**在JDK1.6，JDK1.7中，HashMap采用位桶+链表实现，即使用链表处理冲突，**同一**hash**值的链表都存储在一个链表里。但是当位于一个桶中的元素较多，即**hash**值相等的元素较多时，通过key值依次查找的效率较低。**而JDK1.8中，HashMap采用位桶+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树**，这样大大减少了查找时间

**简单说下HashMap的实现原理：**

**首先有一个每个元素都是链表（可能表述不准确）的数组，当添加一个元素（key-value）时，就首先计算元素key的hash值，以此确定插入数组中的位置，但是可能存在同一hash值的元素已经被放在数组同一位置了，这时就添加到同一hash值的元素的后面，他们在数组的同一位置，但是形成了链表，同一各链表上的Hash值是相同的，所以说数组存放的是链表。而当链表长度太长时，链表就转换为红黑树，这样大大提高了查找的效率。**

2当链表数组的容量超过初始容量的0.75时，再散列将链表数组扩大2倍，把原链表数组的搬移到新的数组中。

在经过resize这个方法之后，元素的位置要么就是在原来的位置，要么就是在原来的位置移动2次幂的位置上。 源码上的注释也是可以翻译出来的

//The next size value at which to resize (capacity \* load factor).

newThr = oldThr << 1; // double threshold

在HashMap刚初始化的时候，使用默认的构造初始化，会返回一个空的table，并且 thershold为0，因此第一次扩容的时候默认值就会是16. 同时再去计算thershold = DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY = 16\*0.75 = 12.

如果说指定初始容量的初始HashMap的时候，那么这时候计算这个threshold的时候就变成了 threshold = DEFAULT\_LOAD\_FACTOR \* threshold(当前的容量)

如果HashMap不是第一次扩容，已经扩容过了，那么每次table的容量

threshold也会变成原来的2倍。

之前看1.7的源码的时候，是没有这个红黑树的，而是在1.8 之后做了相应的优化。 使用的是2次幂的扩展(指长度扩为原来2倍)。 而且在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，这样子他就剩下了计算hash的时间了。

https://www.liangzl.com/get-article-detail-136823.html

部分源码：

{

https://www.jianshu.com/p/17177c12f849

//默认初始容量为16，这里这个数组的容量必须为2的n次幂。

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // aka 16

1左移4位，即为10000（二进制）=2^4=16

//最大容量为2的30次方

static final int MAXIMUM\_CAPACITY = 1 << 30;

//默认加载因子

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

//以Node<K,V>为元素的数组，也就是上图HashMap的纵向的长链数组，起长度必须为2的n次幂

transient Node<K,V>[] table;

**Node<K,V>[] table**，他是整个HashMap的组成子元素。

**hash**字段用来定位桶的索引位置，**key**和**value**就是我们的数据内容，需要注意的是，我们的**key**是**final**的，也就是不允许更改，这也好理解，因为**HashMap**使用**key**的**hashCode**来寻找桶的索引位置，一旦**key**被改变了，那么**key**的**hashCode**很可能就会改变了，所以随意改变**key**会使得我们丢失记录（无法找到记录）。**next**字段指向链表的下一个节点。

resize机制：

HashMap的扩容机制就是重新申请一个容量是当前的2倍的桶数组，然后将原先的记录逐个重新映射到新的桶里面，然后将原先的桶逐个置为null使得引用失效。

在resize操作的时候会造成线程不安全。

**resize**核心**transfer**方法的功能是将原来的记录重新计算在新桶的位置，然后迁移过去。

static class Node<K,V> implements Map.Entry<K,V> {

final int hash; //每个储存元素key的哈希值

final K key; //key

V value; //value

Node<K,V> next; //链表下一个node

Node(int hash, K key, V value, Node<K,V> next) {

this.hash = hash;

......

}

public final K getKey() { return key; }

public final V getValue() { return value; }

在jdk1.7中，当HashMap创建的时候，table这个数组确实会初始化；但是到了jdk1.8中，我们观察上面四个构造函数，除了第四个构造函数调用了resize()外，其他三个常用的构造函数都没有与table初始化相关的迹象，而真正table初始化的地方是在我们上面讲的putVal()方法中，即首次向HashMap添加元素时，调用resize()创建并初始化了一个table数组。

如果(e.hash & oldCap) == 0，则 newTab[j] = loHead = e = oldTab[j]，即索引位置没变。反之 (e.hash & oldCap) != 0, newTab[j + oldCap] = hiHead = e = oldTab[j],也就是说，此时把原数组[j]位置上的桶移到了新数组[j+原数组长度]的位置上了。

移位运算就是—— **a % (2^n) 等价于 a & (2^n - 1)**，也即是位运算与取模运算的转化，且位运算比取模运算具有更高的效率，这也是为什么HashMap中数组长度要求为2^n的原因。

}

源码中的数据域

**加载因子（默认0.75）：为什么需要使用加载因子，为什么需要扩容呢**？**因为如果填充比很大，说明利用的空间很多，如果一直不进行扩容的话，链表就会越来越长，这样查找的效率很低，因为链表的长度很大（当然最新版本使用了红黑树后会改进很多），扩容之后，将原来链表数组的每一个链表分成奇偶两个子链表分别挂在新链表数组的散列位置，这样就减少了每个链表的长度，增加查找效率**

