# 第11章 集合

## 1.集合概述

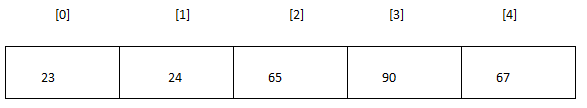
集合是容器，专门用来装对象的。

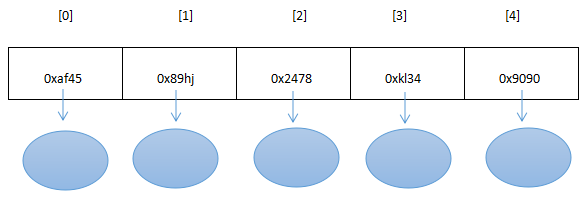
对于集合来说，也就是增删改查集合中的元素，因此最重要的是需要了解这些各种各样的容器是如何**存储元素的，如何获取元素**

### 1.1数组

在集合之前，先了解数组的特征：

* 逻辑结构：线性的
* 物理结构：顺序的存储结构
* 申请内存：一次申请一大段连续的空间，一旦申请到了，内存就固定不变了。
* 存储特点：所有元素存储在这个连续的空间中，数组中的每一个元素都是一个具体的数据（或对象），所有数据都紧密排布，不能有间隔。





* 操作
  + 查询：每一个元素都有一个数值下标，可以通过下标瞬间定位到某个元素
  + 增加：
    - 先使用total变量辅助记录实际存储元素个数
    - 从尾部增加：数组名[total++]=新元素
    - 从其他位置插入：先把index位置开始所有元素后移，然后数组名[index]=新元素
  + 删除：先把index后面的元素前移，然后数组名[total--]=null
  + 改：直接数组名[index]=新元素
* 优缺点
  + 优点：**按索引查询，效率高**
  + 缺点：**添加/删除效率低**，因为都涉及到移动元素；无法直接获取有效元素的个数，需要total来辅助，如图9-1-3和9-1-4

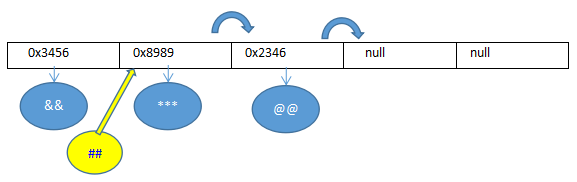


图 9-1- 3

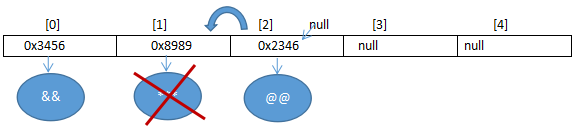


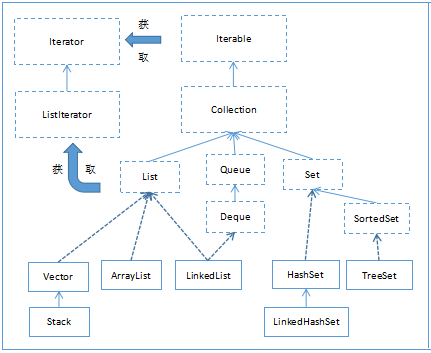
图 9-1- 4

### 1.2集合体系[☆]

集合体系就是一系列的接口和类。这些接口和类都在java.util包中。

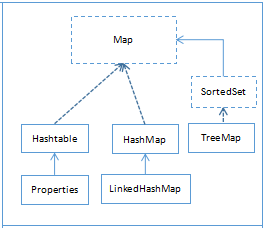
图中虚线为实现，实线为继承。

#### （1）单value型集合体系



* Collection 是单值集合体系中的根接口。他继承了Iterable接口，表示可以用迭代器进行迭代。Iterator接口提供了遍历的方法，而Iterable提供了获取迭代器的方法。
* Collection 表示一组对象，这些对象也称为 collection 的元素。一些 collection 允许有重复的元素，而另一些则不允许。一些 collection 是有序的，而另一些则是无序的。JDK 不提供此接口的任何直接实现：它提供更具体的子接口（如 Set 和 List、Queue）实现。此接口通常用来传递 collection，并在需要最大普遍性的地方操作这些 collection。
  + **List族：**序列，有序的 collection。此接口的用户可以对列表中每个元素的插入位置进行精确地控制。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。
  + **Queue族：**队列，有序；Queue族是在list基础上进行了一定的限定。Queue并不是狭义的先进先出的队列，普通队列以 FIFO（先进先出）的方式排序各个元素，还有优先级队列和 LIFO 队列，前者根据提供的比较器或元素的自然顺序对元素进行排序，后者按 LIFO（后进先出）的方式对元素进行排序。
  + **Set族：**一个不包含重复元素的 collection。对于满足 e1.equals(e2) 的元素对 e1 和 e2，set 不能共存e1和e2，并且最多包含一个 null 元素。Set同时也无法保证元素的顺序。
    - SortedSet进一步提供关于元素的总体排序 的 Set。这些元素使用其自然顺序进行排序，或者根据通常在创建有序 set 时提供的 Comparator进行排序。该 set 的迭代器将按元素升序遍历 set。提供了一些附加的操作来利用这种排序。

#### （2）Key-value型集合体系



* Map：将键映射到值(key,value)的对象。**一个映射不能包含重复的键**；**每个键最多只能映射到一个值**。 Map 接口提供三种collection 视图，允许以键集、值集或键-值映射关系集的形式查看某个映射的内容。映射顺序 定义为迭代器在映射的 collection 视图上返回其元素的顺序。某些映射实现可明确保证其顺序，如 TreeMap 类；另一些映射实现则不保证顺序，如 HashMap 类。
  + SortedMap进一步提供关于键的总体排序 的 Map。该映射是根据其键的自然顺序进行排序的，或者根据通常在创建有序映射时提供的 Comparator 进行排序。

## 2.Collection集合

### 2.1常用方法

**1、添加元素**

（1）add(Object obj)：添加元素对象到当前集合中

（2）addAll(Collection other)：添加other集合中的所有元素对象到当前集合中，即this = this ∪ other

**2、删除元素**

(1) boolean remove(Object obj) ：从当前集合中删除第一个找到的与obj对象equals返回true的元素。

(2)boolean removeAll(Collection coll)：从当前集合中删除所有与coll集合中相同的元素。即this = this - this ∩ coll

(3) cleaer() 清空Collection中的元素

**3、判断元素**

(1) boolean isEmpty()：判断当前集合是否为空集合。

(2) boolean contains(Object obj)：判断当前集合中是否存在一个与obj对象equals返回true的元素。

(3) boolean containsAll(Collection c)：判断c集合中的元素是否在当前集合中都存在。即c集合是否是当前集合的“子集”。

**4、其他**

（1）int size()：获取当前集合中实际存储的元素个数

（2）boolean retainAll(Collection coll)：当前集合仅保留与coll集合中的元素相同的元素，即当前集合中仅保留两个集合的交集，即this = this ∩ coll；

（3） Object[] toArray()：返回包含当前集合中所有元素的数组

#### 1、添加元素

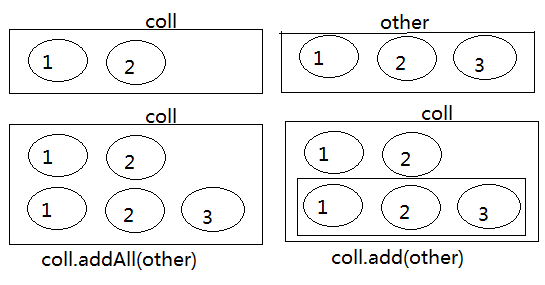
示例代码：

|  |
| --- |
| **package** com.atguigu.part01.collection;  **import** java.util.ArrayList;  **import** java.util.Collection;  **import** org.junit.Test;  @SuppressWarnings("all")  **public** **class** TestCollection {  @Test  **public** **void** test1(){  Collection coll = **new** ArrayList();//ArrayList是Collection的子接口List的实现类  coll.add("张三");  coll.add("李四");  coll.add("王五");  coll.add("张三");    System.***out***.println("coll集合元素的个数：" + coll.size());  }  } |
| @Test  **public** **void** test2(){  Collection coll = **new** ArrayList();  coll.add(1);  coll.add(2);    System.***out***.println("coll集合元素的个数：" + coll.size());    Collection other = **new** ArrayList();  other.add(1);  other.add(2);  other.add(3);    coll.addAll(other); //5  // coll.add(other); //3  System.***out***.println("coll集合元素的个数：" + coll.size());  } |

注意：

coll.addAll(other)：元素变成5个

coll.add(other): 元素变成3个，other以数组形式加入



coll.add(other)的情况对coll进行打印：

System.*out*.println(Arrays.*toString*(coll.toArray())); *//[1, 2, [1, 2, 3]]*

#### 2、删除元素

|  |
| --- |
| @Test  **public** **void** test3(){  Collection coll = **new** ArrayList();  coll.add("张三");  coll.add("李四");  coll.add("王五");  coll.add("张三");    System.***out***.println("coll集合元素的个数：" + coll.size());  coll.remove("张三");  System.***out***.println("coll集合元素的个数：" + coll.size());  } |
| @Test  **public** **void** test4(){  Collection coll = **new** ArrayList();  coll.add(1);  coll.add(2);  coll.add(3);  coll.add(4);  coll.add(5);  System.***out***.println("coll集合元素的个数：" + coll.size());//5    Collection other = **new** ArrayList();  other.add(1);  other.add(2);  other.add(8);    coll.removeAll(other);  System.***out***.println("coll集合元素的个数：" + coll.size());//3  } |

#### 3、保留交集

|  |
| --- |
| @Test  **public** **void** test5(){  Collection coll = **new** ArrayList();  coll.add(1);  coll.add(2);  coll.add(3);  coll.add(4);  coll.add(5);  System.***out***.println("coll集合元素的个数：" + coll.size());//5    Collection other = **new** ArrayList();  other.add(1);  other.add(2);  other.add(8);    coll.retainAll(other);  System.***out***.println("coll集合元素的个数：" + coll.size());//2  } |

### 2.2 Collection集合的遍历

(1)迭代器遍历

(2)forEach遍历

(3)Collection的toArray()方法，将集合变为数组，然后遍历。

#### 迭代器遍历

* Collection接口继承了java.lang.Iterable接口,该接口有一个iterator()方法,那么所有实现了Collection接口的集合类都有一个iterator()方法;此方法返回一个实现了Iterator接口的对象，该对象可以用于迭代集合中的元素。

由于Collection接口继承了iterator接口，因此其所有子类都可以用迭代器遍历，只不过不同的实现类的迭代器实现方式不同。

(1)Iterable接口

|  |
| --- |
|  |

此接口仅是提供iterator()方法，用于获取迭代器

(2)Iterator接口

* Iterator接口提供了一些方法用于遍历集合, Iterator实现类本身并不提供承装对象的能力。Iterator对象是依赖于集合对象的，如果要创建 Iterator 对象，则必须有一个被迭代的集合类。

|  |
| --- |
| 其实每个Collection实现类的Iterator对象都依赖于集合类是因为每个Collection的实现类中，都配备有自己的Iterator接口的内部实现类，如图所示：    所以Collection实现类的iterator()方法返回的是其内部类的对象罢了！  之所以这么设计，是因为每个Collection实现类的内部实现方式不同，对迭代器的实现也是不同的，因此需要设计内部类，此内部类能访问外部类的所有属性，从而可以为外部类提供定制化服务！ |

