HashMap和HashTable到底哪不同?

05 July 2016

HashMap和HashTable有什么不同？在面试和被面试的过程中，我问过也被问过这个问题，也见过了不少回答，今天决定写一写自己心目中的理想答案。

代码版本

JDK每一版本都在改进。本文讨论的HashMap和HashTable基于JDK 1.7.0_67。源码见这里(https://github.com/ZhaoX/jdk-1.7-annotated)

1. 时间

HashTable产生于JDK 1.1，而HashMap产生于JDK 1.2。从时间的维度上来看，HashMap要比HashTable出现得晚一些。

2. 作者

以下是HashTable的作者：

```
以下代码及注释来自java.util.HashTable

* @author  Arthur van Hoff
* @author  Josh Bloch
* @author  Neal Gafter
```

以下是HashMap的作者：

```
以下代码及注释来自java.util.HashMap

 * @author  Doug Lea
 * @author  Josh Bloch
 * @author  Arthur van Hoff
 * @author  Neal Gafter
```

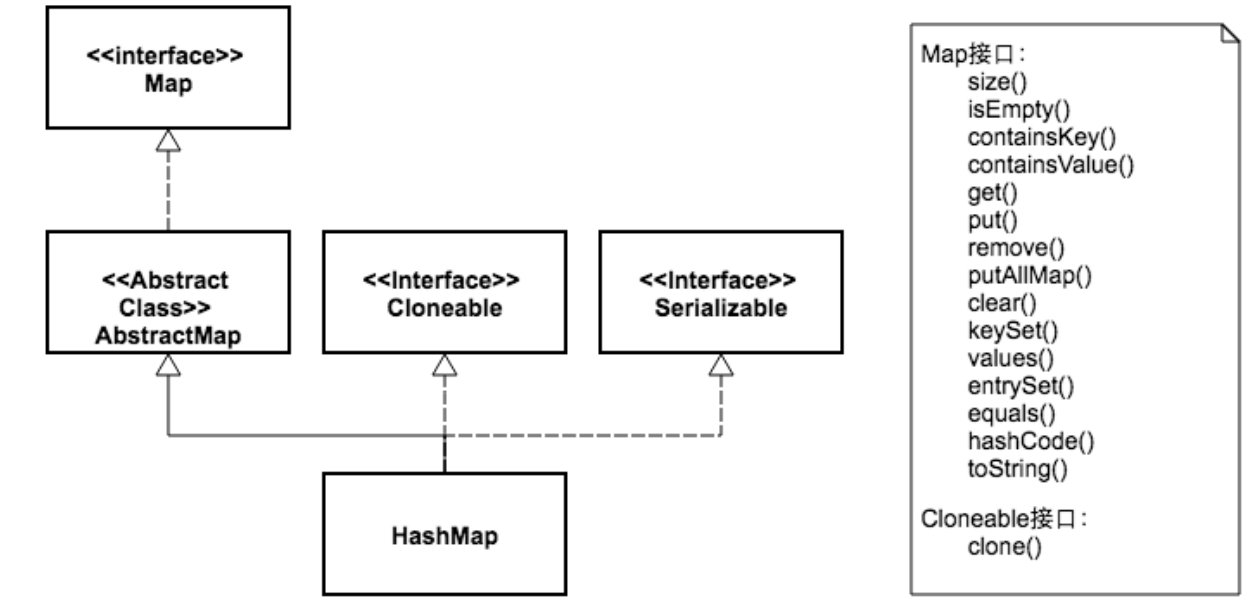
可以看到HashMap的作者多了大神Doug Lea。不了解Doug Lea的，可以看这里(https://en.wikipedia.org/wiki/Doug_Lea)。

