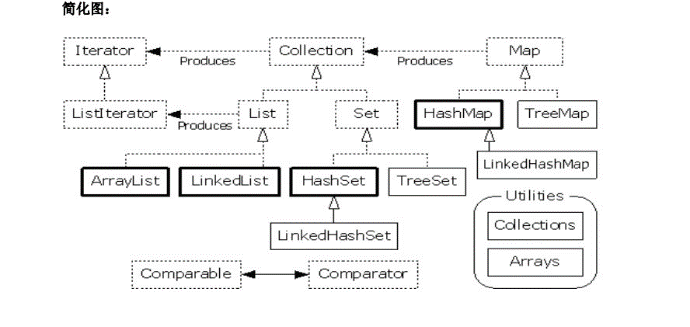
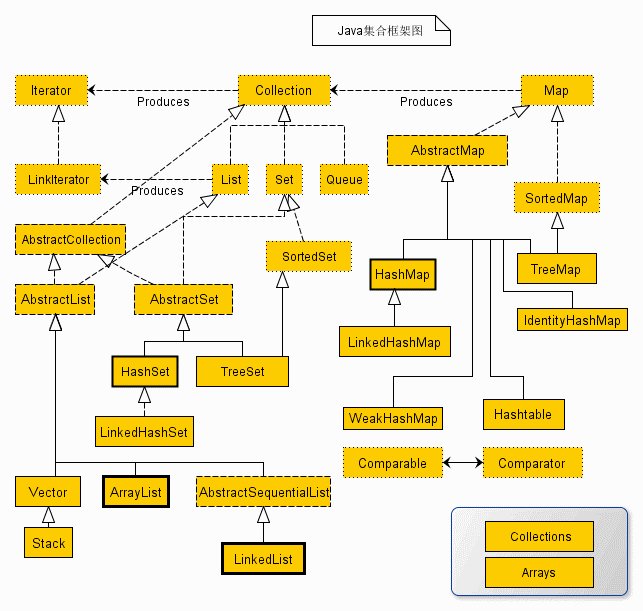
# 纯手写集合框架

## 集合框架介绍



说明：对于以上的框架图有如下几点说明

1.所有集合类都位于java.util包下。Java的集合类主要由两个接口派生而出：Collection和Map，Collection和Map是Java集合框架的根接口，这两个接口又包含了一些子接口或实现类。

2. 集合接口：6个接口（短虚线表示），表示不同集合类型，是集合框架的基础。

3. 抽象类：5个抽象类（长虚线表示），对集合接口的部分实现。可扩展为自定义集合类。

4. 实现类：8个实现类（实线表示），对接口的具体实现。

5. Collection 接口是一组允许重复的对象。

6. Set 接口继承 Collection，集合元素不重复。

7. List 接口继承 Collection，允许重复，维护元素插入顺序。

8. Map接口是键－值对象，与Collection接口没有什么关系。

9.Set、List和Map可以看做集合的三大类：

List集合是有序集合，集合中的元素可以重复，访问集合中的元素可以根据元素的索引来访问。

Set集合是无序集合，集合中的元素不可以重复，访问集合中的元素只能根据元素本身来访问（也是集合里元素不允许重复的原因）。

Map集合中保存Key-value对形式的元素，访问时只能根据每项元素的key来访问其value。

## 纯手写List框架

List集合代表一个有序集合，集合中每个元素都有其对应的顺序索引。List集合允许使用重复元素，可以通过索引来访问指定位置的集合元素。

List接口继承于Collection接口，它可以定义一个允许重复的有序集合。因为List中的元素是有序的，所以我们可以通过使用索引（元素在List中的位置，类似于数组下标）来访问List中的元素，这类似于Java的数组。

List接口为Collection直接接口。List所代表的是有序的Collection，即它用某种特定的插入顺序来维护元素顺序。用户可以对列表中每个元素的插入位置进行精确地控制，同时可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。实现List接口的集合主要有：ArrayList、LinkedList、Vector、Stack。

