Java集合学习指南

极客学院出版

前言

该系列文章是在研究 JDK7 源码过程中对于集合的学习总结。

其包含的内容有：

Java 集合学习 1：HashMap 的实现原理

Java 集合学习 2：HashSet 的实现原理

Java 集合学习 3：Hashtable 的实现原理

Java 集合学习 4：LinkedHashMap 的实现原理

Java 集合学习 5：LinkedHashSet 的实现原理

Java 集合学习 6：ArrayList 的实现原理

Java 集合学习 7：LinkedList 的实现原理

Java 集合学习 8：ConcurrentHashMap 的实现原理

集合应用：LinkedHashMap 与 LRUcache

Java 集合比较：HashSet 和 HashMap 的比较

除此之外，还会增加 TreeMap 等集合的学习以及集合之间的对比。我们学习集合源码的目的一个是为了学习其中的思想，另一个也是为了让我们能够更好的了解该集合，在适合的地方使用恰当的集合实现。

适用人群

适合 Java 初学者的进阶学习，集合类在 Java 中有很重要的意义，保存临时数据，管理对象，泛型，Web 框架等，很多都大量用到了集合类。

学习前提

学习本指南前，你需要达到 Java 语言的入门级。

鸣谢作者：李辉

目录

1

HashMap 的实现原理

HashMap 概述

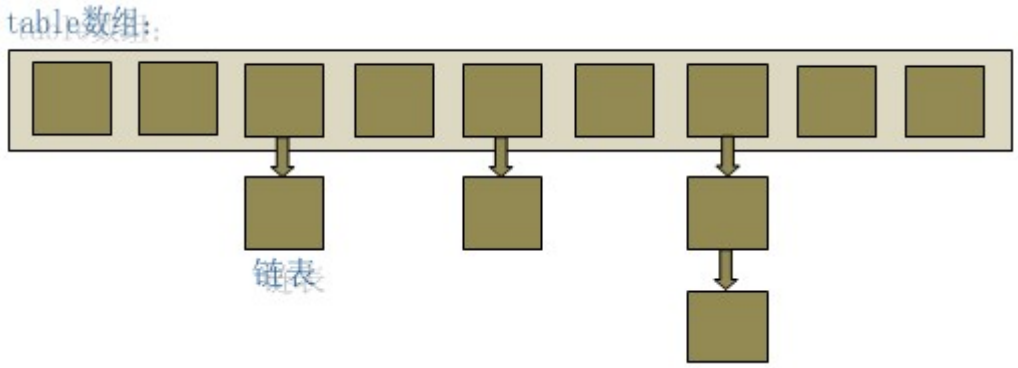
HashMap 是基于哈希表的 Map 接口的非同步实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

此实现假定哈希函数将元素适当地分布在各桶之间，可为基本操作（get 和 put）提供稳定的性能。迭代 collection 视图所需的时间与 HashMap 实例的“容量”（桶的数量）及其大小（键-值映射关系数）成比例。所以，如果迭代性能很重要，则不要将初始容量设置得太高或将加载因子设置得太低。也许大家开始对这段话有一点不太懂，不过不用担心，当你读完这篇文章后，就能深切理解这其中的含义了。

需要注意的是：Hashmap 不是同步的，如果多个线程同时访问一个 HashMap，而其中至少一个线程从结构上（指添加或者删除一个或多个映射关系的任何操作）修改了，则必须保持外部同步，以防止对映射进行意外的非同步访问。

HashMap 的数据结构

在 Java 编程语言中，最基本的结构就是两种，一个是数组，另外一个是指针（引用），HashMap 就是通过这两个数据结构进行实现。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。



图片 1.1 图1

从上图中可以看出，HashMap 底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个 HashMap 的时候，就会初始化一个数组。

我们通过 JDK 中的 HashMap 源码进行一些学习，首先看一下构造函数：

```
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);

    // Find a power of 2 >= initialCapacity
    int capacity = 1;
    while (capacity < initialCapacity)
        capacity <<= 1;

    this.loadFactor = loadFactor;
    threshold = (int)Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
    table = new Entry[capacity];
    useAltHashing = sun.misc.VM.isBooted() &&
            (capacity >= Holder.ALTERNATIVE_HASHING_THRESHOLD);
    init();
}
```

我们着重看一下第 18 行代码 `table = new Entry[capacity];` 。这不就是 Java 中数组的创建方式吗？也就是说在构造函数中，其创建了一个 Entry 的数组，其大小为 capacity（目前我们还不需要太了解该变量含义），那么 Entry 又是什么结构呢？看一下源码：

```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;
    final int hash;
    ......
}
```

我们目前还是只着重核心的部分，Entry 是一个 static class，其中包含了 key 和 value，也就是键值对，另外还包含了一个 next 的 Entry 指针。我们可以总结出：Entry 就是数组中的元素，每个 Entry 其实就是一个 key-value 对，它持有一个指向下一个元素的引用，这就构成了链表。

HashMap 的核心方法解读

存储

```
/**
     * Associates the specified value with the specified key in this map.
     * If the map previously contained a mapping for the key, the old
     * value is replaced.
     *
     * @param key key with which the specified value is to be associated
     * @param value value to be associated with the specified key
     * @return the previous value associated with <tt>key</tt>, or
     *         <tt>null</tt> if there was no mapping for <tt>key</tt>.
     *         (A <tt>null</tt> return can also indicate that the map
     *         previously associated <tt>null</tt> with <tt>key</tt>.)
     */
public V put(K key, V value) {
        //其允许存放null的key和null的value，当其key为null时，调用putForNullKey方法，放入到table[0]的这个位置
        if (key == null)
            return putForNullKey(value);
        //通过调用hash方法对key进行哈希，得到哈希之后的数值。该方法实现可以通过看源码，其目的是为了尽可能的让键值对可以分
        int hash = hash(key);
        //根据上一步骤中求出的hash得到在数组中是索引i
        int i = indexFor(hash, table.length);
        //如果i处的Entry不为null，则通过其next指针不断遍历e元素的下一个元素。
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            Object k;
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }

        modCount++;
        addEntry(hash, key, value, i);
        return null;
}
```

我们看一下方法的标准注释：在注释中首先提到了，当我们 put 的时候，如果 key 存在了，那么新的 value 会代替旧的 value，并且如果 key 存在的情况下，该方法返回的是旧的 value，如果 key 不存在，那么返回 null。

从上面的源代码中可以看出：当我们往 HashMap 中 put 元素的时候，先根据 key 的 hashCode 重新计算 hash 值，根据 hash 值得到这个元素在数组中的位置（即下标），如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

addEntry(hash, key, value, i)方法根据计算出的 hash 值，将 key-value 对放在数组 table 的 i 索引处。addEntry 是 HashMap 提供的一个包访问权限的方法，代码如下：

```
/**
    * Adds a new entry with the specified key, value and hash code to
    * the specified bucket.  It is the responsibility of this
    * method to resize the table if appropriate.
    *
    * Subclass overrides this to alter the behavior of put method.
    */
void addEntry(int hash, K key, V value, int bucketIndex) {
      if ((size >= threshold) && (null != table[bucketIndex])) {
         resize(2 * table.length);
         hash = (null != key) ? hash(key) : 0;
         bucketIndex = indexFor(hash, table.length);
      }

      createEntry(hash, key, value, bucketIndex);
}
void createEntry(int hash, K key, V value, int bucketIndex) {
      // 获取指定 bucketIndex 索引处的 Entry
      Entry<K,V> e = table[bucketIndex];
      // 将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entr
      table[bucketIndex] = new Entry<>(hash, key, value, e);
      size++;
}
```

当系统决定存储 HashMap 中的 key-value 对时，完全没有考虑 Entry 中的 value，仅仅只是根据 key 来计算并决定每个 Entry 的存储位置。我们完全可以把 Map 集合中的 value 当成 key 的附属，当系统决定了 key 的存储位置之后，value 随之保存在那里即可。

hash(int h)方法根据 key 的 hashCode 重新计算一次散列。此算法加入了高位计算，防止低位不变，高位变化时，造成的 hash 冲突。

```
final int hash(Object k) {
      int h = 0;
      if (useAltHashing) {
         if (k instanceof String) {
            return sun.misc.Hashing.stringHash32((String) k);
         }
         h = hashSeed;
      }
      //得到k的hashcode值
      h ^= k.hashCode();
      //进行计算
      h ^= (h >>> 20) ^ (h >>> 12);
      return h ^ (h >>> 7) ^ (h >>> 4);
}
```

我们可以看到在 HashMap 中要找到某个元素，需要根据 key 的 hash 值来求得对应数组中的位置。如何计算这个位置就是 hash 算法。前面说过 HashMap 的数据结构是数组和链表的结合，所以我们当然希望这个 HashMap 里面的 元素位置尽量的分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用 hash 算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，而不用再去遍历链表，这样就大大优化了查询的效率。

对于任意给定的对象，只要它的 hashCode() 返回值相同，那么程序调用 hash(int h) 方法所计算得到的 hash 码值总是相同的。我们首先想到的就是把 hash 值对数组长度取模运算，这样一来，元素的分布相对来说是比较

均匀的。但是，“模”运算的消耗还是比较大的，在 HashMap 中是这样做的：调用 indexFor(int h, int lengt h) 方法来计算该对象应该保存在 table 数组的哪个索引处。indexFor(int h, int length) 方法的代码如下：

```
/**
     * Returns index for hash code h.
     */
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

这个方法非常巧妙，它通过 h & (table.length -1) 来得到该对象的保存位，而 HashMap 底层数组的长度总是 2 的 n 次方，这是 HashMap 在速度上的优化。在 HashMap 构造器中有如下代码：

```
// Find a power of 2 >= initialCapacity
int capacity = 1;
    while (capacity < initialCapacity)
        capacity <<= 1;
```

这段代码保证初始化时 HashMap 的容量总是 2 的 n 次方，即底层数组的长度总是为 2 的 n 次方。

当 length 总是 2 的 n 次方时，h& (length-1)运算等价于对 length 取模，也就是 h%length，但是 & 比 % 具有更高的效率。这看上去很简单，其实比较有玄机的，我们举个例子来说明：

假设数组长度分别为 15 和 16，优化后的 hash 码分别为 8 和 9，那么 & 运算后的结果如下：

h & (table.length-1)	hash		table.length-1	
8 & (15-1)：	0100	&	1110	= 0100
9 & (15-1)：	0101	&	1110	= 0100
8 & (16-1)：	0100	&	1111	= 0100
9 & (16-1)：	0101	&	1111	= 0101

从上面的例子中可以看出：当它们和 15-1（1110）“与”的时候，产生了相同的结果，也就是说它们会定位到数组中的同一个位置上去，这就产生了碰撞，8 和 9 会被放到数组中的同一个位置上形成链表，那么查询的时候就需要遍历这个链 表，得到8或者9，这样就降低了查询的效率。同时，我们也可以发现，当数组长度为 15 的时候，hash 值会与 15-1（1110）进行“与”，那么最后一位永远是 0，而 0001，0011，0101，1001，1011，0111，1101 这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！而当数组长度为16时，即为2的n次方时，2n-1 得到的二进制数的每个位上的值都为 1，这使得在低位上&时，得到的和原 hash 的低位相同，加之 hash(int h)方法对 key 的 hashCode 的进一步优化，加入了高位计算，就使得只有相同的 hash 值的两个值才会被放到数组中的同一个位置上形成链表。

所以说，当数组长度为 2 的 n 次幂的时候，不同的 key 算得得 index 相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

根据上面 put 方法的源代码可以看出，当程序试图将一个key-value对放入HashMap中时，程序首先根据该 key 的 hashCode() 返回值决定该 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() 返回值相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry 的 value，但key不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部——具体说明继续看 addEntry() 方法的说明。

读取

