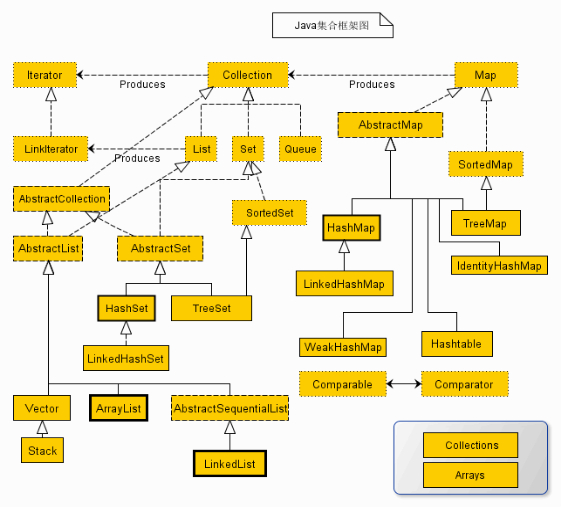
## 1: 集合

以下是Java集合框架图：实线边框的是实现类，折线边框的是抽象类，点线边框的是接口；



Collection是Java.util包中的一个集合类的根接口，Java中没有提供这个接口的直接的实现类。但是却让其被继承产生了两个接口，就是Set和List。Set中不能包含重复的元素。List是一个有序的集合，可以包含重复的元素，提供了按索引访问的方式。

Iterator，所有的集合类，都实现了Iterator接口，这是一个用于遍历集合中元素的接口，主要包含以下三种方法：1.hasNext()是否还有下一个元素。2.next()返回下一个元素。3.remove()删除当前元素。

对于集合，关注的点主要有四点: 1、是否允许空; 2、是否允许重复数据; 3、是否有序，有序的意思是读取数据的顺序和存放数据的顺序是否一致; 4、是否线程安全 ;

Map是Java.util包中的另一个接口，它和Collection接口没有关系，是相互独立的，但是都属于集合类的一部分。Map包含了key-value对。Map不能包含重复的key，但是可以包含相同的value。

### ArrayList

ArrayList就是一个以数组形式实现的集合；允许为空(add null size也加1)、允许重复

数据、有序、非线程安全；

元素: private transient Object[] elementData

作用: ArrayList是基于数组的一个实现，elementData就是底层的数组；

元素: private int size ()

作用: ArrayList里面元素的个数，size是按照add、remove方法的次数进行自增或自减；

优点：

1. ArrayList底层以数组实现，是一种随机访问模式，再加上它实现了RandomAccess接

口，因此查找也就是get的时候非常快；

2、ArrayList在顺序添加一个元素的时候非常方便，只是往数组里面添加了一个元素而已

缺点：

1、删除元素的时候，涉及到一次元素复制，如果要复制的元素很多，那就会比较耗费性能；

2、插入元素的时候，涉及到一次元素复制，如果要复制的元素很多，那就会比较耗费性能；

因此，ArrayList比较适合顺序添加、随机访问的场景。

#### 添加元素与扩容

调用add方法的时候就是给elementData的某个位置添加了一个数据；构造ArrayList的时候，默认的底层数组大小是10；底层数组的大小不够时，底层基于动态数组(底层的数组大小并不是固定的)实现的ArrayList会根据添加的元素大小进行一个判断，不够的话就动态扩容，jdk1.7中扩容newCapacity = oldCapacity + (oldCapacity >> 1) ，即扩容1.5倍；

最后调用到的是Arrays的copyOf方法，将元素组里面的内容复制到新的数组里面去；

(注： << 左移运算符，相当于乘以2； >> 右移运算符，相当于除以2； >>> 无符号右移，忽略符号位，空位都以0补齐)

#### 删除元素

ArrayList支持两种删除方式：1、按照下标删除；2、按照元素删除，这会删除ArrayList

中与指定要删除的元素匹配的第一个元素；其实做的事情就是两件: 1、把指定元素后面位置的所有元素，利用System.arraycopy方法整体向前移动一个位置；2、最后一个位置的元素指定为null，这样让gc可以去回收它；

#### 插入元素

插入的时候，按照指定位置，把从指定位置开始的所有元素利用System,arraycopy方法做一个整体的复制，向后移动一个位置（当然先要用ensureCapacity方法进行判断，加了一个元素之后数组会不会不够大），然后指定位置的元素设置为需要插入的元素，完成了一次插入的操作。

#### 1.1.4 线程非安全

ArrayList是线程非安全的，一个方法是用Collections.synchronizedList方法把ArrayList变成一个线程安全的List，比如：

List<String> synchronizedList = Collections.synchronizedList(list);

synchronizedList.add("aaa");

synchronizedList.add("bbb");

for (int i = 0; i < synchronizedList.size(); i++)

{

    System.out.println(synchronizedList.get(i));

}

另一个方法就是Vector，它是ArrayList的线程安全版本，其实现90%和ArrayList都完全一样，区别在于：1、Vector是线程安全的，ArrayList是线程非安全的；2、Vector可以指定增长因子，如果该增长因子指定了，那么扩容的时候会每次新的数组大小会在原数组的大小基础上加上增长因子；如果不指定增长因子，那么就给原数组大小；

#### 1.1.5 transient

ArrayList的elementData是用transient修饰的？

ArrayList的定义：

public class ArrayList<E> extends AbstractList<E>

        implements List<E>, RandomAccess, Cloneable, java.io.Serializable

看到ArrayList实现了Serializable接口，这意味着ArrayList是可以被序列化的，用transient修饰elementData意味着不希望elementData数组被序列化。因为序列化ArrayList的时候，ArrayList里面的elementData未必是满的，比方说elementData有10的大小，但是只用了其中的3个，那么是否有必要序列化整个elementData呢？显然没有这个必要，因此ArrayList中重写了writeObject方法：

private void writeObject(java.io.ObjectOutputStream s)

        throws java.io.IOException{

// Write out element count, and any hidden stuff

int expectedModCount = modCount;

s.defaultWriteObject();

        // Write out array length

       s.writeInt(elementData.length);

    // Write out all elements in the proper order.

for (int i=0; i<size; i++)

           s.writeObject(elementData[i]);

    if (modCount != expectedModCount) {

           throw new ConcurrentModificationException();

    }

}

每次序列化的时候调用这个方法，先调用defaultWriteObject()方法序列化ArrayList中的非transient元素，elementData不去序列化它，然后遍历elementData，只序列化那些有的元素，这样：

1、加快了序列化的速度

2、减小了序列化之后的文件大小

不失为一种聪明的做法，如果以后开发过程中有遇到这种情况，也是值得学习、借鉴的一种思路。

#### 1.1.6 重要

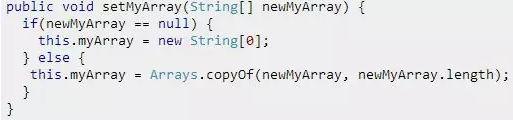
**1、当传递ArrayList到某个方法中，或者某个方法返回ArrayList，什么时候要考虑安全隐患？如何修复安全违规这个问题呢？**

当array被当做参数传递到某个方法中，如果array在没有被复制的情况下直接被分配给了成员变量，那么就可能发生这种情况，即当原始的数组被调用的方法改变的时候，传递到这个方法中的数组也会改变。下面的这段代码展示的就是安全违规以及如何修复这个问题。

ArrayList被直接赋给成员变量——安全隐患：



修复这个安全隐患：



**2、如何复制某个ArrayList到另一个ArrayList中去？写出你的代码？**

下面就是把某个ArrayList复制到另一个ArrayList中去的几种技术：

1：使用clone()方法，比如ArrayList newArray = oldArray.clone();

2：使用ArrayList构造方法，比如：ArrayList myObject = new ArrayList(myTempObject);

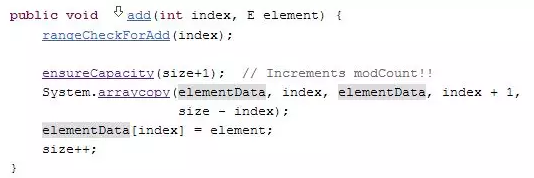
3：使用Collection的copy方法。

注意1和2是浅拷贝(shallow copy)。

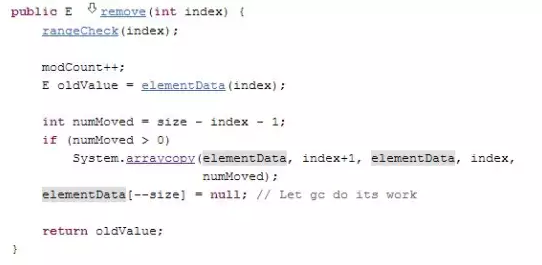
**3、在索引中ArrayList的增加或者删除某个对象的运行过程？效率很低吗？解释为什么？**