### ArrayList底层实现原理

1. Arraylist底层基于数组实现

|  |
| --- |
| **private** Object[] elementData; |

1. Arraylist底层默认数组初始化大小为10个object数组

|  |
| --- |
| **public** ExtArraylist() **throws** Exception {  **this**(10);  }  **public** ExtArraylist(**int** initialCapacity) **throws** Exception {  **if** (initialCapacity < 0) {  **throw** **new** IllegalArgumentException("初始容量不能小于0 " + initialCapacity);  }  elementData = **new** Object[initialCapacity];  } |

1. 添加元素后大于当前数组的长度，则进行扩容,将数组的长度增加原来数组的一半。

|  |
| --- |
| // 增大数组空间  **private** **void** grow(**int** minCapacity) {  // overflow-conscious code  **int** oldCapacity = elementData.length;  **int** newCapacity = oldCapacity + (oldCapacity >> 1); // 在原来容量的基础上加上  // oldCapacity/2  **if** (newCapacity - minCapacity < 0)  newCapacity = minCapacity; // 最少保证容量和minCapacity一样  **if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0)  newCapacity = *hugeCapacity*(minCapacity); // 最多不能超过最大容量  // minCapacity is usually close to size, so this is a win:  elementData = Arrays.*copyOf*(elementData, newCapacity);  } |

### Vector底层实现原理

Vector是线程安全的，但是性能比ArrayList要低。

ArrayList，Vector主要区别为以下几点：

（1）：Vector是线程安全的，源码中有很多的synchronized可以看出，而ArrayList不是。导致Vector效率无法和ArrayList相比；

（2）：ArrayList和Vector都采用线性连续存储空间，当存储空间不足的时候，ArrayList默认增加为原来的50%，Vector默认增加为原来的一倍；

（3）：Vector可以设置capacityIncrement，而ArrayList不可以，从字面理解就是capacity容量，Increment增加，容量增长的参数。

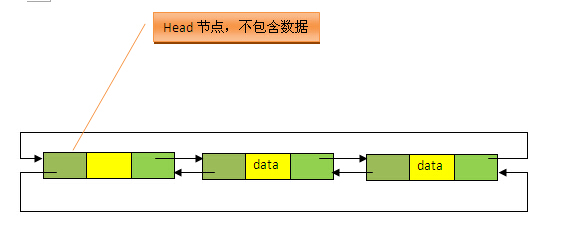
|  |
| --- |
| **private** **void** grow(**int** minCapacity) {  // overflow-conscious code  **int** oldCapacity = elementData.length;  **int** newCapacity = oldCapacity + (oldCapacity >> 1); //扩充的空间增加原来的50%（即是原来的1.5倍）  **if** (newCapacity - minCapacity < 0) //如果容器扩容之后还是不够，那么干脆直接将minCapacity设为容器的大小  newCapacity = minCapacity;  **if** (newCapacity - MAX\_ARRAY\_SIZE > 0) //如果扩充的容器太大了的话，那么就执行hugeCapacity  newCapacity = hugeCapacity(minCapacity);  // minCapacity is usually close to size, so this is a win:  elementData = Arrays.copyOf(elementData, newCapacity);  } |