```
/**
     * Returns the value to which the specified key is mapped,
     * or {@code null} if this map contains no mapping for the key.
     *
     * <p>More formally, if this map contains a mapping from a key
     * {@code k} to a value {@code v} such that {@code (key==null ? k==null :
     * key.equals(k))}, then this method returns {@code v}; otherwise
     * it returns {@code null}.  (There can be at most one such mapping.)
     *
     * <p>A return value of {@code null} does not <i>necessarily</i>
     * indicate that the map contains no mapping for the key; it's also
     * possible that the map explicitly maps the key to {@code null}.
     * The {@link #containsKey containsKey} operation may be used to
     * distinguish these two cases.
     *
     * @see #put(Object, Object)
     */
    public V get(Object key) {
        if (key == null)
            return getForNullKey();
        Entry<K,V> entry = getEntry(key);

        return null == entry ? null : entry.getValue();
    }
    final Entry<K,V> getEntry(Object key) {
        int hash = (key == null) ? 0 : hash(key);
        for (Entry<K,V> e = table[indexFor(hash, table.length)];
             e != null;
             e = e.next) {
            Object k;
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                return e;
        }
        return null;
    }
```

有了上面存储时的 hash 算法作为基础，理解起来这段代码就很容易了。从上面的源代码中可以看出：从 HashMap 中 get 元素时，首先计算 key 的 hashCode，找到数组中对应位置的某一元素，然后通过 key 的 equals 方法在对应位置的链表中找到需要的元素。

归纳

简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据 hash 算法来决定其在数组中的存储位置，在根据 equals 方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry 时，也会根据 hash 算法找到其在数组中的存储位置，再根据 equals 方法从该位置上的链表中取出该Entry。

HashMap 的 resize（rehash）

当 HashMap 中的元素越来越多的时候，hash 冲突的几率也就越来越高，因为数组的长度是固定的。所以为了提高查询的效率，就要对 HashMap 的数组进行扩容，数组扩容这个操作也会出现在 ArrayList 中，这是一个常用的操作，而在 HashMap 数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是 resize。

那么 HashMap 什么时候进行扩容呢？当 HashMap 中的元素个数超过数组大小 `*loadFactor` 时，就会进行数组扩容，loadFactor的默认值为 0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为 16，那么当 HashMap 中元素个数超过 `16*0.75=12` 的时候，就把数组的大小扩展为 `2*16=32`，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知 HashMap 中元素的个数，那么预设元素的个数能够有效的提高 HashMap 的性能。

HashMap 的性能参数

HashMap 包含如下几个构造器：

- HashMap()：构建一个初始容量为 16，负载因子为 0.75 的 HashMap。
- ashMap(int initialCapacity)：构建一个初始容量为 initialCapacity，负载因子为 0.75 的 HashMap。
- HashMap(int initialCapacity, float loadFactor)：以指定初始容量、指定的负载因子创建一个 HashMap。

HashMap 的基础构造器 HashMap(int initialCapacity, float loadFactor) 带有两个参数，它们是初始容量 initialCapacity 和负载因子 loadFactor。

负载因子 loadFactor 衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是 O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。

HashMap 的实现中，通过 threshold 字段来判断 HashMap 的最大容量：

```
    threshold = (int)(capacity * loadFactor);
```

结合负载因子的定义公式可知，threshold 就是在此 loadFactor 和 capacity 对应下允许的最大元素数目，超过这个数目就重新 resize，以降低实际的负载因子。默认的的负载因子 0.75 是对空间和时间效率的一个平衡选择。当容量超出此最大容量时， resize 后的 HashMap 容量是容量的两倍：

Fail-Fast 机制

原理

我们知道 java.util.HashMap 不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了 map，那么将抛出 ConcurrentModificationException，这就是所谓 fail-fast 策略。

ail-fast 机制是 java 集合(Collection)中的一种错误机制。 当多个线程对同一个集合的内容进行操作时，就可能会产生 fail-fast 事件。

例如：当某一个线程 A 通过 iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程 A 访问集合时，就会抛出 ConcurrentModificationException 异常，产生 fail-fast 事件。

这一策略在源码中的实现是通过 modCount 域，modCount 顾名思义就是修改次数，对 HashMap 内容（当然不仅仅是 HashMap 才会有，其他例如 ArrayList 也会）的修改都将增加这个值（大家可以再回头看一下其源码，在很多操作中都有 modCount++ 这句），那么在迭代器初始化过程中会将这个值赋给迭代器的 expectedModCount。

```
HashIterator() {
    expectedModCount = modCount;
    if (size > 0) { // advance to first entry
    Entry[] t = table;
    while (index < t.length && (next = t[index++]) == null)
        ;
    }
}
```

在迭代过程中，判断 modCount 跟 expectedModCount 是否相等，如果不相等就表示已经有其他线程修改了 Map：

注意到 modCount 声明为 volatile，保证线程之间修改的可见性。

```
final Entry<K,V> nextEntry() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
```

在 HashMap 的 API 中指出：

由所有 HashMap 类的“collection 视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒在将来不确定的时间发生任意不确定行为的风险。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出 ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

解决方案

在上文中也提到，fail-fast 机制，是一种错误检测机制。它只能被用来检测错误，因为 JDK 并不保证 fail-fast 机制一定会发生。若在多线程环境下使用 fail-fast 机制的集合，建议使用“java.util.concurrent 包下的类”去取代“java.util 包下的类”。

HashMap 的两种遍历方式

第一种

```
Map map = new HashMap();
Iterator iter = map.entrySet().iterator();
while (iter.hasNext()) {
Map.Entry entry = (Map.Entry) iter.next();
Object key = entry.getKey();
Object val = entry.getValue();
}
```

效率高,以后一定要使用此种方式!

第二种

```
Map map = new HashMap();
Iterator iter = map.keySet().iterator();
while (iter.hasNext()) {
Object key = iter.next();
Object val = map.get(key);
}
```

效率低,以后尽量少使用!

2

HashSet 的实现原理

HashSet 概述

对于 HashSet 而言，它是基于 HashMap 实现的，底层采用 HashMap 来保存元素，所以如果对 HashMap 比较熟悉了，那么学习 HashSet 也是很轻松的。

我们先通过 HashSet 最简单的构造函数和几个成员变量来看一下，证明咱们上边说的，其底层是 HashMap：

```
private transient HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();

/**
 * Constructs a new, empty set; the backing <tt>HashMap</tt> instance has
 * default initial capacity (16) and load factor (0.75).
 */
public HashSet() {
    map = new HashMap<>();
}
```

其实在英文注释中已经说的比较明确了。首先有一个HashMap的成员变量，我们在 HashSet 的构造函数中将其初始化，默认情况下采用的是 initial capacity为16，load factor 为 0.75。

HashSet 的实现

对于 HashSet 而言，它是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，相关 HashSet 的操作，基本上都是直接调用底层 HashMap 的相关方法来完成，我们应该为保存到 HashSet 中的对象覆盖 hashCode() 和 equals()

构造方法

```
/**
 * 默认的无参构造器，构造一个空的HashSet。
 *
 * 实际底层会初始化一个空的HashMap，并使用默认初始容量为16和加载因子0.75。
 */
public HashSet() {
    map = new HashMap<E,Object>();
}

/**
 * 构造一个包含指定collection中的元素的新set。
 *
 * 实际底层使用默认的加载因子0.75和足以包含指定collection中所有元素的初始容量来创建一个HashMap。
 * @param c 其中的元素将存放在此set中的collection。
 */
public HashSet(Collection<? extends E> c) {
    map = new HashMap<E,Object>(Math.max((int) (c.size()/.75f) + 1, 16));
    addAll(c);
}

/**
 * 以指定的initialCapacity和loadFactor构造一个空的HashSet。
 *
 * 实际底层以相应的参数构造一个空的HashMap。
 * @param initialCapacity 初始容量。
 * @param loadFactor 加载因子。
 */
public HashSet(int initialCapacity, float loadFactor) {
    map = new HashMap<E,Object>(initialCapacity, loadFactor);
}

/**
 * 以指定的initialCapacity构造一个空的HashSet。
 *
 * 实际底层以相应的参数及加载因子loadFactor为0.75构造一个空的HashMap。
 * @param initialCapacity 初始容量。
 */
public HashSet(int initialCapacity) {
    map = new HashMap<E,Object>(initialCapacity);
}

/**
 * 以指定的initialCapacity和loadFactor构造一个新的空链接哈希集合。此构造函数为包访问权限，不对外公开，
 * 实际只是是对LinkedHashSet的支持。
 *
 * 实际底层会以指定的参数构造一个空LinkedHashMap实例来实现。
```

```
 * @param initialCapacity 初始容量。
 * @param loadFactor 加载因子。
 * @param dummy 标记。
 */
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
   map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);
}
```

add 方法

```
/**

 * @param e 将添加到此set中的元素。
 * @return 如果此set尚未包含指定元素，则返回true。
 */
public boolean add(E e) {
   return map.put(e, PRESENT)==null;
}
```

如果此 set 中尚未包含指定元素，则添加指定元素。更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2)) 的元素 e2，则向此 set 添加指定的元素 e。如果此 set 已包含该元素，则该调用不更改 set 并返回 false。但底层实际将将该元素作为 key 放入 HashMap。思考一下为什么?

由于 HashMap 的 put() 方法添加 key-value 对时，当新放入 HashMap 的 Entry 中 key 与集合中原有 Entry 的 key 相同（hashCode()返回值相等，通过 equals 比较也返回 true），新添加的 Entry 的 value 会将覆盖原来 Entry 的 value（HashSet 中的 value 都是 PRESENT），但 key 不会有任何改变，因此如果向 HashSet 中添加一个已经存在的元素时，新添加的集合元素将不会被放入 HashMap中，原来的元素也不会有任何改变，这也就满足了 Set 中元素不重复的特性。

该方法如果添加的是在 HashSet 中不存在的，则返回 true；如果添加的元素已经存在，返回 false。其原因在于我们之前提到的关于 HashMap 的 put 方法。该方法在添加 key 不重复的键值对的时候，会返回 null。

其余方法

```
   /**
    * 如果此set包含指定元素，则返回true。
    * 更确切地讲，当且仅当此set包含一个满足(o==null ? e==null : o.equals(e))的e元素时，返回true。
    *
    * 底层实际调用HashMap的containsKey判断是否包含指定key。
    * @param o 在此set中的存在已得到测试的元素。
    * @return 如果此set包含指定元素，则返回true。
    */
   public boolean contains(Object o) {
   return map.containsKey(o);
   }
   /**
    * 如果指定元素存在于此set中，则将其移除。更确切地讲，如果此set包含一个满足(o==null ? e==null : o.equals(e))的元素e，
    * 则将其移除。如果此set已包含该元素，则返回true
    *
    * 底层实际调用HashMap的remove方法删除指定Entry。
```

```
    * @param o 如果存在于此set中则需要将其移除的对象。
    * @return 如果set包含指定元素，则返回true。
    */
   public boolean remove(Object o) {
   return map.remove(o)==PRESENT;
   }
   /**
    * 返回此HashSet实例的浅表副本：并没有复制这些元素本身。
    *
    * 底层实际调用HashMap的clone()方法，获取HashMap的浅表副本，并设置到HashSet中。
    */
   public Object clone() {
      try {
         HashSet<E> newSet = (HashSet<E>) super.clone();
         newSet.map = (HashMap<E, Object>) map.clone();
         return newSet;
      } catch (CloneNotSupportedException e) {
         throw new InternalError();
      }
   }
}
```

相关说明

1. 相关 HashMap 的实现原理，请参考我的上一遍总结：HashMap的实现原理。
2. 对于 HashSet 中保存的对象，请注意正确重写其 equals 和 hashCode 方法，以保证放入的对象的唯一性。这两个方法是比较重要的，希望大家在以后的开发过程中需要注意一下。

3

Hashtable 的实现原理

概述

和 HashMap 一样，Hashtable 也是一个散列表，它存储的内容是键值对。

Hashtable 在 Java 中的定义为：

