# 第11章 集合

## 1.集合概述

不管是哪一种数据结构，其实本质上都是容器来着，就是用来**装对象**的。

因此，我们就要搞清楚两点：**（1）容器如何存储（2）存储特点**

### 1.1数组

**(1)数组的一些特征：**

* 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，内存就固定了。
* 存储特点：所有数据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。如同9-1-1和9-1-2

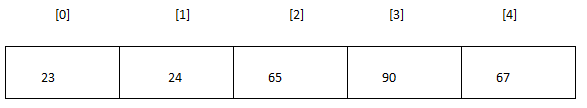


图 9-1- 1

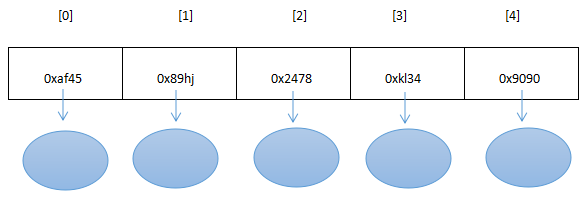


图 9-1- 2

**(2)数组的相关操作：**

* 查询：

1. 按照索引查询，瞬间定位 O(1)
2. 查找某个元素，需要遍历数组 O(n)
3. 查找某个元素，如果是有序数组，可以用二分查找 O(logn)

* 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

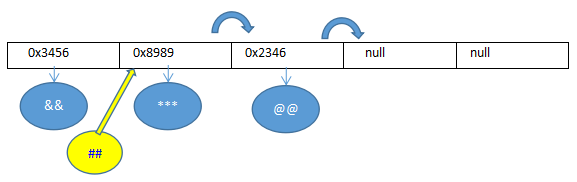


图 9-1- 3

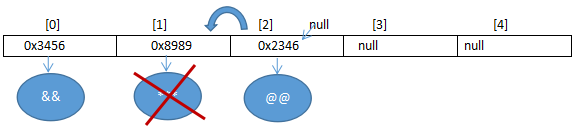


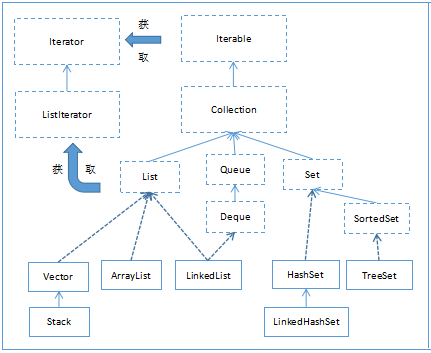
图 9-1- 4

### 1.2集合体系[背]

集合体系就是一系列的接口和类，这些类是可变的聚合的抽象数据类型，专门用于装对象。这些接口和类都在java.util包中，如图所示。

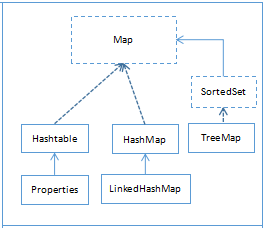
**提前说明：**图中虚线为实现，实线为继承。

**单value型集合体系：**



* Collection 是单值集合体系中的根接口。对于Iterator和Iterable两个接口是提供集合类遍历的作用。
* 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。

**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c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；

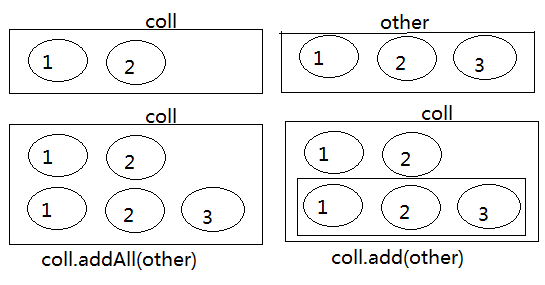
（3） Object[] toArray()：返回包含当前集合中所有元素的数组

#### 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);  // coll.add(other);  System.***out***.println("coll集合元素的个数：" + coll.size());  } |

注意：coll.addAll(other);与coll.add(other);