在ArrayList中增加或者是删除元素，要调用System.arraycopy这种效率很低的操作，如果遇到了需要频繁插入或者是删除的时候，可以选择其他的Java集合，比如LinkedList。看一下下面的代码：

在ArrayList的某个索引i处添加元素：



删除ArrayList的某个索引i处的元素：



### 1.2 LinkedList

LinkedList是基于链表(链表原先是C/C++的概念，是一种线性的存储结构，意思是将要存储的数据存在一个存储单元里面，这个存储单元里面除了存放有待存储的数据以外，还存储有其下一个存储单元的地址，每次查找数据的时候，通过某个存储单元中的下一个存储单元的地址寻找其后面的那个存储单元)实现的；允许为空(add null size也加1)、允许重复数据、有序、非线程安全；

LinkedList是一种双向链表，1、链表中任意一个存储单元都可以通过向前或者向后寻址的方式获取到其前一个存储单元和其后一个存储单元；2、链表的尾节点的后一个节点是链表的头结点，链表的头结点的前一个节点是链表的尾节点；

LinkedList既然是一种双向链表，必然有一个存储单元，以下是LinkedList的基本存储单元(是LinkedList中的一个内部类)：

private static class Node<E> {

    E item;

    Node <E> next;

    Node <E> prev;

    ...

}

LinkedList中的”E item”，就是它真正存储的数据。”Node<E> next”和” Node <E> prev”表示的就是这个存储单元的前一个存储单元的引用地址和后一个存储单元的引用地址。

#### 1.2.1 添加元素

LinkedList添加一个元素：

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++;

}

类属性有一个Node first和Node last，add方法执行时把last赋给本地变量l，然后new一个Node(l,e,null)，其中prev为l的引用地址，e为要新增的元素，next为空；然后把newNode赋给last；判断l是否为空，为空则newNode赋给first，否则更新l的next属性为newNode；

#### 1.2.2 查找元素

LinkedList底层维护的是一个双向链表，所以根据索引查找元素时可以判断索引值与size的关系，index如果小于二分之一size则从头开始查找，否则从最后倒叙查找；

#### **1.2.4 LinkedList和ArrayList的对比**

1、顺序插入速度ArrayList会比较快，因为ArrayList是基于数组实现的，数组是事先new好的，只要往指定位置塞一个数据就好了；LinkedList则不同，每次顺序插入的时候LinkedList将new一个对象出来，如果对象比较大，那么new的时间势必会长一点，再加上一些引用赋值的操作，所以顺序插入LinkedList必然慢于ArrayList；

2、基于上一点，因为LinkedList里面不仅维护了待插入的元素，还维护了Entry的前置Entry和后继Entry，如果一个LinkedList中的Entry非常多，那么LinkedList将比ArrayList更耗费一些内存；

3、数据遍历的速度，使用各自遍历效率最高的方式，ArrayList的遍历效率会比LinkedList的遍历效率高一些；

4、有些说法认为LinkedList做插入和删除更快，这种说法其实是不准确的：

（1）LinkedList做插入、删除的时候，慢在寻址，快在只需要改变前后Entry的引用地址

（2）ArrayList做插入、删除的时候，慢在数组元素的批量copy，快在寻址

所以，如果待插入、删除的元素是在数据结构的前半段尤其是非常靠前的位置的时候，LinkedList的效率将大大快过ArrayList，因为ArrayList将批量copy大量的元素；越往后，对于LinkedList来说，因为它是双向链表，所以在第2个元素后面插入一个数据和在倒数第2个元素后面插入一个元素在效率上基本没有差别，但是ArrayList由于要批量copy的元素越来越少，操作速度必然追上乃至超过LinkedList。

从这个分析看出，如果你十分确定你插入、删除的元素是在前半段，那么就使用LinkedList；如果你十分确定你删除、删除的元素在比较靠后的位置，那么可以考虑使用ArrayList。如果你不能确定你要做的插入、删除是在哪儿呢？那还是建议你使用LinkedList吧，因为一来LinkedList整体插入、删除的执行效率比较稳定，没有ArrayList这种越往后越快的情况；二来插入元素的时候，弄得不好ArrayList就要进行一次扩容，记住，ArrayList底层数组扩容是一个既消耗时间又消耗空间的操作；

最后一点，一切都是纸上谈兵，在选择了List后，有条件的最好可以做一些性能测试，比如在你的代码上下文记录List操作的时间消耗。

#### **1.2.5 对LinkedList以及ArrayList的迭代**

ArrayList使用最普通的for循环遍历，LinkedList使用foreach循环比较快：

public class ArrayList<E> extends AbstractList<E>

        implements List<E>, RandomAccess, Cloneable, java.io.Serializable

public class LinkedList<E>

    extends AbstractSequentialList<E>

    implements List<E>, Deque<E>, Cloneable, java.io.Serializable

注意到ArrayList是实现了RandomAccess接口而LinkedList则没有实现这个接口，关于RandomAccess这个接口的作用，

虽然上面的例子用的Iterator，但是做foreach循环的时候，编译器默认会使用这个集合的Iterator，具体可参见foreach循环原理。测试代码如下：

public class TestMain

{

    private static int SIZE = 111111;

    private static void loopList(List<Integer> list)

    {

        long startTime = System.currentTimeMillis();

        for (int i = 0; i < list.size(); i++)

        {

            list.get(i);

        }

        System.out.println(list.getClass().getSimpleName() + "使用普通for循环遍历时间为" +

                (System.currentTimeMillis() - startTime) + "ms");

        startTime = System.currentTimeMillis();

        for (Integer i : list)

        {

        }

        System.out.println(list.getClass().getSimpleName() + "使用foreach循环遍历时间为" +

                (System.currentTimeMillis() - startTime) + "ms");

    }

    public static void main(String[] args)

    {

        List<Integer> arrayList = new ArrayList<Integer>(SIZE);

        List<Integer> linkedList = new LinkedList<Integer>();

        for (int i = 0; i < SIZE; i++)

        {

            arrayList.add(i);

            linkedList.add(i);

        }

        loopList(arrayList);

        loopList(linkedList);

        System.out.println();

    }

}

我截取三次运行结果：

ArrayList使用普通for循环遍历时间为6ms

ArrayList使用foreach循环遍历时间为12ms

LinkedList使用普通for循环遍历时间为38482ms

LinkedList使用foreach循环遍历时间为11ms

ArrayList使用普通for循环遍历时间为5ms

ArrayList使用foreach循环遍历时间为12ms

LinkedList使用普通for循环遍历时间为43287ms

LinkedList使用foreach循环遍历时间为9ms

ArrayList使用普通for循环遍历时间为4ms

ArrayList使用foreach循环遍历时间为12ms

LinkedList使用普通for循环遍历时间为22370ms

LinkedList使用foreach循环遍历时间为5ms

ArrayList用for循环遍历要快，linkedList用for each循环比较快；

如果使用普通for循环遍历LinkedList，其遍历速度将慢得令人发指。

### 1.3 CopyOnWriteArrayList

CopyOnWriteArrayList位于java.util.concurrent包下，这个类是为并发而设计的；CopyOnWriteArrayList，即Write时总是要Copy，也就是说对于CopyOnWriteArrayList，任何可变的操作（add、set、remove等等）都是伴随复制这个动作的，后面会解读CopyOnWriteArrayList的底层实现机制。

CopyOnWriteArrayList允许为空，允许重复数据，有序，线程安全；

CopyOnWriteArrayList适用于读操作远多于修改操作的并发场景中。

### 1.4 HashMap

HashMap是一种用哈希值来存储和查找键值对（key-value pair，也称作entry）的一种数据结构。每一个键值对也叫做Entry，这些个键值对（Entry）分散存储在一个数组当中，这个数组就是HashMap的主干。

HashMap的key和value都可以为空；key重复会覆盖，value可以重复；遍历HashMap时是无序的；非线程安全；

#### 1.4.1 底层原理

HashMap的一个存储单元Entry：

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

    final K key;

    V value;

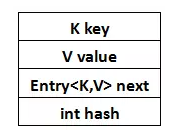
    Entry<K,V> next;

    int hash;

    ...

