# 第12章 集合

## 12.1 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）如何存储（2）存储特点

### 12.1.1数组

* 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

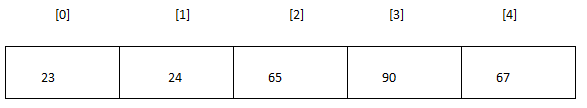


图 9-1- 1

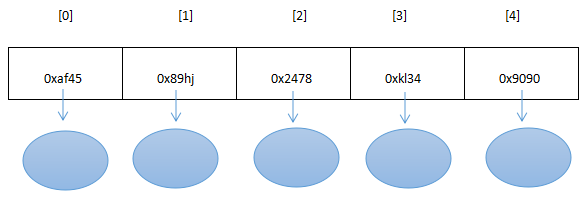


图 9-1- 2

* 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

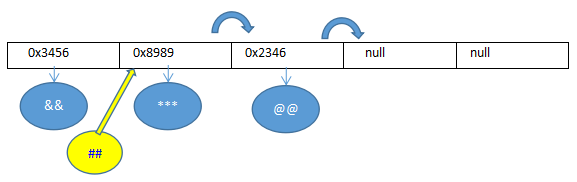


图 9-1- 3

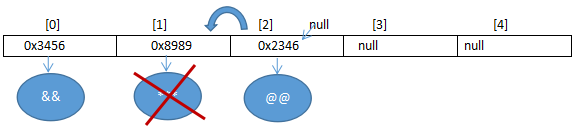


图 9-1- 4

### 12.1.2集合

为了可以满足用户数据更多种的逻辑关系，而设计的一系列的不同于数组的可变的聚合的抽象数据类型。这些接口和类在java.util包中，因为类型很丰富，因此我们通常称为集合框架集。如图9-1-5和9-1-6

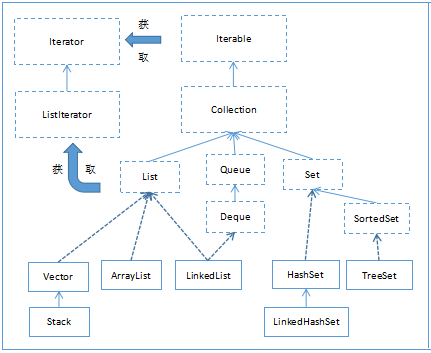


图 9-1- 5

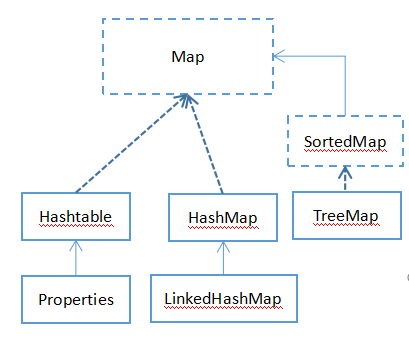


图 9-1- 6

* Collection 层次结构中的根接口。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：有序的 collection（也称为序列）。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。用户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：队列通常（但并非一定）以 FIFO（先进先出）的方式排序各个元素。不过优先级队列和 LIFO 队列（或堆栈）例外，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：一个不包含重复元素的 collection。更确切地讲，set 不包含满足 e1.equals(e2) 的元素对 e1 和 e2，并且最多包含一个 null 元素。正如其名称所暗示的，此接口模仿了数学上的 set 抽象。
    - 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。
* 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。

## 12.2 Collection集合

### 12.2.1 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、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；

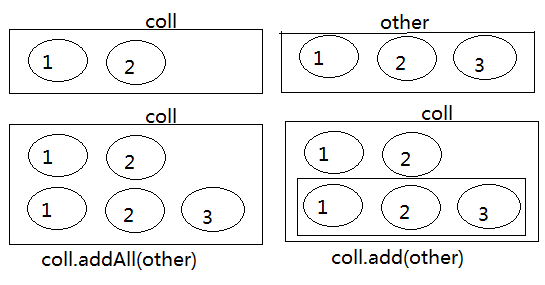
（3） Object[] toArray()：返回包含当前集合中所有元素的数组

#### 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  } |

### 12.2.2 Collection集合的遍历

#### 1、foreach循环遍历

Java 5时Collection接口继承了java.lang.Iterable接口，因此Collection系列的集合就可以直接使用foreach循环遍历。格式：

|  |
| --- |
| 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。

#### 2、Iterator迭代器遍历

因为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，用以返回一个实现了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。如果需要创建 Iterator 对象，则必须有一个被迭代的集合。
* 集合对象每次调用iterator()方法都得到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。