**ArrayList类：**

|  |
| --- |
| (1)实现获取迭代器的方法：     1. 内部迭代器实现类：     迭代器内部有三个属性：   1. 游标：指向下一个待返回的元素 2. lastRet：指向最后返回的元素，如果是-1就没有。 3. expectedModCount：期望的操作次数   三个方法：  （1）boolean hasNext()：如果仍有元素可以迭代，则返回 true  （2）Object next()：返回迭代的下一个元素。  （3）void remove()：从迭代器指向的 collection 中移除迭代器返回的最后一个元素（可选操作）。  当一个迭代器遍历完了之后，就不能再用次迭代器遍历了，如果还想要遍历，需要重新调用iterator()方法获取一个全新的迭代器对象，默认游标都在集合的第一个元素位置。 |

注意：

在使用ArrayList过程中，当使用iterator迭代获取元素并需要移除元素时，需使用iterator的remove()方法移除元素。如果使用list.remove()，将在iterator下一次调用next()时抛出ConcurrentModificationException。如果中途调用list.add()，同样会修改modCount，导致抛出异常。

#### 2、foreach循环遍历

增强for循环，可以替代iterator迭代器遍历，简化版的；

实现了java.lang.Iterable接口的集合或者才能支持foreach遍历。所以本质上foreach遍历还是迭代器遍历。

|  |
| --- |
| for(元素的类型 迭代变量 : 数组/集合名称){  //每一次循环迭代变量依次代表集合中的一个元素  } |

代码示例：

|  |
| --- |
| @Test  **public** **void** test6(){  Collection coll = **new** ArrayList();  coll.add(1);  coll.add(2);  coll.add(3);  coll.add(4);  **for**(Object obj : coll){  System.***out***.println(obj);  }  //foreach循环4次，obj每一次代表一个元素  } |

**foreach遍历只适用于查找集合中的元素，不能在遍历集合的过程中对集合进行增删。**

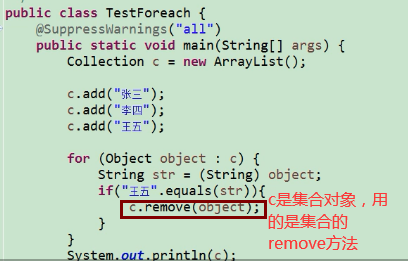
|  |
| --- |
| for(Object obj : coll){  coll.remove(obj);//错误  coll.add(5);//错误  } |

* 遍历数组：

如果涉及下标的操作，用for循环

如果仅仅查看数组的元素，用foreach更简洁

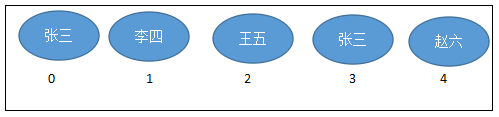
* 遍历Collection集合
  + 如果是含有下标的集合[list],有索引信息,用for也可,没有索引信息不能用for.
  + 注意Collection接口拥有remove方法，而迭代器Iterator也拥有remove方法，如果使用foreach来遍历集合，涉及到删除操作的时候，调用的是Collection的remove方法，这导致一种情况：遍历用的是Iterator，而删除用的是集合的方法，因此可能会导致出错。所以在涉及删除操作的时候，用Iterator来遍历不要用foreach.



提示：**集合中只能添加对象，不能添加基本数据类型的数据，如果添加基本数据类型的数据，它会自动装箱为对应包装类型的对象。**

## 3. List集合

List集合类中元素有序、且可重复。



|  |
| --- |
| 注意：  在JavaSE中List名称的类型有两个，一个是java.util.List集合接口，一个是java.awt.List图形界面的组件，别导错包了。  **List集合关心元素是否有序，而不关心是否重复，请大家记住这个原则。** |

### 3.1 List接口的方法

List除了从Collection集合继承的方法外，List 集合里添加了一些根据索引来操作集合元素的方法。

1、添加元素

* + void add(int index, Object obj)
  + void add(Object obj)
  + boolean addAll(int index, Collection eles)

2、获取元素

* + Object get(int index)

3、获取指定元素索引

* + int indexOf(Object obj)
  + int lastIndexOf(Object obj)

4、删除和替换元素

* + Object remove(int index)
  + Object set(int index, Object obj)

5、获取子list

* + List subList(int start,int stop)

左闭右开

### 3.2 List集合的遍历

1.Iterator遍历

2.增强for循环

3.普通fori循环遍历

fori循环，结合List的get(int index)方法获取索引元素.

代码示例：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List c = **new** ArrayList();  c.add(**new** Student(1,"张三"));  c.add(**new** Student(2,"李四"));  c.add(**new** Student(3,"王五"));  c.add(**new** Student(4,"赵六"));  c.add(**new** Student(5,"钱七"));    **for** (**int** i = 0; i < c.size(); i++) {  System.***out***.println(c.get(i));  }  } |

#### 5.ListIterator迭代器

List 集合额外提供了一个listIterator()方法,此方法是继承listIterator()接口的,该方法返回一个 ListIterator 对象。

ListIterator 接口继承了 Iterator 接口。

ListIterator接口额外提供了专门增删改List 的方法：

* void add()：通过迭代器添加元素到对应集合
* void set(Object obj)：通过迭代器替换正迭代的元素
* void remove()：通过迭代器删除刚迭代的元素
* boolean hasPrevious()：如果以逆向遍历列表，往前是否还有元素。
* Object previous()：返回列表中的前一个元素。
* int previousIndex()：返回列表中的前一个元素的索引
* boolean hasNext()
* Object next()
* int nextIndex()

代码示例：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List c = **new** ArrayList();  c.add(**new** Student(1,"张三"));  c.add(**new** Student(2,"李四"));  c.add(**new** Student(3,"王五"));  c.add(**new** Student(4,"赵六"));  c.add(**new** Student(5,"钱七"));    //从指定位置往前遍历  ListIterator listIterator = c.listIterator(c.size());  **while**(listIterator.hasPrevious()){  Object previous = listIterator.previous();  System.***out***.println(previous);  }  } |

### 3.3 List接口的实现类

#### 1.ArrayList源码

* Arraylist可以放多个null值。
* 方法都是线程不安全的。

##### 初始化机制

|  |
| --- |
| //1. Arraylist底层实际存储元素的数组  **transient** Object[] elementData;    //2.无参构造器生成的ArrayList的底层为空数组  **private** **static** **final** Object[] ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*** = {};  **public** ArrayList() {  //初始化为一个默认的空数组  **this**.elementData = ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***;  }  public ArrayList(int initialCapacity) {  if (initialCapacity > 0) {  this.elementData = new Object[initialCapacity];  } else if (initialCapacity == 0) {  this.elementData = *EMPTY\_ELEMENTDATA*;  } else {  throw new IllegalArgumentException("Illegal Capacity: "+  initialCapacity);  } }  public ArrayList(Collection<? extends E> c) {  elementData = c.toArray();  if ((size = elementData.length) != 0) {  // c.toArray might (incorrectly) not return Object[] (see 6260652)  if (elementData.getClass() != Object[].class)  elementData = Arrays.*copyOf*(elementData, size, Object[].class);  } else {  // replace with empty array.  this.elementData = *EMPTY\_ELEMENTDATA*;  } } |

1. ArrayList中维护了一个Object类型的数组，变量叫做elementData
2. ArrayList默认无参构造初始化容量为0；
3. ArrayList可以指定数组初始化大小；
4. ArrayList可以传进一个Collection集合，生成ArrayList

**额外说明：**

JDK1.6中,ArrayList初始化容量为10

JDK1.7中,初始化容量为EMPTY\_ELEMENTDATA 为0

JDK1.8中,初始化容量为DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA 也是0

因为开发中，很多时候创建了ArrayList的对象，但是没有装元素，这个时候初始化为10就会浪费空间。

##### 添加元素

add(E e)

|  |
| --- |
| **private** **static** **final** **int** ***DEFAULT\_CAPACITY*** = 10;//默认容量  **public** **boolean** add(E e) {  //1.确保当前数组的容量足够  ensureCapacityInternal(size + 1); // Increments modCount!!  //将新元素添加到[size++]的位置  elementData[size++] = e;  **return** **true**;  }  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++; } |

Arraylist在每一次添加元素前都要调用ensureCapacityInternal(size+1)方法来确保容量足够;

##### 扩容

|  |
| --- |
| **private** **void** ensureCapacityInternal(**int** minCapacity) {  //如果是第一次添加，扩容为默认容量大小：10  **if** (elementData == ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***) {  minCapacity = Math.*max*(***DEFAULT\_CAPACITY***, minCapacity);  }    //每一次添加都要判断是否需要扩容  ensureExplicitCapacity(minCapacity);  }  **private** **void** ensureExplicitCapacity(**int** minCapacity) {  modCount++;  // 如果需要扩容  **if** (minCapacity - elementData.length > 0)  grow(minCapacity);  }  **private** **void** grow(**int** minCapacity) {  // 先获取当前数组的容量  **int** oldCapacity = elementData.length;  //新容量为当前容量 + 当前容量的一半  **int** newCapacity = oldCapacity + (oldCapacity >> 1);  **if** (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  **if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0)  newCapacity = *hugeCapacity*(minCapacity);  // 拷贝原数组中的元素至新数组，并返回新数组的引用  elementData = Arrays.*copyOf*(elementData, newCapacity);  } |

1. 如果是第一次添加，就初始化容量为10

(2)无论是否是第一次添加还是不是第一次添加，只要是添加，都要走这个ensureExplicitCapacity进行判断是否需要扩容

(3)扩容判断中维护了一个modCount，记录当前Arraylist被修改的次数（增删改）

(4)如果size + 1大于现有的数组长度elementData.length,就执行grow(mincapacity)

(5)grow(mincapacity)中，先声明两个容量：旧容量和新的容量，其中新容量为旧容量的1.5倍，此时比较新容量和待变容量

对于第一次添加来说，新旧容量都是0，因此经过判断后，新容量还是10

对于非第一次添加，新容量变为1.5倍

(6)扩容后，使用Arrays.coppy方法 拷贝原数组中的元素至新数组，并返回新数组的引用;扩容后的新数组保留原有数据，后面的为null

##### 4、结论

1.ArrayList物理结构是数组，决定了它的存储特点是：需要开辟连续的存储空间来存储元素

2.默认构造参数是空数组

3.add方法会先进行容量判断，只有size+1 > 数组容量的时候，需要扩容，增加容量为原来的1.5倍。

类似的，Vector的物理结构也是数组，当存储容量不够时，需要扩容为原来的2倍。那么是1.5倍好呢？还是2倍好呢？1.5倍使得数组空间使用率提高了，但是这也增加了扩容的频率。所以，建议大家在选择动态数组时，如果对要存储的元素个数有一个预估时，那么可以在创建ArrayList时，就使用ArrayList(int initialCapacity) 构造器，避免反复扩容。