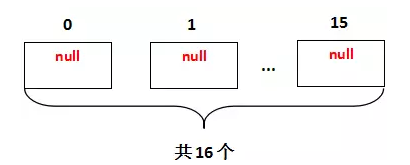
}

LinkedList是一个双向链表，HashMap的Entry组成的是一个单向链表，因为里面只有Entry的后继Entry，而没有Entry的前驱Entry。用图表示应该是以下一个数据结构：



#### 1.4.2 添加元素

DEFAULT\_INITIAL\_CAPACITY为16(最大容量2的30次方，即1073741824)，也就是说，HashMap在new的时候构造出了一个大小为16的Entry数组，Entry内所有数据都取默认值，如下图示：



看到new出了一个大小为16的Entry数组来，以下代码是put方法源码：

public V put(K key, V value) {

    if (key == null)

        return putForNullKey(value);

    int hash = hash(key.hashCode());

    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;

}

static int hash(int h) {

    // 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);

}

static int indexFor(int h, int length) {

     return h & (length-1);

 }

看一下put方法的几个步骤：

1、第2，3行是HashMap允许Key值为空的原因，空的Key会默认放在数组的第0位；

2、第4行拿到Key值的HashCode，由于HashCode是Object的方法，因此每个对象都

有一个HashCode，对这个HashCode做一次hash计算。按JDK源码注释的说法，

这次hash的作用是根据给定的HashCode对它做一次打乱的操作，防止一些糟糕的

Hash算法产生的糟糕的Hash值；

3、第5行根据重新计算的HashCode，对Entry数组的大小取模得到一个Entry数组的位

置。看到这里使用了&，移位加快一点代码运行效率。另外，这个取模操作的正确性依

赖于length必须是2的N次幂，这个熟悉二进制的朋友一定理解，因此注意HashMap

构造函数中，如果指定HashMap初始数组的大小initialCapacity，如果initialCapacity

不是2的N次幂，HashMap会算出大于initialCapacity的最小2的N次幂的值，作为

Entry数组的初始化大小。这里为讲解方便，我们假定字符串111和字符串222算出来

的i都是1

4、第6行~第14行会先判断一下原数据结构中是否存在相同的Key值，存在则覆盖并返

回，不执行后面的代码。注意一下recordAccess这个方法，它也是HashMap的子类

比如LinkedHashMap用的，HashMap中这个方法为空。另外，注意一点，对比Key

是否相同，是先比HashCode是否相同，HashCode相同再判断equals是否为true，

这样大大增加了HashMap的效率；

5、第16行的modeCount++是用于fail-fast机制的，每次修改HashMap数据结构的时候

都会自增一次这个值

然后就到了关键的addEntry方法了：

void addEntry(int hash, K key, V value, int bucketIndex) {

    Entry<K,V> e = table[bucketIndex];

    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

    if (size++ >= threshold)

        resize(2 \* table.length);

}

Entry(int h, K k, V v, Entry<K,V> n) {

    value = v;

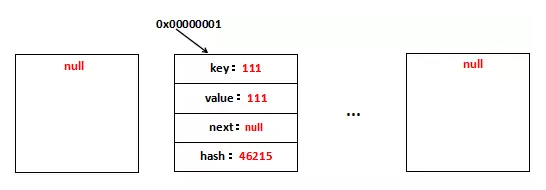
    next = n;

    key = k;

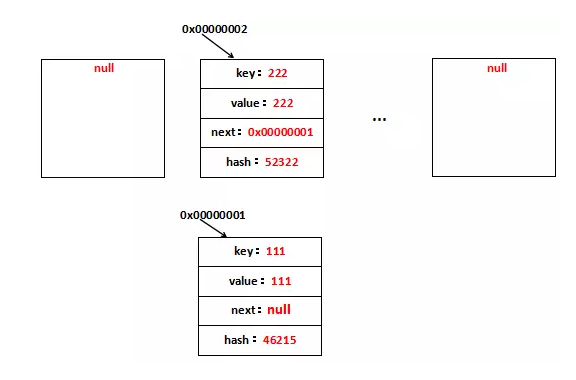
    hash = h;

}

假设new出来的Entry地址为0×00000001，那么，put(“111″, “111″)用图表示应该是这样的：



每一个新增的Entry都位于table[1]上，另外，里面的hash是rehash之后的hash而不是Key最原始的hash。看到table[1]上存放了111—->111这个键值对，它持有原table[1]的引用地址，因此可以寻址到原table[1]，这就是单向链表。 再看一下put(“222″, “222″)做了什么，一张图就可以理解了：



新的Entry再次占据table[1]的位置，并且持有原table[1]，也就是111—->111这个键值对。至此，HashMap进行put数据的过程就呈现清楚了。不过还有一个问题，就是HashMap如何进行扩容，再看一下addEntry方法：

void addEntry(int hash, K key, V value, int bucketIndex) {

      Entry<K,V> e = table[bucketIndex];

      table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

      if (size++ >= threshold)

          resize(2 \* table.length);

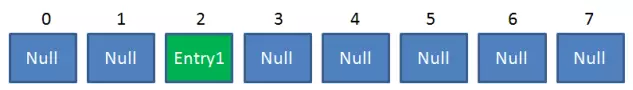
 }

看到第4行~第5行，也就是说在每次放置完Entry之后都会判断是否需要扩容。这里不讲扩容是因为HashMap扩容在不正确的使用场景下将会导致死循环，这是一个值得探讨的话题，也是我工作中实际遇到过的一个问题，因此下一篇文章将会详细说明为什么不正确地使用HashMap会导致死循环。

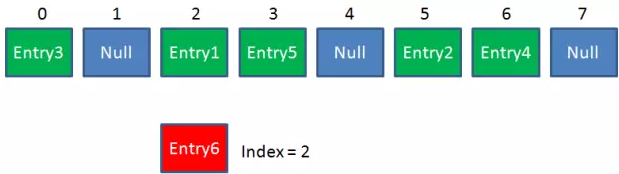
**Put方法的原理：**

HashMap数组每一个元素的初始值都是Null。调用 hashMap.put("apple", 0) ，插入一个Key为“apple"的元素。这时需要利用一个哈希函数来确定Entry的插入位置index：

index =  Hash（“apple”）；假定最后计算出的index是2，那么结果如下：



但是，因为HashMap的长度是有限的，当插入的Entry越来越多时，再完美的Hash函数也难免会出现index冲突的情况。比如下面这样：



HashMap数组的每一个元素不止是一个Entry对象，也是一个链表的头节点。每一个Entry对象通过Next指针指向它的下一个Entry节点。当新来的Entry映射到冲突的数组位置时，只需要插入（“头插法”）到对应的链表即可：



HashMap的默认初始长度是16(JDK1.8 1<<4)，每次自动扩展或手动初始化时，长度必须是2的N次幂；其中16是为了服务于从key映射到index的Hash算法；

index =  HashCode(Key) & (Length – 1)

其中2的N次幂减1得到的二进制数每个位上都是1，可以说，Hash算法最终得到的index结果，完全取决于Key的Hashcode值的最后几位，只要输入的HashCode本身分布均匀，Hash算法的结果就是均匀的；效果等同于取模但是效率更高；如果初始长度不是2的N次幂，HashMap会算出大于initialCapacity的最小2的N次幂的值，作为Entry数组的初始化大小。

#### 1.4.3 删除数据

HashMap删除指定键值对的源代码是：

public V remove(Object key) {

      Entry<K,V> e = removeEntryForKey(key);

      return (e == null ? null : e.value);

 }

final Entry<K,V> removeEntryForKey(Object key) {

    int hash = (key == null) ? 0 : hash(key.hashCode());

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

    Entry<K,V> prev = table[i];

    Entry<K,V> e = prev;

    while (e != null) {

        Entry<K,V> next = e.next;

        Object k;

        if (e.hash == hash &&

            ((k = e.key) == key || (key != null && key.equals(k)))) {

            modCount++;

            size--;

            if (prev == e)

                table[i] = next;

            else

                prev.next = next;

            e.recordRemoval(this);

            return e;

        }

        prev = e;

        e = next;

    }

    return e;

}

分析一下remove元素的时候做了几步：

1、根据key的hash找到待删除的键值对位于table的哪个位置上

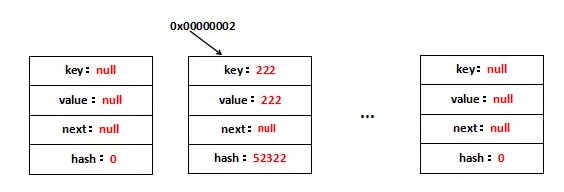
2、记录一个prev表示待删除的Entry的前一个位置Entry，e可以认为是当前位置

3、从table[i]开始遍历链表，假如找到了匹配的Entry，要判断这个Entry是不是table[i]：

（1）是的话，也就是第14行~第15行，table[i]就直接是table[i]的下一个节点，后面的都不需要动

（2）不是的话，也就是第16行~第17行，e的前一个Entry也就是prev，prev的next指向e的后一个节点，也就是next，这样，e所代表的Entry就被踢出了，e的前后Entry就连起来了

remove(“111″)用图表示就是：



整个过程只需要修改一个节点的next的值即可，非常方便。

#### 1.4.4 修改数据

修改元素也是put，看一下源代码：

public V put(K key, V value) {

    if (key == null)

        return putForNullKey(value);

    int hash = hash(key.hashCode());