|  |
| --- |
| /\*\*  \* 自定List接口<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*  \* **@param** <E>  \*/  **public** **interface** ExtList<E> {  **public** **void** add(E object);  **public** **void** add(**int** index, E object);  **public** Object remove(**int** index);  **public** **boolean** remove(E object);  **public** **int** getSize();  **public** Object get(**int** index);  }  /\*\*  \* 纯手写ArrayList<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **public** **class** ExtArrayList<E> **implements** ExtList<E> {  // 保存ArrayList中数据的数组  **private** **transient** Object[] elementData;  // ArrayList实际数量  **private** **int** size;  **public** ExtArrayList() {  // 默认初始容量为10  **this**(10);  }  **public** ExtArrayList(**int** initialCapacity) {  **if** (initialCapacity < 0) {  **throw** **new** IllegalArgumentException("Illegal Capacity: " + initialCapacity);  }  // 初始化数组容量  elementData = **new** Object[initialCapacity];  }  // 添加方法实现  **public** **void** add(Object object) {  ensureExplicitCapacity(size + 1);  elementData[size++] = object;  }  **public** **void** add(**int** index, Object object) {  rangeCheck(index);  ensureExplicitCapacity(size + 1);  System.*arraycopy*(elementData, index, elementData, index + 1, size - index);  elementData[index] = object;  size++;  }  **public** **void** ensureExplicitCapacity(**int** minCapacity) {  // 如果存入的数据,超出了默认数组初始容量 就开始实现扩容  **if** (size == elementData.length) {  // 获取原来数组的长度 2  **int** oldCapacity = elementData.length;  // oldCapacity >> 1 理解成 oldCapacity/2 新数组的长度是原来长度1.5倍  **int** newCapacity = oldCapacity + (oldCapacity >> 1); // 3  **if** (newCapacity < minCapacity) {  // 最小容量比新容量要小的,则采用初始容量minCapacity  newCapacity = minCapacity;  }  // System.out.println("oldCapacity:" + oldCapacity + ",newCapacity:"  // + newCapacity);  elementData = Arrays.*copyOf*(elementData, newCapacity);  }  }  // public void add(Object object) {  // elementData[size++] = object;  // // 如果存入的数据,超出了默认数组初始容量 就开始实现扩容  // if (size == elementData.length) {  // // 新的数组容量  // int newCapacity = 2 \* size;  // // 实现数组扩容  // Object[] newObjct = new Object[newCapacity];  // for (int i = 0; i < elementData.length; i++) {  // newObjct[i] = elementData[i];  // }  // elementData = newObjct;  // }  // }  // 获取数据  **public** Object get(**int** index) {  rangeCheck(index);  **return** elementData[index];  }  **public** Object remove(**int** index) {  Object object = get(index);  **int** numMoved = elementData.length - index - 1;  **if** (numMoved > 0)  System.*arraycopy*(elementData, index + 1, elementData, index, numMoved);  elementData[--size] = **null**;  **return** object;  }  **public** **boolean** remove(E object) {  **for** (**int** i = 0; i < elementData.length; i++) {  Object element = elementData[i];  **if** (element.equals(object)) {  remove(i);  **return** **true**;  }  }  **return** **false**;  }  **private** **void** rangeCheck(**int** index) {  **if** (index >= size) {  **throw** **new** IndexOutOfBoundsException("数组越界啦!");  }  }  **public** **int** getSize() {  **return** size;  }  } |

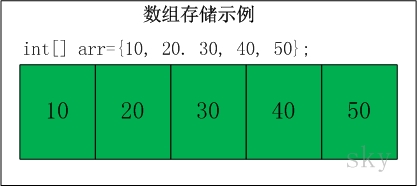
### LinkeList原理

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。

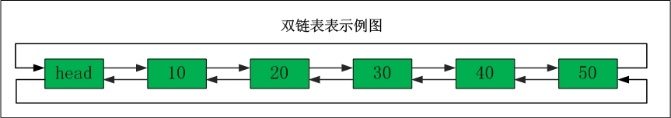
#### LinkedList数据结构原理

LinkedList底层的数据结构是基于双向循环链表的，且头结点中不存放数据,如下：   
   
既然是双向链表，那么必定存在一种数据结构——我们可以称之为节点，节点实例保存业务数据，前一个节点的位置信息和后一个节点位置信息，如下图所示：   


#### 数组和链表结构对比



数组　是将元素在内存中连续存放，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。但是如果要在数组中增加一个元素，需要移动大量元素，在内存中空出一个元素的空间，然后将要增加的元素放在其中。同样的道理，如果想删除一个元素，同样需要移动大量元素去填掉被移动的元素。如果应用需要快速访问数据，很少插入和删除元素，就应该用数组。



链表　中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起，每个结点包括两个部分：一个是存储 数据元素 的　数据域，另一个是存储下一个结点地址的 指针。

　如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要经常插入和删除元素你就需要用链表。

##### 内存存储区别

数组从栈中分配空间, 对于程序员方便快速,但自由度小。

链表从堆中分配空间, 自由度大但申请管理比较麻烦.