#### 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。

|  |
| --- |
|  |

* Iterator 仅用于遍历集合，Iterator 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。

|  |
| --- |
| 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！ |

* Iterator接口中提供了两个方法用于遍历集合中的元素：hastNext()、next()

|  |
| --- |
| ArrayList中的Iterator：     1. 私有类，实现iterator接口，实现next()和haseNext(),remove()方法 2. Cursor是游标，指向下一个待消费元素，初始位置为0 3. LasetRet 指向最后一个被返回的元素；初始为-1 4. 当游标<size的时候，先将游标前移，然后lastRet为游标移动之前的位置，返回elementData[lastRet]，也就是说cursor用于探路，lastRet用于消费 5. ExpectedModCount初始化为对应的集合类的modCount，当Iter中的Expected和modCount不一致就报错；因此获取迭代器后，不能再修改集合，会导致二者不一致；     （6）只能用iterator的remove(),此remove是无参的，移除的是lastRet指向的元素，并且注意iterator的remove()维护了ExpectedModCount和modcount保持一致，因此不会出现问题。 |

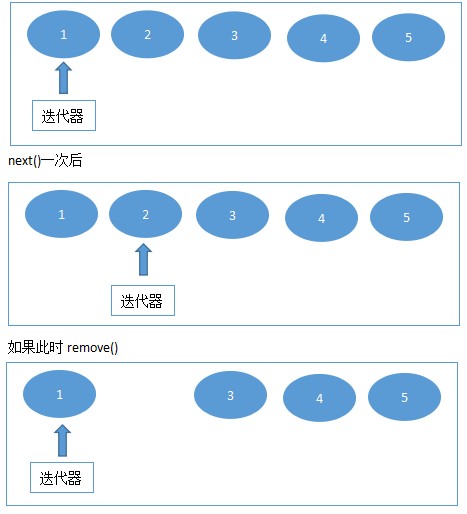
**常用方法：**

（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）。

|  |
| --- |
| 注意：   * 在调用it.next()方法之前必须要调用it.hasNext()进行检测。若不调用，且下一条记录无效，直接调用it.next()会抛出NoSuchElementException异常。 * 如果还未调用next()或在上一次调用 next 方法之后已经调用了 remove 方法，再调用remove都会报IllegalStateException |



代码示例：

|  |
| --- |
| @Test  **public** **void** test7(){  Collection 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"));    Iterator iterator = c.iterator();  **while**(iterator.hasNext()){  Student next = (Student) iterator.next();  //例如：要删除学号为1的学生对象  **if**(next.getId()==1){  iterator.remove();  }  }  } |

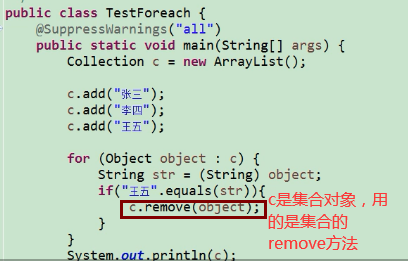
遍历数组：

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

1. 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。
2. 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.**



#### 1、foreach循环遍历

**实现了java.lang.Iterable接口的集合或者容器才能支持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  } |

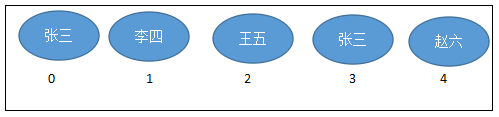
**foreach遍历只适用于查看/查找集合中的元素，不能在遍历集合时有任何影响集合元素个数的操作，**否则报异常或者操作结果将不确定。

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

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。**

## 11.3 List集合

List集合类中元素有序、且可重复。这就像银行门口客服，给每一个来办理业务的客户分配序号：第一个来的是“张三”，客服给他分配的是0；第二个来的是“李四”，客服给他分配的1；以此类推，最后一个序号应该是“总人数-1”。



|  |
| --- |
| 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。例如“张三”可以领取两个号。 |

### 11.3.1 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 ele)**
  + boolean addAll(int index, Collection eles)

2、获取元素

* + **Object get(int index)**
  + List subList(int fromIndex, int toIndex)