#### 2.Vector

底层也是Object[]，动态数组

方法都是synchronized，做了线程同步，所以多线程操作此集合安全;但是效率低

##### （1）构造参数

protected Object[] elementData;

public Vector(int initialCapacity, int capacityIncrement) {  
 super();  
 if (initialCapacity < 0)  
 throw new IllegalArgumentException("Illegal Capacity: "+  
 initialCapacity);  
 this.elementData = new Object[initialCapacity];  
 this.capacityIncrement = capacityIncrement;  
}

public Vector(int initialCapacity) {  
 this(initialCapacity, 0);  
}  
  
*/\*\*  
 \* Constructs an empty vector so that its internal data array  
 \* has size {****@code*** *10} and its standard capacity increment is  
 \* zero.  
 \*/*public Vector() {  
 this(10);  
}

public Vector(Collection<? extends E> c) {  
 elementData = c.toArray();  
 elementCount = elementData.length;  
 // c.toArray might (incorrectly) not return Object[] (see 6260652)  
 if (elementData.getClass() != Object[].class)  
 elementData = Arrays.*copyOf*(elementData, elementCount, Object[].class);  
}

默认无参构造初始化容量为10个

##### 扩容机制

private void grow(int minCapacity) {  
 // overflow-conscious code  
 int oldCapacity = elementData.length;  
 int newCapacity = oldCapacity + ((capacityIncrement > 0) ?  
 capacityIncrement : oldCapacity);  
 if (newCapacity - minCapacity < 0)  
 newCapacity = minCapacity;  
 if (newCapacity - *MAX\_ARRAY\_SIZE* > 0)  
 newCapacity = *hugeCapacity*(minCapacity);  
 elementData = Arrays.*copyOf*(elementData, newCapacity);  
}

是2倍扩容

#### 3.ArrayList与Vector的区别与联系？



相同点：

底层存储结构相同，都是数组，我们称为动态数组。

不同点：

初始化容量：Vector默认初始化容量为10,而ArrayList在JDK1.6之后的版本初始化为默认的空数组；

扩容：如果ArrayList一为默认的空数组，那么添加第一个元素时，扩容为默认大小为10的数组；其他情况下按照50%扩容。Vector扩容增加原来的1倍或按照用户指定的capacityIncrement增长。

线程安全：ArrayList线程不安全（方法没有synchronized修饰），效率高，Vector线程安全，效率低。

提示：在List列表集合实现中都有一个变量modCount，每次添加、删除都会使得modCount变量增加，因此用foreach和Iterator迭代时，如果再通过集合的add和remove等方法修改集合，会抛出 ConcurrentModificationException。

#### 4.Vector与Stack的区别与联系？

Stack类是Vector的子类。

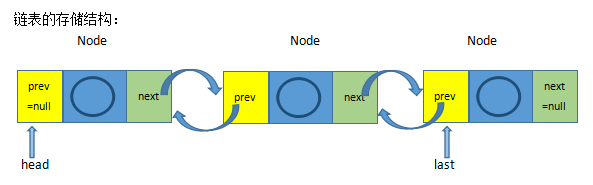
Stack类表示后进先出(LIFO)的对象堆栈。它通过五个操作对类 Vector 进行了扩展 ，允许将向量视为堆栈。它提供了通常的 push 和 pop 操作，以及取堆栈顶点的 peek 方法、测试堆栈是否为空的 empty 方法、在堆栈中查找项并确定到堆栈顶距离的 search 方法。

#### 5.LinkedList类

##### 1.内部类Node

节点有两个指针，说明是双向链表

|  |
| --- |
| private static class Node<E> {  E item;  Node<E> next;  Node<E> prev;   Node(Node<E> prev, E element, Node<E> next) {  this.item = element;  this.next = next;  this.prev = prev;  } } |



##### 2.实现的接口

除了实现 List 接口外，LinkedList类还实现了Deque接口，双向队列接口；

|  |
| --- |
| public class LinkedList<E>  extends AbstractSequentialList<E>  implements List<E>, Deque<E>, Cloneable, java.io.Serializable |

##### 3.LinkedList维护的属性

Linkedlist维护了first和last，是双向链表。

|  |
| --- |
| transient int size = 0;  */\*\*  \* Pointer to first node.  \* Invariant: (first == null && last == null) ||  \* (first.prev == null && first.item != null)  \*/* transient Node<E> first;  */\*\*  \* Pointer to last node.  \* Invariant: (first == null && last == null) ||  \* (last.next == null && last.item != null)  \*/* transient Node<E> last; |

##### 4.作为List的常用方法

###### (1)增

封装好的私有增加元素的方法，这些方法不对外提供，对内提供，属于底层实现。

* 从newNode的创建可以看出，是双向不循环链表

|  |
| --- |
| */\*\*  \* Links e as first element.  \*/* private void linkFirst(E e) {  final Node<E> f = first;  final Node<E> newNode = new Node<>(null, e, f);  first = newNode;  if (f == null)  last = newNode;  else  f.prev = newNode;  size++;  modCount++; }  */\*\*  \* 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++; }  */\*\*  \* 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++; } |

* 对外提供的增加元素的方法：

|  |
| --- |
| 继承自List接口的添加方法  public boolean add(E e) {  linkLast(e);  return true; }  public boolean addAll(Collection<? extends E> c) {  return addAll(size, c); }  public boolean addAll(int index, Collection<? extends E> c) {...}  public void add(int index, E element) {  checkPositionIndex(index);   if (index == size)  linkLast(element);  else  linkBefore(element, node(index)); } |

add(E e):默认添加元素是在尾部添加

###### (2)查

|  |
| --- |
| public E get(int index) {  checkElementIndex(index);  return node(index).item; } |

###### (3)删

私有的不对外提供的删除元素方法：

|  |
| --- |
| */\*\*  \* Unlinks non-null first node f.  \*/* private E unlinkFirst(Node<E> f) {  *// assert f == first && f != null;* final E element = f.item;  final Node<E> next = f.next;  f.item = null;  f.next = null; *// help GC* first = next;  if (next == null)  last = null;  else  next.prev = null;  size--;  modCount++;  return element; }  */\*\*  \* Unlinks non-null last node l.  \*/* private E unlinkLast(Node<E> l) {  *// assert l == last && l != null;* final E element = l.item;  final Node<E> prev = l.prev;  l.item = null;  l.prev = null; *// help GC* last = prev;  if (prev == null)  first = null;  else  prev.next = null;  size--;  modCount++;  return element; }  */\*\*  \* Unlinks non-null node x.  \*/* E unlink(Node<E> x) {  *// assert x != null;* final E element = x.item;  final Node<E> next = x.next;  final Node<E> prev = x.prev;   if (prev == null) {  first = next;  } else {  prev.next = next;  x.prev = null;  }   if (next == null) {  last = prev;  } else {  next.prev = prev;  x.next = null;  }   x.item = null;  size--;  modCount++;  return element; } |

作为list提供的删除方法

|  |
| --- |
| public boolean remove(Object o) {...}  public E remove(int index) {  checkElementIndex(index);  return unlink(node(index)); } |

###### (4)改

|  |
| --- |
| public E set(int index, E element) {  checkElementIndex(index);  Node<E> x = node(index);  E oldVal = x.item;  x.item = element;  return oldVal; } |

LinkedList 类还为在列表的开头及结尾 get、remove 和 insert 元素提供了统一的命名方法。这些操作允许将链接列表用作堆栈、队列或双端队列。

代码示例一：作为List的普通列表形式

|  |
| --- |
| **package** com.atguigu.part02.list;  **import** java.util.LinkedList;  **import** org.junit.Test;  **public** **class** TestLinkedList {  @Test  **public** **void** test1(){  LinkedList list = **new** LinkedList();  list.add(1);  list.add(2);  list.add(3);  list.add(4);  list.add(5);    **for** (Object object : list) {  System.***out***.println(object);  }  }  } |

##### 5.作为栈来使用

JDK1.6之后LinkedList实现了Deque接口。双端队列也可用作 LIFO（后进先出）堆栈。如果要使用堆栈结构的集合,可以考虑使用LinkedList,而不是Stack。

|  |  |
| --- | --- |
| 从Deque继承来的堆栈方法 | LinkedList作为双向链表自身的等效方法;这些方法也继承自Deque接口 |
| push(e) | addFirst(e) |
| pop() | removeFirst() |
| peek() | peekFirst() |

**注意：作为栈使用时，增删都在first节点操作**

代码示例二：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  //入栈  list.addFirst(1);  list.addFirst(2);  list.addFirst(3);    //出栈： LIFO（后进先出）  System.***out***.println(list.removeFirst());//3  System.***out***.println(list.removeFirst());//2  System.***out***.println(list.removeFirst());//1  //栈空了，会报异常java.util.NoSuchElementException  System.***out***.println(list.removeFirst());  } |

##### 6.作为队列使用

用作队列时，将得到 FIFO（先进先出）行为。将元素添加到双端队列的末尾，从双端队列的开头移除元素。

|  |  |
| --- | --- |
| Queue 方法 | 等效 Deque 方法 |
| add(e)：队尾入队 | addLast(e) |
| **offer(e)：队尾入队** | offerLast(e) |
| remove()：队头出队 | removeFirst() |
| **poll()** | pollFirst() |
| element()：查询队头 | getFirst() |
| **peek()** | peekFirst() |

注意：作为队列使用时,入队是在last添加，出队是从first删。

remove()方法时删除first元素

代码示例三：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  //入队  list.addLast(1);  list.addLast(2);  list.addLast(3);    //出队， FIFO（先进先出）  System.***out***.println(list.pollFirst());//1  System.***out***.println(list.pollFirst());//2  System.***out***.println(list.pollFirst());//3  //队空了，返回null  System.***out***.println(list.pollFirst());//null  } |

每种方法都存在两种形式：一种形式在操作失败时抛出异常，另一种形式返回一个特殊值（null 或 false,具体取决于操作）。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 第一个元素（头部） | | 最后一个元素（尾部） | |
| 抛出异常 | 特殊值 | 抛出异常 | 特殊值 |
| 插入 | addFirst(e) | offerFirst(e) | addLast(e) | offerLast(e) |
| 移除 | removeFirst() | pollFirst() | removeLast() | pollLast() |
| 检查 | getFirst() | peekFirst() | getLast() | peekLast() |

#### 7、结论

频繁插入和删除，建议使用LinkedList类，效率较高。因为不涉及到移动元素，只需要修改前后结点的关系。也不需要涉及到扩容。

如果查询多的场景，那么建议使用动态数组。

## 4. Queue集合

Queue接口源码分析：

|  |
| --- |
| **public** **interface** Queue<E> **extends** Collection<E> {    **boolean** add(E e);    **boolean** offer(E e);    E remove();  E poll();    E element();  E peek();  } |
| **public** **interface** Deque<E> **extends** Queue<E> {      **void** addFirst(E e);  **void** addLast(E e);    **boolean** offerFirst(E e);    **boolean** offerLast(E e);    E removeFirst();    E removeLast();  E pollFirst();    E pollLast();    E getFirst();  E getLast();      E peekFirst();  E peekLast();    **boolean** removeFirstOccurrence(Object o);    **boolean** removeLastOccurrence(Object o);  **boolean** add(E e);  **boolean** offer(E e);    E remove();  E poll();  .    E element();    E peek();    **void** push(E e);  E pop();  **boolean** remove(Object o);      **boolean** contains(Object o);    **public** **int** size();    Iterator<E> iterator();    Iterator<E> descendingIterator();  } |

