Hashtable、HashMap、ConcurrentHashMap的区别

# 区别几个方面

### 线程安全:不加锁、全部加锁、部分加锁

### Null值

Hashtable已经是一个过时的类，建议在新的项目中不要再使用Hashtable，一般被ConcurrentHashMap取代。

# 实现原理:Hashtable与HashMap的hash的作用

讨论HashMap和HashTable在数据结构和算法层面，有什么不同。

## 数据结构：

HashMap和HashTable都使用**哈希表**来存储**键值对**。在数据结构上是基本相同的，都创建了一个继承自**Map.Entry**的私有的**内部类Node<K,V>或Entry<K,V>**，每一个**Entry对象**表示**存储在哈希表中的一个键值对**。

一个Entry对象唯一表示一个键值对，有四个属性：

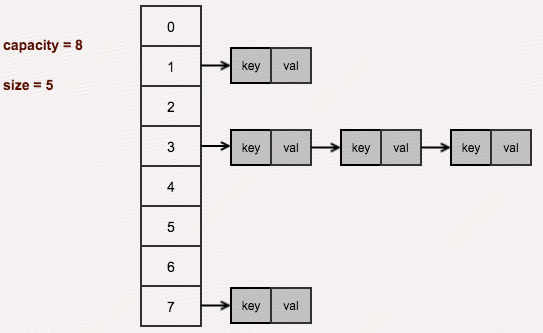
-K key 键对象

-V value 值对象

-int hash 键对象的hash值

-Entry entry 指向链表中下一个Entry对象，可为null，表示当前Entry对象在链表尾部

可以说，有多少个键值对，就有多少个Entry对象，那么在HashMap和HashTable中是怎么存储这些Entry对象，以方便我们快速查找和修改的呢？请看下图。



上图画出的是一个桶数量为8，存有5个键值对的HashMap/HashTable的内存布局情况。可以看到HashMap/HashTable内部创建有一个Entry类型的引用数组，用来表示哈希表，数组的长度，即是哈希桶的数量。而数组的每一个元素都是一个Entry引用，从Entry对象的属性里，也可以看出其是链表的节点，**每一个Entry对象内部又含有另一个Entry对象的引用**。

总结：

HashMap/HashTable内部用**Entry数组**实现哈希表，而对于映射到同一个哈希桶（数组的同一个位置）的键值对，使用**Entry链表**来存储(**解决hash冲突**)。

## 具体源码分析：

### 在HashMap中，存在

**transient Node<K,V>[] table;//对应Entry数组**

HashMap内部封装了一个**Node<K,V>**，其源代码为：

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

final int hash;

final K key;

V value;

Node<K,V> next;

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

this.hash = hash;

this.key = key;

this.value = value;

this.next = next;

}

### 在Hashtable中，存在

**private transient Entry<?,?>[] table;**

Hashtable的内部封装了一个**Entry<K,V>**,其源代码为

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

**final int hash;**

**final K key;**

**V value;**

**Entry<K,V> next;**

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

this.hash = hash;

this.key = key;

this.value = value;

this.next = next;

}

**从源代码可以看出，对于哈希桶的内部表示，Hashtable与HashMap的实现是一致的。**

## 算法

HashMap/HashTable还需要有算法来将给定的键key，映射到确定的**hash桶**（数组位置）。需要有算法在哈希桶内的键值对多到一定程度时，扩充哈希表的大小（数组的大小）。本小节比较这两个类在算法层面有哪些不同。

初始容量大小和每次扩充容量大小的不同。

### HashMap的初始容量：

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

/\*\*

\* Constructs an empty <tt>HashMap</tt> with **the default initial capacity**

\* **(16)** and the default load factor (0.75).