3、获取元素索引

* + **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 ele)**

### 11.3.2 List接口的实现类

**Vector 类：动态数组**

**ArrayList类：动态数组**

**Stack类：堆栈**

**LinkedList：双向链表，从JDK1.6之后又实现了双端队列**

#### 1、ArrayList与Vector的区别与联系？

* 底层物理结构：都是数组，二者都是**动态数组。**
* 无参构造初始化容量：
  + Vector初始化为10，而ArrayList在JDK1.6时也是10；
  + 1.6之后，ArrayList初始化容量为空数组；
* 扩容机制：
  + 如果ArrayList一开始初始化为默认的空数组，那么添加第一个元素时，扩容为默认大小为10的数组。此后当容量不够时，ArrayList扩容增加原来的50%；
  + 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，效率高
  + Vector线程安全，效率低。
* Vector因为版本古老，支持Enumeration 迭代器。但是该迭代器不支持快速失败。而Iterator和ListIterator迭代器支持快速失败。如果在迭代器创建后的任意时间从结构上修改了向量（通过迭代器自身的 remove 或 add 方法之外的任何其他方式），则迭代器将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就完全失败，而不是冒着在将来不确定的时间任意发生不确定行为的风险。

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。

#### 2、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）的对象栈。

Stack类通过五个操作对类 Vector 进行了扩展 ，允许将向量视为栈：

Push

Pop

Peek

Empty

在堆栈中查找项并确定到堆栈顶距离的 search 方法。

#### 3、LinkedList类

LinkedList类不仅实现了List接口，还实现了Dequeue接口，提供了一些列方法，因此LinkedList可以通过这些方法将LinkedList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);  }  }  } |

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。

|  |  |
| --- | --- |
| 堆栈方法 | 等效Deque方法 |
| push(e) | addFirst(e) |
| pop() | removeFirst() |
| peek() | peekFirst() |

代码示例二：作栈

|  |
| --- |
| **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());  } |

用作单端队列时，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) | addLast(e) |
| offer(e) | offerLast(e) |
| remove() | removeFirst() |
| poll() | pollFirst() |
| element() | getFirst() |
| peek() | peekFirst() |

代码示例三：单端队列

|  |
| --- |
| **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() |

### 11.3.3 List集合的遍历

#### 1、普通for循环遍历

对于ArrayList和Vector这样的动态数组，也可以使用普通for循环进行遍历。

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));  }  } |

#### 2、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);  }  } |

## 11.4 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。

## 11.5 Set集合

Set接口是Collection的子接口，**set接口没有提供额外的方法。因此Set集合支持的遍历方式和Collection集合一样：foreach和Iterator。**

Set 集合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。

Set的常用实现类有：HashSet、TreeSet、LinkedHashSet。

### 11.5.1 HashSet和LinkedHashSet

* HashSet 是 Set 接口的典型实现，大多数时候使用 Set 集合时都使用这个实现类。
* HashSet 和LinkedHashSet按 Hash 算法来存储集合中的元素，因此存取和查找效率高。
* 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的元素要重写元素的hashCode和equals方法
* LinkedHashSet是HashSet的子类，它在HashSet的基础上，在结点中增加两个属性before和after维护了结点的前后添加顺序。因此LinkedHashSet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。

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 |

* **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

### 11.5.2 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。排序规则自定义
* 定义排序规则有两种方式：
  + 使用元素的自然顺序(Comparable)对元素进行排序；
  + 创建 TreeSet 时提供的 Comparator 进行排序;

1. TreeSet() 无参构造：当用无参构造来创建TreeSet的时候，要求传入的对象必须实现Comparable方法，实现int ComparTo(Object obj)方法
2. TreeSet(Comprator com): 此参数为构造器接口，当用此构造方法来创建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

使用自然排序，意味着添加进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。

代码示例一：

|  |
| --- |
| @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 |

代码示例二：

|  |
| --- |
| @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，不使用equals方法。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，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。

