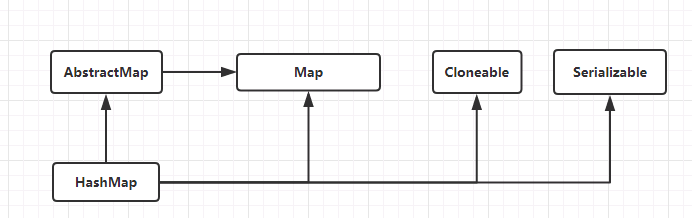
# HashMap

## 继承体系



实现Serializable接口，可以序列化

实现Cloneable接口，可以克隆

java.util.AbstractMap<K, V>接口：这个类实现了java.util.Map,提供了Map的骨干实现，从而最大限度的减少了实现map接口所需要的工作；

实现不可修改的映射时，程序员只需要扩展此类并提供 entrySet方法的实现即可 ，这个方法返回映射关系的视图

实现可修改的映射时，程序员还必须实现put方法，（在AbstractMap中put方法抛出的是不支持操作异常）并且由 entrySet().iterator() 所返回的[迭代器](https://baike.baidu.com/item/%E8%BF%AD%E4%BB%A3%E5%99%A8)必须另外实现其 remove 方法。

HashMap：基于哈希表的Map接口实现的，**允许key和value为null**,不能保证映射的顺序，而且不能保证在哈希表中的顺序是一直不变的

HashMap不是同步的，如果多线程同时访问了HashMap,而且其中一个线程修改了映射，则他必须保持外部同步，以防止对映射进行意外的非同步访问，如下：

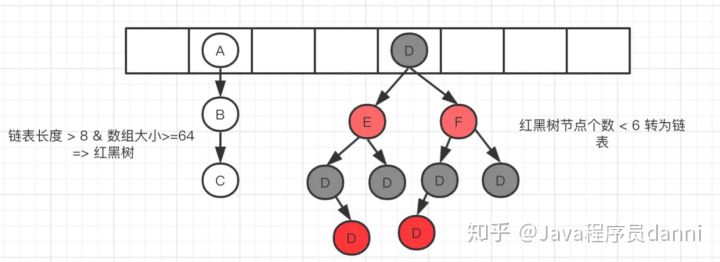
Map m = Collections.synchronizedMap(new HashMap(...));

使用Collections.synchronizedMap来保证hashMap的同步访问

## 内部结构

Jdk1.7:采用数组+链表的形式

Jdk1.8:采用数组+链表+红黑树的形式，当链表长度大于8时并且数组长度大于等于64时，将链表转换为红黑树，当红黑树节点数小于6时，将红黑树转换为链表



## 类成员变量

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // aka 16 数组初始化长度，也就是使用无参构造函数时数组的初始化长度，如果自己传入初始大小k，初始化大小为 大于k的 2的整数次方，例如如果传10，大小为16

在tableSizeFor方法中实现的获取输入参数的最近的2的整数次方

static final int MAXIMUM\_CAPACITY = 1 << 30;数组的最大容量，必须是2的幂次方

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;加载因子

static final int TREEIFY\_THRESHOLD = 8;添加时，每个数组位置上，当存放的节点的数量超过这个之值并且当前结构为链表，就会将当前的链表结构转换为红黑树结构

static final int UNTREEIFY\_THRESHOLD = 6; 删除时，每个数组位置上，当存放的节点的数量小于这个之值并且当前数据结构是红黑树时，将红黑树转换为链表

static final int MIN\_TREEIFY\_CAPACITY = 64;数组长度超过这个值时，添加节点数超过转换红黑数结构的阈值8才会转换为红黑树

transient Node<K,V>[] table;节点数组，hashmap通过将创建的节点存放在这个数组上

transient Set<Map.Entry<K,V>> entrySet;

transient int size; 存在的key-vlaue键值对数

transient int modCount;修改的次数

int threshold;下一个调整大小的值（容量\*加载因子）

final float loadFactor; 接收加载因子的值

## 初始化

在使用无参构造函数时，默认数组长度为16，加载因子为0.75；如果自己传入数组初始长度，这个初始长度为大于传入值的2的整数次方，例如传入10，那就通过方法**tableSizeFor**

