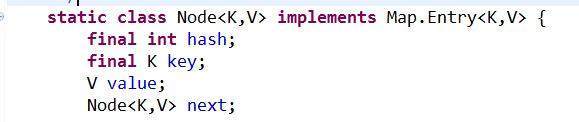
# 1.HashMap用什么数据结构实现的？

答:数组。什么样的数组？答：Node[] table这样的。Node是什么？答：看下图：



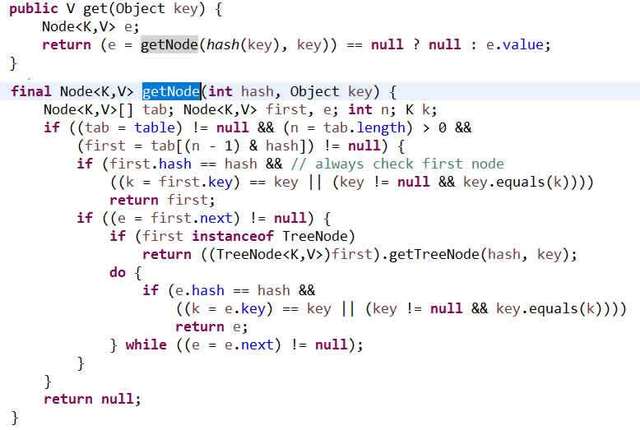
也就是说这个数组每个元素都是个单向链表。

# 2.HashMap的get过程是?

答:先得到key的hash值，再把这个hash值与length-1按位与（取余），得到table数组的下标。取出这个下标值的key，与传入的key比较，如果相同那就是这个了。如果不同呢，那就沿着这个单向链表向后找，直到找到或找到结束也找不到。

等等，这个hash值与length-1按位与就是等同取余么？为什么？

答:是的，因为这个length是有特点的，是2的n次方，length-1就是二进制全部为1。那么一个数的二进制按位与这些全为1的位，就会得到与这些全为1的位对齐的另一个数，如（15&6=1111&0110=6=0110）。而这个数正好是取2的n次的余数（6%15=6=15&6）。 至于为什么这个length是2的n次方，下面的问题会回答 。贴下源码:



# 3.HashMap初始化传入的容量参数的值就是HashMap实际分配的空间么？

答：不是。那是什么呢？是比传入容量参数值大的最小的2的n次方，比如传入6，实际分配8。看下计算容量的源码。虽然这里源码计算的值只是threshold，而没有代码说明它是实际值，但本文最后的源码可以说明它就是实际值。



看到这个这个tableSizeFor方法,那么规则的1、2、4、8、16，是不是有点蒙。算法长成这样，是画画么？先了解下基础知识，一个整形由4个字节构成，那它转为二进制应该是32(4\*8)位。但这是有符号的，其实正数和负数各占一半，也就是说各有2的31次方，最高位是符号位。我们这里是无符号操作就不用管这个符号了。上面的无符号右移16位就可以覆盖32位，16就是这么来的。其实这5个>>>(无符号右移)是为了把cap-1的最高二进制位及其后面的位都置成1。