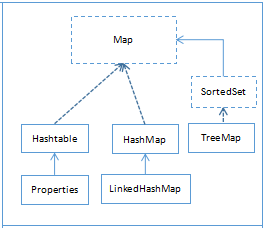
代码示例：

|  |
| --- |
| @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。

## 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
* 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，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。

### 11.5.2 Map的方法

1、添加操作

* Object put(Object key,Object value)
* void putAll(Map t)

1. 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、元视图操作的方法：

* **Set keySet()**
* **Collection values()**
* **Set entrySet()**

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的遍历依赖于Collection的遍历方式：

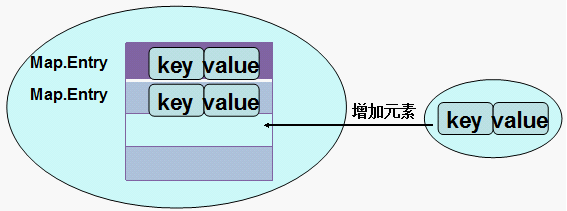
（**1）分开遍历： 单独遍历所有key keySet()**

**单独遍历所有value values()**

**（2）成对遍历：遍历的是映射关系Map.Entry entrySet()**

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：



* 三种遍历Map方式代码示例：

|  |
| --- |
| **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

Map接口的常用实现类：HashMap、TreeMap、LinkedHashMap和Properties。其中HashMap是 Map 接口使用频率最高的实现类。

#### 1、HashMap和Hashtable的区别与联系

* HashMap和Hashtable都是哈希表。
* HashMap和Hashtable判断两个 key 相等的标准：
  + 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。
* 线程安全：
  + Hashtable是线程安全的
  + HashMap是线程不安全的
* null:
  + Hashtable 的key和value不能为null
  + HasmMap 的key和value可以为null

#### 2、LinkedHashMap

LinkedHashMap 是 HashMap 的子类。此实现与 HashMap 的不同之处在于，**后者维护着一个运行于所有条目的双重链接列表。**此链接列表定义了迭代顺序，该迭代顺序通常就是将键插入到映射中的顺序（插入顺序）。

#### 3、TreeMap

基于红黑树（Red-Black tree）的 NavigableMap 实现。该映射根据其键的自然顺序进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。

#### 4、Properties

Properties 类是 Hashtable 的子类，Properties 可保存在流中或从流中加载。**属性列表中每个键及其对应值都是一个字符串。**

存取数据时，建议使用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 深入源码分析

### 11.6.1 Set的源码分析

* Set的内部实现其实是一个Map。
  + HashSet的内部实现是一个HashMap
  + TreeSet的内部实现是一个TreeMap
  + LinkedHashSet的内部实现是一个LinkedHashMap

#### HashSet部分源代码摘要：

1.HashSet底层存储结构： value为static final 的Object对象 的HashMap

|  |
| --- |
| private transient HashMap<E,Object> map;  private static final Object *PRESENT* = new Object(); |

1. HashSet的构造器：

|  |
| --- |
| public HashSet() {  map = new HashMap<>(); }  public HashSet(Collection<? extends E> c) {  map = new HashMap<>(Math.*max*((int) (c.size()/.75f) + 1, 16));  addAll(c); }  public HashSet(int initialCapacity, float loadFactor) {  map = new HashMap<>(initialCapacity, loadFactor); }  public HashSet(int initialCapacity) {  map = new HashMap<>(initialCapacity); } |

1. HashSet 增加元素

|  |
| --- |
| public boolean add(E e) {  return map.put(e, *PRESENT*)==null; } |

1. HashSet的迭代器

|  |
| --- |
| public Iterator<E> iterator() {  return map.keySet().iterator(); } |