##### 逻辑结构区别

数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费。

链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。（数组中插入、删除数据项时，需要移动其它数据项）

##### 总结

1、存取方式上，数组可以顺序存取或者随机存取，而链表只能顺序存取；

2、存储位置上，数组逻辑上相邻的元素在物理存储位置上也相邻，而链表不一定；

3、存储空间上，链表由于带有指针域，存储密度不如数组大；

4、按序号查找时，数组可以随机访问，时间复杂度为O(1)，而链表不支持随机访问，平均需要O(n)；

5、按值查找时，若数组无序，数组和链表时间复杂度均为O(1)，但是当数组有序时，可以采用折半查找将时间复杂度降为O(logn)；

6、插入和删除时，数组平均需要移动n/2个元素，而链表只需修改指针即可；

7、空间分配方面：

　　数组在静态存储分配情形下，存储元素数量受限制，动态存储分配情形下，虽然存储空间可以扩充，但需要移动大量元素，导致操作效率降低，而且如果内存中没有更大块连续存储空间将导致分配失败；

链表存储的节点空间只在需要的时候申请分配，只要内存中有空间就可以分配，操作比较灵活高效；

### 相关代码

|  |
| --- |
| **package** com.itmayiedu;  /\*\*  \* 纯手写 LinkeList<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*  \* **@param** <E>  \*/  **public** **class** LinkeList<E> {  // 第一个元素  **private** Node first;  // 最后一个元素  **private** Node last;  // 实际存放在长度  **private** **int** size;  **class** Node {  // 上一个节点  Node prev;  // 节点内容  Object object;  // 下一个节点  Node next;  }  **public** **void** add(E e) {  // 创建新的节点  Node node = **new** Node();  // 节点内容  node.object = e;  **if** (first == **null**) {  // // 上一个节点  // node.prev = null;  // // 下一个节点  // node.next = null;  // 第一个元素和最后一个元素都是为node  first = node;  } **else** {  // 存放上一个节点内容  node.prev = last;  // 设置上一个节点的next为当前节点  last.next = node;  }  last = node;  size++;  }  **public** **void** add(**int** index, E e) {  // 1.循环遍历到当前index位置Node  // 2.新增当前节点  Node newNode = **new** Node();  newNode.object = e;  // 获取原来的节点  LinkeList<E>.Node olNode = getNode(index);  // 获取原来上一个节点  LinkeList<E>.Node olNodePrev = olNode.prev;  // 4.新增节点的上一个还是当前Node节点的 上一个节点,下一个就是原来的节点  // 原来上一个节点变为当前节点  olNode.prev = newNode;  **if** (olNodePrev == **null**)  first = newNode;  **else**  // 原来上一个节点的下一个节点变为当前节点  olNodePrev.next = newNode;  // 新节点的下一个节点为原来节点  newNode.next = olNode;  size++;  }  **public** E get(**int** index) {  LinkeList<E>.Node node = getNode(index);  **return** (E) node.object;  }  **public** Node getNode(**int** index) {  Node node = **null**;  **if** (first != **null**) {  node = first;  **for** (**int** i = 0; i < index; i++) {  node = node.next;  }  }  **return** node;  }  **public** **void** remove(**int** index) {  checkElementIndex(index);  LinkeList<E>.Node node = getNode(index);  **if** (node != **null**) {  LinkeList<E>.Node prevNode = node.prev;  LinkeList<E>.Node nextNode = node.next;  // 设置上一个节点的next为当前删除节点的next  **if** (prevNode != **null**) {  prevNode.next = nextNode;  }  // 判断是否是最后一个节点  **if** (nextNode != **null**) {  nextNode.prev = prevNode;  }  }  size--;  }  **public** **static** **void** main(String[] args) {  LinkeList<String> linkeList = **new** LinkeList<String>();  linkeList.add("张三");  linkeList.add("李四");  linkeList.add("wawngma");  linkeList.add("wangai");  linkeList.add(2, "ll");  **for** (**int** i = 0; i < linkeList.size; i++) {  System.***out***.println(linkeList.get(i));  }  }  **private** **boolean** isElementIndex(**int** index) {  **return** index >= 0 && index < size;  }  **private** **void** checkElementIndex(**int** index) {  **if** (!isElementIndex(index))  **throw** **new** IndexOutOfBoundsException("越界啦!");  }  } |