    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;

}

这个其实前面已经提到过了，第6行~第14行就是修改元素的逻辑，如果某个Key已经在数据结构中存在的话，那么就会覆盖原value，也就是第10行的代码。

**插入数据**

所谓”插入元素”，一定是基于数据结构是有序的前提下的。像ArrayList、LinkedList，再远点说就是数据库，一条一条都是有序的。而HashMap，它的顺序是基于HashCode，HashCode是一个随机性很强的数字，所以HashMap中的Entry完全是随机存放的。HashMap又不像LinkedHashMap这样维护了插入元素的顺序，所以对HashMap这个数据结构谈插入元素是没有意义的。所以，HashMap并没有插入的概念。

#### 1.4.5 重要问题

##### ****1.4.5.1 再谈HashCode的重要性****

HashMap中对Key的HashCode要做一次rehash，防止一些糟糕的Hash算法生成的糟糕的HashCode，那么为什么要防止糟糕的HashCode？

糟糕的HashCode意味着的是Hash冲突，即多个不同的Key可能得到的是同一个HashCode，糟糕的Hash算法意味着的就是Hash冲突的概率增大，这意味着HashMap的性能将下降，表现在两方面：

1. 有10个Key，可能6个Key的HashCode都相同，另外四个Key所在的Entry均匀分布在table的位置上，而某一个位置上却连接了6个Entry。这就失去了HashMap的意义，HashMap这种数据结构性高性能的前提是，Entry均匀地分布在table位置上，但现在确是1 1 1 1 6的分布。所以，我们要求HashCode有很强的随机性，这样就尽可能地可以保证了Entry分布的随机性，提升了HashMap的效率。
2. HashMap在一个某个table位置上遍历链表的时候的代码：

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

由于采用了”&&”运算符，因此先比较HashCode，HashCode都不相同就直接pass了，不会再进行equals比较了。HashCode因为是int值，比较速度非常快，而equals方法往往会对比一系列的内容，速度会慢一些。Hash冲突的概率大，意味着equals比较的次数势必增多，必然降低了HashMap的效率了。

##### ****1.4.5.2 HashMap的table为什么是transient的****

transient Entry[] table;

table用了transient修饰，也就是说table里面的内容全都不会被序列化，因为HashMap是基于HashCode的，HashCode作为Object的方法，是native的：

public native int hashCode();

这意味着的是：HashCode和底层实现相关，不同的虚拟机可能有不同的HashCode算法。如果table不被transient修饰，在虚拟机A上可以用的程序到虚拟机B上可以用的程序就不能用了，失去了跨平台性，因为：

1、Key在虚拟机A上的HashCode=100，连在table[4]上

2、Key在虚拟机B上的HashCode=101，这样，就去table[5]上找Key，明显找不到

整个代码就出问题了。因此，为了避免这一点，Java采取了重写自己序列化table的方法，在writeObject选择将key和value追加到序列化的文件最后面：

private void writeObject(java.io.ObjectOutputStream s)

        throws IOException

{

Iterator<Map.Entry<K,V>> i =

    (size > 0) ? entrySet0().iterator() : null;

// Write out the threshold, loadfactor, and any hidden stuff

s.defaultWriteObject();

// Write out number of buckets

s.writeInt(table.length);

// Write out size (number of Mappings)

s.writeInt(size);

    // Write out keys and values (alternating)

if (i != null) {

 while (i.hasNext()) {

    Map.Entry<K,V> e = i.next();

    s.writeObject(e.getKey());

    s.writeObject(e.getValue());

    }

    }

}

而在readObject的时候重构HashMap数据结构：

private void readObject(java.io.ObjectInputStream s)

         throws IOException, ClassNotFoundException

{

// Read in the threshold, loadfactor, and any hidden stuff

s.defaultReadObject();

// Read in number of buckets and allocate the bucket array;

int numBuckets = s.readInt();

table = new Entry[numBuckets];

    init();  // Give subclass a chance to do its thing.

// Read in size (number of Mappings)

int size = s.readInt();

// Read the keys and values, and put the mappings in the HashMap

for (int i=0; i<size; i++) {

    K key = (K) s.readObject();

    V value = (V) s.readObject();

    putForCreate(key, value);

}

}

一种麻烦的方式，但却保证了跨平台性。

这个例子也告诉了我们：尽管使用的虚拟机大多数情况下都是HotSpot，但是也不能对其它虚拟机不管不顾，有跨平台的思想是一件好事。

##### **1.4.5.3 HashMap和Hashtable的区别**

HashMap和Hashtable是一组相似的键值对集合，它们的区别也是面试常被问的问题之一，这里简单总结一下HashMap和Hashtable的区别：

1、Hashtable是线程安全的，Hashtable所有对外提供的方法都使用了synchronized，也就是同步，而HashMap则是线程非安全的

2、Hashtable不允许空的value，空的value将导致空指针异常，而HashMap则无所谓，没有这方面的限制

3、两个的rehash算法不同，Hashtable的是：

private int hash(Object k) {

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

    return hashSeed ^ k.hashCode();

}

这个hashSeed是使用sun.misc.Hashing类的randomHashSeed方法产生的。HashMap的rehash算法是：

static int hash(int h) {

    // This function ensures that hashCodes that differ only by

    // constant multiples at each bit position have a bounded

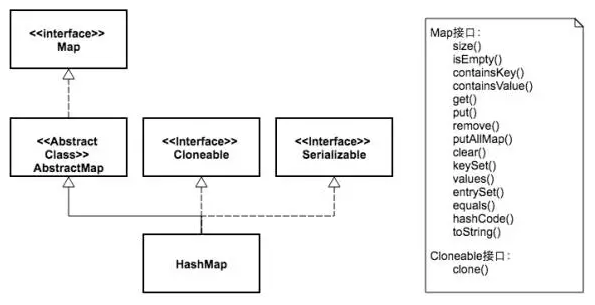
    // number of collisions (approximately 8 at default load factor).

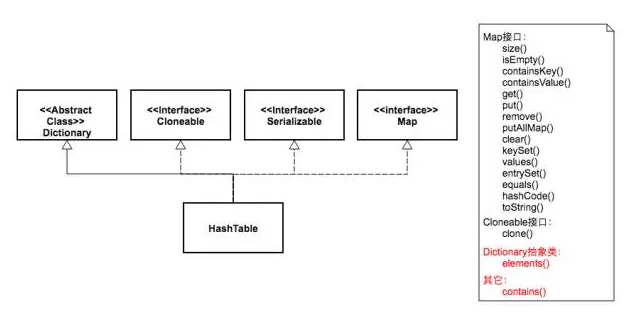
    h ^= (h >>> 20) ^ (h >>> 12);

    return h ^ (h >>> 7) ^ (h >>> 4);

}

1. HashTable产生于JDK 1.1，而HashMap产生于JDK 1.2。HashMap的作者多了大神Doug Lea；
2. HashMap和HashTable都是基于哈希表来实现键值映射的工具类。





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；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。

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。

##### **1.4.5.4 在HashMap中使用可变对象作为Key带来的问题**

如果HashMap Key的哈希值在存储键值对后发生改变，Map可能再也查找不到这个Entry了。如果Key对象是可变的，那么Key的哈希值就可能改变。在HashMap中可变对象作为Key会造成数据丢失。

在HashMap中使用不可变对象。在HashMap中，使用String、Integer等不可变类型用作Key是非常明智的。如果可变对象在HashMap中被用作键，那就要小心在改变对象状态的时候，不要改变它的哈希值了。

##### 1.4.5.5 HashMap是线程不安全

**resize死循环，**HashMap初始容量大小为16,一般来说，当有数据要插入时，都会检查容量有没有超过设定的thredhold，如果超过，需要增大Hash表的尺寸，但是这样一来，整个Hash表里的元素都需要被重算一遍。这叫rehash，这个成本相当的大。

void resize(int newCapacity) {

        Entry[] oldTable = table;

        int oldCapacity = oldTable.length;

        if (oldCapacity == MAXIMUM\_CAPACITY) {

            threshold = Integer.MAX\_VALUE;

            return;

        }

        Entry[] newTable = new Entry[newCapacity];

        transfer(newTable, initHashSeedAsNeeded(newCapacity));

        table = newTable;

        threshold = (int)Math.min(newCapacity \* loadFactor, MAXIMUM\_CAPACITY + 1);

}

void transfer(Entry[] newTable, boolean rehash) {

        int newCapacity = newTable.length;

        for (Entry<K,V> e : table) {

            while(null != e) {

                Entry<K,V> next = e.next;

                if (rehash) {

                    e.hash = null == e.key ? 0 : hash(e.key);

                }

                int i = indexFor(e.hash, newCapacity);

                e.next = newTable[i];

                newTable[i] = e;

                e = next;