结论：

1. HashSet其实就是value为Object对象的HashMap
2. Value是Object静态常量

#### LinkedHashSet部分源码摘要

* LinkedHashSet是HashSet子类

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

* LinkedHashSet底层存储结构

|  |
| --- |
| public LinkedHashSet(int initialCapacity, float loadFactor) {  super(initialCapacity, loadFactor, true); } public LinkedHashSet(int initialCapacity) {  super(initialCapacity, .75f, true); } public LinkedHashSet() {  super(16, .75f, true); } public LinkedHashSet(Collection<? extends E> c) {  super(Math.*max*(2\*c.size(), 11), .75f, true);  addAll(c); } |
| 上面三个LinkedHashSet的构造器实际上调用的都是下面这个：  HashSet(int initialCapacity, float loadFactor, boolean dummy) {  map = new LinkedHashMap<>(initialCapacity, loadFactor); } |

结论：LinkedHashset底层是LinkedHashMap

#### TreeSet

关注两个构造器即可

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

### 11.6.2 ArrayList的源码分析

以下源码分析基于JDK1.8源码：

#### 无参构造

|  |
| --- |
| **transient** Object[] elementData; //实际存储元素的数组  **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***;  } |

JDK1.6中，ArrayList初始化容量为10

JDK1.7，初始化容量为EMPTY\_ELEMENTDATA 为0

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

**记住：jdk1.6之前初始化为10，之后为0**

#### 添加元素

add(Object obj)

|  |
| --- |
| **private** **static** **final** **int** ***DEFAULT\_CAPACITY*** = 10;//默认容量  **public** **boolean** add(E e) {  //确保当前数组的容量是够得  ensureCapacityInternal(size + 1); // Increments modCount!!  //将新元素添加到[size++]的位置  elementData[size++] = e;  **return** **true**;  }  **private** **void** ensureCapacityInternal(**int** minCapacity) {  //如果是第一次添加  **if** (elementData == ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***) {  //扩容为默认容量大小：10  minCapacity = Math.*max*(***DEFAULT\_CAPACITY***, minCapacity);  }    //每一次添加都要判断是否需要扩容  ensureExplicitCapacity(minCapacity);  } |

每一次添加元素之前都要确保容量足够：ensureCapacityInternal(size+1)

如果是第一次添加，就让容量变为10

如果size + 1大于现有的数组长度，,就进行扩容

扩容为1.5倍

#### 扩容

|  |
| --- |
| **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);  } |

#### 4、结论

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，当存储容量不够时，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。

### 11.6.4 LinkedList的源码分析

以下源码分析基于JDK1.8源码：

#### 1、初始化

|  |
| --- |
| **public** LinkedList() {  } |

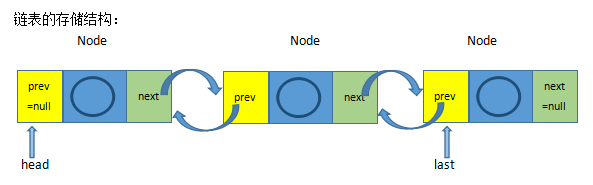
并未开辟任何类似于数组一样的存储空间，那么链表是如何存储元素的呢？

#### 2、内部类Node

**1.LinkedList内部维护了两个属性first和last指向首尾节点**

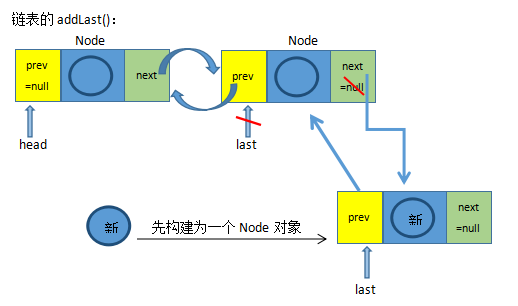
**2.Node类型是一个双向链表，前后指针都有**

|  |
| --- |
| **transient** Node<E> first;//指向链表的第一个结点  **transient** Node<E> last;//指向链表的最后一个结点    //LinkedList中有一个内部类Node类型  **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;  }  } |