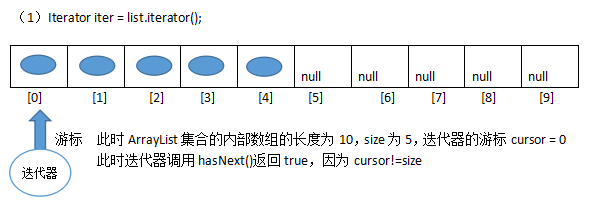
常用方法：

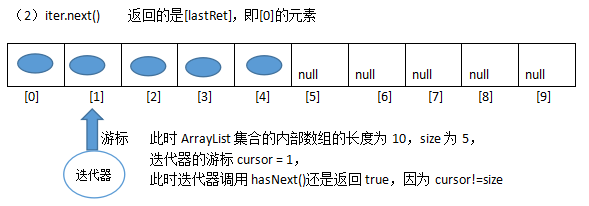
（1）boolean hasNext()：如果仍有元素可以迭代，则返回 true

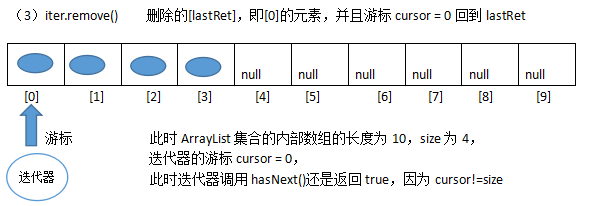
（2）Object next()：返回迭代的下一个元素。

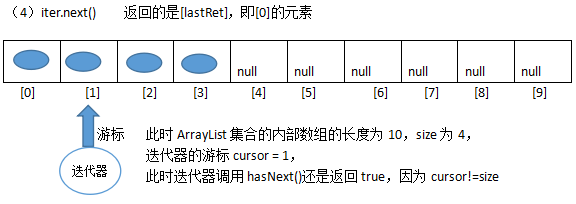
（3）void remove()：从迭代器指向的 collection 中移除迭代器返回的最后一个元素（可选操作）。

|  |
| --- |
| 注意：   * 建议在调用it.next()方法之前调用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 |







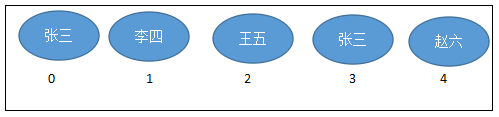


代码示例：

|  |
| --- |
| @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();  }  }  } |

## 12.3 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。 |

### 12.3.1 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)

示例代码：

|  |
| --- |
| @SuppressWarnings("all")  @Test  **public** **void** test1(){  ArrayList list = **new** ArrayList();  list.add("张三");  list.add(0,"李四");//把“李四”添加到[0]位置  list.add(1,"王五");//把“王五”添加到[1]位置  //结果[李四、王五、张三]  **for** (Object object : list) {  System.***out***.println(object);  }  } |
| @SuppressWarnings("all")  @Test  **public** **void** test2(){  ArrayList list = **new** ArrayList();  list.add("张三");  list.add("李四");  list.add("王五");    Object object = list.get(1);//返回[1]的元素  System.***out***.println(object);//李四  } |
| @SuppressWarnings("all")  @Test  **public** **void** test3(){  ArrayList list = **new** ArrayList();  list.add("张三");  list.add("李四");  list.add("王五");  list.add("李四");    //从[0]开始查找，返回第一次出现的"李四"的索引  **int** index = list.indexOf("李四");  System.***out***.println("index = " + index);//1  } |
| @SuppressWarnings("all")  @Test  **public** **void** test4(){  ArrayList list = **new** ArrayList();  list.add("张三");  list.add("李四");  list.add("王五");    list.remove(0);//删除的是[0]的元素，即删除“张三”    **for** (Object object : list) {  System.***out***.println(object);  }  } |

### 12.3.2 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？

它们的底层物理结构都是数组，我们称为动态数组。

* ArrayList是新版的动态数组，线程不安全，效率高，Vector是旧版的动态数组，线程安全，效率低。
* 动态数组的扩容机制不同，ArrayList扩容为原来的1.5倍，Vector扩容增加为原来的2倍。
* 数组的初始化容量，如果在构建ArrayList与Vector的集合对象时，没有显式指定初始化容量，那么Vector的内部数组的初始容量默认为10，而ArrayList在JDK1.6及之前的版本也是10，而JDK1.7之后的版本ArrayList初始化为长度为0的空数组，之后在添加第一个元素时，再创建长度为10的数组。
* 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。

#### 3、LinkedList类

除了实现 List 接口外，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);  }  }  } |

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：

|  |
| --- |
| **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：

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

### 12.3.3 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，该方法返回一个 ListIterator 对象， ListIterator 接口继承了 Iterator 接口，提供了专门操作 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);  }  } |