计算出初始大小16，

static final int tableSizeFor(int cap) {

int n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

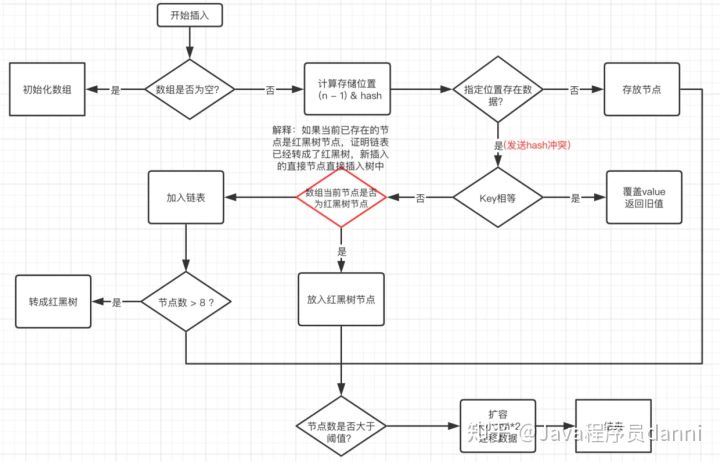
return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}

算法就是让初始二进制右移1，2，4，8，16位，分别与自己异或，把高位第一个为1的数通过不断右移，把高位为1的后几位全变为1，111111 + 1 = 1000000

## Put方法

### P方法的流程



1. 判断数组是否为空，为空进行初始化;
2. 不为空，计算 k 的 hash 值，通过(n - 1) & hash计算应当存放在数组中的下标 index;
3. 查看 table[index] 是否存在数据，没有数据就构造一个Node节点存放在 table[index] 中；
4. 存在数据，说明发生了hash冲突(存在二个节点key的hash值一样), 继续判断key是否相等，相等，用新的value替换原数据(onlyIfAbsent为false)；
5. 如果不相等，判断当前节点类型是不是树型节点，如果是树型节点，创造树型节点插入红黑树中；
6. 如果不是树型节点，创建普通Node加入链表中；判断链表长度是否大于 8， 大于的话链表转换为红黑树；
7. 插入完成之后判断当前节点数是否大于阈值，如果大于开始扩容为原数组的二倍。

### 计算key的hash值

调用hash方法

**static** **final** **int** hash(Object key) {

**int** h;

**return** (key == **null**) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

当key为null时，hash值为0

Key不会null时，向调用hashCode方法返回key本来的hashcode值，然后让hashcode的高16位和低16位进行异或操作；

这样做的目的有2点

1. 一定要尽可能降低hash碰撞，越分散越好；
2. Hash方法调用比较频繁, 因此采用位运算来实现高效操作

key.hashCode()函数调用的是key键值类型自带的哈希函数，返回int型散列值。int值范围为-2147483648~2147483647，前后加起来大概40亿的映射空间。只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。

右位移16位，正好是32bit的一半，自己的高半区和低半区做异或，就是为了混合原始哈希码的高位和低位，以此来加大低位的随机性。而且混合后的低位掺杂了高位的部分特征，这样高位的信息也被变相保留下来。

### resize()方法

put方法中会调用

1.在数组没有初始化时，调用这个方法来初始化数组

2.在put之后，判断当前put key的数量是否超过了扩容的阈值，如果超过了就调用来扩容

根据数组是否初始化计算出新数组的初始长度，创建数组，如果是初始化，那么就直接放回

如果是扩容那么就执行后续的扩容操作（扩容长度为原来的2倍）

扩容

就是将原数组上的数据转移到新数组上；

遍历原数组，取出root节点，判断节点

如果只有一个节点，那么获取这个节点value的hash值和（数组长度-1）进行或运算，结果就是新数组上的下标位置

如果节点类型是TreeNode，那么调用TreeNode的split方法

遍历红黑树结构，将树节点的以链表的形式存储到新的引用中，并计算出树节点的数量；

计算新下标：将hash值和原数组长度进行&或运算，如果结果为0，那么节点在新数组的下标位置就为原下标位置；不为0，那么就是原下标位置加上原数组之和

遍历树结构，将原链表从头开始依次转移指向新的引用对象，最终链表loHead或者hiHead指向链表的头，loTail或者hiTail指向链表的尾

loHead,loTail指向下标位置不变的树结构

**if** ((e.hash & bit) == 0) {

**if** ((e.prev = loTail) == **null**)

loHead = e;

**else**

loTail.next = e;

loTail = e;

++lc;

}

hiHead,hiTail指向下标位置要改变的树结构

**if** ((e.prev = hiTail) == **null**)

hiHead = e;

**else**

hiTail.next = e;

hiTail = e;

++hc;