容器LinkedList继承了Deque接口，实现了双端队列的一些方法，所以LinkedList可以当作单端口队列和双端口队列使用。

## 5. Set集合

* Set接口是Collection的子接口，但是set接口没有提供额外的方法。
* 不允许包含相同的元素，如果试把两个相同的元素加入同一个 Set 集合中，则添加操作失败。
* 最多只包含一个null
* Set集合不支持索引遍历，支持的遍历方式和Collection集合一样：foreach和Iterator。
* Set集合无序，取出的顺序和添加顺序不同。不过元素一旦添加进去了，位置是固定的
* Set的实现类有：HashSet、TreeSet、LinkedHashSet。

### 5.1 HashSet/LinkedHashSet/TreeSet

* HashSet和HashSet其子类都是按照 Hash 算法来存储集合中的元素，因此具有高效的写入和查找性能。
* 三个Set实现类元素存储的顺序
  + HashSet完全无序，存储的位置按照hash值来
  + LinkedHashSet保证了元素按照添加顺序存储
  + TreeSet按照元素的大小进行存储
* 由于LinkedHashSet元素的存储维护了添加顺序，因此插入性能略低于 HashSet，但在迭代访问 Set 里的全部元素时有很好的性能。
* 相应的三者的应用场景：
  + HashSet 如果是单纯的不可重复，即可使用
  + LinkedHashSet 不可重复但有添加顺序的要求
  + TreeSet 不可重复但要求存储有序

代码示例：

|  |  |
| --- | --- |
| HashSet set = **new** HashSet();  set.add("张三");  set.add("李四");  set.add("王五");  set.add("张三");    System.***out***.println("元素个数：" + set.size());  **for** (Object object : set) {  System.***out***.println(object);  } | LinkedHashSet set = **new** LinkedHashSet();  set.add("张三");  set.add("李四");  set.add("王五");  set.add("张三");    System.***out***.println("元素个数：" + set.size());  **for** (Object object : set) {  System.***out***.println(object);  } |
| 运行结果：  元素个数：3  李四  张三  王五 | 运行结果：  元素个数：3  张三  李四  王五 |
| @Test  **public** **void** test1(){  TreeSet set = **new** TreeSet();  set.add("zhangsan"); //String它实现了java.lang.Comparable接口  set.add("lisi");  set.add("wangwu");  set.add("zhangsan");    System.***out***.println("元素个数：" + set.size());  **for** (Object object : set) {  System.***out***.println(object);  }  } | |
| 运行结果：  元素个数：3  lisi  wangwu  zhangsan | |

### 5.2 Set集合如何保证元素不重复

* HashSet和LinkedHashSet：在add元素的时候，会先后调用hashcode和equals方法;先比较hash值，hash值不同，一定不同；hash值一样，再调用equals比较;

|  |
| --- |
|  |

* TreeSet：大小相同则认为相同

### 5.2 TreeSet

* TreeSet 是 SortedSet 接口的实现类,TreeSet 可以确保集合元素处于排序状态。
* 要实现集合内元素有序必须要使用自然排序或者定制排序接口：
  + Java.lang.Comparable接口
  + java.util.Comparator接口

TreeSet中存储的元素要比较大小，因此有两种方式来实现。

1. 如果使用TreeSet() 无参构造方法来创建TreeSet的时候，则必须要求传入的元素类型必须实现Comparable接口,并实现int ComparTo(Object obj)方法
2. 当用TreeSet(Comprator com)方法来创建TreeSet的时候，需要为传入的对象定制比较器，实现**int** compare(Object o1, Object o2)方法

#### 1、自然排序

* 自然排序的话，需要用TreeSet的无参构造，并且TreeSet的元素类型必须实现 Comparable 接口。实现 Comparable 的类必须实现 compareTo(Object obj) 方法，两个对象即通过 compareTo(Object obj) 方法的返回值来比较大小。
* 对于 TreeSet 集合而言，它判断两个对象是否相等的唯一标准是：两个对象通过 compareTo(Object obj) 方法比较返回值为0。

代码示例：

|  |
| --- |
| @Test  **public** **void** test2(){  TreeSet set = **new** TreeSet();  set.add(**new** Student(3,"张三")); //Student实现了java.lang.Comparable接口  set.add(**new** Student(1,"李四"));  set.add(**new** Student(2,"王五"));  set.add(**new** Student(3,"张三风"));    System.***out***.println("元素个数：" + set.size());  **for** (Object object : set) {  System.***out***.println(object);  }  } |
| **public class** Student **implements** Comparable{  **private** **int** id;  **private** String name;  **public** Student(**int** id, String name) {  **super**();  **this**.id = id;  **this**.name = name;  }  //......这里省略了get/set方法    @Override  **public** **int** compareTo(Object o) {  Student other = (Student) o;  **return** **this**.id - other.id;  }  @Override  **public** String toString() {  **return** "Student [id=" + id + ", name=" + name + "]";  }  } |
| 运行结果：  元素个数：3  Student [id=1, name=李四]  Student [id=2, name=王五]  Student [id=3, name=张三] |

提示：

虽然添加到TreeSet时，虽然不使用equals方法比较元素是否相等。但当元素实现java.lang.Comparable接口时，重写compareTo方法时，也建议重写 equals() 方法，并且应保证该方法与 compareTo(Object obj) 方法有一致的结果：如果两个对象通过 equals() 方法比较返回 true，则通过 compareTo(Object obj) 方法比较应返回 0。否则让人难以理解。

#### 2、定制排序

* 如果放到TreeSet中的元素的自然排序(Comparable)规则不符合当前排序需求时，或者元素的类型没有实现Comparable接口。那么在创建TreeSet时，可以单独指定一个Comparator的对象。
* 使用定制排序判断两个元素相等的标准是：通过Comparator比较两个元素返回了0。

代码示例：

|  |
| --- |
| @Test  **public** **void** test3(){  TreeSet set = **new** TreeSet(**new** Comparator(){  @Override  **public** **int** compare(Object o1, Object o2) {  Student s1 = (Student) o1;  Student s2 = (Student) o2;  **return** s1.getId() - s2.getId();  }    });  set.add(**new** Student(3,"张三"));  set.add(**new** Student(1,"李四"));  set.add(**new** Student(2,"王五"));  set.add(**new** Student(3,"张三风"));    System.***out***.println("元素个数：" + set.size());  **for** (Object object : set) {  System.***out***.println(object);  }  } |
| **public class** Student{  **private** **int** id;  **private** String name;  **public** Student(**int** id, String name) {  **super**();  **this**.id = id;  **this**.name = name;  }  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  //......这里省略了name属性的get/set  @Override  **public** String toString() {  **return** "Student [id=" + id + ", name=" + name + "]";  }  } |
| 运行结果：  元素个数：3  Student [id=1, name=李四]  Student [id=2, name=王五]  Student [id=3, name=张三] |

注意：

当使用具有与 equals 不一致的强行排序能力的 Comparator 对有序 set（或有序映射）进行排序时，应该小心谨慎。

### 5.2 Set的源码分析

#### 1.Set和Map的关系

Set的内部实现其实是一个Map

1. HashSet的内部实现是一个HashMap

|  |
| --- |
| **private** **static** **final** Object ***PRESENT*** = **new** Object();  **public** **boolean** add(E e) {  **return** map.put(e, ***PRESENT***)==**null**;  }  **public** Iterator<E> iterator() {  **return** map.keySet().iterator();  }  public HashSet() {  map = new HashMap<>(); }  HashSet的元素是作为Map的key使用的，其value是一个Object类型的static final对象PERSENT,**所有的hashSet都共用同一个PERSENT，因为此对象是静态的**.  同理，TreeSet和LinkedHashSet也是如此。  Set的迭代器，其实就是map.keySet().iterator |

1. TreeSet的内部实现是一个TreeMap

|  |
| --- |
| public TreeSet() {  this(new TreeMap<E,Object>()); } |

1. LinkedHashSet的内部实现是一个LinkedHashMap

|  |
| --- |
| public LinkedHashSet() {  super(16, .75f, true); }  HashSet(int initialCapacity, float loadFactor, boolean dummy) {  map = new LinkedHashMap<>(initialCapacity, loadFactor); } |

#### 2.Set如何判断元素是否重复

**(1)代码演示，引入问题**

|  |
| --- |
| package \_collection; import java.util.ArrayList; import java.util.HashSet; import java.util.LinkedList; import java.util.Vector;  public class \_01 {   public static void main(String[] args) {   HashSet<String> h = new HashSet<>();    *//hashSet的add方法返回boolean值，添加成功为true* System.*out*.println(h.add("john"));*//true* System.*out*.println(h.add("lucy"));*//true* System.*out*.println(h.add("john"));*//false* System.*out*.println(h.add("jack"));*//true* System.*out*.println(h.add("Rose"));*//true* h.remove("john");  System.*out*.println("set=" + h);   HashSet<Dog> d = new HashSet<>();  System.*out*.println(d.add(new Dog("snoopy"))); *//true* System.*out*.println(d.add(new Dog("snoopy"))); *//true* System.*out*.println(d);   *//todo 非常经典的面试题* h.add(new String("hmp"));*//true* h.add(new String("hmp"));*//false* System.*out*.println("set=" + h);*//set=[Rose, hmp, lucy, jack]* }  } class Dog{  private String name;  public Dog(){}  public Dog(String name){  this.name = name;  } } |

HashSet 和LinkedHashSet集合判断两个元素相等的标准：**两个对象通过 hashCode() 方法比较相等，并且两个对象的 equals() 方法返回值也相等。**因此，存储到HashSet和LinkedHashSet的元素要重写hashCode和equals方法。

重写equals()方法的原则：

* 对称性：如果x.equals(y)返回是“true”，那么y.equals(x)也应该返回是“true”。
* 自反性：x.equals(x)必须返回是“true”。
* 类推性：如果x.equals(y)返回是“true”，而且y.equals(z)返回是“true”，那么z.equals(x)也应该返回是“true”。
* 一致性：如果x.equals(y)返回是“true”，只要x和y内容一直不变，不管你重复x.equals(y)多少次，返回都是“true”。
* 任何情况下，x.equals(null)，永远返回是“false”；x.equals(和x不同类型的对象)永远返回是“false”。

重写 hashCode() 方法的基本原则

* 在程序运行时，同一个对象多次调用 hashCode() 方法应该返回相同的值
* 当两个对象的 equals() 方法比较返回 true 时，这两个对象的 hashCode() 方法的返回值也应相等
* 对象中用作 equals() 方法比较的属性Field，都应该用来计算 hashCode 值

#### 3.HashSet的add方法

|  |
| --- |
| public boolean add(E e) {  return map.put(e, *PRESENT*)==null; } |
| public V put(K key, V value) {  return putVal(*hash*(key), key, value, false, true); } |

* HashSet的add方法实际上调用的是map.put(e,PERSENT)
* 添加一个元素时，先得到hash值，根据hash值可以得到该元素在数组中对应的索引值