## 纯手写Map框架

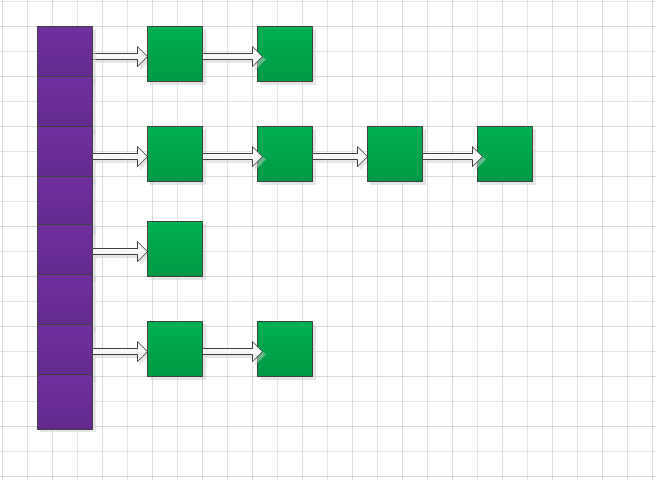
HashMap的介绍

HashMap的实现原理

从底层结构、put和get方法、hash数组索引、扩容机制等几个方面来分析HashMap的实现原理：

### 底层结构

HashMap的底层结构是由数组+链表构成的。



数组（紫色）：hash数组（桶），数组元素是每个链表的头节点

链表（绿色）：解决hash冲突，不同的key映射到了数组的同一索引处，则形成链表。

### put和get方法

put()方法大概过程如下：

如果添加的key值为null，那么将该键值对添加到数组索引为0的链表中，不一定是链表的首节点。

如果添加的key不为null，则根据key计算数组索引的位置：

**数组索引处存在链表，则遍历该链表，如果发现key已经存在，那么将新的value值替换旧的value值**

数组索引处不存在链表，将该key-value添加到此处，成为头节点

get()方法的大概过程：   
1. 如果key为null，那么在数组索引table[0]处的链表中遍历查找key为null的value   
2. 如果key不为null，根据key找到数组索引位置处的链表，遍历查找key的value，找到返回value，若没找到则返回null

### 扩容机制

先看一个例子，创建一个HashMap，初始容量默认为16，负载因子默认为0.75，那么什么时候它会扩容呢？

来看以下公式：

实际容量 = 初始容量 × 负载因子

1

计算可知，16×0.75=12，也就是当实际容量超过12时，这个HashMap就会扩容。

初始容量

当构造一个hashmap时，初始容量设为不小于指定容量的2的次方的一个数（new HashMap(5)， 指定容量为5，那么实际初始容量为8，2^3=8>5），且最大值不能超过2的30次方。

负载因子

负载因子是哈希数组在其容量自动增加之前可以达到多满的一种尺度。（时间与空间的折衷） 当哈希数组中的条目数超出了加载因子与初始容量的乘积时，则要对该哈希数组进行扩容操作（即resize）。

特点：

1\*16=16

0.75\*16=12 0.5\*16=8 **负载因子越小， 容易扩容—容易扩容的时候 产生新的空数组位置**

**三次扩容**

负载因子1 负载因子0.75 负载因子0.5

1.16\*2=32 16 1.16\*2=32 12 16\*2=32 8

2.32\*2=64 32 2.32\*2=64 24 32\*2=64 16

2.64\*2=128 64 3.64\*2=128 48 64\*2=128 32

|  |  |  |
| --- | --- | --- |
| 存放数据10000万个 >13毫秒 | 存放数据10000万个 13毫秒 | 存放数据10000万个 21毫秒 |
| 存放数据10000万个 76左右毫秒 | 存放数据10000万个 77左右毫秒 |  |

**负载因子越小，容易扩容，浪费空间，但查找效率高 链表特别短 减少hash冲突**

**负载因子越大，不易扩容，对空间的利用更加充分，查找效率低（链表拉长）hash冲突比较多，链表比较长**

扩容过程

HashMap在扩容时，新数组的容量将是原来的2倍，由于容量发生变化，原有的每个元素需要重新计算数组索引Index，再存放到新数组中去，这就是所谓的rehash。

### eqauls方法和hashCode方法

1 如果两个对象相同，那么它们的hashCode值一定要相同。也告诉我们重写equals方法，一定要重写hashCode方法，也就是说hashCode值要和类中的成员变量挂上钩，对象相同–>成员变量相同—->hashCode值一定相同。