```
public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable{}
```

从源码中，我们可以看出，Hashtable 继承于 Dictionary 类，实现了 Map, Cloneable, java.io.Serializable接口。其中Dictionary类是任何可将键映射到相应值的类（如 Hashtable）的抽象父类，每个键和值都是对象（源码注释为：The `Dictionary` class is the abstract parent of any class, such as `Hashtable`, which maps keys to values. Every key and every value is an object.）。但在这一点我开始有点怀疑，因为我查看了HashMap以及TreeMap的源码，都没有继承于这个类。不过当我看到注释中的解释也就明白了，其 Dictionary 源码注释是这样的：NOTE: This class is obsolete. New implementations should implement the Map interface, rather than extending this class. 该话指出 Dictionary 这个类过时了，新的实现类应该实现Map接口。

Hashtable 源码解读

成员变量

Hashtable是通过"拉链法"实现的哈希表。它包括几个重要的成员变量：table, count, threshold, loadFactor, modCount。

- table是一个 Entry[] 数组类型，而 Entry（在 HashMap 中有讲解过）实际上就是一个单向链表。哈希表的"key-value键值对"都是存储在Entry数组中的。
- count 是 Hashtable 的大小，它是 Hashtable 保存的键值对的数量。
- threshold 是 Hashtable 的阈值，用于判断是否需要调整 Hashtable 的容量。threshold 的值="容量*加载因子"。
- loadFactor 就是加载因子。
- modCount 是用来实现 fail-fast 机制的。

关于变量的解释在源码注释中都有，最好还是应该看英文注释。

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

    /**
     * The total number of entries in the hash table.
     */
    private transient int count;

    /**
     * The table is rehashed when its size exceeds this threshold.  (The
     * value of this field is (int)(capacity * loadFactor).)
     *
     * @serial
     */
    private int threshold;

    /**
     * The load factor for the hashtable.
     *
     * @serial
     */
    private float loadFactor;

    /**
     * The number of times this Hashtable has been structurally modified
     * Structural modifications are those that change the number of entries in
     * the Hashtable or otherwise modify its internal structure (e.g.,
     * rehash).  This field is used to make iterators on Collection-views of
     * the Hashtable fail-fast.  (See ConcurrentModificationException).
```

```
     */
    private transient int modCount = 0;
```

构造方法

Hashtable 一共提供了 4 个构造方法：

- `public Hashtable(int initialCapacity, float loadFactor)`： 用指定初始容量和指定加载因子构造一个新的空哈希表。useAltHashing 为 boolean，其如果为真，则执行另一散列的字符串键，以减少由于弱哈希计算导致的哈希冲突的发生。
- `public Hashtable(int initialCapacity)`：用指定初始容量和默认的加载因子 (0.75) 构造一个新的空哈希表。
- `public Hashtable()`：默认构造函数，容量为 11，加载因子为 0.75。
- `public Hashtable(Map<? extends K, ? extends V> t)`：构造一个与给定的 Map 具有相同映射关系的新哈希表。

```
/**
     * Constructs a new, empty hashtable with the specified initial
     * capacity and the specified load factor.
     *
     * @param      initialCapacity   the initial capacity of the hashtable.
     * @param      loadFactor        the load factor of the hashtable.
     * @exception  IllegalArgumentException  if the initial capacity is less
     *             than zero, or if the load factor is nonpositive.
     */
    public Hashtable(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal Load: "+loadFactor);

        if (initialCapacity==0)
            initialCapacity = 1;
        this.loadFactor = loadFactor;
        table = new Entry[initialCapacity];
        threshold = (int)Math.min(initialCapacity * loadFactor, MAX_ARRAY_SIZE + 1);
        useAltHashing = sun.misc.VM.isBooted() &&
                (initialCapacity >= Holder.ALTERNATIVE_HASHING_THRESHOLD);
    }

    /**
     * Constructs a new, empty hashtable with the specified initial capacity
     * and default load factor (0.75).
     *
     * @param      initialCapacity   the initial capacity of the hashtable.
     * @exception IllegalArgumentException if the initial capacity is less
     *             than zero.
     */
    public Hashtable(int initialCapacity) {
        this(initialCapacity, 0.75f);
    }

    /**
```

```
    * Constructs a new, empty hashtable with a default initial capacity (11)
    * and load factor (0.75).
    */
   public Hashtable() {
       this(11, 0.75f);
   }

   /**
    * Constructs a new hashtable with the same mappings as the given
    * Map.  The hashtable is created with an initial capacity sufficient to
    * hold the mappings in the given Map and a default load factor (0.75).
    *
    * @param t the map whose mappings are to be placed in this map.
    * @throws NullPointerException if the specified map is null.
    * @since   1.2
    */
   public Hashtable(Map<? extends K, ? extends V> t) {
       this(Math.max(2*t.size(), 11), 0.75f);
       putAll(t);
   }
```

put 方法

put 方法的整个流程为：

1. 判断 value 是否为空，为空则抛出异常；
2. 计算 key 的 hash 值，并根据 hash 值获得 key 在 table 数组中的位置 index，如果 table[index] 元素不为空，则进行迭代，如果遇到相同的 key，则直接替换，并返回旧 value；
3. 否则，我们可以将其插入到 table[index] 位置。

我在下面的代码中也进行了一些注释：

```
public synchronized V put(K key, V value) {
        // Make sure the value is not null确保value不为null
        if (value == null) {
            throw new NullPointerException();
        }

        // Makes sure the key is not already in the hashtable.
        //确保key不在hashtable中
        //首先，通过hash方法计算key的哈希值，并计算得出index值，确定其在table[]中的位置
        //其次，迭代index索引位置的链表，如果该位置处的链表存在相同的key，则替换value，返回旧的value
        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)) {
                V old = e.value;
                e.value = value;
                return old;
            }
        }

        modCount++;
        if (count >= threshold) {
```

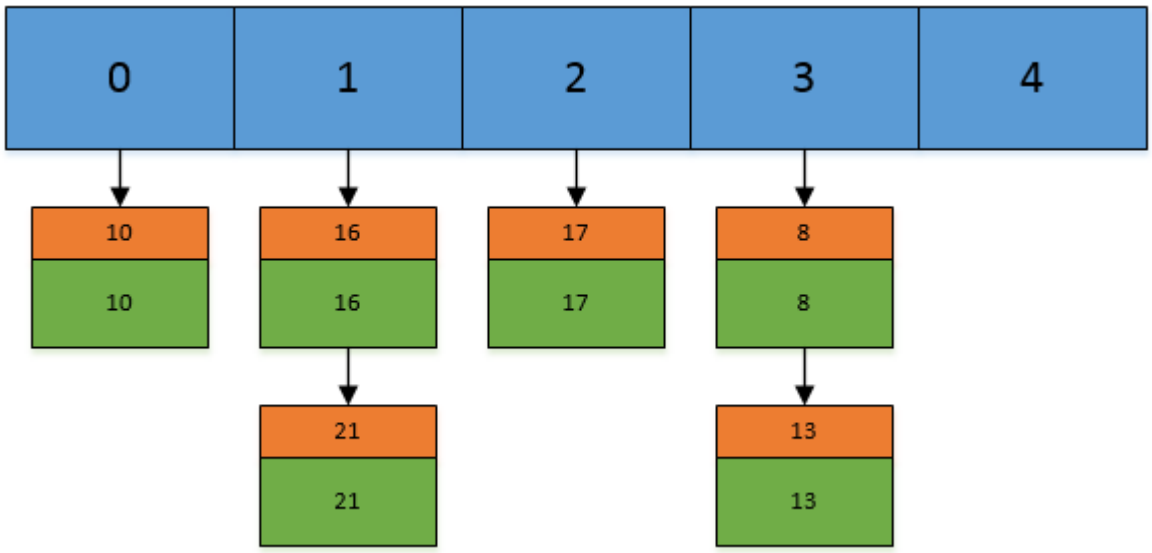
```
            // Rehash the table if the threshold is exceeded
            //如果超过阀值，就进行rehash操作
            rehash();

            tab = table;
            hash = hash(key);
            index = (hash & 0x7FFFFFFF) % tab.length;
        }

        // Creates the new entry.
        //将值插入，返回的为null
        Entry<K,V> e = tab[index];
        // 创建新的Entry节点，并将新的Entry插入Hashtable的index位置，并设置e为新的Entry的下一个元素
        tab[index] = new Entry<>(hash, key, value, e);
        count++;
        return null;
    }
```

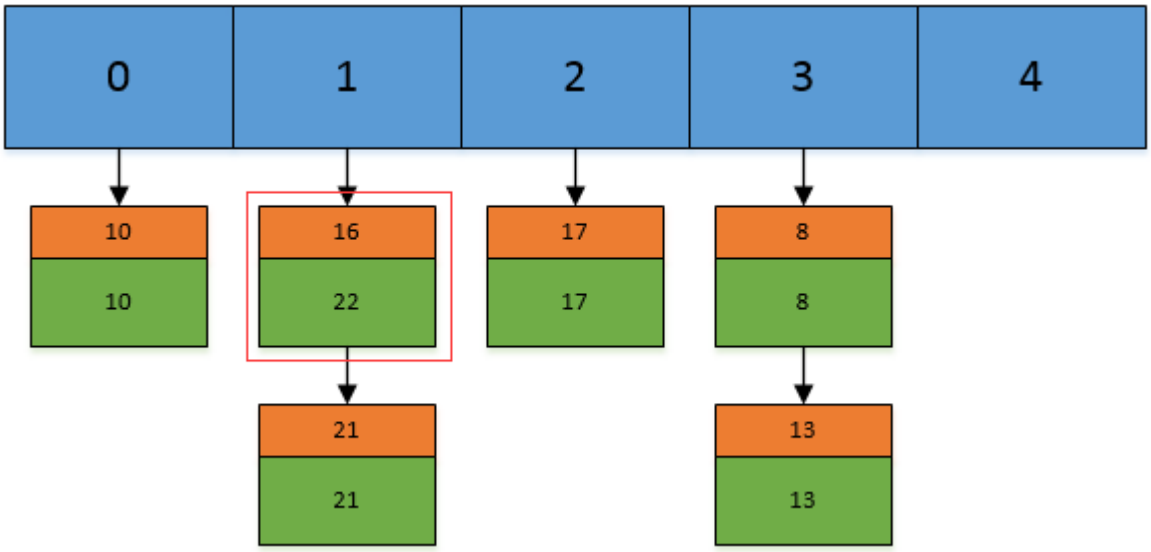
通过一个实际的例子来演示一下这个过程：

假设我们现在Hashtable的容量为5，已经存在了(5,5)，(13,13)，(16,16)，(17,17)，(21,21)这 5 个键值对，目前他们在Hashtable中的位置如下：