#### 4.HashMap中的散列函数hash()

putVal中hash(key)决定了这个元素放到数组的哪个位置，那么hash是如何实现的？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| JDK1.7和JDK1.8关于hash()的实现代码不一样，但是不管怎么样都是为了让元素之间尽量不出现hash碰撞，更加均匀的分布。   |  |  | | --- | --- | | JDK1.7 | **final** **int** hash(Object k) {  **int** h = hashSeed;  **if** (0 != h && k **instanceof** String) {  **return** sun.misc.Hashing.*stringHash32*((String) k);  }  h ^= k.hashCode();  h ^= (h >>> 20) ^ (h >>> 12);  **return** h ^ (h >>> 7) ^ (h >>> 4);  } | | JDK1.8 | **static** **final** **int** hash(Object key) {  **int** h;  **return** (key == **null**) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  } |   这里用JDK1.8的示例分析一下：     * **1.8中hash的值**   元素的hashCode 与 其无符号右移16位按位与  如果元素为null,那么hash为0   * **JDK1.8为什么要保留高16位呢？**   因为index的求值方式为： index = hash & (n-1),n=table.length  而一个HashMap的table数组一般不会特别大,至少在不断扩容之前,那么对于32位的int类型的table.length-1来说,大部分高位都是0,直接用hashCode和table.length-1进行&运算的话,就会导致总是只有最低的几位是有效的,那么就算你的hashCode()实现的再好也难以避免发生碰撞,这时保留高16位的意义就体现出来了。它对hashcode的低位添加了随机性并且混合了高位的部分特征，显著减少了碰撞冲突的发生。 |

## 11.5 Map集合

### 11.5.1 Map概述

将键映射到值的对象。

* Map 中的 key 和 value 都可以是任何引用类型的数据；key通常使用不可变对象，比如String和Integer
* Map 中的 key 不允许重复
* key 和 value 之间存在单向一对一关系，即通过指定的 key 总能找到唯一的、确定的 value。
* Map 接口提供三种collection 视图：

键集

值集

键-值映射关系集

映射顺序定义为迭代器在映射的 collection 视图上返回其元素的顺序。某些映射实现可明确保证其顺序，如 TreeMap 类；另一些映射实现则不保证顺序，如 HashMap 类。

### 11.5.2 Map API

**1、添加操作**

* Object put(Object key,Object value)：如何key重复，会覆盖前面的value
* void putAll(Map t)

**2、删除**

* void clear()
* Object remove(Object key)

**3、元素查询的操作**

* Object get(Object key)
* boolean containsKey(Object key)
* boolean containsValue(Object value)
* boolean isEmpty()

**4、遍历Map的相关方法：**

Map是无法直接遍历的

* Set keySet()： 因为key不可重复，所以组成set集合
* Collection values()：value可能重复也可能不重复所以是个Collection系列的集合
* Set entrySet()：获取key-value对 set集合；一个k-v封装为entry类型

1. 其他方法

* int size()

### 11.5.3 Map的遍历

Collection集合的遍历：（1）foreach（2）通过Iterator对象遍历

Map的遍历：不直接支持遍历

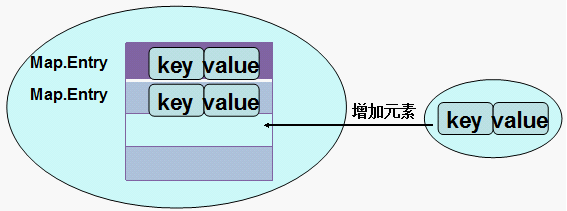
（1）分开遍历：单独遍历所有key

单独遍历所有value

（2）成对遍历：遍历的是映射关系Map.Entry

Map.Entry是Map接口的内部接口。每一种Map内部有自己的Map.Entry的实现类

在Map中存储数据，实际上是将Key---->value的数据存储在Map.Entry接口的实例中，再在Map集合中插入Map.Entry的实例化对象，如图示：



代码示例：

|  |
| --- |
| **public** **class** TestMap {  **public** **static** **void** main(String[] args) {  HashMap map = **new** HashMap();  map.put("许仙", "白娘子");  map.put("董永", "七仙女");  map.put("牛郎", "织女");  map.put("许仙", "小青");    System.***out***.println("所有的key:");  Set keySet = map.keySet();  **for** (Object key : keySet) {  System.***out***.println(key);  }    System.***out***.println("所有的value：");  Collection values = map.values();  **for** (Object value : values) {  System.***out***.println(value);  }    System.***out***.println("所有的映射关系");  Set entrySet = map.entrySet();  **for** (Object entry : entrySet) {  System.***out***.println(entry);  }  }  } |

### 11.5.4 Map的实现类

#### 1.Map接口的常用实现类

Map的底层实现分为两大类：

1. 哈希表：包含了hashMap和hashtable 数组+链表/红黑树结构
2. TreeMap 红黑树结构

Map所有的实现类如下：

* HashMap( Map 接口使用频率最高的实现类)
* TreeMap
* LinkedHashMap
* Properties

#### 2.HashMap和Hashtable的区别与联系

1. HashMap和Hashtable都是哈希表。
2. Hashtable是旧版，hashMap是新版；
3. Hashtable是线程安全的，key和value均不能为nulll
4. HashMap是线程不安全的，并允许key和value使用 null 值

HashMap和Hashtable判断两个 key 相等的标准是：两个 key 的hashCode 值相等，并且 equals() 方法也返回 true。因此，为了成功地在哈希表中存储和获取对象，用作键的对象必须实现 hashCode 方法和 equals 方法。

#### 3.LinkedHashMap

LinkedHashMap 是 HashMap 的子类；

LinkedHashMap维护了映射关系的添加顺序；比hashMap添加效率低

#### 4.TreeMap

* 基于红黑树（Red-Black tree）的 NavigableMap 实现。
* 往TreeMap添加的元素要么实现了Comparable接口，要么创建TreeMap的时候提供了 Comparator 进行排序，具体取决于使用的构造方法。
* 按照key的大小排序

#### 5.Properties

1. Properties 类是 Hashtable 的子类。
2. Properties不允许key和value有null值
3. Properties的key和value都必须为String类型存取数据
4. 通常用于存储配置；通常结合IO读取配置文件中存储的key=value型的数据
5. 提供了使用setProperty(String key,String value)方法和getProperty(String key)方法。

代码示例：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Properties properties = System.*getProperties*();  String p2 = properties.getProperty("file.encoding");//当前源文件字符编码  System.***out***.println(p2);  } |

## 11.6 Map源码分析

存储到HashMap中的映射关系(key,value)，其中的key的hashCode值和equals方法非常重要。

* Map的底层实现：

1. 哈希表系列：

Jdk1.7及之前 数组 + 链表

Jdk1.8及之后 数组 + 链表/红黑树

1. TreeMap：

红黑树

数组优点：访问特别快，直接根据索引定位；确定是增删要移动大量元素

链表优点：元素物理存储不需要紧挨着；添加和删除很方便

HashMap会根据key的hashCode获取数组的索引；链表的目的：当不同元素出现hash碰撞的时候，可以将多个值用链表连在一起。

二叉树的优点：查询比链表快；当一个[index]下的链表过长，导致查询速度减慢；长到一定程度，链表转为树，提高查询速度。

### 11.6.1 JDK1.7的hashMap

#### (1)常量和属性

**static** **final** **float *DEFAULT\_INITIAL\_CAPACITY*** = 16

**static** **final** **float** ***DEFAULT\_LOAD\_FACTOR*** = 0.75f;

**final** **float** loadFactor;

**int** threshold;

**三个核心变量：**

1. initial\_capacity:初始化容量为16
2. Load\_factor:加载因子
3. Threshold: 扩容阈值
   1. 数组需要**考虑**扩容的阈值；注意是考虑，并非一定扩容；
   2. 其值 = capacity \* load factor

**思考：load factor设置为0.9或者0.1有什么区别？**

0.9 扩容太不频繁，导致table[index]下面的链表可能会很长，降低了查询速度

0.1 扩容太频繁

#### (2)new HashMap()

|  |
| --- |
|  |
| **hashMap的无参构造干了三件事：**   1. 创建了空的Entry数组 2. 加载因子loadFactor赋值0.75 3. 阈值threshold赋值**16**   **这里注意：threshold是16不是12** |



#### JDK1.7的put存储过程

##### Step1：如果table数组是否为空

|  |
| --- |
| inflateTable(threshold)中threshold的值为16，因为new HashMap() 中初始化threshold为16 |
| |  | | --- | | 此方法用于获取最接近number的2^n，也就是说保证了数组的长度必然是2的n次蜜 |   inflateTable()：  Capacity为16  threshold 变成了16\*0.75  table变成了容量为16的Entry数组！ |

* put()方法中如果是空的Entry数组所做的事情：
  + threshold 变成了16\*0.75
  + table变成了容量为16的Entry数组！

所以threshold在空参构造中生成为16，在往空数组中put的时候才变为12

* 为什么将capacity纠正为2^n?

如果指定了数组的初始化容量capacity,会检测capacity是不是2^n，不是的话自动纠正为最接近的2^n



##### Step2：处理key为null的情况

|  |  |  |
| --- | --- | --- |
| |  | | --- | |  | | Key为null的，hash为0，index也为0 | |

##### Step3：获取不为null的key的hash值

|  |  |
| --- | --- |
| 根据key求hash值   |  | | --- | | 这里不具体研究hash值的算法，反正目的就是散列hash值，避免hash碰撞 | |

注意：key=null肯定存在table[0],key!=null也有可能存在table[0]

##### Step4：根据hash值获取index值

|  |  |  |
| --- | --- | --- |
| |  | | --- | | 所以：index = hash(key) & (table.lenth -1)  2^n - 1 的特征是： 前面都是0,后面都是连续的1  因此按位与运算，结果一定是在[0,table.length-1] | | 因为hash code是一个整数，而数组的长度也是一个整数，因此要限制hash值映射到table.lenth范围内只有两种思路：  ①hash code值 % table.length会得到一个[0,table.length-1]范围的值，正好是下标范围，但是用%运算，不能保证均匀存放，可能会导致某些table[index]桶中的元素太多，而另一些太少，因此不合适。  ②hash code值 & (table.length-1)，因为table.length是2的幂次方，因此table.length-1是一个二进制低位全是1的数，所以&操作完，也会得到一个[0,table.length-1]范围的值。 | |

##### Step5：查找table[i]下面的链表中是否有Entry的key是和当前put的key是重复的，如果有重复的，用新的换旧的

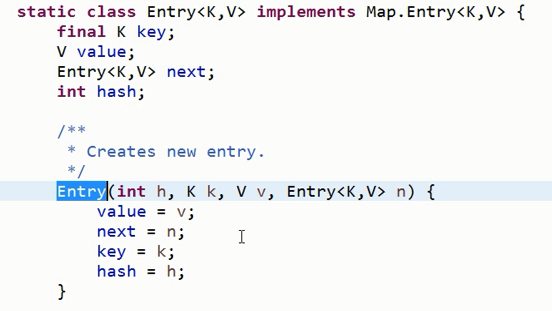
|  |
| --- |
| 当table[i]不为null，说明该index下面已经有东西了  然后遍历table[i]的链表，如果hash值相同 并且 key相同(key地址相同或者equals相同),则用新的value替换旧的value。  如果有相同的key,for循环中直接返回老的value,put()到这里就结束了，如果没有遇到相同的key，那么还有下面第六步 |