2 如果两个对象的hashCode相同，它们并不一定相同，这里的对象相同指的是用eqauls方法比较。

### 基于Linkedlist实现Map

#### 代码实现方式

|  |
| --- |
| /\*\*  \* 1.纯手写HasMap集合 完全采用Arraylist实现 缺点效率低 <br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **public** **class** ExtArrayListMap<Key, Value> {  List<Entry<Key, Value>> tables = **new** ArrayList<Entry<Key, Value>>();  **public** **void** put(Key key, Value value) {  // 判断key是否已经重复  Entry existEntry = getEntry(key);  **if** (existEntry != **null**) {  existEntry.value = value;  **return**;  }  Entry entry = **new** Entry<Key, Value>(key, value);  tables.add(entry);  }  **public** Value get(String key) {  **for** (Entry<Key, Value> entry : tables) {  **if** (entry.key.equals(key)) {  **return** entry.value;  }  }  **return** **null**;  }  **public** **void** remove(Key key) {  Entry existEntry = getEntry(key);  **if** (existEntry != **null**) {  tables.remove(existEntry);  }  }  **public** Entry<Key, Value> getEntry(Key key) {  **for** (Entry<Key, Value> entry : tables) {  **if** (entry.key.equals(key)) {  **return** entry;  }  }  **return** **null**;  }  **public** **static** **void** main(String[] args) {  ExtArrayListMap<String, String> extArrayListMap = **new** ExtArrayListMap<String, String>();  extArrayListMap.put("a", "aaaaa");  extArrayListMap.put("b", "bbb");  extArrayListMap.put("a", "aa");  // extArrayListMap.remove("b");  System.***out***.println(extArrayListMap.get("b"));  }  }  **class** Entry<Key, Value> {  **public** Entry(Key key, Value value) {  **this**.key = key;  **this**.value = value;  }  Key key;  Value value;  }  /\*\*  \* 纯手写HasMap集合(采用 LinkedList)集合实现<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **public** **class** LinkedListMap<Key, Value> {  // 实际存放Map元素  LinkedList<Entry>[] tables = **new** LinkedList[998];  // 实际Map大小  **private** **int** size;  **public** **void** put(Object key, Object value) {  // 创建entry；  Entry newEntry = **new** Entry(key, value);  // hashCode  **int** hash = getHash(key);  // 判断是否已经在数组重存在  LinkedList<Entry> entrylinkedList = tables[hash];  **if** (entrylinkedList == **null**) {  // 数组中没有存放元素  LinkedList<Entry> linkedList = **new** LinkedList<>();  linkedList.add(newEntry);  tables[hash] = linkedList;  } **else** {  // hashCode 相同情况下 存放在链表后面  **for** (Entry entry : entrylinkedList) {  **if** (entry.key.equals(key)) {  // hashCode相同 对象也相同  entry.value = value;  } **else** {  // hashCode 相同，但是对象不同。  entrylinkedList.add(newEntry);  }  }  }  size++;  }  **public** **int** getHash(Object key) {  **int** hashCode = key.hashCode();  **int** hash = hashCode % tables.length;  **return** hash;  }  **public** Value get(Object key) {  **return** (Value) getEntry(key).value;  }  **public** Entry getEntry(Object key) {  // hashCode  **int** hash = getHash(key);  LinkedList<Entry> listEntry = tables[hash];  **if** (listEntry != **null**) {  **for** (Entry entry : listEntry) {  **if** (entry.key.equals(key)) {  **return** entry;  }  }  }  **return** **null**;  }  **public** **static** **void** main(String[] args) {  LinkedListMap linkedListMap = **new** LinkedListMap<String, String>();  linkedListMap.put("a", "644");  linkedListMap.put("b", "123456");  linkedListMap.put("b", "123");  System.***out***.println(linkedListMap.get("b"));  }  }  **class** Entry {  **public** Entry(Object key, Object value) {  **this**.key = key;  **this**.value = value;  }  Object key;  Object value;  } |