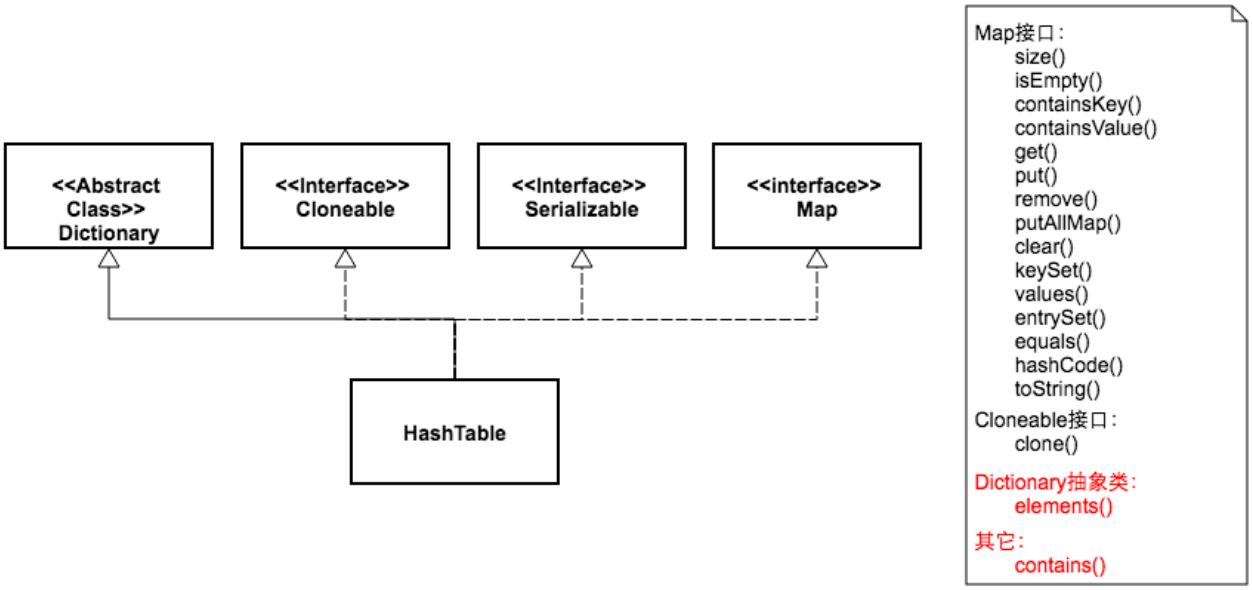
3. 对外的接口（API）

HashMap和HashTable都是基于哈希表来实现键值映射的工具类。讨论他们的不同，我们首先来看一下他们暴露在外的API有什么不同。

3.1 Public Method

下面两张图，我画出了HashMap和HashTable的类继承体系，并列出了这两个类的可供外部调用的公开方法。





从图中可以看出，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，这一点我们可以从它代码的注释中看到：

以下代码及注释来自java.util.Dictionary

```
 * <strong>NOTE: This class is obsolete.  New implementations should
 * implement the Map interface, rather than extending this class.</strong>
```

同时我们看到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。代码为证：

以下代码及注释来自java.util.HashTable

```
public synchronized boolean contains(Object value) {
    if (value == null) {
        throw new NullPointerException();
    }

    Entry tab[] = table;
    for (int i = tab.length ; i-- > 0 ;) {
        for (Entry<K,V> e = tab[i] ; e != null ; e = e.next) {
            if (e.value.equals(value)) {
                return true;
            }
        }
    }
    return false;
}

public boolean containsValue(Object value) {
    return contains(value);
}
```

所以从公开的方法上来看，这两个类提供的，是一样的功能。都提供键值映射的服务，可以增、删、查、改键值对，可以对建、值、键值对提供遍历视图。支持浅拷贝，支持序列化。

3.2 Null Key & Null Value

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。我们一put方法为例，看一看代码的细节：

以下代码及注释来自java.util.HashTable

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

    // 如果value为null，抛出NullPointerException
    if (value == null) {
        throw new NullPointerException();
    }

    // 如果key为null，在调用key.hashCode()时抛出NullPointerException

    // ...
}
```

以下代码及注释来自java.util.HasMap

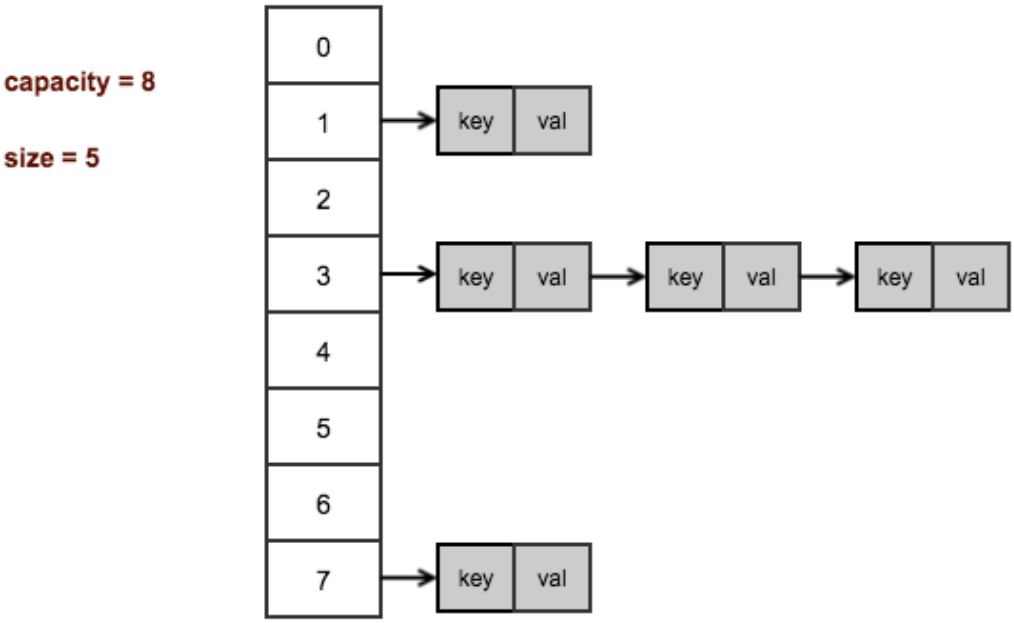
```
public V put(K key, V value) {
    if (table == EMPTY_TABLE) {
        inflateTable(threshold);
    }
    // 当key为null时，调用putForNullKey特殊处理
    if (key == null)
        return putForNullKey(value);
    // ...
}

private V putForNullKey(V value) {
    // key为null时，放到table[0]也就是第0个bucket中
    for (Entry<K,V> e = table[0]; e != null; e = e.next) {
        if (e.key == null) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    addEntry(0, null, value, 0);
    return null;
}
```

4. 实现原理

本节讨论HashMap和HashTable在数据结构和算法层面，有什么不同。

4.1 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Entry，每一个Entry对象表示存储在哈希表中的一个键值对。
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<K, V> 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。
这样就可以得出结论，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)。

以下代码及注释来自java.util.HashTable