\*/

public HashMap() {

this.loadFactor = DEFAULT\_LOAD\_FACTOR; // all other fields defaulted

}

扩容：**newCap = oldCap << 1**

final Node<K,V>[] **resize**() {

Node<K,V>[] oldTab = table;

int oldCap = (oldTab == null) ? 0 : oldTab.length;

int oldThr = threshold;

int newCap, newThr = 0;

if (oldCap > 0) {

if (oldCap >= MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return oldTab;

}//左移1位，扩大2倍

else if ((**newCap = oldCap << 1**) < MAXIMUM\_CAPACITY &&

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

newThr = oldThr << 1; // double threshold

}

else if (oldThr > 0) // initial capacity was placed in threshold

newCap = oldThr;

else { // zero initial threshold signifies using defaults

newCap = DEFAULT\_INITIAL\_CAPACITY;

newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

}

…….

}

}

return newTab;

}

### Hashtable的初始容量：

public Hashtable() {

**this(11, 0.75f);**

}

public Hashtable(int initialCapacity) {

this(initialCapacity, 0.75f);

}

扩容：rehash，重新对key获取hash值。

@SuppressWarnings("unchecked")

**protected void rehash() {**

int oldCapacity = table.length;

**Entry<?,?>[] oldMap = table;**

// overflow-conscious code

**int newCapacity = (oldCapacity << 1) + 1;//每次扩容为原来的2n+1**

if (newCapacity - MAX\_ARRAY\_SIZE > 0) {

if (oldCapacity == MAX\_ARRAY\_SIZE)

// Keep running with MAX\_ARRAY\_SIZE buckets

return;

newCapacity = MAX\_ARRAY\_SIZE;

}

**Entry<?,?>[] newMap = new Entry<?,?>[newCapacity];**

modCount++;

threshold = (int)Math.min(newCapacity \* loadFactor, MAX\_ARRAY\_SIZE + 1);

**table = newMap;**

for (int i = oldCapacity ; i-- > 0 ;) {

for (Entry<K,V> old = (Entry<K,V>)oldMap[i] ; old != null ; ) {

Entry<K,V> e = old;

old = old.next;

**int index = (e.hash & 0x7FFFFFFF) % newCapacity;**

**e.next = (Entry<K,V>)newMap[index];**

newMap[index] = e;

}

}

}

### 总结

可以看到HashTable默认的初始大小为11，之后每次扩充为原来的2n+1。HashMap默认的初始化大小为16，之后每次扩充为原来的2倍。还有就是如果在创建时给定了初始化大小，那么**HashTable会直接使用你给定的大小**，而**HashMap会将其扩充为2的幂次方大小**。

也就是**说HashTable会尽量使用素数、奇数**。而**HashMap则总是使用2的幂作为哈希表的大小**。**我们知道当哈希表的大小为素数时，简单的取模哈希的结果会更加均匀**（具体证明，见这篇文章），所以单从这一点上看，**HashTable的哈希表大小选择，似乎更高明些**。但另一方面我们又知道，**在取模计算时，如果模数是2的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法**。所以从hash计算的效率上，又是HashMap更胜一筹。

所以，事实就是**HashMap为了加快hash的速度，将哈希表的大小固定为了2的幂**，当然这引入了**哈希分布不均匀**的问题，所以HashMap为解决这问题，又对hash算法做了一些改动。

## 如何将hash映射到哈希桶中

具体我们来看看，在获取了key对象的hashCode之后，HashTable和HashMap分别是**怎样将他们hash到确定的哈希桶（Entry数组位置）中的**。

以下代码及注释来自java.util.HashTable

// hash 不能超过Integer.MAX\_VALUE 所以要取其最小的31个bit

int hash = hash(key);

int index = (hash & 0x7FFFFFFF) % tab.length;

// 直接计算key.hashCode()

private int hash(Object k) {

// hashSeed will be zero if alternative hashing is disabled.

return hashSeed ^ k.hashCode();

}

以下代码及注释来自java.util.HashMap

int hash = hash(key);

int i = indexFor(hash, table.length);

// 在计算了key.hashCode()之后，做了一些位运算来减少哈希冲突

final int hash(Object k) {

int h = hashSeed;

if (0 != h && k instanceof String) {

return sun.misc.Hashing.stringHash32((String) k);

}

h ^= k.hashCode();

// This function ensures that hashCodes that differ only by

// constant multiples at each bit position have a bounded

// number of collisions (approximately 8 at default load factor).