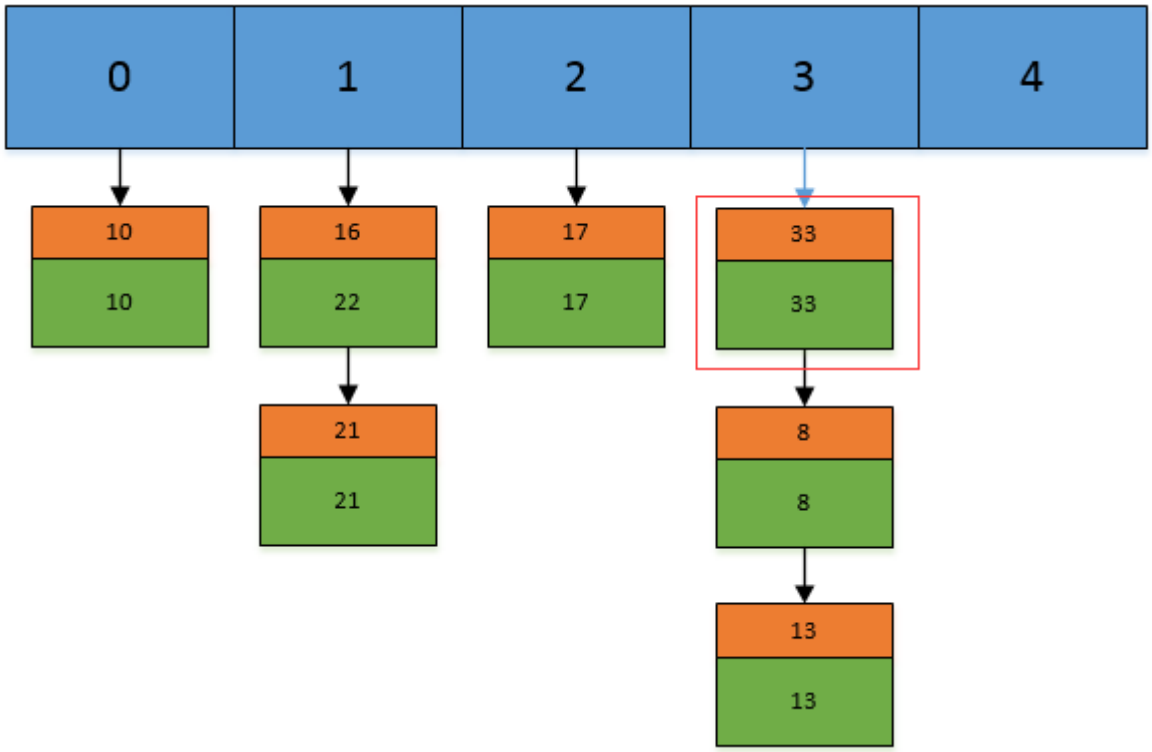
图片 3.1 图1

现在，我们插入一个新的键值对，put(16,22)，假设key=16的索引为1.但现在索引1的位置有两个Entry了，所以程序会对链表进行迭代。迭代的过程中，发现其中有一个Entry的key和我们要插入的键值对的key相同，所以现在会做的工作就是将newValue=22替换oldValue=16，然后返回oldValue=16.



图片 3.2 图2

然后我们现在再插入一个，put(33,33)，key=33的索引为3，并且在链表中也不存在key=33的Entry，所以将该节点插入链表的第一个位置。



图片 3.3 图3

get 方法

相比较于 put 方法，get 方法则简单很多。其过程就是首先通过 hash()方法求得 key 的哈希值，然后根据 hash 值得到 index 索引（上述两步所用的算法与 put 方法都相同）。然后迭代链表，返回匹配的 key 的对应的 value；找不到则返回 null。

```
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;
    }
```

Hashtable 遍历方式

Hashtable 有多种遍历方式：

```
//1、使用keys()
Enumeration<String> en1 = table.keys();
   while(en1.hasMoreElements()) {
   en1.nextElement();
}

//2、使用elements()
Enumeration<String> en2 = table.elements();
   while(en2.hasMoreElements()) {
   en2.nextElement();
}

//3、使用keySet()
Iterator<String> it1 = table.keySet().iterator();
   while(it1.hasNext()) {
   it1.next();
}

//4、使用entrySet()
Iterator<Entry<String, String>> it2 = table.entrySet().iterator();
   while(it2.hasNext()) {
   it2.next();
}
```

Hashtable 与 HashMap 的简单比较

1. HashTable 基于 Dictionary 类，而 HashMap 是基于 AbstractMap。Dictionary 是任何可将键映射到相应值的类的抽象父类，而 AbstractMap 是基于 Map 接口的实现，它以最大限度地减少实现此接口所需的工作。
2. HashMap 的 key 和 value 都允许为 null，而 Hashtable 的 key 和 value 都不允许为 null。HashMap 遇到 key 为 null 的时候，调用 putForNullKey 方法进行处理，而对 value 没有处理；Hashtable遇到 null，直接返回 NullPointerException。
3. Hashtable 方法是同步，而HashMap则不是。我们可以看一下源码，Hashtable 中的几乎所有的 public 的方法都是 synchronized 的，而有些方法也是在内部通过 synchronized 代码块来实现。所以有人一般都建议如果是涉及到多线程同步时采用 HashTable，没有涉及就采用 HashMap，但是在 Collections 类中存在一个静态方法：synchronizedMap()，该方法创建了一个线程安全的 Map 对象，并把它作为一个封装的对象来返回。

4

LinkedHashMap 的实现原理

LinkedHashMap 概述

HashMap 是无序的，HashMap 在 put 的时候是根据 key 的 hashcode 进行 hash 然后放入对应的地方。所以在按照一定顺序 put 进 HashMap 中，然后遍历出 HashMap 的顺序跟 put 的顺序不同（除非在 put 的时候 key 已经按照 hashcode 排序号了，这种几率非常小）

JAVA 在 JDK1.4 以后提供了 LinkedHashMap 来帮助我们实现了有序的 HashMap！

LinkedHashMap 是 HashMap 的一个子类，它保留插入的顺序，如果需要输出的顺序和输入时的相同，那么就选用 LinkedHashMap。

LinkedHashMap 是 Map 接口的哈希表和链接列表实现，具有可预知的迭代顺序。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

LinkedHashMap 实现与 HashMap 的不同之处在于，LinkedHashMap 维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。

注意，此实现不是同步的。如果多个线程同时访问链接的哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须保持外部同步。

根据链表中元素的顺序可以分为：按插入顺序的链表，和按访问顺序(调用 get 方法)的链表。默认是按插入顺序排序，如果指定按访问顺序排序，那么调用get方法后，会将这次访问的元素移至链表尾部，不断访问可以形成按访问顺序排序的链表。

小 Demo

我在最开始学习 LinkedHashMap 的时候，看到访问顺序、插入顺序等等，有点晕了，随着后续的学习才慢慢懂得其中原理，所以我会先在进行做几个 demo 来演示一下 LinkedHashMap 的使用。看懂了其效果，然后再来研究其原理。

HashMap

看下面这个代码：

```
public static void main(String[] args) {
    Map<String, String> map = new HashMap<String, String>();
    map.put("apple", "苹果");
    map.put("watermelon", "西瓜");
    map.put("banana", "香蕉");
    map.put("peach", "桃子");

    Iterator iter = map.entrySet().iterator();
    while (iter.hasNext()) {
        Map.Entry entry = (Map.Entry) iter.next();
        System.out.println(entry.getKey() + "=" + entry.getValue());
    }
}
```

一个比较简单的测试 HashMap 的代码，通过控制台的输出，我们可以看到 HashMap 是没有顺序的。

```
banana=香蕉
apple=苹果
peach=桃子
watermelon=西瓜
```

LinkedHashMap

我们现在将 map 的实现换成 LinkedHashMap，其他代码不变：`Map<String, String> map = new LinkedHashMap<String, String>();`

看一下控制台的输出：

```
apple=苹果
watermelon=西瓜
banana=香蕉
peach=桃子
```

我们可以看到，其输出顺序是完成按照插入顺序的！也就是我们上面所说的保留了插入的顺序。我们不是在上面还提到过其可以按照访问顺序进行排序么？好的，我们还是通过一个例子来验证一下：

```
public static void main(String[] args) {
    Map<String, String> map = new LinkedHashMap<String, String>(16,0.75f,true);
    map.put("apple", "苹果");
    map.put("watermelon", "西瓜");
    map.put("banana", "香蕉");
    map.put("peach", "桃子");

    map.get("banana");
    map.get("apple");

    Iterator iter = map.entrySet().iterator();
    while (iter.hasNext()) {
        Map.Entry entry = (Map.Entry) iter.next();
        System.out.println(entry.getKey() + "=" + entry.getValue());
    }
}
```

代码与之前的都差不多，但我们多了两行代码，并且初始化 LinkedHashMap 的时候，用的构造函数也不相同，看一下控制台的输出结果：

```
watermelon=西瓜
peach=桃子
banana=香蕉
apple=苹果
```

这也就是我们之前提到过的，LinkedHashMap 可以选择按照访问顺序进行排序。

LinkedHashMap 的实现

对于 LinkedHashMap 而言，它继承与 HashMap(public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>)、底层使用哈希表与双向链表来保存所有元素。其基本操作与父类 HashMap 相似，它通过重写父类相关的方法，来实现自己的链接列表特性。下面我们来分析 LinkedHashMap 的源代码：

成员变量

LinkedHashMap 采用的 hash 算法和 HashMap 相同，但是它重新定义了数组中保存的元素 Entry，该 Entry 除了保存当前对象的引用外，还保存了其上一个元素 before 和下一个元素 after 的引用，从而在哈希表的基础上又构成了双向链接列表。看源代码：

```
/**
* The iteration ordering method for this linked hash map: <tt>true</tt>
* for access-order, <tt>false</tt> for insertion-order.
* 如果为true，则按照访问顺序；如果为false，则按照插入顺序。
*/
private final boolean accessOrder;
/**
* 双向链表的表头元素。
 */
private transient Entry<K,V> header;

/**
* LinkedHashMap的Entry元素。
* 继承HashMap的Entry元素，又保存了其上一个元素before和下一个元素after的引用。
 */
private static class Entry<K,V> extends HashMap.Entry<K,V> {
    Entry<K,V> before, after;
    ……
}
```

LinkedHashMap 中的 Entry 集成与 HashMap 的 Entry，但是其增加了 before 和 after 的引用，指的是上一个元素和下一个元素的引用。

初始化

通过源代码可以看出，在 LinkedHashMap 的构造方法中，实际调用了父类 HashMap 的相关构造方法来构造一个底层存放的 table 数组，但额外可以增加 accessOrder 这个参数，如果不设置，默认为 false，代表按照插入顺序进行迭代；当然可以显式设置为 true，代表以访问顺序进行迭代。如：

```
public LinkedHashMap(int initialCapacity, float loadFactor,boolean accessOrder) {
    super(initialCapacity, loadFactor);
```

```
    this.accessOrder = accessOrder;
}
```

我们已经知道 LinkedHashMap 的 Entry 元素继承 HashMap 的 Entry，提供了双向链表的功能。在上述 HashMap 的构造器中，最后会调用 init() 方法，进行相关的初始化，这个方法在 HashMap 的实现中并无意义，只是提供给子类实现相关的初始化调用。

但在 LinkedHashMap 重写了 init() 方法，在调用父类的构造方法完成构造后，进一步实现了对其元素 Entry 的初始化操作。

```
/**
* Called by superclass constructors and pseudoconstructors (clone,
* readObject) before any entries are inserted into the map.  Initializes
* the chain.
*/
@Override
void init() {
  header = new Entry<>(-1, null, null, null);
  header.before = header.after = header;
}
```

存储

LinkedHashMap 并未重写父类 HashMap 的 put 方法，而是重写了父类 HashMap 的 put 方法调用的子方法 void recordAccess(HashMap m) ，void addEntry(int hash, K key, V value, int bucketIndex) 和void createEntry(int hash, K key, V value, int bucketIndex)，提供了自己特有的双向链接列表的实现。我们在之前的文章中已经讲解了HashMap的put方法，我们在这里重新贴一下 HashMap 的 put 方法的源代码：

HashMap.put:

```
public V put(K key, V value) {
        if (key == null)
            return putForNullKey(value);
        int hash = hash(key);
        int i = indexFor(hash, table.length);
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            Object k;
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }

        modCount++;
        addEntry(hash, key, value, i);
        return null;
}
```