```
/**
 * The hash table data.
 */
private transient Entry<K,V>[] table;
```

以下代码及注释来自java.util.HashMap

```
/**
 * The table, resized as necessary. Length MUST Always be a power of two.
 */
transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY_TABLE;
```

从代码可以看到，对于哈希桶的内部表示，两个类的实现是一致的。

4.2 算法

上一小节已经说了用来表示哈希表的内部数据结构。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。先看代码：

以下代码及注释来自java.util.HashTable

```
// 哈希表默认初始大小为11
public Hashtable() {
    this(11, 0.75f);
}

protected void rehash() {
    int oldCapacity = table.length;
    Entry<K,V>[] oldMap = table;

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

以下代码及注释来自java.util.HashMap

```
// 哈希表默认初始大小为2^4=16
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

void addEntry(int hash, K key, V value, int bucketIndex) {
    // 每次扩充为原来的2n
    if ((size >= threshold) && (null != table[bucketIndex])) {
       resize(2 * table.length);
}
```

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。还有我没列出代码的一点，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(http://zhaox.github.io/algorithm/2015/06/29/hash)），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。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。感兴趣的可以看这里(http://stackoverflow.com/questions/9413966/why-initialcapacity-of-hashtable-is-11-while-the-default-initial-capacity-in-has)。
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(http://stackoverflow.com/questions/29918624/what-is-the-use-of-holder-class-in-hashmap)。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。

5. 线程安全

我们说HashTable是同步的，HashMap不是，也就是说HashTable在多线程使用的情况下，不需要做额外的同步，而HashMap则不行。那么HashTable是怎么做到的呢？

以下代码及注释来自java.util.HashTable

```
public synchronized V get(Object key) {
    Entry tab[] = table;
    int hash = hash(key);
    int index = (hash & 0x7FFFFFFF) % tab.length;
    for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            return e.value;
        }
    }
    return null;
}

public Set<K> keySet() {
    if (keySet == null)
        keySet = Collections.synchronizedSet(new KeySet(), this);
    return keySet;
}
```

可以看到，也比较简单，就是公开的方法比如get都使用了synchronized描述符。而遍历视图比如keySet都使用了Collections.synchronizedXXX进行了同步包装。

6. 代码风格

从我的品位来看，HashMap的代码要比HashTable整洁很多。下面这段HashTable的代码，我就觉着有点混乱，不太能接受这种代码复用的方式。

以下代码及注释来自java.util.HashTable

```
/**
 * A hashtable enumerator class.  This class implements both the
 * Enumeration and Iterator interfaces, but individual instances
 * can be created with the Iterator methods disabled.  This is necessary
 * to avoid unintentionally increasing the capabilities granted a user
 * by passing an Enumeration.
 */
private class Enumerator<T> implements Enumeration<T>, Iterator<T> {
    Entry[] table = Hashtable.this.table;
    int index = table.length;
    Entry<K,V> entry = null;
    Entry<K,V> lastReturned = null;
    int type;

    /**
     * Indicates whether this Enumerator is serving as an Iterator
     * or an Enumeration.  (true -> Iterator).
     */
    boolean iterator;

    /**
     * The modCount value that the iterator believes that the backing
     * Hashtable should have.  If this expectation is violated, the iterator
     * has detected concurrent modification.
     */
    protected int expectedModCount = modCount;

    Enumerator(int type, boolean iterator) {
        this.type = type;
        this.iterator = iterator;
    }

    //...

}
```

7. HashTable已经被淘汰了，不要在代码中再使用它。

以下描述来自于HashTable的类注释：

> If a thread-safe implementation is not needed, it is recommended to use HashMap in place of Hashtable. If a thread-safe highly-concurrent implementation is desired, then it is recommended to use java.util.concurrent.ConcurrentHashMap in place of Hashtable.

简单来说就是，如果你不需要线程安全，那么使用HashMap，如果需要线程安全，那么使用ConcurrentHashMap。HashTable已经被淘汰了，不要在新的代码中再使用它。

8. 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。

Reference

- https://github.com/ZhaoX/jdk-1.7-annotated/blob/master/src/java/util/HashMap.java (https://github.com/ZhaoX/jdk-1.7-annotated/blob/master/src/java/util/HashMap.java)
- https://github.com/ZhaoX/jdk-1.7-annotated/blob/master/src/java/util/Hashtable.java (https://github.com/ZhaoX/jdk-1.7-annotated/blob/master/src/java/util/Hashtable.java)

← Previous (/2016/06/24/mysql-architecture) | Archive (/archive.html)
Next → (/security/2016/11/05/how-to-store-users-password-securely)