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

// 取模不再需要做除法

static int indexFor(int h, int length) {

// assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";

return h & (length-1);

}

正如我们所言，HashMap由于使用了2的幂次方，所以在取模运算时不需要做除法，只需要位的与运算就可以了。但是由于引入的hash冲突加剧问题，HashMap在调用了对象的hashCode方法之后，**又做了一些位运算在打散数据**。关于这些位计算为什么可以打散数据的问题，本文不再展开了。

如果你有细心读代码，还可以发现一点，就是HashMap和HashTable在计算hash时都用到了一个叫**hashSeed的变量**。这是因为映射到同一个hash桶内的Entry对象，是以链表的形式存在的，而链表的查询效率比较低，所以HashMap/HashTable的效率对哈希冲突非常敏感，所以可以额外开启一个**可选hash（hashSeed）**，从而减少哈希冲突。因为这是两个类相同的一点，所以本文不再展开了，感兴趣的看这里。事实上，这个优化在JDK 1.8中已经去掉了，因为JDK 1.8中，映射到同一个哈希桶（数组位置）的Entry对象，使用了红黑树来存储，从而大大加速了其查找效率。

# 简单介绍HashMap和Hashtable

HashMap和Hashtable的比较是Java面试中的常见问题，用来考验程序员是否**能够正确使用集合类以及是否可以随机应变使用多种思路解决问题**。

**HashMap的工作原理**、**ArrayList与Vector的比较**以及这个问题是有关Java 集合框架的最经典的问题。**Hashtable是个过时的集合类**，存在于Java API中很久了。在Java 4中被重写了，实现了Map接口，所以自此以后也成了Java集合框架中的一部分。

# HashMap和Hashtable的区别

## 相同点

### HashMap和Hashtable都实现了Map接口；



两个类的继承体系有些不同。虽然都实现了Map、Cloneable、Serializable三个接口。但是HashMap继承自抽象类AbstractMap，而HashTable继承自抽象类Dictionary。其中Dictionary类是一个已经被废弃的类。HashTable比HashMap多了两个公开方法。一个是elements，这来自于抽象类Dictionary，鉴于该类已经废弃，所以这个方法也就没什么用处了。另一个多出来的方法是contains，这个多出来的方法也没什么用，因为它跟containsValue方法功能是一样的

## 不同点

### 线程安全与不安全

HashMap不是线程安全的，适用于单线程操作，或者外加同步措施。

而Hashtable是线程安全的。

### HashMap的键和值都可以为null，而Hashtable的键和值都不允许为null。

HashMap是支持null键和null值的，而Hashtable在遇到null时，会抛出NullPointerException异常。这并不是因为HashTable有什么特殊的实现层面的原因导致不能支持null键和null值，这仅仅是因为HashMap在实现时对null做了特殊处理：将null的hashCode值定为了0，从而**将其存放在哈希表的第0个bucket中**。

HashMap中获取key的hash的源码为：

static final int hash(Object key) {

int h;

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

}

public V put(K key, V value) {

return putVal(**hash(key)**, key, value, false, true);

}

而Hashtable中：

public synchronized V put(K key, V value) {

// Make sure the value is not null

**if (value == null) {**

**throw new NullPointerException();**

**}**

// Makes sure the key is not already in the hashtable.