默认负载因子0.75是**是对空间和时间(纵向横向)效率的一个平衡选择**

HashMap本来是以空间换时间，所以填充比没必要太大。但是填充比太小又会导致空间浪费。如果关注内存，填充比可以稍大，如果主要关注查找性能，填充比可以稍小。

get(key)方法时获取key的**hash**值，计算**hash**&(n-1)得到在链表数组中的位置first=tab[**hash**&(n-1)],先判断first的key是否与参数key相等，不等就遍历后面的链表找到相同的key值返回对应的Value值即可。

HashMap的构造方法有4种，主要涉及到的参数有，指定初始容量，指定填充比和用来初始化的**Map**

null key总是放在Entry数组的第一个元素**.**

**HashMap：**

查询和插入速度极快，但是线程不安全，在多线程情况下在扩容的情况下可能会形成闭环链路，耗光cpu资源。

HashMap允许null key和null value，而hashtable不允许。

Hashtable的方法是Synchronize的，而HashMap不是，在多个线程访问Hashtable时，不需要自己为它的方法实现同步，而HashMap 就必须为之提供外同步。

Hashtable继承自Dictionary类。

TreeMap和TreeSet一样数据都是有序的使用二叉树排序。**TreeMap**基于红黑树（**可以在O(log n)时间内做查找，插入和删除**

）实现。

Synchronized Map与HashTable差别不大，也是在并发中作类似的操作，两者的唯一区别就是Synchronized Map没被遗弃，它可以通过使用Collections.synchronizedMap()来包装Map作为同步容器使用

**>>>**为什么线程不安全，为什么会形成闭环链路？

**https://www.jianshu.com/p/e2f75c8cce01**

**1.put**的时候多线程造成数据不一致，丢失的等情况

**2.resize**的时候，在从老表迁移到新的**hash**表时出现闭环链路。

如果在取链表的时候从头开始取（现在是从尾部开始取）的话，则可以保证节点之间的顺序，那样就不存在这样的问题了。

高并发下建议：使用并发包下的java.util.concurrent.ConcurrentHashMap，ConcurrentHashMap实现了更高级的线程安全；  
或者使用synchronizedMap() 同步方法包装 HashMap object，得到线程安全的Map，并在此Map上进行操作。

**LinkedHashMap**属于**HashMap**的子类，与**HashMap**的区别在于**LinkedHashMap**保存了记录插入的顺序。**TreeMap**实现了**SortedMap**接口，**TreeMap**有能力对插入的记录根据**key**排序，默认按照升序排序，也可以自定义比较强，在使用**TreeMap**的时候，**key**应当实现**Comparable**。

java8的HashMap在java7的基础上增加了红黑树这种数据结构，使得在桶里面查找数据的复杂度从O(n)降到O(logn)，当然还有一些其他的优化，比如resize的优化等。

**LinkedHashMap：**

基本和HashMap实现类似，多了一个链表来维护元素插入的顺序，因此维护的效率会比HashMap略低。但是因为有链表的存在，遍历效率会高于HashMap。

**ConcurrentHashMap：**

线程安全，而且采用分段锁的方式进行数据同步，因此相对于Hashtable来说，效率要高。但是因为引入了段的概念，所以每次元素插入或者获取，需要进行两次哈希算法，第一次确定到该元素位于哪一段，第二次才能真正确定到元素位置。因此效率会低于HashMap。不过在多线程情况下，这种性能的牺牲换取数据安全是非常值得的。因此在多线程的情况下应该首选ConcurrentHashMap。

ConcurrentMap：不可以有null键，线程安全，原子操作。

ConcurrentMap继承了ReentrantLock 类，所以本身是一个锁。当多线程对ConcurrentHashMap 操作时，不是完全锁住map， 而是锁住相应的segment 。这样提高了并发效率。缺点：当遍历ConcurrentMap中的元素时，需要获取所有的segment 的锁，使用遍历时慢。锁的增多，占用了系统的资源。使得对整个集合进行操作的一些方法（例如 size() 或 isEmpty() ）的实现更加困难，因为这些方法要求一次获得许多的锁，并且还存在返回不正确的结果的风险。

ConcurrentHashMap的设计有点特别，表现在多个线程操作上。它不用做外的同步的情况下默认同时允许16个线程读和写这个Map容器。因为其内部的实现剥夺了锁，使它有很好的扩展性。

为什么 **ConcurrentHashMap**在多线程下效率更高？

**HashTable**使用一把锁处理并发问题，当有多个线程访问时，需要多个线程竞争一把锁，导致阻塞。

**ConcurrentHashMap**则使用分段，相当于把一个**HashMap**分成多个，然后每个部分分配一把锁，这样就可以支持多线程访问。

在并发编程中，证明一个类是线程安全的很难，所以我们经常试着去证明类是线程不安全，证明的方法一般有如下几种: 1. 对象不能安全发布，构造过程逃逸； 2. 内存的可见性，内容不能及时发布； 3. 操作不是原子的； 4. 读写不能同步； 5. 存在死锁的可能性；

**<https://blog.csdn.net/yiifaa/article/details/77072622>**