##### Step6：将entry存入到table中

如果没有相同的key,则会执行下面的方法：

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 添加entry前，首先要进行size判断是否要给数组扩容：  如果size达到阈值并且新的entry所要放置的table[index]不为空，那么数组扩容为2倍，并且重新计算hash值、index值  扩容条件：size>=threshold && table[i] != null;所以threshold不是唯一扩容条件  注意：resize()不是简单的扩容，会将table中原有的元素都会重新计算位置   |  | | --- | | e是table[index]的entry，也就是说让原本的table[index]连接到新的entry的next，自己变成table[index]的”头结点” | | |

##### Entry类：



可以看出来Entry是Map内部类，维护了四个属性

1. Key
2. Value
3. 下一个节点的指针 也是Entry类型(说明Entry是个单向链表的节点类型)
4. Hash值

##### 总结存储过程

1. 创建HashMap对象

new HashMap() 创建了一个空的Entry数组，扩容阈值threshold初始化为16，加载因子初始化为0.75

put元素：

1. 初始化table数组

先判断table是否为空数组，如果是，先初始化Entry数组长度为16;并且把threshold初始化为12

如果指定了capacity，会纠正capacity为2^n

1. 求出元素的hash值，并求出存储index

如果key是null,则hash值为0,并且index=0;

如果key不是null，那么先计算hash(key)，在通过处理过的hash值&(table.length-1)计算index

1. 判断table[index]下的节点是否有相同key元素

* 判断table[index]桶下是否存在某个Entry的key与新的key的“相同”

**相同的条件：hash值相同 并且 (key的地址相同 或 key的equals返回true)**

* 如果存在相同的，用新的value替换原来的value;put结束
* 如果不存在，决定将新的元素添加到table[index]下：先判断是否需扩容：

**扩容条件： size达到阈值(threshold) 并且 table[index]不是null**

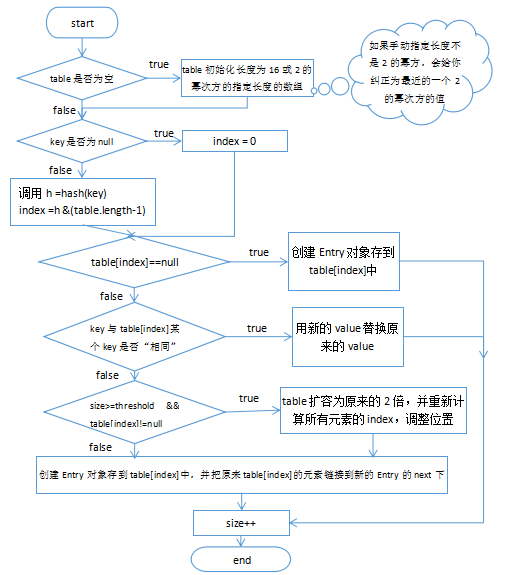
* 如果满足了扩容条件，那么会扩容会导致原来table中的所有元素都会重新计算hash值和index，并调整存储位置；

(5)添加元素

把新的k-v封装成一个新的Entry对象至table[index](注意，这个index也是重新计算过的)中，并且把当前table[index]下的所有元素都连接到新的Entry的next下。

**添加元素的时候，新元素做头结点**

(6)size++，元素个数增加



### 11.6.2 JDK1.8的hashMap

#### 1.维护的属性

|  |
| --- |
| **常量们：**  **//1.默认初始容量**  static final int *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4; *// aka 16*  **//2.数组最大容量**static final int *MAXIMUM\_CAPACITY* = 1 << 30;  **//3.默认加载因子**static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f;  **//4.树化阈值 当链表长度达到这个值，考虑树化**  static final int *TREEIFY\_THRESHOLD* = 8;  **//5.退树化阈值 链表长度降到这个值，考虑树变为链表**static final int *UNTREEIFY\_THRESHOLD* = 6;  **//6.最小树化容量 这个指的是数组的长度**static final int *MIN\_TREEIFY\_CAPACITY* = 64;  **说明：当链表长度达到了8并且数组长度达到了64，此时该链表会树化**  **当链表长度达到了8但是数组长度到达64，此时会对数组进行扩容**  **变量们：**  **//1.HashMap底层真正的存储结构 Node数组**  transient Node<K,V>[] table; transient Set<Map.Entry<K,V>> entrySet;  transient int size ;  **//2.记录HashMap结构变化次数**transient int modCount;  **//3.阈值，当size达到此值，考虑扩容**  **(capacity \* load factor)**  int threshold;  **//4.加载因子 影响扩容频率**final float loadFactor; |

#### 2.内部类Entry

|  |
| --- |
| static class Node<K,V> implements Map.Entry<K,V> {  final int hash;  final K key;  V value;  Node<K,V> next;   Node(int hash, K key, V value, Node<K,V> next) {  this.hash = hash;  this.key = key;  this.value = value;  this.next = next;  }   public final K getKey() { return key; }  public final V getValue() { return value; }  public final String toString() { return key + "=" + value; }   public final int hashCode() {  return Objects.*hashCode*(key) ^ Objects.*hashCode*(value);  }   public final V setValue(V newValue) {  V oldValue = value;  value = newValue;  return oldValue;  }   public final boolean equals(Object o) {  if (o == this)  return true;  if (o instanceof Map.Entry) {  Map.Entry<?,?> e = (Map.Entry<?,?>)o;  if (Objects.*equals*(key, e.getKey()) &&  Objects.*equals*(value, e.getValue()))  return true;  }  return false;  } } |
| (1)四个属性：  final int hash;  final K key;  V value;  Node<K,V> next;  可以看出是个单链表的节点  (2)获取entry的key和value的方法  public final K getKey() { return key; } public final V getValue() { return value; }  (3)重写了toString  public final String toString() { return key + "=" + value; } (4)hashCode算法  public final int hashCode() {  return Objects.*hashCode*(key) ^ Objects.*hashCode*(value);  } key和value的hash值的异或  (5)设置value值  返回老的value  (6)重写了equals方法，判断两个Entry是否一致  是按照key和value两个一起比较的。  public final boolean equals(Object o) {  if (o == this)  return true;  if (o instanceof Map.Entry) {  Map.Entry<?,?> e = (Map.Entry<?,?>)o;  if (Objects.*equals*(key, e.getKey()) &&  Objects.*equals*(value, e.getValue()))  return true;  }  return false;  } |

#### 3.new HashMap()

|  |
| --- |
| public HashMap() {  this.loadFactor = *DEFAULT\_LOAD\_FACTOR*; *// all other fields defaulted* } |

无参构造器只做了一件事：给加载因子赋默认值0.75;

其他都是默认值因此table为null、threshold为0

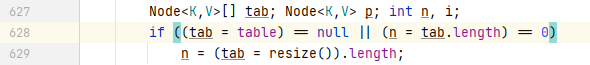
#### 4.public V put(K key, V value)

|  |
| --- |
| final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  boolean evict) {  Node<K,V>[] tab; Node<K,V> p; int n, i;  //step1.初始化阶段，如果数组不为null就不用初始化。  //初始化的结果就是table.length = 16 threshold = 12  if ((tab = table) == null || (n = tab.length) == 0)  n = (tab = resize()).length; //如果是第一次添加，那么n=16  //step2.table数组有了，那么接下来就是讲newNode放进table中了  //两种情况：table[i] == null 直接放进去  if ((p = tab[i = (n - 1) & hash]) == null)  tab[i] = newNode(hash, key, value, null);  //table[i]不为null p为table[i]节点  else {  Node<K,V> e; K k;  if (p.hash == hash &&  ((k = p.key) == key || (key != null && key.equals(k))))  e = p;  else if (p instanceof TreeNode)  e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  else {  for (int binCount = 0; ; ++binCount) {  if ((e = p.next) == null) {  p.next = newNode(hash, key, value, null);  if (binCount >= *TREEIFY\_THRESHOLD* - 1) *// -1 for 1st* treeifyBin(tab, hash);  break;  }  if (e.hash == hash &&  ((k = e.key) == key || (key != null && key.equals(k))))  break;  p = e;  }  }  //e到这个时候为null，说明没有重复的key,  //不为null,则e为key重复的node  //这里是处理key重复的情况，哪新的value替换旧的，然后返回老的值 put就此结束  if (e != null) { *// existing mapping for key* V oldValue = e.value;  if (!onlyIfAbsent || oldValue == null)  e.value = value;  afterNodeAccess(e);  return oldValue;  }  }  //每put一次，modCount都要+1  ++modCount;  if (++size > threshold)  resize();  afterNodeInsertion(evict);  return null; } |

##### Step1.求key的hash值

|  |  |
| --- | --- |
| public V put(K key, V value) {  return putVal(*hash*(key), key, value, false, true); }   * put调用putVal,三个参数： 1：hash(key) 2.就是key和value  |  | | --- | | static final int hash(Object key) {  int h;  return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16); }   * Key的hash值的计算：   如果key为null, 那么hash值为0  如果key不为null,(h = key.hashCode()) ^ (h >>> 16)  对比JDK1.7中求index的方式 : index=hash & table.length-1  Jdk1.8hash算法的优势：1.7的hash值将key的hash值和table.length-1按位与，这样高16位没有机会参与hash值的计算，会增加冲突概率；为了降低冲突概率，将高16位加入到hash的计算 | |
| putVal方法：  **参数列表：**  hash – hash for key  key – the key  value – the value to put  onlyIfAbsent – if true, don't change existing value  evict – if false, the table is in creation mode.  **返回值：**  previous value, or null if none  final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  boolean evict) {....} |