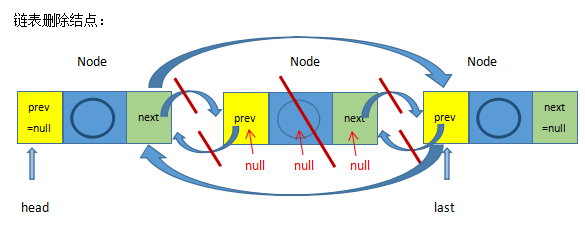
#### 3、其中一个添加元素的方法：add(E e)

|  |
| --- |
| **public** **boolean** add(E e) {  //默认链接到链表末尾  linkLast(e);  **return** **true**;  }  **void** linkLast(E e) {  //用l记录当前链表的最后一个结点对象  **final** Node<E> l = last;  //创建一个新结点对象，并且指定当前新结点的前一个结点为l  **final** Node<E> newNode = **new** Node<>(l, e, **null**);  //当前新结点就变成了链表的最后一个结点  last = newNode;  **if** (l == **null**)  //如果当前链表是空的，那么新结点对象，同时也是链表的第一个结点  first = newNode;  **else**  //如果当前链表不是空的，那么最后一个结点的next就指向当前新结点  l.next = newNode;  //元素个数增加  size++;  //修改次数增加  modCount++;  } |



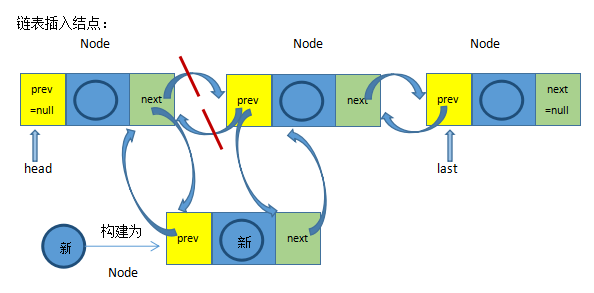
#### 4、其中一个删除方法：remove（Object obj）

|  |
| --- |
| **public** **boolean** remove(Object o) {  //分o是否是null讨论，从头到尾找到要删除的元素o对应的Node结点对象，然后删除  **if** (o == **null**) {  **for** (Node<E> x = first; x != **null**; x = x.next) {  **if** (x.item == **null**) {  unlink(x);  **return** **true**;  }  }  } **else** {  **for** (Node<E> x = first; x != **null**; x = x.next) {  **if** (o.equals(x.item)) {  unlink(x);  **return** **true**;  }  }  }  **return** **false**;  }  E unlink(Node<E> x) {  **final** E element = x.item;  //用next记录被删除结点的后一个结点  **final** Node<E> next = x.next;  //用prev记录被删除结点的前一个结点  **final** Node<E> prev = x.prev;    **if** (prev == **null**) {  //如果删除的是第一个结点，那么被删除的结点的后一个结点将成为第一个结点  first = next;  } **else** {  //否则被删除结点的前一个结点的next应该指向被删除结点的后一个结点  prev.next = next;  //断开被删除结点与前一个结点的关系  x.prev = **null**;  }  **if** (next == **null**) {  //如果删除的是最后一个结点，那么被删除结点的前一个结点将变成最后一个结点  last = prev;  } **else** {  //否则被删除结点的后一个结点的prev应该指向被删除结点的额前一个结点  next.prev = prev;  //断开被删除结点与后一个结点的关系  x.next = **null**;  }  //彻底把被删除结点变成垃圾对象  x.item = **null**;  //元素个数减少  size--;  //修改次数增加  modCount++;  **return** element;  } |



#### 5、指定位置插入元素

|  |
| --- |
| **public** **void** add(**int** index, E element) {  **//检查索引位置的合理性**  checkPositionIndex(index);  **if** (index == size)  **//如果位置是在最后，那么链接到链表的最后**  linkLast(element);  **else**  //否则在链表中间插入  //node(index)表示找到index位置的Node对象  linkBefore(element, node(index));  }  **void** linkBefore(E e, Node<E> succ) {  // pred记录被插入位置的前一个结点  **final** Node<E> pred = succ.prev;  //构建一个新结点  **final** Node<E> newNode = **new** Node<>(pred, e, succ);  //把新结点插入到succ的前面  succ.prev = newNode;  //如果被插入点是链表的开头，那么新结点变成了链表头  **if** (pred == **null**)  first = newNode;  **else**  //否则pred的next就变成了新结点  pred.next = newNode;  //元素个数增加  size++;  //修改次数增加  modCount++;  } |