            }

        }

}

transfer：

对索引数组中的元素遍历

1. 对链表上的每一个节点遍历：用 next 取得要转移那个元素的下一个，将 e 转移到新

Hash 表的头部，使用头插法插入节点。

2、循环2，直到链表节点全部转移

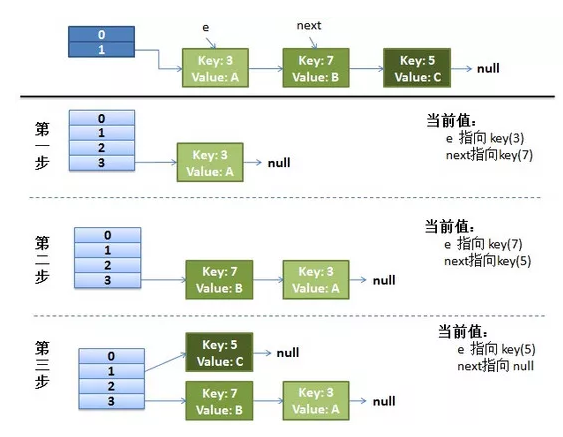
3、循环1，直到所有索引数组全部转移

其中转移的时候是逆序的。假如转移前链表顺序是1->2->3，那么转移后就会变成3->2->1。死锁问题不就是因为1->2的同时2->1造成的吗？所以，HashMap 的死锁问题就出在这个transfer()函数上。

单线程情况下，rehash 不会出现任何问题：

* 假设hash算法就是最简单的 key mod table.length（也就是数组的长度）。
* 最上面的是old hash 表，其中的Hash表的 size = 2, 所以 key = 3, 7, 5，在 mod 2以后碰撞发生在 table[1]
* 接下来的三个步骤是 Hash表 resize 到4，并将所有的 <key,value> 重新rehash到新 Hash 表的过程

如图所示：



**多线程 rehash 详细演示**

为了思路更清晰，我们只将关键代码展示出来

while(null != e) {

    Entry<K,V> next = e.next;

    e.next = newTable[i];

    newTable[i] = e;

    e = next;

}

1. Entry<K,V> next = e.next;——因为是单链表，如果要转移头指针，一定要保存下一个

结点，不然转移后链表就丢了；

2、e.next = newTable[i];——e 要插入到链表的头部，所以要先用 e.next 指向新的 Hash

表第一个元素（为什么不加到新链表最后？因为复杂度是 O（N））

3、newTable[i] = e;——现在新 Hash 表的头指针仍然指向 e 没转移前的第一个元素，

所以需要将新 Hash 表的头指针指向 e；

4、e = next——转移 e 的下一个结点

假设这里有两个线程同时执行了put()操作，并进入了transfer()环节

while(null != e) {

    Entry<K,V> next = e.next; //线程1执行到这里被调度挂起了

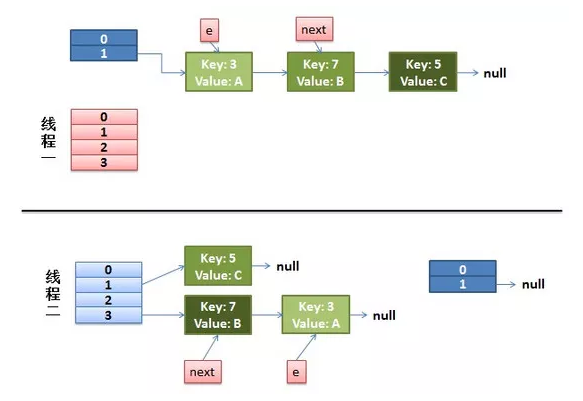
    e.next = newTable[i];

    newTable[i] = e;

    e = next;

}

那么现在的状态为：



从上面的图我们可以看到，因为线程1的 e 指向了 key(3)，而 next 指向了 key(7)，在线程2 rehash 后，就指向了线程2 rehash 后的链表。

然后线程1被唤醒了：

执行e.next = newTable[i]，于是 key(3)的 next 指向了线程1的新 Hash 表，因为新 Hash 表为空，所以e.next = null，

执行newTable[i] = e，所以线程1的新 Hash 表第一个元素指向了线程2新 Hash 表的 key(3)。好了，e 处理完毕。

执行e = next，将 e 指向 next，所以新的 e 是 key(7)

然后该执行 key(3)的 next 节点 key(7)了:

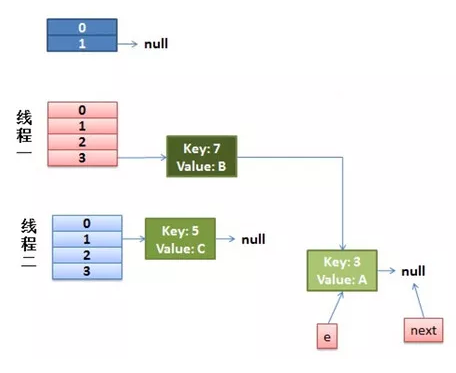
现在的 e 节点是 key(7)，首先执行Entry<K,V> next = e.next,那么 next 就是 key(3)了

执行e.next = newTable[i]，于是key(7) 的 next 就成了 key(3)

执行newTable[i] = e，那么线程1的新 Hash 表第一个元素变成了 key(7)

执行e = next，将 e 指向 next，所以新的 e 是 key(3)

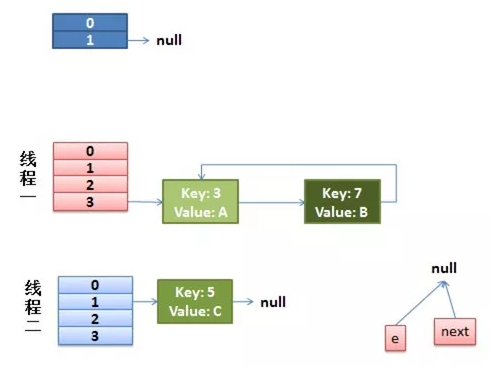
这时候的状态图为：



然后又该执行 key(7)的 next 节点 key(3)了：

1. 现在的 e 节点是 key(3)，首先执行Entry<K,V> next = e.next,那么 next 就是 null
2. 执行e.next = newTable[i]，于是key(3) 的 next 就成了 key(7)
3. 执行newTable[i] = e，那么线程1的新 Hash 表第一个元素变成了 key(3)
4. 执行e = next，将 e 指向 next，所以新的 e 是 key(7)

这时候的状态如图所示：



很明显，环形链表出现了！！当然，现在还没有事情，因为下一个节点是 null，所以transfer()就完成了，等put()的其余过程搞定后，HashMap 的底层实现就是线程1的新 Hash 表了。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。

##### 1.4.5.6 不正确地使用 HashMap 引发死循环及元素丢失

HashMap在多线程环境下会引发死循环的这个问题，下面先用一段代码简单模拟出HashMap的死循环：

public class HashMapThread extends Thread

{

    private static AtomicInteger ai = new AtomicInteger(0);

    private static Map<Integer, Integer> map = new HashMap<Integer, Integer>(1);

    public void run()

    {

        while (ai.get() < 100000)

        {

            map.put(ai.get(), ai.get());

            ai.incrementAndGet();

        }

    }

}

这个线程的作用很简单，给AtomicInteger不断自增并写入HashMap中，其中AtomicInteger和HashMap都是全局共享的，也就是说所有线程操作的都是同一个AtomicInteger和HashMap。开5个线程操作一下run方法中的代码：

public static void main(String[] args)

{

    HashMapThread hmt0 = new HashMapThread();

    HashMapThread hmt1 = new HashMapThread();

    HashMapThread hmt2 = new HashMapThread();

    HashMapThread hmt3 = new HashMapThread();

    HashMapThread hmt4 = new HashMapThread();

    hmt0.start();

    hmt1.start();

    hmt2.start();

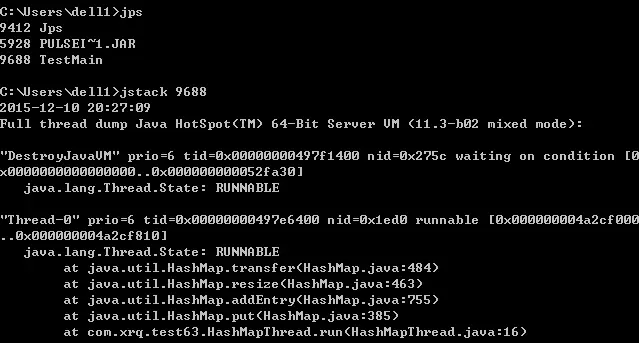
    hmt3.start();

    hmt4.start();

