衡**，**引入红黑树就是为了查找数据快，解决链表查询深度的问题**，我们知道红黑树属于平衡二叉树，但是**为了保持“平衡”是需要付出代价的，但是该代价所损耗的资源要比遍历线性链表要少**，所以当长度大于8的时候，会使用红黑树，如果链表长度很短的话，根本不需要引入红黑树，引入反而会慢。

1. 红黑树的见解

每个节点非红即黑

根节点总是黑色的

如果节点是红色的，则它的子节点必须是黑色的（反之不一定）

每个叶子节点都是黑色的空节点（NIL节点）

从根节点到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点（即相同的黑色高度）

1. Jdk1.8对hashmap做了那些改变

在java 1.8中，如果链表的长度超过了8，那么链表将转换为红黑树。（桶的数量必须大于64，小于64的时候只会扩容）

发生hash碰撞时，java 1.7 会在链表的头部插入，而java 1.8会在链表的尾部插入

在java 1.8中，Entry被Node替代

1. Hashmap，linkedhashmap，treemap有什么区别

LinkedHashMap 保存了记录的插入顺序，在用 Iterator 遍历时，先取到的记录肯定是先插入的；遍历比 HashMap 慢；

TreeMap 实现 SortMap 接口，能够把它保存的记录根据键排序（默认按键值升序排序，也可以指定排序的比较器）

1. Hashmap，linkedhashmap，treemap的使用场景

HashMap：在 Map 中插入、删除和定位元素时；

TreeMap：在需要按自然顺序或自定义顺序遍历键的情况下；

LinkedHashMap：在需要输出的顺序和输入的顺序相同的情况下。

1. Hashmap和hashtable的区别

①、HashMap 是线程不安全的，HashTable 是线程安全的；

②、由于线程安全，所以 HashTable 的效率比不上 HashMap；

③、HashMap最多只允许一条记录的键为null，允许多条记录的值为null，而 HashTable不允许；

④、HashMap 默认初始化数组的大小为16，HashTable 为 11，前者扩容时，扩大两倍，后者扩大两倍+1；

⑤、HashMap 需要重新计算 hash 值，而 HashTable 直接使用对象的 hashCode

1. Java中另一个线程安全的与hashmap类似的类是什么，同样是线程安全，它与hashtable有什么不同

ConcurrentHashMap 类（是 Java并发包 java.util.concurrent 中提供的一个线程安全且高效的 HashMap 实现）。

HashTable 是使用 synchronize 关键字加锁的原理（就是对对象加锁）；

而针对 ConcurrentHashMap，在 JDK 1.7 中采用 分段锁的方式；JDK 1.8 中直接采用了CAS（无锁算法）+ synchronized。

1. Hashmap和Concurrenthashmap有什么区别

除了加锁，原理上无太大区别。另外，HashMap 的键值对允许有null，但是ConCurrentHashMap 都不允许。

1. 为什么Concurrenthashmap比hashtable效率高

HashTable 使用一把锁（锁住整个链表结构）处理并发问题，多个线程竞争一把锁，容易阻塞；

ConcurrentHashMap

JDK 1.7 中使用分段锁（ReentrantLock + Segment + HashEntry），相当于把一个 HashMap 分成多个段，每段分配一把锁，这样支持多线程访问。锁粒度：基于 Segment，包含多个 HashEntry。

JDK 1.8 中使用 CAS + synchronized + Node + 红黑树。锁粒度：Node（首结

点）（实现 Map.Entry）。锁粒度降低了。

1. Concurrenthashmap在jdk1.8中的实现（1.7和1.8的区别）

JDK 1.7 中，采用分段锁的机制，实现并发的更新操作，底层采用数组+链表的存储结构，包括两个核心静态内部类 Segment 和 HashEntry。

①、Segment 继承 ReentrantLock（重入锁） 用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶；

②、HashEntry 用来封装映射表的键-值对；

③、每个桶是由若干个 HashEntry 对象链接起来的链表

JDK 1.8 中，采用Node + CAS + Synchronized来保证并发安全。取消类 Segment，直接用 table 数组存储键值对；当 HashEntry 对象组成的链表长度超过 TREEIFY\_THRESHOLD 时，链表转换为红黑树，提升性能。底层变更为数组 + 链表 + 红黑树。

1. Concurrenthashmap在jdk1.8中为什么要使用内置锁Synchronized来代替重入锁

①、粒度降低了；

②、JVM 开发团队没有放弃 synchronized，而且基于 JVM 的 synchronized 优化空间更大，更加自然。

③、在大量的数据操作下，对于 JVM 的内存压力，基于 API 的 ReentrantLock 会开销更多的内存。

1. Concurrenthashmap简介

**①、重要的常量：**

private transient volatile int sizeCtl;

当为负数时，-1 表示正在初始化，-N 表示 N - 1 个线程正在进行扩容；

当为 0 时，表示 table 还没有初始化；

当为其他正数时，表示初始化或者下一次进行扩容的大小。

**②、数据结构：**

Node 是存储结构的基本单元，继承 HashMap 中的 Entry，用于存储数据；

TreeNode 继承 Node，但是数据结构换成了二叉树结构，是红黑树的存储结构，用于红黑树中存储数据；

TreeBin 是封装 TreeNode 的容器，提供转换红黑树的一些条件和锁的控制。

**③、存储对象时（put() 方法）：**

如果没有初始化，就调用 initTable() 方法来进行初始化；

如果没有 hash 冲突就直接 CAS 无锁插入；

如果需要扩容，就先进行扩容；

如果存在 hash 冲突，就加锁来保证线程安全，两种情况：一种是链表形式就直接遍历

到尾端插入，一种是红黑树就按照红黑树结构插入；

如果该链表的数量大于阀值 8，就要先转换成红黑树的结构，break 再一次进入循环

如果添加成功就调用 addCount() 方法统计 size，并且检查是否需要扩容。

**④、扩容方法 transfer()：默认容量为 16，扩容时，容量变为原来的两倍。**

helpTransfer()：调用多个工作线程一起帮助进行扩容，这样的效率就会更高。

**⑤、获取对象时（get()方法）：**

计算 hash 值，定位到该 table 索引位置，如果是首结点符合就返回；

如果遇到扩容时，会调用标记正在扩容结点 ForwardingNode.find()方法，查找该结点，匹配就返回；

以上都不符合的话，就往下遍历结点，匹配就返回，否则最后就返回 null。

1. Concurrenthashmap的并发度是什么

程序运行时能够同时更新 ConccurentHashMap 且不产生锁竞争的最大线程数。默认为 16，且可以在构造函数中设置。

当用户设置并发度时，ConcurrentHashMap 会使用大于等于该值的最小2幂指数作为实际并发度（假如用户设置并发度为17，实际并发度则为32）