最高二进制位是什么意思呢？ 一个整形转为二进制后，最左边的那位一定是1，前面补的0不能算哈，这个最左的1就是最高二进制位。

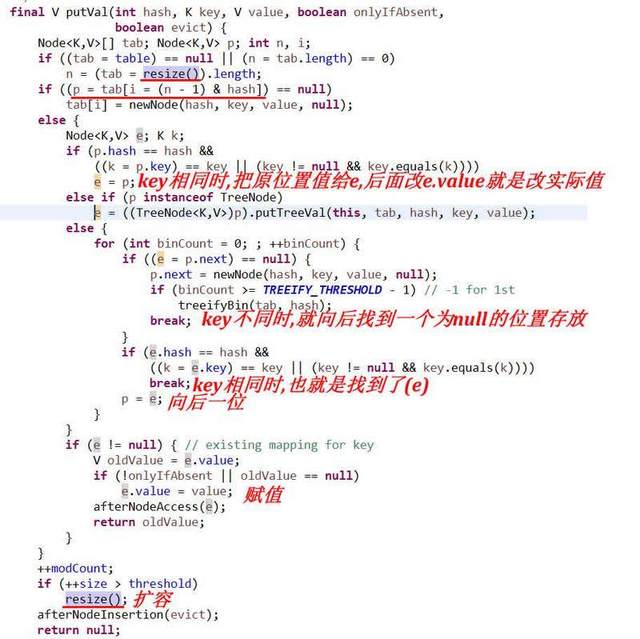
按上面的算法，第一次无符号右移1位再和原二进制按位或，至少把最左边和次最左边的两个位置为1了吧。那第二次再只移一位就没意义了，因为前两位都是1了，不需计算了，所以第二次是右移2位，这样就有4个1了。同理4位变8位，8位变16位，最后16位变32位，实际是只有31位（最前的是符号位）。这样就把cap-1变为一个从其二进制形式的最高位到0位全为1的二进制数了，而这个二进制数正好是比cap-1大一点的那个2的n次方的数-1（如2的3次方减1就是1000-1=0111=7）。

最后一行再+1就是2的n次方了(7+1=8=2的3次方).而第一行的n=cap-1,这也就是防止传入参数就是2的n次的情况（如8的二进制是1000），再把这个最高二进制位的后面都置成1，最后再加1就变成16了。这就不是与传入参数最接近的2的n次方了。

这里也就回答了为什么map.length是2的n次方了，后面扩容的机制会证明它总是保持着2的n次方。

# 4.HashMap扩容机制是什么，什么时候扩，每次扩多少？

答:在put时，容量不够用的时候。因为每个元素都是一个单向链表，所以map里放的实际数量总是大于等于申请的空间。那么容量不够用就是实际存放的数量等于申请的空间大小的时候。看下源码put->putVal：



上面我们已经知道，HashMap的实际容量是2的n次方了。那么扩容也就是原来的容量\*2，也就是扩到2的n+1次方。构造函数里有个加载因子（loadFactor),我曾经以为扩容就是扩这个数(默认0.75)+1倍，看到这里的源码才发现错了。这个加载因子只是计算下一个2的n次用的中间变量。默认的是0.75，那计算下一个2的n次方的基准值就是old\*2\*0.75,那么下一个2的n次方就是比这个基准值大的最少值（old\*2）。如下图：



考点：

1.hashing的概念

2.HashMap中解决碰撞的方法

3.equals()和hashCode()的应用，以及它们在HashMap中的重要性

4.不可变对象的好处

5.HashMap多线程的条件竞争

6.重新调整HashMap的大小

常见面试问题：

# 1.“你知道HashMap的工作原理吗？” “你知道HashMap的get()方法的工作原理吗？”

HashMap基于hashing原理，我们通过put()和get()方法储存和获取对象。当我们将键值对传递给put()方法时，它调用键对象的hashCode()方法来计算hashcode，让后找到bucket位置来储存Entry对象。当获取对象时，通过键对象的equals()方法找到正确的键值对，然后返回值对象。HashMap使用链表来解决碰撞问题，当发生碰撞了，对象将会储存在链表的下一个节点中。 HashMap在每个链表节点中储存键值对对象。

这里关键点在于指出，HashMap是在bucket中储存键对象和值对象，作为Map.Entry。这一点有助于理解获取对象的逻辑。

# 2.我们能否让HashMap同步？

HashMap可以通过下面的语句进行同步：  
Map m = Collections.synchronizeMap(hashMap);