#### 6、结论

LinkedList适用于频繁的插入、删除，效率较高；因为不涉及到移动元素，也不涉及到扩容。查询效率低，按索引查询和按照元素查询都只能遍历查询

ArrayList适用于查询，如果需要按索引操作，那么建议使用动态数组。

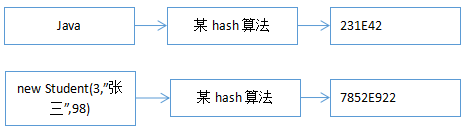
### 11.6.5 HashMap的源码分析

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。

#### 1、hashCode值

hashCode值在HashMap中的作用为求索引index；

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。

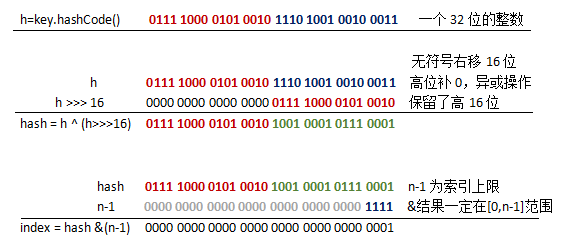


#### 2、HashMap中的散列函数hash()

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提高hash code值与 (table.length-1)的按位与完的结果，尽量的均匀分布。

|  |  |
| --- | --- |
| 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：



* JDK1.8的hash值的算法为什么要保留高16位呢？

首先index的求解方式 index = hash & (n-1),而一个HashMap的table数组在不断扩容之前一般不会特别大，那么table.length-1的大部分高位都是0。此时如果直接用hashCode和table.length-1进行&运算的话，就会导致hashCode总是只有最低的几位是参与index的计算的，那么就算你的hashCode()实现的再好也难以避免发生index碰撞，这时保留高16位的意义就体现出来了。 Hash=h^(h>>>16) 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。

#### 3、HashMap存储结构

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  } |

Entry是一个单向链表节点。

##### （2）初始容量：16

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

##### （3）扩容为原来的2倍

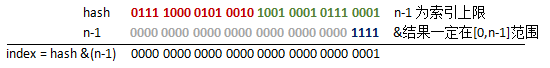
|  |  |
| --- | --- |
| JDK1.7 | **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {  **if** ((size >= threshold) && (**null** != table[bucketIndex])) {  **resize(2 \* table.length);**  hash = (**null** != key) ? hash(key) : 0;  bucketIndex = *indexFor*(hash, table.length);  }  createEntry(hash, 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）那么HashMap是如何决定某个映射关系存在哪个桶的呢？

首先要求出hashCode，然后利用hashCode根据一定的算法，确定数据放在table数组的index

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 % 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 & (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。 |



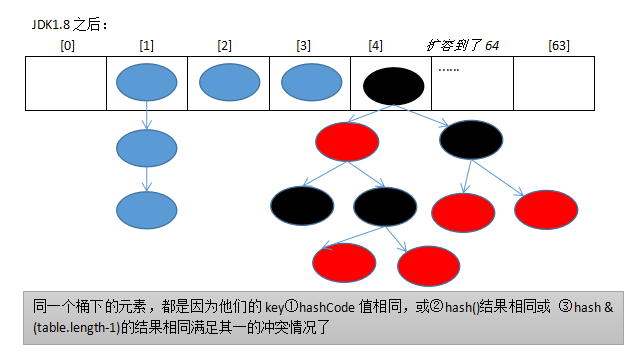
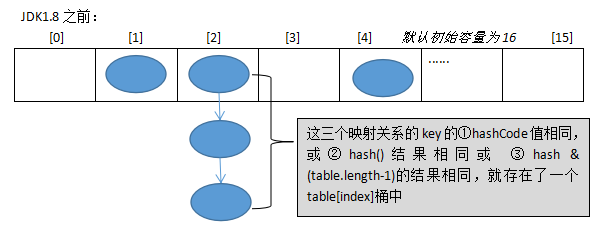
|  |  |
| --- | --- |
| JDK1.7 | **static** **int** indexFor(**int** h, **int** length) {  // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";  **return** **h & (length-1)**;  } |
| JDK1.8 | **final** V putVal(**int** hash, K key, V value, **boolean** onlyIfAbsent,  **boolean** 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、hash冲突的解决