##### Step2：处理table为空的情况



tab是数组

p是一个节点

n是数组长度

i是下标

如果table为空，表示是第一次添加；这里是判断为空的两种形式：

1. {} 此时tab.length = 0
2. 只是声明没初始化则为null

###### table为空的时候resize()的执行逻辑分析

|  |
| --- |
| **1.该部分重点分析new HashMap之后第一次put的情况，没有注释的说明此情况下不会执行这些代码**  final Node<K,V>[] resize() {  //1.获取旧的table 要么为null要么为{}  Node<K,V>[] oldTab = table;  //2.旧table的容量 oldCap = 0  int oldCap = (oldTab == null) ? 0 : oldTab.length;  //3.旧阈值 threshold在默认构造器中不赋值，为0  int oldThr = threshold;  //4.新容量 新阈值  int newCap, newThr = 0;  ~~//5.处理老table不为空的情况  if (oldCap > 0) {  if (oldCap >=~~ *~~MAXIMUM\_CAPACITY~~*~~) {~~**~~//数组是否达到最大容量~~** ~~threshold = Integer.~~*~~MAX\_VALUE~~*~~;  return oldTab;  }  else if ((newCap = oldCap << 1) <~~ *~~MAXIMUM\_CAPACITY~~* ~~&&  oldCap >=~~ *~~DEFAULT\_INITIAL\_CAPACITY~~*~~)~~**~~//老容量>16~~**  ~~newThr = oldThr << 1;~~***~~// double threshold~~***~~}~~  ~~//6.处理老table的阈值不为0的情况  else if (oldThr > 0)~~ *~~// initial capacity was placed in threshold~~* ~~newCap = oldThr;~~  **//7.这是处理一个new HashMap的情况**  else { *// zero initial threshold signifies using defaults* newCap = *DEFAULT\_INITIAL\_CAPACITY*;  newThr = (int)(*DEFAULT\_LOAD\_FACTOR* \* *DEFAULT\_INITIAL\_CAPACITY*);  //赋予新容量和新阈值，都为默认值； 16和16\*0.75  }  ~~if (newThr == 0) {  float ft = (float)newCap \* loadFactor;  newThr = (newCap <~~ *~~MAXIMUM\_CAPACITY~~* ~~&& ft < (float)~~*~~MAXIMUM\_CAPACITY~~* ~~?  (int)ft : Integer.~~*~~MAX\_VALUE~~*~~);  }~~  threshold = newThr;//将新阈值赋值给阈值属性  @SuppressWarnings({"rawtypes","unchecked"})  //创建容量为16的新Node数组  Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];  table = newTab;   ~~if (oldTab != null) {~~  **~~//把原来table中的映射关系倒腾到新的table中~~**  ~~for (int j = 0; j < oldCap; ++j) {  Node<K,V> e;  if ((e = oldTab[j]) != null) {  oldTab[j] = null;  if (e.next == null)  newTab[e.hash & (newCap - 1)] = e;  else if (e instanceof TreeNode)  ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  else {~~ *~~// preserve order~~* ~~Node<K,V> loHead = null, loTail = null;  Node<K,V> hiHead = null, hiTail = null;  Node<K,V> next;  do {  next = e.next;  if ((e.hash & oldCap) == 0) {  if (loTail == null)  loHead = e;  else  loTail.next = e;  loTail = e;  }  else {  if (hiTail == null)  hiHead = e;  else  hiTail.next = e;  hiTail = e;  }  } while ((e = next) != null);  if (loTail != null) {  loTail.next = null;  newTab[j] = loHead;  }  if (hiTail != null) {  hiTail.next = null;  newTab[j + oldCap] = hiHead;  }  }  }  }  }~~  return newTab; } |

结论：对于一个空table，resize()干了两件事  
 1.创建一个容量为16的Node[] table

2.threshold赋予 16\*0.75

##### Step3：处理table[i]=null的情况，意味着此位置第一次插入



new Node()

Node<K,V> newNode(int hash, K key, V value, Node<K,V> next) {  
 return new Node<>(hash, key, value, next);  
}

结论：

1. 插入位置的求法： i = (n-1) & hash
2. 创建一个新的Node,其next指向null

##### **Step4：**处理table[i] != null的情况

|  |
| --- |
| Node<K,V> e; K k;  **//p是table[i]位置的第一个Node**  **//case1:处理table[i]位置第一个Node和 新元素的key相同的情况** if (p.hash == hash &&  ((k = p.key) == key || (key != null && key.equals(k))))  e = p;  **//case2:如果table[i]是个树节点(已经树化了)** else if (p instanceof TreeNode)  e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  **//case3:如果table[i]不是树节点，而且key不和table[i]重复** else {  **//binCount记录了table[i]下面节点个数**  for (int binCount = 0; ; ++binCount) {  **//这里，是将新节点放在table[i]下面的最后一个，e=p.next e==null**  if ((e = p.next) == null) {  p.next = newNode(hash, key, value, null);  **//当节点个数>=8-1的时候就进行树化（并不一定树化，要么扩容要么树化）**  if (binCount >= *TREEIFY\_THRESHOLD* - 1) *// -1 for 1st* treeifyBin(tab, hash);  break;  }  **//如果e(p.next)的key重复了，就跳出循环，就是什么都没干**  if (e.hash == hash &&  ((k = e.key) == key || (key != null && key.equals(k))))  break;  p = e;  } }  //如果e不为空，说明有key重复，则替换value if (e != null) { *// existing mapping for key* V oldValue = e.value;  if (!onlyIfAbsent || oldValue == null)  e.value = value;  afterNodeAccess(e);  return oldValue; } |

**树化方法：**

|  |
| --- |
| final void treeifyBin(Node<K,V>[] tab, int hash) {  int n, index; Node<K,V> e;  **//case1:如果table为空，或者table长度没有达到64，先扩容**  if (tab == null || (n = tab.length) < *MIN\_TREEIFY\_CAPACITY*)  resize();  **//case2: table不为空并且table长度达到了树化容量**  else if ((e = tab[index = (n - 1) & hash]) != null) {  TreeNode<K,V> hd = null, tl = null;  **//do while 把table[index]链表变成了红黑树**  do {  TreeNode<K,V> p = replacementTreeNode(e, null);  if (tl == null)  hd = p;  else {  p.prev = tl;  tl.next = p;  }  tl = p;  } while ((e = e.next) != null);  if ((tab[index] = hd) != null)  hd.treeify(tab);  } } |

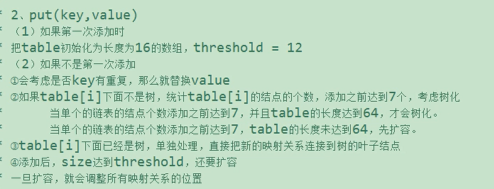
* 当链表长度>=8-1的时候，会执行treeifyBin方法：
  + 如果table为空，或者table长度没有达到64，**先扩容**
  + Table[(n - 1) & hash]不为空并且table长度达到了树化容量,则链表会树化

###### table不为空的时候，resize()的执行逻辑分析

|  |
| --- |
| final Node<K,V>[] resize() {  //1.获取旧的table 16  Node<K,V>[] oldTab = table;  //2.旧table的容量 oldTab.length  int oldCap = (oldTab == null) ? 0 : oldTab.length;  //3.旧阈值 threshold 为12  int oldThr = threshold;  //4.新容量 新阈值  int newCap, newThr = 0;  if (oldCap > 0) {  if (oldCap >= *MAXIMUM\_CAPACITY*) {**//数组是否达到最大容量**  threshold = Integer.*MAX\_VALUE*;  return oldTab;  }  else if ((newCap = oldCap << 1) < *MAXIMUM\_CAPACITY* &&  oldCap >= *DEFAULT\_INITIAL\_CAPACITY*) newThr = oldThr << 1;  newCap = 旧容量\*2,  新容量<最大数组长度限制 && oldCap >=初始容量16 则 threshold\*2  所以这里，新容量和新阈值都乘以2倍  }  ~~else if (oldThr > 0)~~ *~~// initial capacity was placed in threshold~~* ~~newCap = oldThr;  else {~~ *~~// zero initial threshold signifies using defaults~~* ~~newCap =~~ *~~DEFAULT\_INITIAL\_CAPACITY~~*~~;  newThr = (int)(~~*~~DEFAULT\_LOAD\_FACTOR~~* ~~\*~~ *~~DEFAULT\_INITIAL\_CAPACITY~~*~~);  }~~  ~~if (newThr == 0) {  float ft = (float)newCap \* loadFactor;  newThr = (newCap <~~ *~~MAXIMUM\_CAPACITY~~* ~~&& ft < (float)~~*~~MAXIMUM\_CAPACITY~~* ~~?  (int)ft : Integer.~~*~~MAX\_VALUE~~*~~);  }~~  threshold = newThr;// 变为24  @SuppressWarnings({"rawtypes","unchecked"})  //创建容量为32的新Node数组  Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];  table = newTab;  **//把原来table中的映射关系倒腾到新的table中**  if (oldTab != null) { for (int j = 0; j < oldCap; ++j) {  Node<K,V> e;  if ((e = oldTab[j]) != null) {  oldTab[j] = null;  if (e.next == null)  newTab[e.hash & (newCap - 1)] = e;  else if (e instanceof TreeNode)  ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  else { *// preserve order* Node<K,V> loHead = null, loTail = null;  Node<K,V> hiHead = null, hiTail = null;  Node<K,V> next;  do {  next = e.next;  if ((e.hash & oldCap) == 0) {  if (loTail == null)  loHead = e;  else  loTail.next = e;  loTail = e;  }  else {  if (hiTail == null)  hiHead = e;  else  hiTail.next = e;  hiTail = e;  }  } while ((e = next) != null);  if (loTail != null) {  loTail.next = null;  newTab[j] = loHead;  }  if (hiTail != null) {  hiTail.next = null;  newTab[j + oldCap] = hiHead;  }  }  }  }  }  return newTab; } |

**总结：不为空，resize() =》**新容量和新阈值都乘以2倍

|  |
| --- |
| ++modCount;  元素个数增加，达到阈值 if (++size > threshold)  resize();  把新的节点放到某个node后面 afterNodeInsertion(evict); return null; |



两种情况下会扩容：

1. table[i]下面的链表节点达到8个，但是size不超过64
2. 添加元素后，table中元素个数>阈值(threshold),会扩容

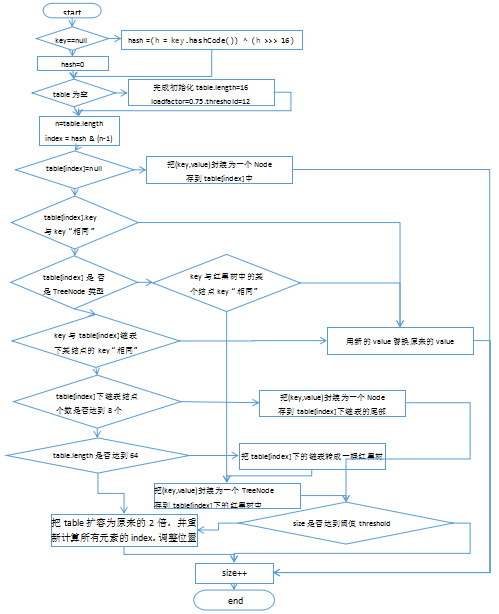
最少11个节点会变树：

8个 不变树

9个 满足上面条件1，扩容 32

10个 满足上面条件1 扩容64

11个 不满足条件1了 变树



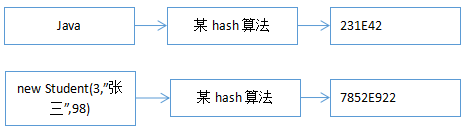
#### 5.modCount

* 记录了当前集合被修改（add,remove）的次数
* 作用：当我们是用迭代器或者foreach遍历的时候，如果foreach遍历的时候，自动调用迭代器的迭代方法。此时在遍历过程中调用了集合的add，remove方法，modCount会改变，而迭代器所记录的modCount是开始迭代之前的，如果两个不一致，就会报异常。除非调用迭代器自身的add和remove

引入modCOunt，是为了让集合在并发场景下做增删，很快完全失败，而不是冒着在将来某个不确定时间发生意外的不确定性行为的风险。

### 1、hashCode值

hash算法是一种可以从任何数据中提取出其“指纹”的数据摘要算法，它将任意大小的数据映射到**一个固定大小的序列上**，这个序列被称为hash code、数据摘要或者指纹。比较出名的hash算法有MD5、SHA。**hash是具有唯一性且不可逆的，唯一性是指相同的“对象”产生的hash code永远是一样的。**