# 3.什么是HashSet？

HashSet实现了Set接口，它不允许集合中有重复的值，当我们提到HashSet时，第一件事情就是在将对象存储在HashSet之前，要先确保对象重写equals()和hashCode()方法，这样才能比较对象的值是否相等，以确保set中没有储存相等的对象。如果我们没有重写这两个方法，将会使用这个方法的默认实现。

public boolean add(Object o)方法用来在Set中添加元素，当元素值重复时则会立即返回false，如果成功添加的话会返回true。

# 4.“你用过HashMap吗？” “什么是HashMap？你为什么用到它？”

HashMap实现了Map接口，Map接口对键值对进行映射。Map中不允许重复的键。Map接口有两个基本的实现，HashMap和TreeMap。TreeMap保存了对象的排列次序，而HashMap则不能。HashMap存储的是键值对，允许键和值为null。HashMap是非synchronized的，但collection框架提供方法能保证HashMap synchronized，这样多个线程同时访问HashMap时，能保证只有一个线程更改Map。

public Object put(Object Key,Object value)方法用来将元素添加到map中。

# 5.HashSet与HashMap的区别？



# 6.关于HashMap中的碰撞探测(collision detection)以及碰撞的解决方法？

当两个对象的hashcode相同会发生什么？

因为hashcode相同，所以它们的bucket位置相同，‘碰撞’会发生。因为HashMap使用链表存储对象，这个Entry(包含有键值对的Map.Entry对象)会存储在链表中。

# 7.如果两个键的hashcode相同，你如何获取值对象？

当我们调用get()方法，HashMap会使用键对象的hashcode找到bucket位置，找到bucket位置之后，会调用keys.equals()方法去找到链表中正确的节点，最终找到要找的值对象。

注意：面试官会问因为你并没有值对象去比较，你是如何确定确定找到值对象的？除非面试者直到HashMap在链表中存储的是键值对，否则他们不可能回答出这一题。

一些优秀的开发者会指出使用不可变的、声明作final的对象，并且采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择。

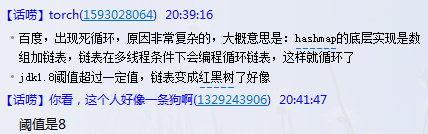
# 8.如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？

除非你真正知道HashMap的工作原理，否则你将回答不出这道题。默认的负载因子大小为0.75，也就是说，当一个map填满了75%的bucket时候，和其它集合类(如ArrayList等)一样，将会创建原来HashMap大小的两倍的bucket数组，来重新调整map的大小，并将原来的对象放入新的bucket数组中。这个过程叫作rehashing，因为它调用hash方法找到新的bucket位置。

# 9.你了解重新调整HashMap大小存在什么问题吗？

当多线程的情况下，可能产生条件竞争。

**当重新调整HashMap大小的时候，确实存在条件竞争，因为如果两个线程都发现HashMap需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免尾部遍历(tail traversing)。如果条件竞争发生了，那么就死循环了。这个时候，你可以质问面试官，为什么这么奇怪，要在多线程的环境下使用HashMap呢？：）-------未理解**

****

注意，jdk1.8阈值是8，面试时被问到过！

# 10.为什么String, Interger这样的wrapper类适合作为键？

String, Interger这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。如果你可以仅仅通过将某个field声明成final就能保证hashCode是不变的，那么请这么做吧。因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能。

# 11.我们可以使用自定义的对象作为键吗？

这是前一个问题的延伸。当然你可能使用任何对象作为键，只要它遵守了equals()和hashCode()方法的定义规则，并且当对象插入到Map中之后将不会再改变了。如果这个自定义对象时不可变的，那么它已经满足了作为键的条件，因为当它创建之后就已经不能改变了。

# 12.我们可以使用CocurrentHashMap来代替Hashtable吗？

这是另外一个很热门的面试题，因为ConcurrentHashMap越来越多人用了。我们知道Hashtable是synchronized的，但是ConcurrentHashMap同步性能更好，因为它仅仅根据同步级别对map的一部分进行上锁。ConcurrentHashMap当然可以代替HashTable，但是HashTable提供更强的线程安全性。