重写方法：

```
void recordAccess(HashMap<K,V> m) {
    LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
    if (lm.accessOrder) {
        lm.modCount++;
        remove();
        addBefore(lm.header);
        }
}

void addEntry(int hash, K key, V value, int bucketIndex) {
    // 调用create方法，将新元素以双向链表的的形式加入到映射中。
    createEntry(hash, key, value, bucketIndex);

    // 删除最近最少使用元素的策略定义
    Entry<K,V> eldest = header.after;
    if (removeEldestEntry(eldest)) {
        removeEntryForKey(eldest.key);
    } else {
        if (size >= threshold)
            resize(2 * table.length);
    }
}

void createEntry(int hash, K key, V value, int bucketIndex) {
    HashMap.Entry<K,V> old = table[bucketIndex];
    Entry<K,V> e = new Entry<K,V>(hash, key, value, old);
    table[bucketIndex] = e;
    // 调用元素的addBrefore方法，将元素加入到哈希、双向链接列表。
    e.addBefore(header);
    size++;
}

private void addBefore(Entry<K,V> existingEntry) {
    after  = existingEntry;
    before = existingEntry.before;
    before.after = this;
    after.before = this;
}
```

读取

LinkedHashMap 重写了父类 HashMap 的 get 方法，实际在调用父类 getEntry() 方法取得查找的元素后，再判断当排序模式 accessOrder 为 true 时，记录访问顺序，将最新访问的元素添加到双向链表的表头，并从原来的位置删除。由于的链表的增加、删除操作是常量级的，故并不会带来性能的损失。

```
public V get(Object key) {
    // 调用父类HashMap的getEntry()方法，取得要查找的元素。
    Entry<K,V> e = (Entry<K,V>)getEntry(key);
    if (e == null)
        return null;
    // 记录访问顺序。
    e.recordAccess(this);
    return e.value;
}

void recordAccess(HashMap<K,V> m) {
    LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
    // 如果定义了LinkedHashMap的迭代顺序为访问顺序，
```

```
    // 则删除以前位置上的元素，并将最新访问的元素添加到链表表头。
    if (lm.accessOrder) {
        lm.modCount++;
        remove();
        addBefore(lm.header);
    }
}


/**
* Removes this entry from the linked list.
*/
private void remove() {
    before.after = after;
    after.before = before;
}

/**clear链表，设置header为初始状态*/
public void clear() {
 super.clear();
 header.before = header.after = header;
}
```

排序模式

LinkedHashMap 定义了排序模式 accessOrder，该属性为 boolean 型变量，对于访问顺序，为 true；对于插入顺序，则为 false。一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。

这些构造方法都会默认指定排序模式为插入顺序。如果你想构造一个 LinkedHashMap，并打算按从近期访问最少到近期访问最多的顺序（即访问顺序）来保存元素，那么请使用下面的构造方法构造 LinkedHashMap：public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)

该哈希映射的迭代顺序就是最后访问其条目的顺序，这种映射很适合构建 LRU 缓存。LinkedHashMap 提供了 removeEldestEntry(Map.Entry<K,V> eldest) 方法。该方法可以提供在每次添加新条目时移除最旧条目的实现程序，默认返回 false，这样，此映射的行为将类似于正常映射，即永远不能移除最旧的元素。

我们会在后面的文章中详细介绍关于如何用 LinkedHashMap 构建 LRU 缓存。

总结

其实 LinkedHashMap 几乎和 HashMap 一样：从技术上来说，不同的是它定义了一个 Entry<K,V> header，这个 header 不是放在 Table 里，它是额外独立出来的。LinkedHashMap 通过继承 hashMap 中的 Entry<K,V>,并添加两个属性 Entry<K,V> before,after,和 header 结合起来组成一个双向链表，来实现按插入顺序或访问顺序排序。

在写关于 LinkedHashMap 的过程中，记起来之前面试的过程中遇到的一个问题，也是问我 Map 的哪种实现可以做到按照插入顺序进行迭代？当时脑子是突然短路的，但现在想想，也只能怪自己对这个知识点还是掌握的不够扎实，所以又从头认真的把代码看了一遍。

不过，我的建议是，大家首先首先需要记住的是：LinkedHashMap 能够做到按照插入顺序或者访问顺序进行迭代，这样在我们以后的开发中遇到相似的问题，才能想到用 LinkedHashMap 来解决，否则就算对其内部结构非常了解，不去使用也是没有什么用的。

我们学习的目的是为了更好的应用。

5

LinkedHashSet 的实现原理

LinkedHashSet 概述

思考了好久，到底要不要总结 LinkedHashSet 的内容 = = 我在之前的博文中，分别写了 HashMap 和 HashSet，然后我们可以看到 HashSet 的方法基本上都是基于 HashMap 来实现的，说白了，HashSet内部的数据结构就是一个 HashMap，其方法的内部几乎就是在调用 HashMap 的方法。

LinkedHashSet 首先我们需要知道的是它是一个 Set 的实现，所以它其中存的肯定不是键值对，而是值。此实现与 HashSet 的不同之处在于，LinkedHashSet 维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可为插入顺序或是访问顺序。

看到上面的介绍，是不是感觉其与 HashMap 和 LinkedHashMap 的关系很像?

注意，此实现不是同步的。如果多个线程同时访问链接的哈希Set，而其中至少一个线程修改了该 Set，则它必须保持外部同步。

小 Demo

在LinkedHashMap的实现原理中，通过例子演示了 HashMap 和 LinkedHashMap 的区别。举一反三，我们现在学习的LinkedHashSet与之前的很相同，只不过之前存的是键值对，而现在存的只有值。

所以我就不再具体的贴代码在这边了，但我们可以肯定的是，LinkedHashSet 是可以按照插入顺序或者访问顺序进行迭代。

LinkedHashSet 的实现

对于 LinkedHashSet 而言，它继承与 HashSet、又基于 LinkedHashMap 来实现的。

LinkedHashSet 底层使用 LinkedHashMap 来保存所有元素，它继承与 HashSet，其所有的方法操作上又与 HashSet 相同，因此 LinkedHashSet 的实现上非常简单，只提供了四个构造方法，并通过传递一个标识参数，调用父类的构造器，底层构造一个 LinkedHashMap 来实现，在相关操作上与父类 HashSet 的操作相同，直接调用父类 HashSet 的方法即可。LinkedHashSet 的源代码如下：

```
public class LinkedHashSet<E>
    extends HashSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {

    private static final long serialVersionUID = -2851667679971038690L;

    /**
     * 构造一个带有指定初始容量和加载因子的新空链接哈希set。
     *
     * 底层会调用父类的构造方法，构造一个有指定初始容量和加载因子的LinkedHashMap实例。
     * @param initialCapacity 初始容量。
     * @param loadFactor 加载因子。
     */
    public LinkedHashSet(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor, true);
    }

    /**
     * 构造一个带指定初始容量和默认加载因子0.75的新空链接哈希set。
     *
     * 底层会调用父类的构造方法，构造一个带指定初始容量和默认加载因子0.75的LinkedHashMap实例。
     * @param initialCapacity 初始容量。
     */
    public LinkedHashSet(int initialCapacity) {
        super(initialCapacity, .75f, true);
    }

    /**
     * 构造一个带默认初始容量16和加载因子0.75的新空链接哈希set。
     *
     * 底层会调用父类的构造方法，构造一个带默认初始容量16和加载因子0.75的LinkedHashMap实例。
     */
    public LinkedHashSet() {
        super(16, .75f, true);
    }

    /**
     * 构造一个与指定collection中的元素相同的新链接哈希set。
     *
     * 底层会调用父类的构造方法，构造一个足以包含指定collection
     * 中所有元素的初始容量和加载因子为0.75的LinkedHashMap实例。
     * @param c 其中的元素将存放在此set中的collection。
     */
    public LinkedHashSet(Collection<? extends E> c) {
        super(Math.max(2*c.size(), 11), .75f, true);
        addAll(c);
```

```
    }
}
```

以上几乎就是 LinkedHashSet 的全部代码了，那么读者可能就会怀疑了，不是说 LinkedHashSet 是基于 LinkedHashMap 实现的吗？那我为什么在源码中甚至都没有看到出现过 LinkedHashMap。不要着急，我们可以看到在 LinkedHashSet 的构造方法中，其调用了父类的构造方法。我们可以进去看一下：

```
/**
     * 以指定的initialCapacity和loadFactor构造一个新的空链接哈希集合。
     * 此构造函数为包访问权限，不对外公开，实际只是是对LinkedHashSet的支持。
     *
     * 实际底层会以指定的参数构造一个空LinkedHashMap实例来实现。
     * @param initialCapacity 初始容量。
     * @param loadFactor 加载因子。
     * @param dummy 标记。
     */
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
    map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);
}
```

在父类 HashSet 中，专为 LinkedHashSet 提供的构造方法如下，该方法为包访问权限，并未对外公开。

由上述源代码可见，LinkedHashSet 通过继承 HashSet，底层使用 LinkedHashMap，以很简单明了的方式来实现了其自身的所有功能。

总结

以上就是关于 LinkedHashSet 的内容，我们只是从概述上以及构造方法这几个方面介绍了，并不是我们不想去深入其读取或者写入方法，而是其本身没有实现，只是继承于父类 HashSet 的方法。

所以我们需要注意的点是：

- LinkedHashSet 是 Set 的一个具体实现，其维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可为插入顺序或是访问顺序。
- LinkedHashSet 继承与 HashSet，并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的LinkedHashMap 其内部是基于 Hashmap 实现一样，不过还是有一点点区别的（具体的区别大家可以自己去思考一下）。
- 如果我们需要迭代的顺序为插入顺序或者访问顺序，那么 LinkedHashSet 是需要你首先考虑的。

6

ArrayList 的实现原理

ArrayList 概述

ArrayList 可以理解为动态数组，用 MSDN 中的说法，就是 Array 的复杂版本。与 Java 中的数组相比，它的容量能动态增长。ArrayList 是 List 接口的可变数组的实现。实现了所有可选列表操作，并允许包括 null 在内的所有元素。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。（此类大致上等同于 Vector 类，除了此类是不同步的。）

每个 ArrayList 实例都有一个容量，该容量是指用来存储列表元素的数组的大小。它总是至少等于列表的大小。随着向 ArrayList 中不断添加元素，其容量也自动增长。自动增长会带来数据向新数组的重新拷贝，因此，如果可预知数据量的多少，可在构造 ArrayList 时指定其容量。在添加大量元素前，应用程序也可以使用 ensureCapacity 操作来增加 ArrayList 实例的容量，这可以减少递增式再分配的数量。

注意，此实现不是同步的。如果多个线程同时访问一个 ArrayList 实例，而其中至少一个线程从结构上修改了列表，那么它必须保持外部同步。（结构上的修改是指任何添加或删除一个或多个元素的操作，或者显式调整底层数组的大小；仅仅设置元素的值不是结构上的修改。）

我们先学习了解其内部的实现原理，才能更好的理解其应用。

ArrayList 的实现

对于 ArrayList 而言，它实现 List 接口、底层使用数组保存所有元素。其操作基本上是对数组的操作。下面我们来分析 ArrayList 的源代码：

实现的接口

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
}
```

ArrayList 继承了 AbstractList，实现了 List。它是一个数组队列，提供了相关的添加、删除、修改、遍历等功能。

ArrayList 实现了 RandmoAccess 接口，即提供了随机访问功能。RandmoAccess 是 java 中用来被 List 实现，为 List 提供快速访问功能的。在 ArrayList 中，我们即可以通过元素的序号快速获取元素对象；这就是快速随机访问。

ArrayList 实现了 Cloneable 接口，即覆盖了函数 clone()，能被克隆。 ArrayList 实现 java.io.Serializable 接口，这意味着 ArrayList 支持序列化，能通过序列化去传输。

底层使用数组实现