Entry<?,?> tab[] = table;

int hash = key.hashCode();

**int index = (hash & 0x7FFFFFFF) % tab.length;**

@SuppressWarnings("unchecked")

Entry<K,V> entry = (Entry<K,V>)tab[index];

for(; entry != null ; entry = entry.next) {

if ((entry.hash == hash) && entry.key.equals(key)) {

V old = entry.value;

entry.value = value;

return old;

}

}

**addEntry(hash, key, value, index);**

return null;

}

### 性能：速度

单线程时：HashMap快于Hashtable。由于Hashtable是线程安全的，所以在单线程环境下它比HashMap要慢。

**HashMap的迭代器(Iterator)是fail-fast迭代器**，而**Hashtable的enumerator迭代器不是fail-fast的**。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。这条同样也是**Enumeration和Iterator的区别**。

官方建议不要使用Hashtable：

As of the Java 2 platform v1.2, this class was retrofitted to implement the **Map** interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, **Hashtable is synchronized.** ①If a thread-safe implementation is not needed, it is recommended to use **HashMap** in place of Hashtable. ②If a thread-safe highly-concurrent implementation is desired, then it is recommended to use **ConcurrentHashMap** in place of Hashtable.

**Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。**

持续优化：

虽然HashMap和HashTable的公开接口应该不会改变，或者说改变不频繁。但每一版本的JDK，都会对HashMap和HashTable的内部实现做优化，比如上文曾提到的**JDK 1.8的红黑树优化**。所以，尽可能的使用新版本的JDK吧，除了那些炫酷的新功能，普通的API也会有性能上有提升。

为什么HashTable已经淘汰了，还要优化它？因为有老的代码还在使用它，所以优化了它之后，这些老的代码也能获得性能提升。

# 要注意的一些重要术语：

## synchronized

**synchronized意味着在一次仅有一个线程能够更改Hashtable**。就是说任何线程要更新Hashtable时要首先获得同步锁，其它线程要等到同步锁被释放之后才能再次获得同步锁更新Hashtable。

## Fail-safe和iterator迭代器相关。

如果某个集合对象创建了**Iterator或者ListIterator**，然后其它的线程试图**“结构上”**更改集合对象，将会抛出**ConcurrentModificationException异常。**但其它线程可以通过**set()**方法更改集合对象是允许的，因为这并没有从“结构上”更改集合。但是假如已经从结构上进行了更改，再调用set()方法，将会抛出**IllegalArgumentException**异常。

3) 结构上的更改指的是删除或者插入一个元素，这样会影响到map的结构。

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**);

# ConcurrentHashMap与HashMap的区别

主要区别：HashMap是对整个哈希表加锁，而ConcurrentHashHashMap将哈希表分割，只对部分加锁。

Hashtable和ConcurrentHashMap都可以用于**多线程的环境**，但是当Hashtable的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。

因为ConcurrentHashMap引入了**分割(segmentation)**，不论它变得多么大，仅仅需要锁定map的某个部分，而其它的线程不需要等到迭代完成才能访问map。简而言之，在迭代的过程中，ConcurrentHashMap仅仅锁定map的某个部分，而Hashtable则会锁定整个map。

为什么我们需要ConcurrentHashMap和CopyOnWriteArrayList

同步的集合类（Hashtable和Vector），同步的封装类（使用Collections.synchronizedMap()方法和Collections.synchronizedList()方法返回的对象）可以创建出线程安全的Map和List。但是有些因素使得它们不适合高并发的系统。**它们仅有单个锁，对整个集合加锁，以及为了防止ConcurrentModificationException异常经常要在迭代的时候要将集合锁定一段时间，这些特性对可扩展性来说都是障碍。**

ConcurrentHashMap和CopyOnWriteArrayList保留了线程安全的同时，也提供了更高的并发性。ConcurrentHashMap和CopyOnWriteArrayList并不是处处都需要用，大部分时候你只需要用到HashMap和ArrayList，它们用于应对一些普通的情况。

# ConcurrentHashMap的源码分析

内部同HashMap一样存在一个实现Map.Entry的Node类：

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

final int hash;

final K key;

volatile V val;

volatile Node<K,V> next;

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

this.hash = hash;

this.key = key;

this.val = val;

this.next = next;

}

分段可重入锁：

static class **Segment<K,V>** extends **ReentrantLock** implements Serializable {

private static final long serialVersionUID = 2249069246763182397L;

final float loadFactor;

**Segment(float lf) { this.loadFactor = lf; }**

}