# 13.hashmap的存储过程？

HashMap内部维护了一个存储数据的Entry数组，HashMap采用链表解决冲突，每一个Entry本质上是一个单向链表。当准备添加一个key-value对时，首先通过hash(key)方法计算hash值，然后通过indexFor(hash,length)求该key-value对的存储位置，计算方法是先用hash&0x7FFFFFFF后，再对length取模，这就保证每一个key-value对都能存入HashMap中，当计算出的位置相同时，由于存入位置是一个链表，则把这个key-value对插入链表头。

 HashMap中key和value都允许为null。key为null的键值对永远都放在以table[0]为头结点的链表中。

# 14.hashMap扩容问题？

扩容是是新建了一个HashMap的底层数组，而后调用transfer方法，将就HashMap的全部元素添加到新的HashMap中（要重新计算元素在新的数组中的索引位置）。 很明显，扩容是一个相当耗时的操作，因为它需要重新计算这些元素在新的数组中的位置并进行复制处理。因此，我们在用HashMap的时，最好能提前预估下HashMap中元素的个数，这样有助于提高HashMap的性能。

HashMap共有四个构造方法。构造方法中提到了两个很重要的参数：初始容量和加载因子。这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中槽的数量（即哈希数组的长度），初始容量是创建哈希表时的容量（从构造函数中可以看出，如果不指明，则默认为16），加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 resize 操作（即扩容）。

默认加载因子为0.75，如果加载因子越大，对空间的利用更充分，但是查找效率会降低（链表长度会越来越长）；如果加载因子太小，那么表中的数据将过于稀疏（很多空间还没用，就开始扩容了），对空间造成严重浪费。如果我们在构造方法中不指定，则系统默认加载因子为0.75，这是一个比较理想的值，一般情况下我们是无需修改的。

# 15.什么是hash，什么是碰撞，什么是equals ？

Hash：是一种信息摘要[算法](http://www.java123.net/v/list-192-1.html" \t "_blank)，它还叫做哈希，或者散列。我们平时使用的MD5,SHA1都属于Hash算法，通过输入key进行Hash计算，就可以获取key的HashCode()，比如我们通过校验MD5来验证文件的完整性。

对于HashCode，它是一个本地方法，实质就是地址取样运算

碰撞：好的Hash算法可以出计算几乎出独一无二的HashCode，如果出现了重复的hashCode，就称作碰撞;

就算是MD5这样优秀的算法也会发生碰撞，即两个不同的key也有可能生成相同的MD5。

HashCode，它是一个本地方法，实质就是地址取样运算；

==是用于比较指针是否在同一个地址；

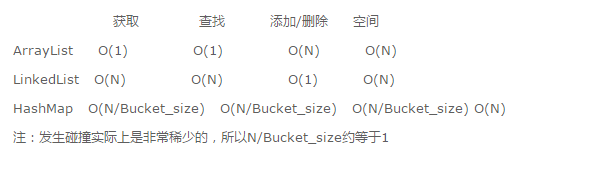
equals与==是相同的。

# 16.如何减少碰撞？

使用不可变的、声明作final的对象，并且采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择

# 17.HashMap的复杂度

HashMap整体上性能都非常不错，但是不稳定，为O(N/Buckets)，N就是以数组中没有发生碰撞的元素。



# 18.为什么HashMap是线程不安全的，实际会如何体现？

第一，如果多个线程同时使用put方法添加元素

假设正好存在两个put的key发生了碰撞(hash值一样)，那么根据HashMap的实现，这两个key会添加到数组的同一个位置，这样最终就会发生其中一个线程的put的数据被覆盖。

第二，如果多个线程同时检测到元素个数超过数组大小\*loadFactor

这样会发生多个线程同时对hash数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给table，也就是说其他线程的都会丢失，并且各自线程put的数据也丢失。且会引起死循环的错误。

# 19.能否让HashMap实现线程安全，如何做？

1、直接使用Hashtable，但是当一个线程访问HashTable的同步方法时，其他线程如果也要访问同步方法，会被阻塞住。举个例子，当一个线程使用put方法时，另一个线程不但不可以使用put方法，连get方法都不可以，效率很低，现在基本不会选择它了。