### 基于JDK1.7版本实现HashMap

#### 创建Map接口

|  |
| --- |
| /\*\*  \* 纯手写Map集合<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **public** **interface** ExtMap<K, V> {  // 向集合中插入数据  **public** V put(K k, V v);  // 根据k 从Map集合中查询元素  **public** V get(K k);  // 获取集合元素个数  **public** **int** size();  **interface** Entry<K, V> {  K getKey();  V getValue();  V setValue(V value);  }  } |

#### 创建HashMap实现

|  |
| --- |
| /\*\*  \* 纯手写HashMap集合 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **public** **class** ExtHashMap<K, V> **implements** ExtMap<K, V> {  // 1.定义table 存放HasMap 数组元素 默认是没有初始化容器 懒加载  Node<K, V>[] table = **null**;  // 2. 实际用到table 存储容量 大小  **int** size;  // 3.HashMap默认负载因子，负载因子越小，hash冲突机率越低， 根据每个链表的个数  **float** DEFAULT\_LOAD\_FACTOR = 0.75f;  // 4.table默认初始大小 16  **static** **int** *DEFAULT\_INITIAL\_CAPACITY* = 16; // 16  **public** V put(K key, V value) {  // 1.判断table 数组大小是否为空（如果为空的情况下 ，做初始化操作）  **if** (table == **null**) {  table = **new** Node[*DEFAULT\_INITIAL\_CAPACITY*];  }  // 2.判断数组是否需要扩容 实际存储容量=负载因子0.75\*初始容量16 =12 相当于如果长度大于12的时候就需要开始数组扩容  **if** (size >= (DEFAULT\_LOAD\_FACTOR \* *DEFAULT\_INITIAL\_CAPACITY*)) {  System.***out***.println("开始扩容啦！！！！");  resize();  }  // 3.计算hash值指定下标位置  **int** index = getIndex(key, *DEFAULT\_INITIAL\_CAPACITY*);  Node<K, V> node = table[index];  **if** (node == **null**) {  // 没有发生hash冲突问题  node = **new** Node<K, V>(key, value, **null**);  size++;  } **else** {  Node<K, V> newNode = node;  **while** (newNode != **null**) {  // // 已经发生hash冲突问题key 直接添加(冲突node)到前面了 不是往后面加  **if** (newNode.getKey().equals(key) || newNode.getKey() == key) {  // hashCodoe 相同，而且equals 相等情况 说明是同一个对象 修改值  // node.value = value;  **return** newNode.setValue(value);  } **else** {  // 继续添加，排在前面 hascode 取模余数相同 index 存放在链表 或者hashCode 相同但是对象不同  // 新的node 的next 原来的node  **if** (newNode.next == **null**) {  node = **new** Node<K, V>(key, value, node);  size++;  }  }  newNode = newNode.next;  }  }  table[index] = node;  **return** **null**;  }  // hashMap数组扩容机制  **private** **void** resize() {  // 1.定义新的数组元素  Node[] newTables = **new** Node[*DEFAULT\_INITIAL\_CAPACITY* << 1];  // 2. 将老的table的key index重新计算下标  **for** (**int** i = 0; i < table.length; i++) {  // 老的Node节点  Node<K, V> oldNode = table[i];  **while** (oldNode != **null**) {  table[i] = **null**;  // 重新计算index 索引下标位置  K oldKey = oldNode.key;  **int** index = getIndex(oldKey, newTables.length);  // 老的next  Node oldnext = oldNode.next;  // 判断newTables数组中,是否存在下标相同，如果下标相同则存放在原来的.next  oldNode.next = newTables[index];  // 将原来的node赋值给新的node  newTables[index] = oldNode;  // 获取下一个节点,判断是否继续循环  oldNode = oldnext;  }  }  // 3.将newTable赋值给table;  table = newTables;  *DEFAULT\_INITIAL\_CAPACITY* = newTables.length;  newTables = **null**;// 将 对象变为不可达对象  }  // hasMap数组扩容  // private void resize() {  //  // // 1.定于新数组（） 按照两倍进行扩容 // 1.使用取模算法定位数组链表 DEFAULT\_INITIAL\_CAPACITY)\*2  // Node[] newTables = new Node[DEFAULT\_INITIAL\_CAPACITY << 1];  // // 2.hashCode不会发生改变.但是index会发生改变,重新计算index 重新取模  // for (int i = 0; i < table.length; i++) {  // Node<K, V> node = table[i];  // while (node != null) {  // table[i] = null;  // int index = getIndex(node.key, newTables.length);  // // 获取下一个节点  // ExtHashMap<K, V>.Node<K, V> next = node.next;  // // 判断是否有下一个节点  // node.next = newTables[index];  // // 将之前node 计算的inde下标存放在 newTables  // newTables[index] = node;  // // 判断是否继续循环  // node = next;  // }  //  // }  // // 将新的tables 赋值改老的table  // table = newTables;  // DEFAULT\_INITIAL\_CAPACITY = newTables.length;  // newTables = null;// 变为不可达对象  // }  // 测试方法.打印所有的链表元素  **void** print() {  **for** (**int** i = 0; i < table.length; i++) {  Node<K, V> node = table[i];  System.***out***.print("下标位置[" + i + "]");  **while** (node != **null**) {  System.***out***.print("[ key:" + node.getKey() + ",value:" + node.getValue() + "]");  node = node.next;  // if (node.next != null) {  // node = node.next;  // } else {  // // 结束循环  // node = null;  // }  }  System.***out***.println();  }  }  **public** **int** getIndex(K k, **int** length) {  // System.out.println("k:" + k + ",hashCode=" + hashCode);  **int** index = k == **null** ? 0 : k.hashCode() % length;  **return** index;  }  **public** V get(K k) {  Node<K, V> node = getNode(table[getIndex(k, *DEFAULT\_INITIAL\_CAPACITY*)], k);  **return** node == **null** ? **null** : node.value;  }  **public** Node<K, V> getNode(Node<K, V> node, K k) {  **while** (node != **null**) {  **if** (node.getKey().equals(k)) {  **return** node;  }  node = node.next;  }  **return** **null**;  }  **public** **int** size() {  **return** size;  }  // 定义节点  **class** Node<K, V> **implements** Entry<K, V> {  // 存放Map 集合 key  **private** K key;  // 存放Map 集合 value  **private** V value;  // 下一个节点Node  **private** Node<K, V> next;  **public** Node(K key, V value, Node<K, V> next) {  **super**();  **this**.key = key;  **this**.value = value;  **this**.next = next;  }  **public** K getKey() {  **return** **this**.key;  }  **public** V getValue() {  **return** **this**.value;  }  **public** V setValue(V value) {  // 设置新值的返回老的 值  V oldValue = **this**.value;  **this**.value = value;  **return** oldValue;  }  }  } |