```
/**
* The array buffer into which the elements of the ArrayList are stored.
* The capacity of the ArrayList is the length of this array buffer.
*/
private transient Object[] elementData;
```

构造方法

```
    /**
     * Constructs an empty list with an initial capacity of ten.
     */
    public ArrayList() {
        this(10);
    }
    /**
     * Constructs an empty list with the specified initial capacity.
     *
     * @param  initialCapacity  the initial capacity of the list
```

```
     * @throws IllegalArgumentException if the specified initial capacity
     *         is negative
     */
    public ArrayList(int initialCapacity) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        this.elementData = new Object[initialCapacity];
    }

    /**
     * Constructs a list containing the elements of the specified
     * collection, in the order they are returned by the collection's
     * iterator.
     *
     * @param c the collection whose elements are to be placed into this list
     * @throws NullPointerException if the specified collection is null
     */
    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        size = elementData.length;
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    }
```

ArrayList 提供了三种方式的构造器：

1. `public ArrayList()` 可以构造一个默认初始容量为10的空列表；
2. `public ArrayList(int initialCapacity)` 构造一个指定初始容量的空列表；
3. `public ArrayList(Collection<? extends E> c)` 构造一个包含指定 collection 的元素的列表，这些元素按照该collection的迭代器返回它们的顺序排列的。

存储

ArrayList 中提供了多种添加元素的方法，下面将一一进行讲解：

1.set(int index, E element)：该方法首先调用 `rangeCheck(index)` 来校验 index 变量是否超出数组范围，超出则抛出异常。而后，取出原 index 位置的值，并且将新的 element 放入 Index 位置，返回 oldValue。

```
    /**
     * Replaces the element at the specified position in this list with
     * the specified element.
     *
     * @param index index of the element to replace
     * @param element element to be stored at the specified position
     * @return the element previously at the specified position
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    public E set(int index, E element) {
        rangeCheck(index);
```

```
    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;
}
/**
 * Checks if the given index is in range.  If not, throws an appropriate
 * runtime exception.  This method does *not* check if the index is
 * negative: It is always used immediately prior to an array access,
 * which throws an ArrayIndexOutOfBoundsException if index is negative.
 */
private void rangeCheck(int index) {
  if (index >= size)
  throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

2.add(E e)：该方法是将指定的元素添加到列表的尾部。当容量不足时，会调用 grow 方法增长容量。

```
/**
 * Appends the specified element to the end of this list.
 *
 * @param e element to be appended to this list
 * @return <tt>true</tt> (as specified by {@link Collection#add})
 */
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
private void ensureCapacityInternal(int minCapacity) {
    modCount++;
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

3.add(int index, E element)：在 index 位置插入 element。

```
/**
 * Inserts the specified element at the specified position in this
 * list. Shifts the element currently at that position (if any) and
 * any subsequent elements to the right (adds one to their indices).
 *
 * @param index index at which the specified element is to be inserted
 * @param element element to be inserted
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
```

```
                size - index);
        elementData[index] = element;
        size++;
    }
```

4. addAll(Collection<? extends E> c) 和 addAll(int index, Collection<? extends E> c)：将特定 Collection 中的元素添加到 Arraylist 末尾。

```
/**
     * Appends all of the elements in the specified collection to the end of
     * this list, in the order that they are returned by the
     * specified collection's Iterator.  The behavior of this operation is
     * undefined if the specified collection is modified while the operation
     * is in progress.  (This implies that the behavior of this call is
     * undefined if the specified collection is this list, and this
     * list is nonempty.)
     *
     * @param c collection containing elements to be added to this list
     * @return <tt>true</tt> if this list changed as a result of the call
     * @throws NullPointerException if the specified collection is null
     */
    public boolean addAll(Collection<? extends E> c) {
        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacityInternal(size + numNew);  // Increments modCount
        System.arraycopy(a, 0, elementData, size, numNew);
        size += numNew;
        return numNew != 0;
    }

    /**
     * Inserts all of the elements in the specified collection into this
     * list, starting at the specified position.  Shifts the element
     * currently at that position (if any) and any subsequent elements to
     * the right (increases their indices).  The new elements will appear
     * in the list in the order that they are returned by the
     * specified collection's iterator.
     *
     * @param index index at which to insert the first element from the
     *              specified collection
     * @param c collection containing elements to be added to this list
     * @return <tt>true</tt> if this list changed as a result of the call
     * @throws IndexOutOfBoundsException {@inheritDoc}
     * @throws NullPointerException if the specified collection is null
     */
    public boolean addAll(int index, Collection<? extends E> c) {
        rangeCheckForAdd(index);

        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacityInternal(size + numNew);  // Increments modCount

        int numMoved = size - index;
        if (numMoved > 0)
            System.arraycopy(elementData, index, elementData, index + numNew,
                             numMoved);

        System.arraycopy(a, 0, elementData, index, numNew);
        size += numNew;
        return numNew != 0;
    }
```

在 ArrayList 的存储方法，其核心本质是在数组的某个位置将元素添加进入。但其中又会涉及到关于数组容量不够而增长等因素。

读取

这个方法就比较简单了，ArrayList 能够支持随机访问的原因也是很显然的，因为它内部的数据结构是数组，而数组本身就是支持随机访问。该方法首先会判断输入的index值是否越界，然后将数组的 index 位置的元素返回即可。

```
/**
* Returns the element at the specified position in this list.
*
* @param  index index of the element to return
* @return the element at the specified position in this list
* @throws IndexOutOfBoundsException {@inheritDoc}
*/
public E get(int index) {
   rangeCheck(index);
   return (E) elementData[index];
}
private void rangeCheck(int index) {
   if (index >= size)
   throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

删除

ArrayList 提供了根据下标或者指定对象两种方式的删除功能。需要注意的是该方法的返回值并不相同，如下：

```
   /**
    * Removes the element at the specified position in this list.
    * Shifts any subsequent elements to the left (subtracts one from their
    * indices).
    *
    * @param index the index of the element to be removed
    * @return the element that was removed from the list
    * @throws IndexOutOfBoundsException {@inheritDoc}
    */
   public E remove(int index) {
      rangeCheck(index);

      modCount++;
      E oldValue = elementData(index);

      int numMoved = size - index - 1;
      if (numMoved > 0)
         System.arraycopy(elementData, index+1, elementData, index,
                          numMoved);
      elementData[--size] = null; // Let gc do its work

      return oldValue;
   }
```

```
/**
     * Removes the first occurrence of the specified element from this list,
     * if it is present.  If the list does not contain the element, it is
     * unchanged.  More formally, removes the element with the lowest index
     * <tt>i</tt> such that
     * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>
     * (if such an element exists).  Returns <tt>true</tt> if this list
     * contained the specified element (or equivalently, if this list
     * changed as a result of the call).
     *
     * @param o element to be removed from this list, if present
     * @return <tt>true</tt> if this list contained the specified element
     */
    public boolean remove(Object o) {
        if (o == null) {
            for (int index = 0; index < size; index++)
                if (elementData[index] == null) {
                    fastRemove(index);
                    return true;
                }
        } else {
            for (int index = 0; index < size; index++)
                if (o.equals(elementData[index])) {
                    fastRemove(index);
                    return true;
                }
        }
        return false;
    }
```

注意：从数组中移除元素的操作，也会导致被移除的元素以后的所有元素的向左移动一个位置。

调整数组容量

从上面介绍的向 ArrayList 中存储元素的代码中，我们看到，每当向数组中添加元素时，都要去检查添加后元素的个数是否会超出当前数组的长度，如果超出，数组将会进行扩容，以满足添加数据的需求。数组扩容有两个方法，其中开发者可以通过一个 public 的方法 `ensureCapacity(int minCapacity)` 来增加 ArrayList 的容量，而在存储元素等操作过程中，如果遇到容量不足，会调用priavte方法 `private void ensureCapacityInternal(int minCapacity)` 实现。

```
    public void ensureCapacity(int minCapacity) {
        if (minCapacity > 0)
            ensureCapacityInternal(minCapacity);
    }

    private void ensureCapacityInternal(int minCapacity) {
        modCount++;
        // overflow-conscious code
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }
    /**
     * Increases the capacity to ensure that it can hold at least the
     * number of elements specified by the minimum capacity argument.
     *
     * @param minCapacity the desired minimum capacity
```

```
     */
    private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

从上述代码中可以看出，数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长大约是其原容量的 1.5 倍（从 `int newCapacity = oldCapacity + (oldCapacity >> 1)` 这行代码得出）。这种操作的代价是很高的，因此在实际使用时，我们应该尽量避免数组容量的扩张。当我们可预知要保存的元素的多少时，要在构造 ArrayList 实例时，就指定其容量，以避免数组扩容的发生。或者根据实际需求，通过调用ensure Capacity 方法来手动增加 ArrayList 实例的容量。

Fail-Fast 机制

ArrayList 也采用了快速失败的机制，通过记录 modCount 参数来实现。在面对并发的修改时，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险。 关于 Fail-Fast 的更详细的介绍，我在之前将 HashMap 中已经提到。

7

LinkedList 的实现原理

概述

LinkedList 和 ArrayList 一样，都实现了 List 接口，但其内部的数据结构有本质的不同。LinkedList 是基于链表实现的（通过名字也能区分开来），所以它的插入和删除操作比 ArrayList 更加高效。但也是由于其为基于链表的，所以随机访问的效率要比 ArrayList 差。

看一下 LinkedList 的类的定义：

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{}
```

LinkedList 继承自 AbstractSequenceList，实现了 List、Deque、Cloneable、java.io.Serializable 接口。AbstractSequenceList 提供了List接口骨干性的实现以减少实现 List 接口的复杂度，Deque 接口定义了双端队列的操作。

在 LinkedList 中除了本身自己的方法外，还提供了一些可以使其作为栈、队列或者双端队列的方法。这些方法可能彼此之间只是名字不同，以使得这些名字在特定的环境中显得更加合适。

LinkedList 也是 fail-fast 的（前边提过很多次了）。

LinkedList 源码解读

数据结构

LinkedList 是基于链表结构实现，所以在类中包含了 first 和 last 两个指针(Node)。Node 中包含了上一个节点和下一个节点的引用，这样就构成了双向的链表。每个 Node 只能知道自己的前一个节点和后一个节点，但对于链表来说，这已经足够了。

```
    transient int size = 0;
    transient Node<E> first; //链表的头指针
    transient Node<E> last; //尾指针
    //存储对象的结构 Node, LinkedList的内部类
    private static class Node<E> {
        E item;
        Node<E> next; // 指向下一个节点
        Node<E> prev; //指向上一个节点

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
```

存储

add(E e)

该方法是在链表的 end 添加元素，其调用了自己的方法 linkLast(E e)。

该方法首先将 last 的 Node 引用指向了一个新的 Node(l)，然后根据l新建了一个 newNode，其中的元素就为要添加的 e；而后，我们让 last 指向了 newNode。接下来是自身进行维护该链表。

```
/**
     * Appends the specified element to the end of this list.
     *
     * <p>This method is equivalent to {@link #addLast}.
     *
     * @param e element to be appended to this list
     * @return {@code true} (as specified by {@link Collection#add})
     */
public boolean add(E e) {
    linkLast(e);
    return true;
}
/**
* Links e as last element.
*/
```

```
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

add(int index, E element)

该方法是在指定 index 位置插入元素。如果 index 位置正好等于 size，则调用 linkLast(element) 将其插入末尾；否则调用 linkBefore(element, node(index))方法进行插入。该方法的实现在下面，大家可以自己仔细的分析一下。（分析链表的时候最好能够边画图边分析）