}

多运行几次之后死循环就出来了，我大概运行了7次、8次的样子，其中有几次是数组下标越界异常ArrayIndexOutOfBoundsException。这里面要提一点，多线程环境下代码会出现问题并不意味着多线程环境下一定会出现问题，但是只要出现了问题，或者是死锁、或者是死循环，那么你的项目除了重启就没有什么别的办法了，所以代码的线程安全性在开发、评审的时候必须要重点考虑到。

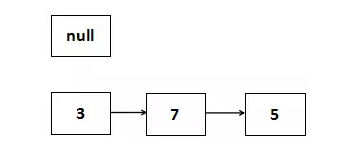
死循环问题的定位一般都是通过jps+jstack查看堆栈信息来定位的：



看到Thread-0处于RUNNABLE，而从堆栈信息上应该可以看出，这次的死循环是由于Thread-0对HashMap进行扩容而引起的。

**正常的扩容过程**

HashMap一次正常的扩容过程。简单一点看吧，假设有三个经过了最终rehash得到的数字，分别是5 7 3，HashMap的table也只有2，那么HashMap把这三个数字put进数据结构了之后应该是这么一个样子的：



然后看一下resize的代码：

void addEntry(int hash, K key, V value, int bucketIndex) {

Entry<K,V> e = table[bucketIndex];

    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

    if (size++ >= threshold)

        resize(2 \* table.length);

}

void resize(int newCapacity) {

    Entry[] oldTable = table;

    int oldCapacity = oldTable.length;

    if (oldCapacity == MAXIMUM\_CAPACITY) {

        threshold = Integer.MAX\_VALUE;

        return;

    }

    Entry[] newTable = new Entry[newCapacity];

    transfer(newTable);

    table = newTable;

    threshold = (int)(newCapacity \* loadFactor);

}

void transfer(Entry[] newTable) {

    Entry[] src = table;

    int newCapacity = newTable.length;

    for (int j = 0; j < src.length; j++) {

        Entry<K,V> e = src[j];

        if (e != null) {

            src[j] = null;

            do {

                Entry<K,V> next = e.next;

                int i = indexFor(e.hash, newCapacity);

                e.next = newTable[i];

                newTable[i] = e;

                e = next;

            } while (e != null);

        }

    }

}

总结一下这三段代码，HashMap一次扩容的过程应该是：

1、取当前table的2倍作为新table的大小；

2、根据算出的新table的大小new出一个新的Entry数组来，名为newTable；

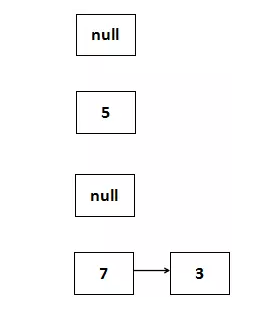
3、轮询原table的每一个位置，将每个位置上连接的Entry，算出在新table上的位置，

并以链表形式连接；

4、原table上的所有Entry全部轮询完毕之后，意味着原table上面的所有Entry已经移

到了新的table上，HashMap中的table指向newTable；

这样就完成了一次扩容，用图表示是这样的：



**扩容导致的死循环**

既然是扩容导致的死循环，那么继续看扩容的代码：

void transfer(Entry[] newTable) {

    Entry[] src = table;

    int newCapacity = newTable.length;

    for (int j = 0; j < src.length; j++) {

        Entry<K,V> e = src[j];

        if (e != null) {

            src[j] = null;

            do {

                Entry<K,V> next = e.next;

                int i = indexFor(e.hash, newCapacity);

                e.next = newTable[i];

                newTable[i] = e;

                e = next;

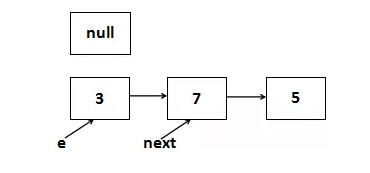
            } while (e != null);

        }

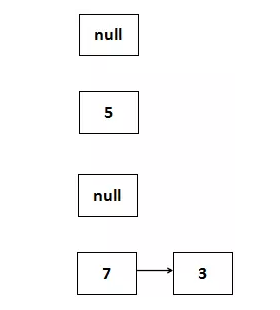
    }

}

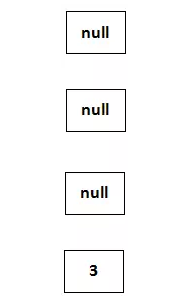
两个线程，线程A和线程B。假设第9行执行完毕，线程A切换，那么对于线程A而言，是这样的：



CPU切换到线程B运行，线程B将整个扩容过程全部执行完毕，于是就形成了：



此时CPU切换到线程A上，执行第8行~第14行的do…while…循环，首先放置3这个Entry：



我们必须要知道，由于线程B已经执行完毕，因此根据Java内存模型（JMM），现在table里面所有的Entry都是最新的，也就是7的next是3，3的next是null。3放置到table[3]的位置上了，下面的步骤是：

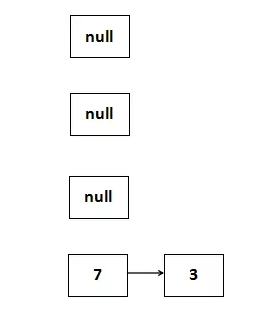
1、e=next，即e=7

2、判断e不等于null，循环继续

3、next=e.next，即next=7的next，也就是3

4、放置7这个Entry

所以，用图表示就是：



放置完7之后，继续运行代码：

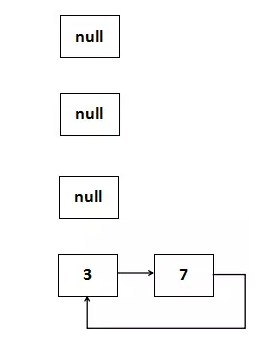
1、e=next，也就是说e=3

2、判断e不等于null，循环继续

3、next=e.next，即3的next，也就是null

4、放置3这个Entry

把3移到table[3]上去，死循环就出来了：



3移到table[3]上去了，3的next指向7，由于原先7的next指向3，这样就成了一个死循环。

此时执行13行的e=next，那么e=null，循环终止。尽管此次循环确实结束了，但是后面的操作，只要涉及轮询HashMap数据结构的，无论是迭代还是扩容，都将在table[3]这个链表处出现死循环。这也就是前面的死循环堆栈出现的原因，transfer的484行，因为这是一次扩容操作，需要遍历HashMap数据结构，transfer方法是扩容的最后一个方法。

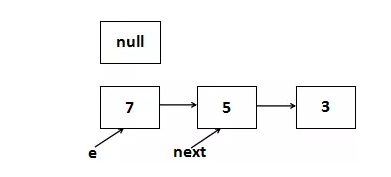
**3 5 7又会有怎样的结果**

可能有人觉得上面的数字5 7 3太巧了，像是专门为了产生HashMap的死循环而故意选择的数字。

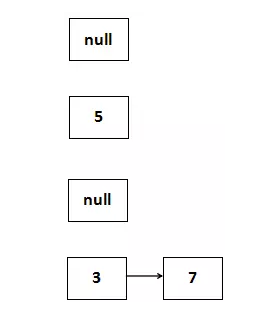
这个问题，我这么回答：我记得在《从Paxos到Zookeeper分布式一致性原理与实践》有一段话大概是这么描述的，有一个被反复实践得出的结论是，任何在多线程下可能发生的错误场景最终一定会发生。

5 7 3这个数字可不巧，扩容前相邻两个Entry被分配到扩容后同样的table位置是很正常的。关键的是，即使这种异常场景发生的可能性再低，只要发生一次，那么你的系统就部分甚至全部不可用了—-除了重启系统没有任何办法。所以，这种可能会发生的异常场景必须提前扼杀。

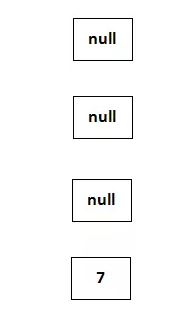
OK，不扯了，前面讲了5 7 3导致了死循环，现在看一下正常的顺序3 5 7，会发生什么问题。简单看一下，就不像上面讲得这么详细了：



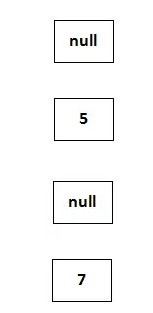
这是扩容前数据结构中的内容，扩容之后正常的应该是：



现在在多线程下遇到问题了，某个线程先放7：



再接着放5：



由于5的next此时为null，因此扩容操作结束，3 5 7造成的结果就是元素丢失。

**如何解决**

把一个线程非安全的集合作为全局共享的，本身就是一种错误的做法，并发下一定会产生错误。

所以，解决这个问题的办法很简单，有两种：

1、使用Collections.synchronizedMap(Mao<K,V> m)方法把HashMap变成一个线程安全的Map

2、使用Hashtable、ConcurrentHashMap这两个线程安全的Map

不过，既然选择了线程安全的办法，那么必然要在性能上付出一定的代价—-毕竟这个世界上没有十全十美的事情，既要运行效率高、又要线程安全。

##### 1.4.5.7 HashMap 遍历方式

JDK8之前，可以使用keySet或者entrySet来遍历HashMap，JDK8中引入了map.foreach来进行遍历。

原因：keySet其实是遍历了2次，一次是转为Iterator对象，另一次是从hashMap中取出key所对应的value。而entrySet只是遍历了一次就把key和value都放到了entry中，效率更高。如果是JDK8，使用Map.foreach方法*。*

