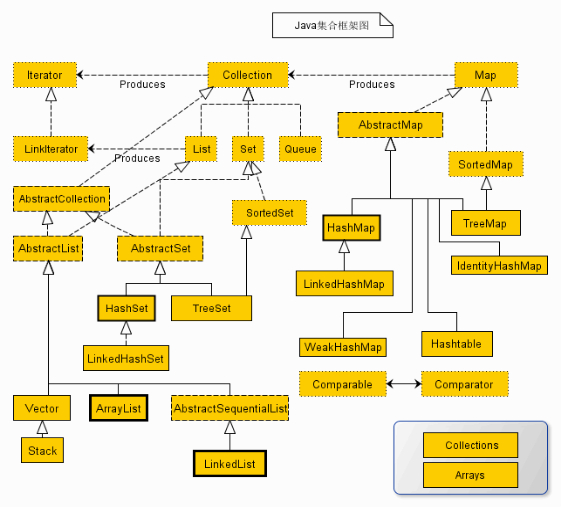
## 1: 集合

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

为什么ArrayList的elementData是用transient修饰的？ArrayList中的数组，是这么定义的：

private transient Object[] elementData;

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。这是为什么？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。

### 1.2 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 Entry<E> {

    E element;

    Entry<E> next;

    Entry<E> previous;

    ...

}

其中LinkedList的Entry中的”E element”，就是它真正存储的数据。”Entry<E> next”和”Entry<E> previous”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：

public static void main(String[] args)

{

     List<String> list = new LinkedList<String>();

     list.add("111");

     list.add("222");

}

逐行分析main函数中的三行代码是如何执行的，首先是第3行，看LinkedList的源码：

public class LinkedList<E>

    extends AbstractSequentialList<E>

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

{

    private transient Entry<E> header = new Entry<E>(null, null, null);

    private transient int size = 0;

    public LinkedList() {

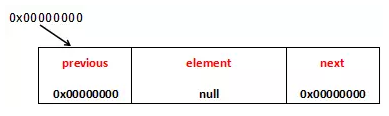
        header.next = header.previous = header;

    }

    ...

}

new了一个Entry出来名为header，Entry里面的previous、element、next都为null，执行构造函数的时候，将previous和next的值都设置为header的引用地址，还是用画图的方式表示。32位JDK的字长为4个字节，而目前64位的JDK一般采用的也是4字长，所以就以4个字长为单位。header引用地址的字长就是4个字节，假设是0×00000000，那执行完”List<String> list = new LinkedList<String>()”之后表示如下：



然后第4行add一个字符串”111″：

public boolean add(E e) {

     addBefore(e, header);

     return true;

}

private Entry<E> addBefore(E e, Entry<E> entry) {

    Entry<E> newEntry = new Entry<E>(e, entry, entry.previous);

    newEntry.previous.next = newEntry;

    newEntry.next.previous = newEntry;

    size++;

    modCount++;

    return newEntry;

}

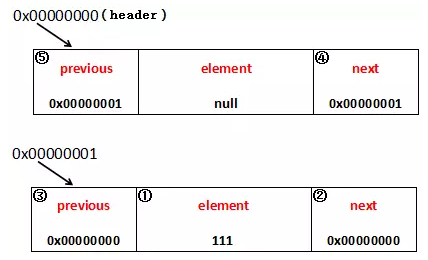
第2行new了一个Entry出来，可能不太好理解，根据Entry的构造函数，我把这句话”翻译”一下，可能就好理解了：

1、newEntry.element = e;

2、newEntry.next = header.next;

3、newEntry.previous = header.previous;

header.next和header.previous上图中已经看到了，都是0×00000000，那么假设new出来的这个Entry的地址是0×00000001，继续画图表示：



一共五步，每一步的操作步骤都用数字表示出来了：

1、新的entry的element赋值为111;

2、新的entry的next是header的next，header的next是0×00000000，所以新的entry的next即0×00000000;

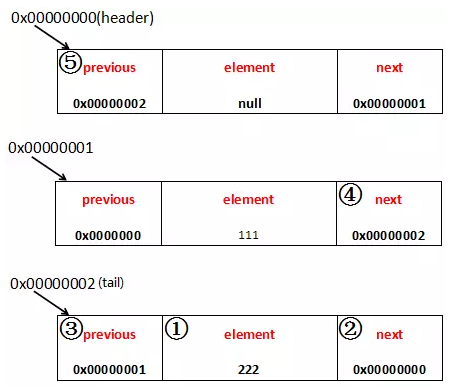
3、新的entry的previous是header的previous，header的previous是0×00000000，所以新的entry的next即0×00000000;

4、”newEntry.previous.next = newEntry”，首先是newEntry的previous，由于newEntry的previous为0×00000000，所以newEntry.previous表示的是header，header的next为newEntry，即header的next为0×00000001;

5、”newEntry.next.previous = newEntry”，和4一样，把header的previous设置为0×00000001;

为什么要这么做？还记得双向链表的两个特点吗，一是任意节点都可以向前和向后寻址，二是整个链表头的previous表示的是链表的尾Entry，链表尾的next表示的是链表的头Entry。现在链表头就是0×00000000这个Entry，链表尾就是0×00000001，可以自己看图观察、思考一下是否符合这两个条件。