```
/**
    * Inserts the specified element at the specified position in this list.
    * Shifts the element currently at that position (if any) and any
    * subsequent elements to the right (adds one to their indices).
    *
    * @param index index at which the specified element is to be inserted
    * @param element element to be inserted
    * @throws IndexOutOfBoundsException {@inheritDoc}
    */
   public void add(int index, E element) {
       checkPositionIndex(index);

       if (index == size)
           linkLast(element);
       else
           linkBefore(element, node(index));
   }
   /**
        * Inserts element e before non-null Node succ.
        */
       void linkBefore(E e, Node<E> succ) {
           // assert succ != null;
           final Node<E> pred = succ.prev;
           final Node<E> newNode = new Node<>(pred, e, succ);
           succ.prev = newNode;
           if (pred == null)
               first = newNode;
           else
               pred.next = newNode;
           size++;
           modCount++;
       }
```

LinkedList 的方法实在是太多，在这没法一一举例分析。但很多方法其实都只是在调用别的方法而已，所以建议大家将其几个最核心的添加的方法搞懂就可以了，比如 linkBefore、linkLast。其本质也就是链表之间的删除添加等。

8

ConcurrentHashMap 的实现原理

概述

我们在之前的博文中了解到关于 HashMap 和 Hashtable 这两种集合。其中 HashMap 是非线程安全的，当我们只有一个线程在使用 HashMap 的时候，自然不会有问题，但如果涉及到多个线程，并且有读有写的过程中，HashMap 就不能满足我们的需要了(fail-fast)。在不考虑性能问题的时候，我们的解决方案有 Hashtable 或者Collections.synchronizedMap(hashMap)，这两种方式基本都是对整个 hash 表结构做锁定操作的，这样在锁表的期间，别的线程就需要等待了，无疑性能不高。

所以我们在本文中学习一个 util.concurrent 包的重要成员，ConcurrentHashMap。

ConcurrentHashMap 的实现是依赖于 Java 内存模型，所以我们在了解 ConcurrentHashMap 的前提是必须了解Java 内存模型。但 Java 内存模型并不是本文的重点，所以我假设读者已经对 Java 内存模型有所了解。

ConcurrentHashMap 分析

ConcurrentHashMap 的结构是比较复杂的，都深究去本质，其实也就是数组和链表而已。我们由浅入深慢慢的分析其结构。

先简单分析一下，ConcurrentHashMap 的成员变量中，包含了一个 Segment 的数组（ `final Segment<K,V>[] segments;` ），而 Segment 是 ConcurrentHashMap 的内部类，然后在 Segment 这个类中，包含了一个 HashEntry 的数组（ `transient volatile HashEntry<K,V>[] table;` ）。而 HashEntry 也是 ConcurrentHashMap 的内部类。HashEntry 中，包含了 key 和 value 以及 next 指针（类似于 HashMap 中 Entry），所以 HashEntry 可以构成一个链表。

所以通俗的讲，ConcurrentHashMap 数据结构为一个 Segment 数组，Segment 的数据结构为 HashEntry 的数组，而 HashEntry 存的是我们的键值对，可以构成链表。

首先，我们看一下 HashEntry 类。

HashEntry

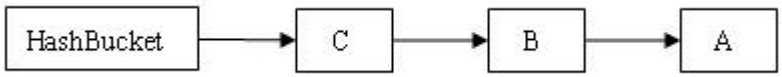
HashEntry 用来封装散列映射表中的键值对。在 HashEntry 类中，key，hash 和 next 域都被声明为 final 型，value 域被声明为 volatile 型。其类的定义为：

```
static final class HashEntry<K,V> {
        final int hash;
        final K key;
        volatile V value;
        volatile HashEntry<K,V> next;

        HashEntry(int hash, K key, V value, HashEntry<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
        ...
        ...
}
```

HashEntry 的学习可以类比着 HashMap 中的 Entry。我们的存储键值对的过程中，散列的时候如果发生“碰撞”，将采用“分离链表法”来处理碰撞：把碰撞的 HashEntry 对象链接成一个链表。

如下图，我们在一个空桶中插入 A、B、C 两个 HashEntry 对象后的结构图（其实应该为键值对，在这进行了简化以方便更容易理解）：



图片 8.1 图1

Segment

Segment 的类定义为 `static final class Segment<K,V> extends ReentrantLock implements Serializable` 。其继承于 ReentrantLock 类，从而使得 Segment 对象可以充当锁的角色。Segment 中包含HashEntry 的数组，其可以守护其包含的若干个桶（HashEntry的数组）。Segment 在某些意义上有点类似于 HashMap 了，都是包含了一个数组，而数组中的元素可以是一个链表。

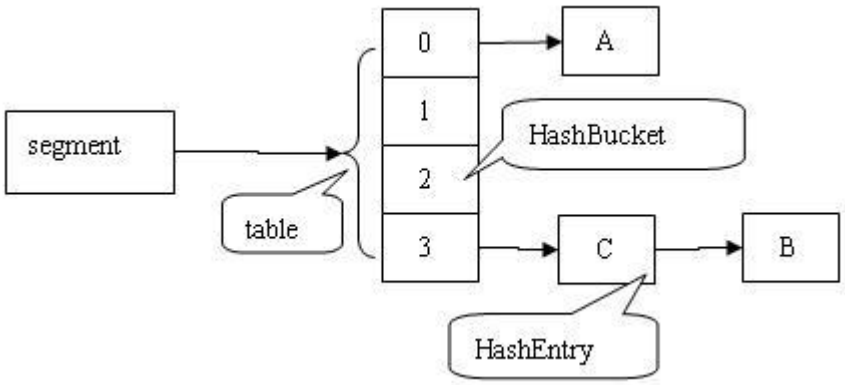
table:table 是由 HashEntry 对象组成的数组如果散列时发生碰撞，碰撞的 HashEntry 对象就以链表的形式链接成一个链表table数组的数组成员代表散列映射表的一个桶每个 table 守护整个 ConcurrentHashMap 包含桶总数的一部分如果并发级别为 16，table 则守护 ConcurrentHashMap 包含的桶总数的 1/16。

count 变量是计算器，表示每个 Segment 对象管理的 table 数组（若干个 HashEntry 的链表）包含的HashEntry 对象的个数。之所以在每个Segment对象中包含一个 count 计数器，而不在 ConcurrentHashMap 中使用全局的计数器，是为了避免出现“热点域”而影响并发性。

```
/**
     * Segments are specialized versions of hash tables.  This
     * subclasses from ReentrantLock opportunistically, just to
     * simplify some locking and avoid separate construction.
     */
    static final class Segment<K,V> extends ReentrantLock implements Serializable {
        /**
         * The per-segment table. Elements are accessed via
         * entryAt/setEntryAt providing volatile semantics.
         */
        transient volatile HashEntry<K,V>[] table;

        /**
         * The number of elements. Accessed only either within locks
         * or among other volatile reads that maintain visibility.
         */
        transient int count;
        transient int modCount;
        /**
         * 装载因子
         */
        final float loadFactor;
    }
```

我们通过下图来展示一下插入 ABC 三个节点后，Segment 的示意图：



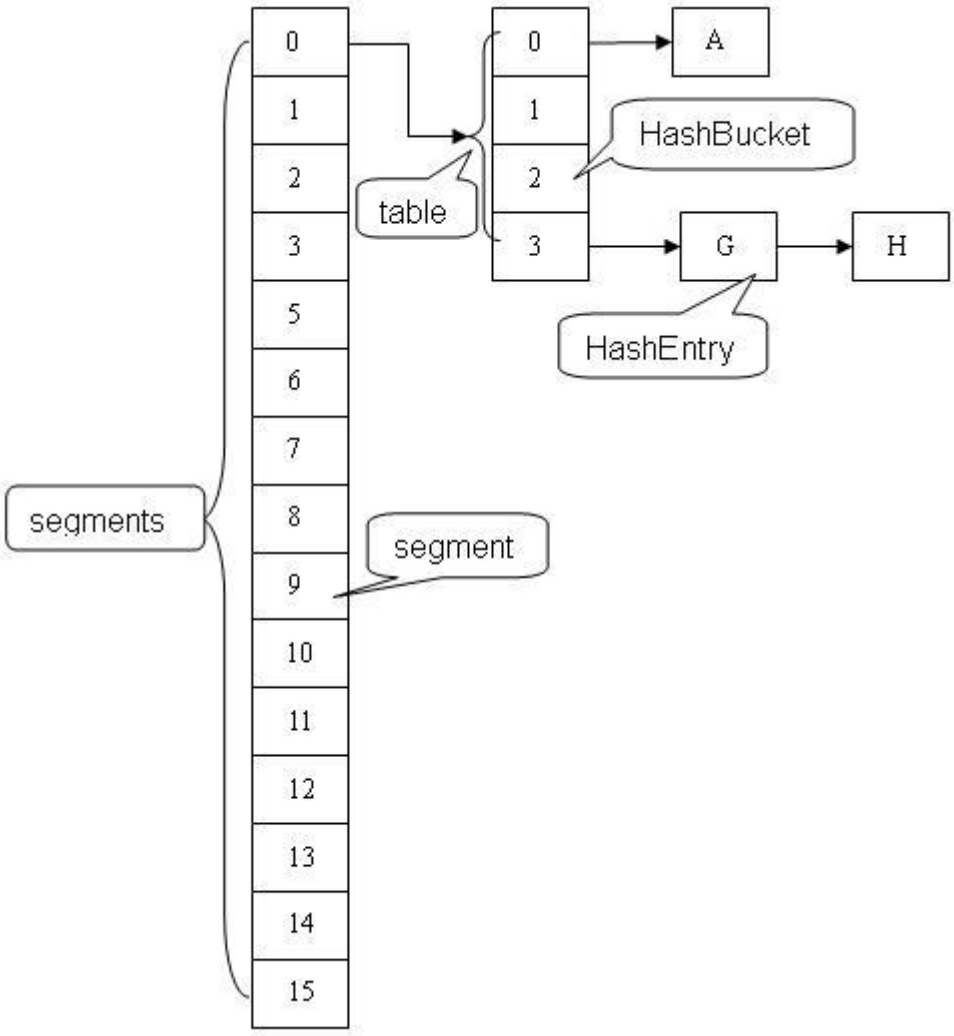
图片 8.2 图2

其实从我个人角度来说，Segment结构是与HashMap很像的。

ConcurrentHashMap

ConcurrentHashMap 的结构中包含的 Segment 的数组，在默认的并发级别会创建包含 16 个 Segment 对象的数组。通过我们上面的知识，我们知道每个 Segment 又包含若干个散列表的桶，每个桶是由 HashEntry 链接起来的一个链表。如果 key 能够均匀散列，每个 Segment 大约守护整个散列表桶总数的 1/16。

下面我们还有通过一个图来演示一下 ConcurrentHashMap 的结构：


0
1
2
3
5
6
7
8
9
10
11
12
13
14
15
segments
segment
table
0
1
2
3
A
HashBucket
G
H
HashEntry

图片 8.3 图3

并发写操作

在 ConcurrentHashMap 中，当执行 put 方法的时候，会需要加锁来完成。我们通过代码来解释一下具体过程： 当我们 new 一个 ConcurrentHashMap 对象，并且执行put操作的时候，首先会执行 ConcurrentHashMap 类中的 put 方法，该方法源码为：