**1. keySet和entrySet**

**1.1 基本用法**

keySet:

Map map=new HashMap();

Iterator it=map.keySet().iterator();

Object key;

Object value;

while(it.hasNext()){

key=it.next();

value=map.get(key);

System.out.println(key+":"+value);

}

entrySet:

Map map=new HashMap();

Iterator it=map.entrySet().iterator();

Object key;

Object value;

while(it.hasNext()){

Map.Entry entry = (Map.Entry)it.next();

key=entry.getKey();

value=entry.getValue();

System.out.println(key+"="+value);

}

源码上看：

keySet:

final class KeyIterator extends HashIterator

        implements Iterator<K> {

        public final K next() { return nextNode().key; }

    }

entrySet:

final class EntryIterator extends HashIterator

        implements Iterator<Map.Entry<K,V>> {

        public final Map.Entry<K,V> next() { return nextNode(); }

    }

其实这里已经很明显了，当要得到某个value时，keySet还需要从HashMap中get，entrySet相比keySet少了遍历table的过程，这也是两者性能上的主要差别。

**2. Map.foreach**

在JDK8以后，引入了Map.foreach。Map.foreach本质仍然是entrySet

default void forEach(BiConsumer<? super K, ? super V> action) {

        Objects.requireNonNull(action);

        for (Map.Entry<K, V> entry : entrySet()) {

            K k;

            V v;

            try {

                k = entry.getKey();

                v = entry.getValue();

            } catch(IllegalStateException ise) {

                // this usually means the entry is no longer in the map.

                throw new ConcurrentModificationException(ise);

            }

            action.accept(k, v);

        }

    }

配合lambda表达式一起使用，操作起来更加方便。

**2.1 使用Java8的foreach+lambda表达式遍历Map**

Map<String, Integer> items = new HashMap<>();

items.put("A", 10);

items.put("B", 20);

items.put("C", 30);

items.put("D", 40);

items.put("E", 50);

items.put("F", 60);

items.forEach((k,v)->System.out.println("Item : " + k + " Count : " + v));

items.forEach((k,v)->{

    System.out.println("Item : " + k + " Count : " + v);

    if("E".equals(k)){

        System.out.println("Hello E");

    }

});

### 1.5 LinkedHashMap

HashMap是一种非常常见、非常有用的集合，并且在多线程情况下使用不当会有线程安全问题。大多数情况下，只要不涉及线程安全问题，Map基本都可以使用HashMap，不过HashMap有一个问题，就是迭代HashMap的顺序并不是HashMap放置的顺序，也就是无序。LinkedHashMap虽然增加了时间和空间上的开销，但是通过维护一个运行于所有条目的双向链表，LinkedHashMap保证了元素迭代的顺序。

LinkedHashMap允许键值对为空(即Key和Value都允许为空)，允许重复数据(Key重复会覆盖、Value允许重复)，有序，非线程安全；

#### 1.5.1 基本数据结构

关于LinkedHashMap，需要注意两点：

1、LinkedHashMap可以认为是HashMap+LinkedList，即它既使用HashMap操作数据结构，又使用LinkedList维护插入元素的先后顺序；

2、LinkedHashMap的基本实现思想就是—-多态。可以说，理解多态，再去理解LinkedHashMap原理会事半功倍；反之也是，对于LinkedHashMap原理的学习，也可以促进和加深对于多态的理解。

为什么可以这么说，首先看一下，LinkedHashMap的定义：

public class LinkedHashMap<K,V>

    extends HashMap<K,V>

    implements Map<K,V>

{

    ...

}

看到，LinkedHashMap是HashMap的子类，自然LinkedHashMap也就继承了HashMap中所有非private的方法。再看一下LinkedHashMap中本身的方法：

看到LinkedHashMap中并没有什么操作数据结构的方法，也就是说LinkedHashMap操作数据结构（比如put一个数据），和HashMap操作数据的方法完全一样，无非就是细节上有一些的不同罢了。

LinkedHashMap和HashMap的区别在于它们的基本数据结构上，看一下LinkedHashMap的基本数据结构，也就是Entry：

private static class Entry<K,V> extends HashMap.Entry<K,V> {

    // These fields comprise the doubly linked list used for iteration.

    Entry<K,V> before, after;

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

        super(hash, key, value, next);

    }

    ...

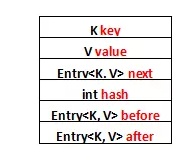
}

列一下Entry里面有的一些属性吧：

* K key
* V value
* Entry<K, V> next
* int hash
* Entry<K, V> before
* Entry<K, V> after

其中前面四个，也就是红色部分是从HashMap.Entry中继承过来的；后面两个，也就是蓝色部分是LinkedHashMap独有的。不要搞错了next和before、After，next是用于维护HashMap指定table位置上连接的Entry的顺序的，before、After是用于维护Entry插入的先后顺序的。

还是用图表示一下，列一下属性而已：



**初始化LinkedHashMap**

假如有这么一段代码：

public static void main(String[] args)

{

    LinkedHashMap<String, String> linkedHashMap =

            new LinkedHashMap<String, String>();

    linkedHashMap.put("111", "111");

    linkedHashMap.put("222", "222");

}

首先是第3行~第4行，new一个LinkedHashMap出来，看一下做了什么：

public LinkedHashMap() {

      super();

      accessOrder = false;

 }

public HashMap() {

     this.loadFactor = DEFAULT\_LOAD\_FACTOR;

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

     table = new Entry[DEFAULT\_INITIAL\_CAPACITY];

     init();

}

void init() {

      header = new Entry<K,V>(-1, null, null, null);

      header.before = header.after = header;

 }

/\*\*

 \* The head of the doubly linked list.

 \*/

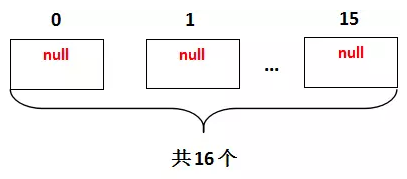
private transient Entry<K,V> header;

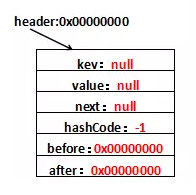
这里出现了第一个多态：init()方法。尽管init()方法定义在HashMap中，但是由于：

1、LinkedHashMap重写了init方法

2、实例化出来的是LinkedHashMap

因此实际调用的init方法是LinkedHashMap重写的init方法。假设header的地址是0×00000000，那么初始化完毕，实际上是这样的：





#### 1.5.2 添加元素

继续看LinkedHashMap添加元素，也就是put(“111″,”111″)做了什么，首先当然是调用HashMap的put方法：

public V put(K key, V value) {

    if (key == null)

        return putForNullKey(value);

    int hash = hash(key.hashCode());

    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;

}

第17行又是一个多态，因为LinkedHashMap重写了addEntry方法，因此addEntry调用的是LinkedHashMap重写了的方法：

void addEntry(int hash, K key, V value, int bucketIndex) {

    createEntry(hash, key, value, bucketIndex);

    // Remove eldest entry if instructed, else grow capacity if appropriate

    Entry<K,V> eldest = header.after;

    if (removeEldestEntry(eldest)) {

        removeEntryForKey(eldest.key);

    } else {

        if (size >= threshold)

            resize(2 \* table.length);

    }

}

因为LinkedHashMap由于其本身维护了插入的先后顺序，因此LinkedHashMap可以用来做缓存，第5行~第7行是用来支持FIFO算法的，这里暂时不用去关心它。看一下createEntry方法：

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;

    e.addBefore(header);

    size++;

}

private void addBefore(Entry<K,V> existingEntry) {

    after  = existingEntry;

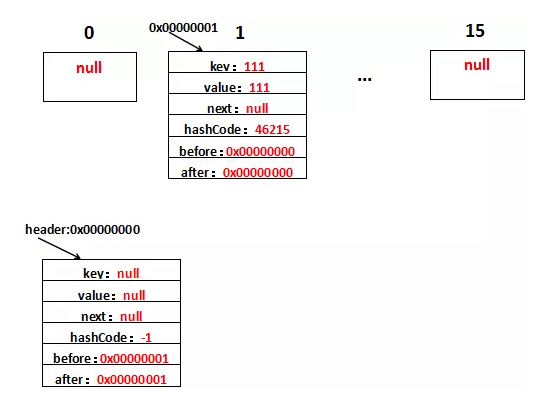
    before = existingEntry.before;

    before.after = this;

    after.before = this;