### HashMap 核心源码部分的分析

1. HashMap只允许一个为null的key。

2. HashMap的扩容：当前table数组的两倍

3. HashMap实际能存储的元素个数： capacity \* loadFactor

4. HashMap在扩容的时候，会重新计算hash值，并对hash的位置进行重新排列， 因此，为了效率，尽量给HashMap指定合适的容量，避免多次扩容

## Java运算符

<< : 左移运算符，num << 1,相当于num乘以2

>> : 右移运算符，num >> 1,相当于num除以2

>>> : 无符号右移，忽略符号位，空位都以0补齐

## 数组拷贝

Arrays.copyOf功能是实现数组的复制，返回复制后的数组。参数是被复制的数组和复制的长度

System.arraycopy System.arraycopy方法：如果是数组比较大，那么使用System.arraycopy会比较有优势，因为其使用的是内存复制，省去了大量的数组寻址访问等时间

复制指定源数组src到目标数组dest。复制从src的srcPos索引开始，复制的个数是length，复制到dest的索引从destPos开始。

src:源数组； srcPos:源数组要复制的起始位置；

dest:目的数组； destPos:目的数组放置的起始位置； length:复制的长度。

注意：src and dest都必须是同类型或者可以进行转换类型的数组．

有趣的是这个函数可以实现自己到自己复制，比如：

int[] fun ={0,1,2,3,4,5,6};

System.arraycopy(fun,0,fun,3,3);

则结果为：{0,1,2,0,1,2,6};

实现过程是这样的，先生成一个长度为length的临时数组,将fun数组中srcPos

到srcPos+length-1之间的数据拷贝到临时数组中，再执行System.arraycopy(临时数组,0,fun,3,3).