```
/**
     * Maps the specified key to the specified value in this table.
     * Neither the key nor the value can be null.
     *
     * <p> The value can be retrieved by calling the <tt>get</tt> method
     * with a key that is equal to the original key.
     *
     * @param key key with which the specified value is to be associated
     * @param value value to be associated with the specified key
     * @return the previous value associated with <tt>key</tt>, or
     *         <tt>null</tt> if there was no mapping for <tt>key</tt>
     * @throws NullPointerException if the specified key or value is null
     */
    @SuppressWarnings("unchecked")
    public V put(K key, V value) {
        Segment<K,V> s;
        if (value == null)
            throw new NullPointerException();
        int hash = hash(key);
        int j = (hash >>> segmentShift) & segmentMask;
        if ((s = (Segment<K,V>)UNSAFE.getObject          // nonvolatile; recheck
             (segments, (j << SSHIFT) + SBASE)) == null) //  in ensureSegment
            s = ensureSegment(j);
        return s.put(key, hash, value, false);
    }
```

我们通过注释可以了解到，ConcurrentHashMap 不允许空值。该方法首先有一个 Segment 的引用 s，然后会通过 hash() 方法对 key 进行计算，得到哈希值；继而通过调用 Segment 的 put(K key, int hash, V value, boolean onlyIfAbsent)方法进行存储操作。该方法源码为：

```
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    //加锁，这里是锁定的Segment而不是整个ConcurrentHashMap
    HashEntry<K,V> node = tryLock() ? null :scanAndLockForPut(key, hash, value);
    V oldValue;
    try {
        HashEntry<K,V>[] tab = table;
        //得到hash对应的table中的索引index
        int index = (tab.length - 1) & hash;
        //找到hash对应的是具体的哪个桶，也就是哪个HashEntry链表
        HashEntry<K,V> first = entryAt(tab, index);
        for (HashEntry<K,V> e = first;;) {
            if (e != null) {
                K k;
                if ((k = e.key) == key ||
                    (e.hash == hash && key.equals(k))) {
                    oldValue = e.value;
                    if (!onlyIfAbsent) {
                        e.value = value;
```

```
                ++modCount;
            }
            break;
        }
        e = e.next;
      }
      else {
        if (node != null)
          node.setNext(first);
        else
          node = new HashEntry<K,V>(hash, key, value, first);
        int c = count + 1;
        if (c > threshold && tab.length < MAXIMUM_CAPACITY)
          rehash(node);
        else
          setEntryAt(tab, index, node);
        ++modCount;
        count = c;
        oldValue = null;
        break;
      }
    }
  } finally {
    //解锁
    unlock();
  }
  return oldValue;
}
```

关于该方法的某些关键步骤，在源码上加上了注释。

需要注意的是：加锁操作是针对的 hash 值对应的某个 Segment，而不是整个 ConcurrentHashMap。因为 put 操作只是在这个 Segment 中完成，所以并不需要对整个 ConcurrentHashMap 加锁。所以，此时，其他的线程也可以对另外的 Segment 进行 put 操作，因为虽然该 Segment 被锁住了，但其他的 Segment 并没有加锁。同时，读线程并不会因为本线程的加锁而阻塞。

正是因为其内部的结构以及机制，所以 ConcurrentHashMap 在并发访问的性能上要比Hashtable和同步包装之后的HashMap的性能提高很多。在理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作（如果并发级别设置为 16），及任意数量线程的读操作。

总结

在实际的应用中，散列表一般的应用场景是：除了少数插入操作和删除操作外，绝大多数都是读取操作，而且读操作在大多数时候都是成功的。正是基于这个前提，ConcurrentHashMap 针对读操作做了大量的优化。通过 HashEntry 对象的不变性和用 volatile 型变量协调线程间的内存可见性，使得 大多数时候，读操作不需要加锁就可以正确获得值。这个特性使得 ConcurrentHashMap 的并发性能在分离锁的基础上又有了近一步的提高。

ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于 HashTable 和用同步包装器包装的 HashMap（ Collections.synchronizedMap(new HashMap()) ），ConcurrentHashMap 拥有更高的并发性。在 HashTable 和由同步包装器包装的 HashMap 中，使用一个全局的锁来同步不同线程间的并发访问。同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器。这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成串行化的了。

ConcurrentHashMap 的高并发性主要来自于三个方面：

- 用分离锁实现多个线程间的更深层次的共享访问。
- 用 HashEntery 对象的不变性来降低执行读操作的线程在遍历链表期间对加锁的需求。
- 通过对同一个 Volatile 变量的写 / 读访问，协调不同线程间读 / 写操作的内存可见性。

使用分离锁，减小了请求 同一个锁的频率。

通过 HashEntery 对象的不变性及对同一个 Volatile 变量的读 / 写来协调内存可见性，使得 读操作大多数时候不需要加锁就能成功获取到需要的值。由于散列映射表在实际应用中大多数操作都是成功的 读操作，所以 2 和 3 既可以减少请求同一个锁的频率，也可以有效减少持有锁的时间。通过减小请求同一个锁的频率和尽量减少持有锁的时间 ，使得 ConcurrentHashMap 的并发性相对于 HashTable 和用同步包装器包装的 HashMap有了质的提高。

9

LinkedHashMap 与 LRUcache

LRU 缓存介绍

我们平时总会有一个电话本记录所有朋友的电话，但是，如果有朋友经常联系，那些朋友的电话号码不用翻电话本我们也能记住，但是，如果长时间没有联系了，要再次联系那位朋友的时候，我们又不得不求助电话本，但是，通过电话本查找还是很费时间的。但是，我们大脑能够记住的东西是一定的，我们只能记住自己最熟悉的，而长时间不熟悉的自然就忘记了。

其实，计算机也用到了同样的一个概念，我们用缓存来存放以前读取的数据，而不是直接丢掉，这样，再次读取的时候，可以直接在缓存里面取，而不用再重新查找一遍，这样系统的反应能力会有很大提高。但是，当我们读取的个数特别大的时候，我们不可能把所有已经读取的数据都放在缓存里，毕竟内存大小是一定的，我们一般把最近常读取的放在缓存里（相当于我们把最近联系的朋友的姓名和电话放在大脑里一样）。

LRU 缓存利用了这样的一种思想。LRU 是 Least Recently Used 的缩写，翻译过来就是“最近最少使用”，也就是说，LRU 缓存把最近最少使用的数据移除，让给最新读取的数据。而往往最常读取的，也是读取次数最多的，所以，利用 LRU 缓存，我们能够提高系统的 performance。

实现

要实现 LRU 缓存，我们首先要用到一个类 LinkedHashMap。

用这个类有两大好处：一是它本身已经实现了按照访问顺序的存储，也就是说，最近读取的会放在最前面，最最不常读取的会放在最后（当然，它也可以实现按照插入顺序存储）。第二，LinkedHashMap 本身有一个方法用于判断是否需要移除最不常读取的数，但是，原始方法默认不需要移除（这是，LinkedHashMap 相当于一个linkedlist），所以，我们需要 override 这样一个方法，使得当缓存里存放的数据个数超过规定个数后，就把最不常用的移除掉。关于 LinkedHashMap 中已经有详细的介绍。

代码如下：（可直接复制，也可以通过LRUcache-Java下载）

```
import java.util.LinkedHashMap;
import java.util.Collection;
import java.util.Map;
import java.util.ArrayList;

/**
 * An LRU cache, based on <code>LinkedHashMap</code>.
 *
 * <p>
 * This cache has a fixed maximum number of elements (<code>cacheSize</code>).
 * If the cache is full and another entry is added, the LRU (least recently
 * used) entry is dropped.
 *
 * <p>
 * This class is thread-safe. All methods of this class are synchronized.
 *
 * <p>
 * Author: Christian d'Heureuse, Inventec Informatik AG, Zurich, Switzerland<br>
 * Multi-licensed: EPL / LGPL / GPL / AL / BSD.
 */
public class LRUCache<K, V> {
   private static final float hashTableLoadFactor = 0.75f;
   private LinkedHashMap<K, V> map;
   private int cacheSize;

   /**
    * Creates a new LRU cache. 在该方法中，new LinkedHashMap<K,V>(hashTableCapacity,
    * hashTableLoadFactor, true)中，true代表使用访问顺序
    *
    * @param cacheSize
    *           the maximum number of entries that will be kept in this cache.
    */
   public LRUCache(int cacheSize) {
      this.cacheSize = cacheSize;
      int hashTableCapacity = (int) Math
            .ceil(cacheSize / hashTableLoadFactor) + 1;
      map = new LinkedHashMap<K, V>(hashTableCapacity, hashTableLoadFactor,
            true) {
         // (an anonymous inner class)
         private static final long serialVersionUID = 1;

         @Override
```

```
        protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
            return size() > LRUCache.this.cacheSize;
        }
    };
}

/**
 * Retrieves an entry from the cache.<br>
 * The retrieved entry becomes the MRU (most recently used) entry.
 *
 * @param key
 *          the key whose associated value is to be returned.
 * @return the value associated to this key, or null if no value with this
 *         key exists in the cache.
 */
public synchronized V get(K key) {
    return map.get(key);
}

/**
 * Adds an entry to this cache. The new entry becomes the MRU (most recently
 * used) entry. If an entry with the specified key already exists in the
 * cache, it is replaced by the new entry. If the cache is full, the LRU
 * (least recently used) entry is removed from the cache.
 *
 * @param key
 *          the key with which the specified value is to be associated.
 * @param value
 *          a value to be associated with the specified key.
 */
public synchronized void put(K key, V value) {
    map.put(key, value);
}

/**
 * Clears the cache.
 */
public synchronized void clear() {
    map.clear();
}

/**
 * Returns the number of used entries in the cache.
 *
 * @return the number of entries currently in the cache.
 */
public synchronized int usedEntries() {
    return map.size();
}

/**
 * Returns a <code>Collection</code> that contains a copy of all cache
 * entries.
 *
 * @return a <code>Collection</code> with a copy of the cache content.
 */
public synchronized Collection<Map.Entry<K, V>> getAll() {
    return new ArrayList<Map.Entry<K, V>>(map.entrySet());
}

// Test routine for the LRUCache class.
public static void main(String[] args) {
    LRUCache<String, String> c = new LRUCache<String, String>(3);
```

```
        c.put("1", "one"); // 1
        c.put("2", "two"); // 2 1
        c.put("3", "three"); // 3 2 1
        c.put("4", "four"); // 4 3 2
        if (c.get("2") == null)
            throw new Error(); // 2 4 3
        c.put("5", "five"); // 5 2 4
        c.put("4", "second four"); // 4 5 2
        // Verify cache content.
        if (c.usedEntries() != 3)
            throw new Error();
        if (!c.get("4").equals("second four"))
            throw new Error();
        if (!c.get("5").equals("five"))
            throw new Error();
        if (!c.get("2").equals("two"))
            throw new Error();
        // List cache content.
        for (Map.Entry<String, String> e : c.getAll())
            System.out.println(e.getKey() + " : " + e.getValue());
    }
}
```

10

HashSet 和 HashMap 的比较

HashMap 和 HashSet 都是 collection 框架的一部分，它们让我们能够使用对象的集合。collection 框架有自己的接口和实现，主要分为 Set 接口，List 接口和 Queue 接口。它们有各自的特点，Set 的集合里不允许对象有重复的值，List 允许有重复，它对集合中的对象进行索引，Queue 的工作原理是 FCFS 算法(First Come, First Serve)。

首先让我们来看看什么是 HashMap 和 HashSet，然后再来比较它们之间的分别。

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。

什么是 HashMap

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 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。

HashSet 和 HashMap 的区别

HashMap	HashSet
HashMap实现了Map接口	HashSet实现了Set接口
HashMap储存键值对	HashSet仅仅存储对象
使用put()方法将元素放入map中	使用add()方法将元素放入set中
HashMap中使用键对象来计算hashcode值	HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode可能相同，所以equals()方法用来判断对象的相等性，如果两个对象不同的话，那么返回false
HashMap比较快，因为是使用唯一的键来获取对象	HashSet较HashMap来说比较慢

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/java-collection/