}

第2行~第4行的代码和HashMap没有什么不同，新添加的元素放在table[i]上，差别在于LinkedHashMap还做了addBefore操作，这四行代码的意思就是让新的Entry和原链表生成一个双向链表。假设字符串111放在位置table[1]上，生成的Entry地址为0×00000001，那么用图表示是这样的：



如果熟悉LinkedList的源码应该不难理解，还是解释一下，注意下existingEntry表示的是header：

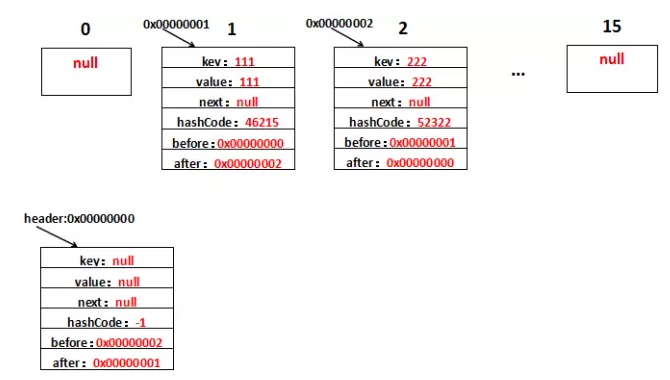
1、after=existingEntry，即新增的Entry的after=header地址，即after=0×00000000

2、before=existingEntry.before，即新增的Entry的before是header的before的地址，header的before此时是0×00000000，因此新增的Entry的before=0×00000000

3、before.after=this，新增的Entry的before此时为0×00000000即header，header的after=this，即header的after=0×00000001

4、after.before=this，新增的Entry的after此时为0×00000000即header，header的before=this，即header的before=0×00000001

这样，header与新增的Entry的一个双向链表就形成了。再看，新增了字符串222之后是什么样的，假设新增的Entry的地址为0×00000002，生成到table[2]上，用图表示是这样的：



就不细解释了，只要before、after清除地知道代表的是哪个Entry的就不会有什么问题。

总得来看，再说明一遍，LinkedHashMap的实现就是HashMap+LinkedList的实现方式，以HashMap维护数据结构，以LinkList的方式维护数据插入顺序。

**利用LinkedHashMap实现LRU算法缓存**

前面讲了LinkedHashMap添加元素，删除、修改元素就不说了，比较简单，和HashMap+LinkedList的删除、修改元素大同小异，下面讲一个新的内容。

LinkedHashMap可以用来作缓存，比方说LRUCache，看一下这个类的代码，很简单，就十几行而已：

public class LRUCache extends LinkedHashMap

{

    public LRUCache(int maxSize)

    {

        super(maxSize, 0.75F, true);

        maxElements = maxSize;

    }

    protected boolean removeEldestEntry(java.util.Map.Entry eldest)

    {

        return size() > maxElements;

    }

    private static final long serialVersionUID = 1L;

    protected int maxElements;

}

顾名思义，LRUCache就是基于LRU算法的Cache（缓存），这个类继承自LinkedHashMap，而类中看到没有什么特别的方法，这说明LRUCache实现缓存LRU功能都是源自LinkedHashMap的。LinkedHashMap可以实现LRU算法的缓存基于两点：

1、LinkedList首先它是一个Map，Map是基于K-V的，和缓存一致

2、LinkedList提供了一个boolean值可以让用户指定是否实现LRU

那么，首先我们了解一下什么是LRU：LRU即Least Recently Used，最近最少使用，也就是说，当缓存满了，会优先淘汰那些最近最不常访问的数据。比方说数据a，1天前访问了；数据b，2天前访问了，缓存满了，优先会淘汰数据b。

我们看一下LinkedList带boolean型参数的构造方法：

public LinkedHashMap(int initialCapacity,

         float loadFactor,

                     boolean accessOrder) {

    super(initialCapacity, loadFactor);

    this.accessOrder = accessOrder;

}

就是这个accessOrder，它表示：

（1）false，所有的Entry按照插入的顺序排列

（2）true，所有的Entry按照访问的顺序排列

第二点的意思就是，如果有1 2 3这3个Entry，那么访问了1，就把1移到尾部去，即2 3 1。每次访问都把访问的那个数据移到双向队列的尾部去，那么每次要淘汰数据的时候，双向队列最头的那个数据不就是最不常访问的那个数据了吗？换句话说，双向链表最头的那个数据就是要淘汰的数据。

“访问”，这个词有两层意思：

1、根据Key拿到Value，也就是get方法

2、修改Key对应的Value，也就是put方法

首先看一下get方法，它在LinkedHashMap中被重写：

public V get(Object key) {

    Entry<K,V> e = (Entry<K,V>)getEntry(key);

    if (e == null)

        return null;

    e.recordAccess(this);

    return e.value;

}

然后是put方法，沿用父类HashMap的：

public V put(K key, V value) {

    if (key == null)

        return putForNullKey(value);

    int hash = hash(key.hashCode());

    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;

}

修改数据也就是第6行~第14行的代码。看到两端代码都有一个共同点：都调用了recordAccess方法，且这个方法是Entry中的方法，也就是说每次的recordAccess操作的都是某一个固定的Entry。

recordAccess，顾名思义，记录访问，也就是说你这次访问了双向链表，我就把你记录下来，怎么记录？把你访问的Entry移到尾部去。这个方法在HashMap中是一个空方法，就是用来给子类记录访问用的，看一下LinkedHashMap中的实现：

void recordAccess(HashMap<K,V> m) {

    LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;

    if (lm.accessOrder) {

        lm.modCount++;

        remove();

        addBefore(lm.header);

    }

}

private void remove() {

    before.after = after;

    after.before = before;

}

private void addBefore(Entry<K,V> existingEntry) {

    after  = existingEntry;

    before = existingEntry.before;

    before.after = this;

    after.before = this;

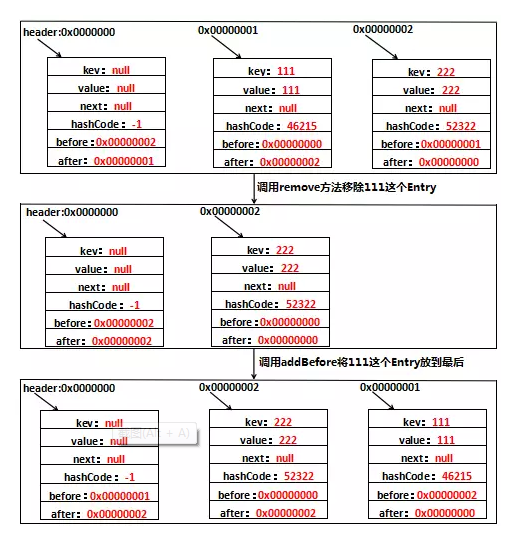
}

看到每次recordAccess的时候做了两件事情：

1、把待移动的Entry的前后Entry相连

2、把待移动的Entry移动到尾部

当然，这一切都是基于accessOrder=true的情况下。最后用一张图表示一下整个recordAccess的过程吧：



**代码演示LinkedHashMap按照访问顺序排序的效果**

最后代码演示一下LinkedList按照访问顺序排序的效果，验证一下上一部分LinkedHashMap的LRU功能：

public static void main(String[] args)

{

    LinkedHashMap<String, String> linkedHashMap =

            new LinkedHashMap<String, String>(16, 0.75f, true);

    linkedHashMap.put("111", "111");

    linkedHashMap.put("222", "222");

    linkedHashMap.put("333", "333");

    linkedHashMap.put("444", "444");

    loopLinkedHashMap(linkedHashMap);

    linkedHashMap.get("111");

    loopLinkedHashMap(linkedHashMap);

    linkedHashMap.put("222", "2222");

    loopLinkedHashMap(linkedHashMap);

}

public static void loopLinkedHashMap(LinkedHashMap<String, String> linkedHashMap)

{

    Set<Map.Entry<String, String>> set = inkedHashMap.entrySet();

    Iterator<Map.Entry<String, String>> iterator = set.iterator();

    while (iterator.hasNext())

    {

        System.out.print(iterator.next() + "\t");

    }

    System.out.println();

}

注意这里的构造方法要用三个参数那个且最后的要传入true，这样才表示按照访问顺序排序。看一下代码运行结果：

111=111    222=222    333=333    444=444

222=222    333=333    444=444    111=111

333=333    444=444    111=111    222=2222

代码运行结果证明了两点：

1、LinkedList是有序的；

1. 每次访问一个元素（get或put），被访问的元素都被提到最后面去了。

### 1.6 红黑树