最后看一下add了一个字符串”222″做了什么，假设新new出来的Entry的地址是0×00000002，画图表示：



还是执行的那5步，图中每一步都标注出来了，只要想清楚previous、next各自表示的是哪个节点就不会出问题了。

至此，往一个LinkedList里面添加一个字符串”111″和一个字符串”222″就完成了。从这张图中应该理解双向链表比较容易：

1、中间的那个Entry，previous的值为0×00000000，即header；next的值为0×00000002，即tail，这就是任意一个Entry既可以向前查找Entry，也可以向后查找Entry

2、头Entry的previous的值为0×00000002，即tail，这就是双向链表中头Entry的previous指向的是尾Entry

3、尾Entry的next的值为0×00000000，即header，这就是双向链表中尾Entry的next指向的是头Entry

#### **1.2.2 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，看最后一部分，这里就不细讲了，结论是：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。

#### **1.2.3 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，看一下JDK API上的说法：

public interface RamdomAcess

为此，我写一段代码证明一下这一点，注意，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

有了JDK API的解释，这个结果并不让人感到意外，最最想要提出的一点是：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适用于读操作远多于修改操作的并发场景中。

### 2.1 HashMap

HashMap是一种键值对（K-V）形式的存储结构，结合了ArrayList的查询快(插入、删除较慢)与LinkedList的插入、删除方便(查询较慢)的优点；

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；

以下是HashMap的一个存储单元Entry：

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

    final K key;

    V value;

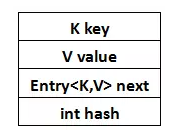
    Entry<K,V> next;

    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。用图表示应该是这么一个数据结构：



接下来，假设我有这么一段代码：

public static void main(String[] args)

{

     Map<String, String> map = new HashMap<String, String>();

     map.put("111", "111");

     map.put("222", "222");

}

看一下做了什么。首先从第3行开始，new了一个HashMap出来：

public HashMap() {

     this.loadFactor = DEFAULT\_LOAD\_FACTOR;

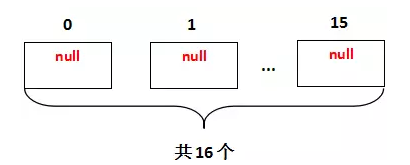
     threshold = (int)(DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR);

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

     init();

}

注意一下第5行的init()是个空方法，它是HashMap的子类比如LinkedHashMap构造的时候使用的。DEFAULT\_INITIAL\_CAPACITY为16，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，如图示为：



看到new出了一个大小为16的Entry数组来。接着第4行，put了一个Key和Value同为111的字符串，看一下put的时候底层做了什么：

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行~第3行就是HashMap允许Key值为空的原因，空的Key会默认放在第0位的数组位置上

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。按照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，至于为什么要防止糟糕的Hash值，HashMap添加元素的最后会讲到

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，如果你指定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;

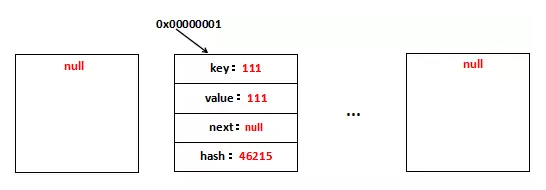
    next = n;

    key = k;

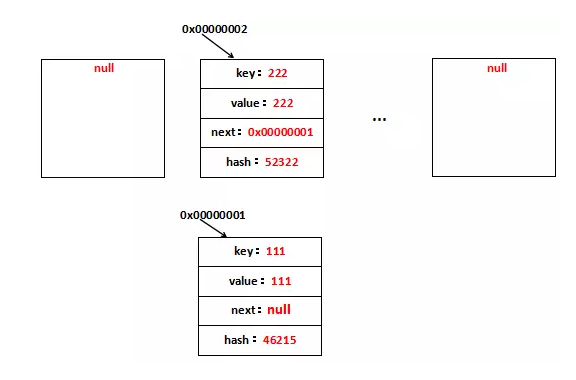
    hash = h;

}

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。

**删除数据**

有一段代码：

public static void main(String[] args)

{

    Map<String, String> map = new HashMap<String, String>();

    map.put("111", "111");

    map.put("222", "222");

    map.remove("111");

}

第6行删除元素，看一下删除元素的时候做了什么，第4行~第5行添加了两个键值对就沿用上面的图，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

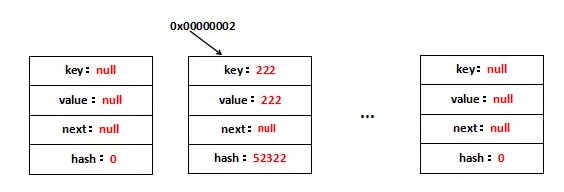
2、记录一个prev表示待删除的Entry的前一个位置Entry，e可以认为是当前位置

3、从table[i]开始遍历链表，假如找到了匹配的Entry，要做一个判断，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。

**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”，在我的理解里，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。

**再谈HashCode的重要性**

前面讲到了，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)))

看到，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。

**HashMap的table为什么是transient的**

一个非常细节的地方：

transient Entry[] table;

看到table用了transient修饰，也就是说table里面的内容全都不会被序列化，不知道大家有没有想过这么写的原因？

在我看来，这么写是非常必要的。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。再进一步说得明白些就是，可能同一个Key在虚拟机A上的HashCode=1，在虚拟机B上的HashCode=2，在虚拟机C上的HashCode=3。

这就有问题了，Java自诞生以来，就以跨平台性作为最大卖点，好了，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。

**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，我这里简单总结一下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算法上面看过了，也就是：

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);

}