2、HashMap可以通过下面的语句进行同步：

Collections.synchronizeMap(hashMap);

3、直接使用JDK 5 之后的 ConcurrentHashMap，如果使用Java 5或以上的话，请使用ConcurrentHashMap。

**Collections.synchronizeMap(hashMap);又是如何保证了HashMap线程安全？**

[复制代码](javascript:void(0);)

// synchronizedMap方法

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {

return new SynchronizedMap<>(m);

}

// SynchronizedMap类

private static class SynchronizedMap<K,V>

implements Map<K,V>, Serializable {

private static final long serialVersionUID = 1978198479659022715L;

private final Map<K,V> m; // Backing Map

final Object mutex; // Object on which to synchronize

SynchronizedMap(Map<K,V> m) {

this.m = Objects.requireNonNull(m);

mutex = this;

}

SynchronizedMap(Map<K,V> m, Object mutex) {

this.m = m;

this.mutex = mutex;

}

public int size() {

synchronized (mutex) {return m.size();}

}

public boolean isEmpty() {

synchronized (mutex) {return m.isEmpty();}

}

public boolean containsKey(Object key) {

synchronized (mutex) {return m.containsKey(key);}

}

public boolean containsValue(Object value) {

synchronized (mutex) {return m.containsValue(value);}

}

public V get(Object key) {

synchronized (mutex) {return m.get(key);}

}

public V put(K key, V value) {

synchronized (mutex) {return m.put(key, value);}

}

public V remove(Object key) {

synchronized (mutex) {return m.remove(key);}

}

// 省略其他方法

}

[复制代码](javascript:void(0);)

从源码中看出 synchronizedMap()方法返回一个SynchronizedMap类的对象，而在SynchronizedMap类中使用了synchronized来保证对Map的操作是线程安全的，故效率其实也不高。

# 19.为什么HashTable的默认大小和HashMap不一样？

 前面分析了，Hashtable 的扩容方法是乘2再+1，不是简单的乘2，故hashtable保证了容量永远是奇数，合之前分析hashmap的重算hash值的逻辑，就明白了，因为在数据分布在等差数据集合(如偶数)上时，如果公差与桶容量有公约数 n，则至少有(n-1)/n 数量的桶是利用不到的，故之前的hashmap 会在取模（使用位与运算代替）哈希前先做一次哈希运算，调整hash值。这里hashtable比较古老，直接使用了除留余数法，那么就需要设置容量起码不是偶数（除（近似）质数求余的分散效果好）。

# 20.对key进行Hash计算

在JDK8中，由于使用了红黑树来处理大的链表开销，所以hash这边可以更加省力了，只用计算hashCode并移动到低位就可以了

static final int hash(Object key) {

    int h;

    //计算hashCode，并无符号移动到低位

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

}

# 21.几个常用的哈希码的算法

1：Object类的hashCode.返回对象的内存地址经过处理后的结构，由于每个对象的内存地址都不一样，所以哈希码也不一样。

[复制代码](javascript:void(0);)

public int hashCode() {

int lockWord = shadow$\_monitor\_;

final int lockWordMask = 0xC0000000; // Top 2 bits.

final int lockWordStateHash = 0x80000000; // Top 2 bits are value 2 (kStateHash).

if ((lockWord & lockWordMask) == lockWordStateHash) {

return lockWord & ~lockWordMask;

}

return System.identityHashCode(this);

}

[复制代码](javascript:void(0);)

2：String类的hashCode.根据String类包含的字符串的内容，根据一种特殊算法返回哈希码，只要字符串所在的堆空间相同，返回的哈希码也相同。

[复制代码](javascript:void(0);)

@Override public int hashCode() {

int hash = hashCode;

if (hash == 0) {

if (count == 0) {

return 0;

}

final int end = count + offset;

final char[] chars = value;

for (int i = offset; i < end; ++i) {

hash = 31\*hash + chars[i];

}

hashCode = hash;

}

return hash;

}

[复制代码](javascript:void(0);)

3：Integer类，返回的哈希码就是Integer对象里所包含的那个整数的数值，例如Integer i1=new Integer(100),i1.hashCode的值就是100 。由此可见，2个一样大小的Integer对象，返回的哈希码也一样。

public int hashCode() {

return value;

}

int，char这样的基本类型，它们不需要hashCode.