### 2、Entry数组

HashMap和Hashtable是散列表，**其中维护了一个长度为2的幂次方的Entry类型的数组table**，数组的每一个元素被称为一个桶(bucket)，你添加的映射关系(key,value)最终都被封装为一个Map.Entry类型的对象，放到了某个table[index]桶中。使用数组的目的是查询和添加的效率高，可以根据索引直接定位到某个table[index]。

#### （1）数组元素类型：Map.Entry

|  |  |
| --- | --- |
| JDK1.7 | **transient** Entry<K,V>[] table = (Entry<K,V>[]) ***EMPTY\_TABLE***;  **static** **class** Entry<K,V> **implements** Map.Entry<K,V> {  **final** K key;  V value;  Entry<K,V> next;  **int** hash;  //...省略  } |
| JDK1.8 | **transient** Node<K,V>[] table;  **static** **class** Node<K,V> **implements** Map.Entry<K,V> {  **final** **int** hash;  **final** K key;  V value;  Node<K,V> next;  //...省略  } |

#### （2）初始容量：16

|  |
| --- |
| **static** **final** **int** ***DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4; |

#### （3）扩容为原来的2倍

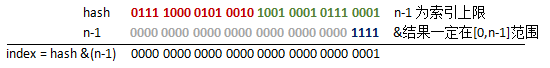
|  |  |
| --- | --- |
| JDK1.7 | **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {  **if** ((size >= threshold) && (**null** != table[bucketIndex])) {  **resize(2 \* table.length);**  hash = (**null** != key) ? hash(key) : 0;  bucketIndex = *indexFor*(hash, table.length);  }  createEntry(hash, key, value, bucketIndex);  } |
| JDK1.8 | **final** Node<K,V>[] resize() {  Node<K,V>[] oldTab = table;  **int** oldCap = (oldTab == **null**) ? 0 : oldTab.length;  **int** oldThr = threshold;  **int** newCap, newThr = 0;  **if** (oldCap > 0) {  **if** (oldCap >= ***MAXIMUM\_CAPACITY***) {  threshold = Integer.***MAX\_VALUE***;  **return** oldTab;  }  **else** **if** ((**newCap = oldCap << 1**) < ***MAXIMUM\_CAPACITY*** &&  oldCap >= ***DEFAULT\_INITIAL\_CAPACITY***)  newThr = oldThr << 1; // double threshold  }  //......此处省略其他代码  } |

#### （4）hash值 索引

因为hash code是一个整数，而数组的长度也是一个整数，有两种思路：

①hash code值 % table.length会得到一个[0,table.length-1]范围的值，正好是下标范围，但是用%运算，不能保证均匀存放，可能会导致某些table[index]桶中的元素太多，而另一些太少，因此不合适。

②hash code值 & (table.length-1)，因为table.length是2的幂次方，因此table.length-1是一个二进制低位全是1的数，所以&操作完，也会得到一个[0,table.length-1]范围的值。



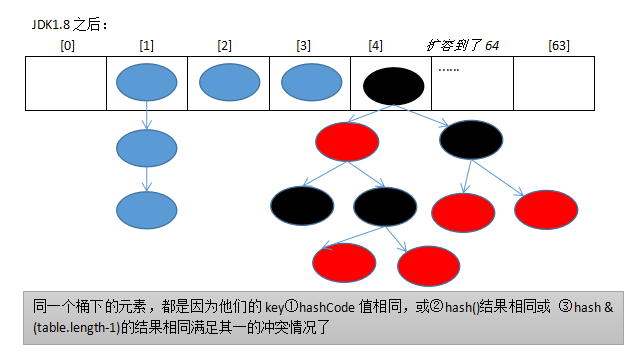
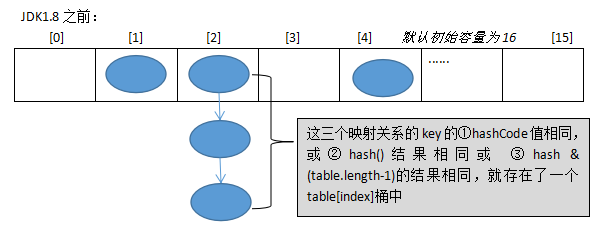
|  |  |
| --- | --- |
| JDK1.7 | **static** **int** indexFor(**int** h, **int** length) {  // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";  **return** **h & (length-1)**;  } |
| JDK1.8 | **final** V putVal(**int** hash, K key, V value, **boolean** onlyIfAbsent,  **boolean** evict) {  Node<K,V>[] tab; Node<K,V> p; **int** n, i;  **if** ((tab = table) == **null** || (n = tab.length) == 0)  n = (tab = resize()).length;  **if** ((p = **tab[i = (n - 1) & hash]**) == **null**)  tab[i] = newNode(hash, key, value, **null**);  //....省略大量代码  } |

#### 4、解决冲突

虽然从设计hashCode()到上面HashMap的hash()函数，都尽量减少冲突，但是仍然存在两个不同的对象返回的hashCode值相同，或者hashCode值就算不同，通过hash()函数计算后，得到的index也会存在大量的相同，因此key分布完全均匀的情况是不存在的。那么发生碰撞冲突时怎么办？

JDK1.8之间使用：数组+链表的结构。

JDK1.8之后使用：数组+链表/红黑树的结构。



#### 6、JDK1.8的put存储过程

（1）几个常量和变量值的作用：

①默认负载因子**static** **final** **float** ***DEFAULT\_LOAD\_FACTOR*** = 0.75f;

②负载因子**final** **float** loadFactor;

③阈值**int** threshold;

当size达到threshold阈值时，会扩容；

④树化阈值**static** **final** **int** ***TREEIFY\_THRESHOLD*** = 8;

该阈值的作用是判断是否需要树化，树化的目的是为了提高查询效率；当某个链表的结点个数达到这个值时，可能会导致树化。

⑤树化最小容量值**static** **final** **int** ***MIN\_TREEIFY\_CAPACITY*** = 64;

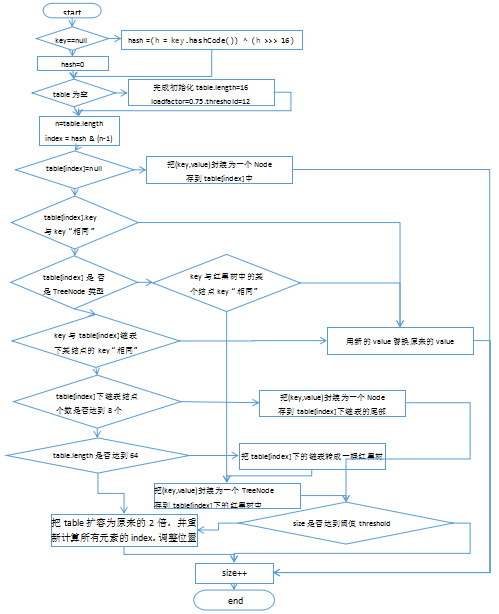
当某个链表的结点个数达到8时，还要检查table的长度是否达到64，如果没有达到，先扩容解决冲突问题

⑥反树化阈值**static** **final** **int** ***UNTREEIFY\_THRESHOLD*** = 6;

当删除了结点时，如果某棵红黑树的结点个数已经低于该值时，会把树重新变成链表，目的是减少复杂度。

（2）存储过程

1. 先计算key的hash值，如果key是null，hash值就是0，如果为null，使用(h = key.hashCode()) ^ (h >>> 16)得到hash值；
2. 如果table是空的，先初始化table数组；
3. 通过hash值计算存储的索引位置index = hash & (table.length-1)
4. 如果table[index]==null，那么直接创建一个Node结点存储到table[index]中即可
5. 如果table[index]!=null，并且table[index]是一个TreeNode结点，说明table[index]下是一棵红黑树，如果该树的某个结点的key与新的key“相同”（hash值相同并且(满足key的地址相同或key的equals返回true)），那么用新的value替换原来的value，否则将(key,value)封装为一个TreeNode结点，连接到红黑树中。
6. 如果table[index]不是一个TreeNode结点，说明table[index]下是一个链表，如果该链表中的某个结点的key与新的key“相同”，那么用新的value替换原来的value，否则需要判断table[index]下结点个数，如果没有达到***TREEIFY\_THRESHOLD(8)***个，那么(key,value)将会封装为一个Node结点直接链接到链表尾部。
7. 如果table[index]下结点个数已经达到***TREEIFY\_THRESHOLD(8)***个，那么再判断table.length是否达到***MIN\_TREEIFY\_CAPACITY(64)***，如果没达到，那么先扩容，扩容会导致所有元素重新计算index，并调整位置；
8. 如果table[index]下结点个数已经达到***TREEIFY\_THRESHOLD(8)***个并table.length也已经达到***MIN\_TREEIFY\_CAPACITY(64)***，那么会将该链表转成一棵自平衡的红黑树，并将结点链接到红黑树中。
9. 如果新增结点而不是替换，那么size++，并且还要重新判断size是否达到threshold阈值，如果达到，还要扩容。



#### 7、关于映射关系的key是否可以修改？

映射关系存储到HashMap中会存储key的hash值，这样就不用在每次查找时重新计算每一个Entry或Node（TreeNode）的hash值了，因此如果已经put到Map中的映射关系，再修改key的属性，而这个属性又参与hashcode值的计算，那么会导致匹配不上。

这个规则也同样适用于LinkedHashMap、HashSet、LinkedHashSet、Hashtable等所有散列存储结构的集合。

|  |  |
| --- | --- |
| JDK1.7 | **transient** Entry<K,V>[] table = (Entry<K,V>[]) ***EMPTY\_TABLE***;  **static** **class** Entry<K,V> **implements** Map.Entry<K,V> {  **final** K key;  V value;  Entry<K,V> next;  **int hash;**  //...省略  } |
| JDK1.8 | **transient** Node<K,V>[] table;  **static** **class** Node<K,V> **implements** Map.Entry<K,V> {  **final int hash;**  **final** K key;  V value;  Node<K,V> next;  //...省略  } |

## 11.7 本章总结

学完了本章内容，你掌握了在开发中最常用的容器——集合的使用方法，也了解各种数据结构的特点，并且还深入源代码分析，我们发现，不管多么复杂的数据结构，其最终的底层物理结构就只有两种：连续存储空间的数组（下标）、非连续存储空间的链表（prev,next）。就算是树（parent,left,right），也是有链表的结构延伸出来的。恭喜你，你的技能指数又提高了2000。

练习

1、请从键盘随机输入10个整数保存到List中，并按倒序、从大到小的顺序显示出来

2、把如下信息添加到Map中，并遍历显示

|  |
| --- |
| 浙江省  绍兴市  温州市  湖州市  嘉兴市  台州市  金华市  舟山市  衢州市  丽水市  海南省  海口市  三亚市  北京市  北京市 |

3、有一个字符串，其中包含中文字符、英文字符和数字字符，请统计和打印出各个字符的字数。

举例说明： String content = “中中国55kkfff”;

统计出：

中：2

国：1

5：2

k：2

f：3