如果数量不超过*UNTREEIFY\_THRESHOLD*，那么就要将当前结构转换为链表（树节点转换为Node），

tab[index] = loHead.untreeify(map);转换为链表的过程

否则就将新引用上的结构（树节点的链表）转换为树结构，

loHead.treeify(tab);转换为树结构的过程

如果节点是链表

遍历链表，将原数组位置上的链表转移到新的链表对象上，然后在新的位置上赋值

计算新下标：将hash值和原数组长度进行&或运算时如果结果为0，

那么就是在新数组上的位置下标就是原来的位置下标；不为0，

那么信息新下标就是原来位置下标加上原数组长度之和（结果满足hash&(newCap-1)，不用逐个计算位置下标）

遍历链表，将原链表从头开始依次转移指向新的引用对象，最终链表loHead或者hiHead指向链表的头，loTail或者hiTail指向链表的尾

Node<K,V> loHead = null, loTail = null;指向下标不变的链表

Node<K,V> hiHead = null, hiTail = null;指向下标要改变的链表（原位置下标加上原数组长度）

if (loTail == null)

loHead = e;

else

loTail.next = e;

loTail = e;

将新链表赋值到指定位置

newTab[j + oldCap] = hiHead;下标为原下标加上原数组长度

newTab[j] = loHead;下标不变

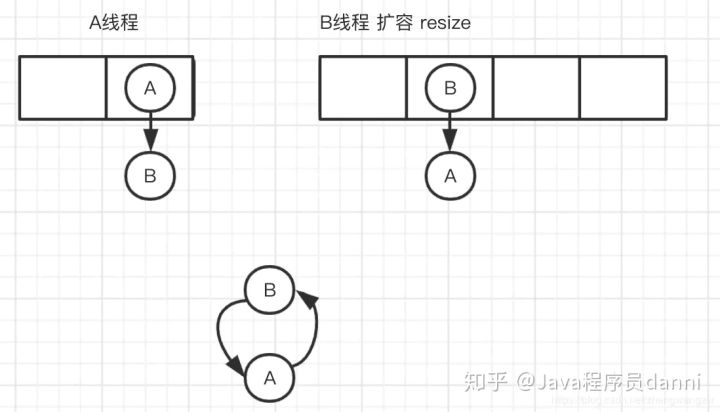
## 1.8相比之前的优化

1. 数组+链表改成了数组+链表或红黑树；

1.8使用红黑树：防止发生hash冲突，链表长度过长，将时间复杂度由O(n)降为O(logn);

1. 链表的插入方式从头插法改成了尾插法，简单说就是插入时，如果数组位置上已经有元素，1.7将新元素放到数组中，原始节点作为新节点的后继节点，1.8遍历链表，将元素放置到链表的最后；

因为1.7头插法扩容时，头插法会使链表发生反转，多线程环境下会产生环；  
A线程在插入节点B，B线程也在插入，遇到容量不够开始扩容，重新hash，放置元素，采用头插法，后遍历到的B节点放入了头部，这样形成了环，如下图所示：



1. 扩容的时候1.7需要对原数组中的元素进行重新hash定位在新数组的位置，1.8采用更简单的判断逻辑，位置不变或索引+旧容量大小；
2. 在插入时，1.7先判断是否需要扩容，再插入，1.8先进行插入，插入完成再判断是否需要扩容；

## HashMap是线程安全的吗

不是，

在多线程环境下，1.7 会产生死循环、数据丢失、数据覆盖的问题，

1.8 中会有数据覆盖的问题，以1.8为例，看👇的代码，当A线程判断index位置为空后正好挂起，B线程开始往index位置的写入节点数据，这时A线程恢复现场，执行赋值操作，就把A线程的数据给覆盖了；还有++size这个地方也会造成多线程同时扩容等问题。

## 平常怎么解决hashmap线程不安全的问题

Java中有HashTable、Collections.synchronizedMap、以及ConcurrentHashMap可以实现线程安全的Map。

HashTable是直接在操作方法上加synchronized关键字，锁住整个数组，锁粒度比较大，Collections.synchronizedMap是使用Collections集合工具的内部类，通过传入Map封装出一个SynchronizedMap对象，内部定义了一个对象锁，方法通过对象锁实现线程安全；ConcurrentHashMap使用分段锁，降低了锁粒度，让并发度大大提高

链表转红黑树是链表长度达到阈值，这个阈值是多少？

阈值是8，红黑树转链表阈值为6

经过计算，在hash函数设计合理的情况下，发生hash碰撞8次的几率为百万分之6，概率说话，因为8够用了（也就是通过计算碰撞发生次数的概率统计得到的），至于为什么转回来是6，因为如果hash碰撞次数在8附近徘徊，会一直发生链表和红黑树的互相转化，为了预防这种情况的发生，设置为6。