插入包装类到数组

(1). 如果输入当前的位置是空的，就插进去

(2). 如果当前位置已经有了node，且它们发生了碰撞，则新的放到前面，旧的放到后面，这叫做链地址法处理冲突。

失败的hashCode算法会导致HashMap的性能下降为链表，所以想要避免发生碰撞，就要提高hashCode结果的均匀性。当然，在JDK8中，采用了红黑二叉树进行了处理，这个我们后面详细介绍。

什么是Hash攻击?

 通过请求大量key不同，但是hashCode相同的数据，让HashMap不断发生碰撞，硬生生的变成了SingleLinkedList

 0

|

1 -> a ->b -> c -> d(撞！撞！撞！复杂度由O(1)变成了O(N))

|

2 -> null(本应该均匀分布，这里却是空的)

|

3 -> null

|

4 -> null

这样put/get性能就从O(1)变成了O(N)，CPU负载呈直线上升，形成了放大版DDOS的效果，这种方式就叫做hash攻击。在Java8中通过使用TreeMap，提升了处理性能，可以一定程度的防御Hash攻击。

扩容

(threshold = capacity \* load factor ) < size

它主要有两个步骤：

1. 容量加倍

左移N位，就是2^n，用位运算取代了乘法运算

newCap = oldCap << 1;

newThr = oldThr << 1;

2. 遍历计算Hash

[复制代码](javascript:void(0);)

for (int j = 0; j < oldCap; ++j) {

Node<K,V> e;

//如果发现当前有Bucket

if ((e = oldTab[j]) != null) {

oldTab[j] = null;

//如果这里没有碰撞

if (e.next == null)

//重新计算Hash，分配位置

newTab[e.hash & (newCap - 1)] = e;

//这个见下面的新特性介绍，如果是树，就填入树

else if (e instanceof TreeNode)

((TreeNode<K,V>)e).split(this, newTab, j, oldCap);

//如果是链表，就保留顺序....目前就看懂这点

else { // preserve order

Node<K,V> loHead = null, loTail = null;

Node<K,V> hiHead = null, hiTail = null;

Node<K,V> next;

do {

next = e.next;

if ((e.hash & oldCap) == 0) {

if (loTail == null)

loHead = e;

else

loTail.next = e;

loTail = e;

}

else {

if (hiTail == null)

hiHead = e;

else

hiTail.next = e;

hiTail = e;

}

} while ((e = next) != null);

if (loTail != null) {

loTail.next = null;

newTab[j] = loHead;

}

if (hiTail != null) {

hiTail.next = null;

newTab[j + oldCap] = hiHead;

}

}

}

}

[复制代码](javascript:void(0);)

由此可以看出扩容需要遍历并重新赋值，成本非常高，所以选择一个好的初始容量非常重要。

如何提升性能？

解决扩容损失：如果知道大致需要的容量，把初始容量设置好以解决扩容损失；

比如我现在有1000个数据，需要 1000/0.75 = 1333 ,又 1024 < 1333 < 2048，所以最好使用2048作为初始容量。

2048=Collections.roundUpToPowerOfTwo(1333)

解决碰撞损失：使用高效的HashCode与loadFactor，这个...由于JDK8的高性能出现，这儿问题也不大了。

解决数据结构选择的错误：在大型的数据与搜索中考虑使用别的结构比如TreeMap，这个就是积累了；

JDK8中HashMap的新特性

如果某个桶中的链表记录过大的话（当前是TREEIFY\_THRESHOLD = 8），就会把这个链动态变成红黑二叉树，使查询最差复杂度由O(N)变成了O(logN)。