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之前使用：数组+链表的结构 解决hash冲突的问题

JDK1.8之后使用：数组+链表/红黑树的结构 解决hash冲突的问题



#### 5、JDK1.7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;

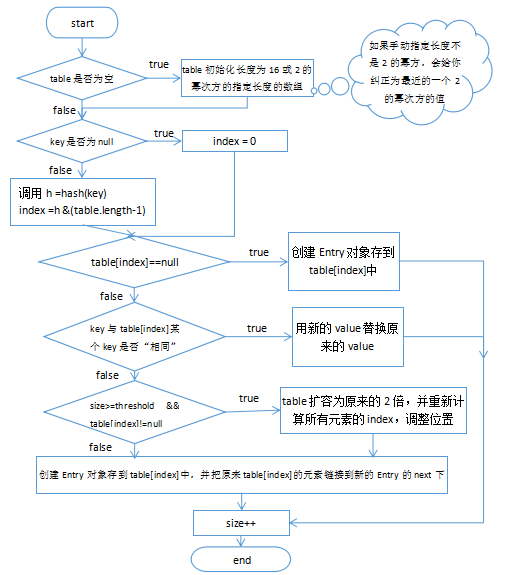
③阈值

**int** threshold;

|  |
| --- |
| threshold = (**int**) Math.*min*(capacity \* loadFactor, ***MAXIMUM\_CAPACITY*** + 1); |
| **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {  **if ((size >= threshold) && (null != table[bucketIndex])) {**  **resize(2 \* table.length); //扩容**  hash = (**null** != key) ? hash(key) : 0;  bucketIndex = *indexFor*(hash, table.length);  }  createEntry(hash, key, value, bucketIndex);  } |

（2）存储过程

1. 先判断table是否为空数组，如果是，先初始化数组，长度为16；
2. 判断key是null，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，决定是在table[index]，index在[0,table.length-1]范围内；
3. 判断table[index]桶下是否存在某个Entry的key与新的key的“相同”（hash值相同并且(满足key的地址相同或key的equals返回true)），如果是，用新的value替换原来的value；
4. 如果不存在，判断是否满足size达到阈值(threshold)并且table[index]不是null，如果是，先扩容；扩容会导致原来table中的所有元素都会重新计算位置，并调整存储位置；
5. 添加一个新的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。
6. size++，元素个数增加



#### 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;

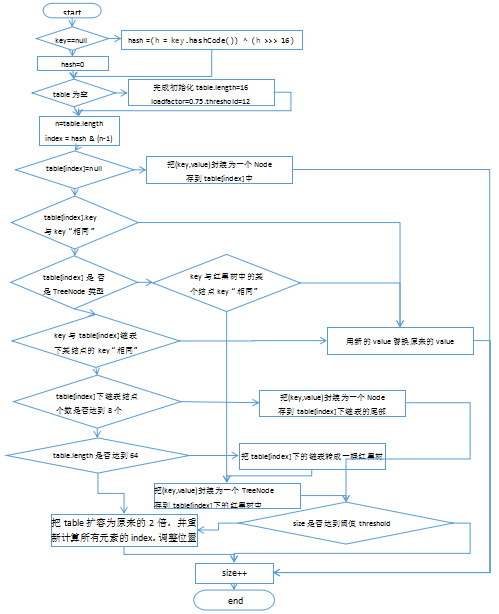
当某个链表的结点个数达到8时，还要检查table的长度是否达到64，如果没有达到，先扩容解决冲突问题

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