### 12.3.4 List案例

案例需求：使用List的实现类添加三本图书，并遍历，打印如下效果

书名：xx ，价格：xx，作者：xx

书名：xx ，价格：xx，作者：xx

完整代码如下：

|  |
| --- |
| **package** com.atguigu.part02.list;  **public** **class** Book {  **private** String title;  **private** **double** price;  **private** String author;  **public** Book(String title, **double** price, String author) {  **super**();  **this**.title = title;  **this**.price = price;  **this**.author = author;  }  **public** Book() {  **super**();  }  **public** String getTitle() {  **return** title;  }  **public** **void** setTitle(String title) {  **this**.title = title;  }  **public** **double** getPrice() {  **return** price;  }  **public** **void** setPrice(**double** price) {  **this**.price = price;  }  **public** String getAuthor() {  **return** author;  }  **public** **void** setAuthor(String author) {  **this**.author = author;  }  @Override  **public** String toString() {  **return** "书名：" + title + ", 价格：" + price + ", 作者：" + author;  }  } |
| **package** com.atguigu.part02.list;  **import** java.util.ArrayList;  **public** **class** TestListDemo {  @SuppressWarnings("all")  **public** **static** **void** main(String[] args) {  ArrayList list = **new** ArrayList();  list.add(**new** Book("《来尚硅谷学Java》",88.8,"宋红康"));  list.add(**new** Book("《来尚硅谷学BigData》",99.9,"缪传海"));  list.add(**new** Book("《来尚硅谷学H5》",66.6,"张晓飞"));    **for** (Object object : list) {  System.***out***.println(object);  }  }  } |

## 12.4 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。

### 12.4.1 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，因此具有很好的存取和查找性能。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。

示例代码：定义一个Employee类，该类包含属性：name, birthday，其中 birthday 为 MyDate类的对象；MyDate为自定义类型，包含年、月、日属性。要求 name和birthday一样的视为同一个员工。

|  |
| --- |
| **package** com.atguigu.part03.set;  **public** **class** Employee {  **private** String name;  **private** MyDate birthday;  **public** Employee(String name, MyDate birthday) {  **super**();  **this**.name = name;  **this**.birthday = birthday;  }  **public** Employee() {  **super**();  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** MyDate getBirthday() {  **return** birthday;  }  **public** **void** setBirthday(MyDate birthday) {  **this**.birthday = birthday;  }  @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + ((birthday == **null**) ? 0 : birthday.hashCode());  result = prime \* result + ((name == **null**) ? 0 : name.hashCode());  **return** result;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj)  **return** **true**;  **if** (obj == **null**)  **return** **false**;  **if** (getClass() != obj.getClass())  **return** **false**;  Employee other = (Employee) obj;  **if** (birthday == **null**) {  **if** (other.birthday != **null**)  **return** **false**;  } **else** **if** (!birthday.equals(other.birthday))  **return** **false**;  **if** (name == **null**) {  **if** (other.name != **null**)  **return** **false**;  } **else** **if** (!name.equals(other.name))  **return** **false**;  **return** **true**;  }  @Override  **public** String toString() {  **return** "姓名：" + name + ", 生日：" + birthday;  }  } |
| **package** com.atguigu.part03.set;  **public** **class** MyDate {  **private** **int** year;  **private** **int** month;  **private** **int** day;  **public** MyDate(**int** year, **int** month, **int** day) {  **super**();  **this**.year = year;  **this**.month = month;  **this**.day = day;  }  **public** MyDate() {  **super**();  }  **public** **int** getYear() {  **return** year;  }  **public** **void** setYear(**int** year) {  **this**.year = year;  }  **public** **int** getMonth() {  **return** month;  }  **public** **void** setMonth(**int** month) {  **this**.month = month;  }  **public** **int** getDay() {  **return** day;  }  **public** **void** setDay(**int** day) {  **this**.day = day;  }  @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + day;  result = prime \* result + month;  result = prime \* result + year;  **return** result;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj)  **return** **true**;  **if** (obj == **null**)  **return** **false**;  **if** (getClass() != obj.getClass())  **return** **false**;  MyDate other = (MyDate) obj;  **if** (day != other.day)  **return** **false**;  **if** (month != other.month)  **return** **false**;  **if** (year != other.year)  **return** **false**;  **return** **true**;  }  @Override  **public** String toString() {  **return** year + "-" + month + "-" + day;  }  } |
| **package** com.atguigu.part03.set;  **import** java.util.HashSet;  **public** **class** TestHashSet {  @SuppressWarnings("all")  **public** **static** **void** main(String[] args) {  HashSet set = **new** HashSet();  set.add(**new** Employee("张三", **new** MyDate(1990,1,1)));  //重复元素无法添加，因为MyDate和Employee重写了hashCode和equals方法  set.add(**new** Employee("张三", **new** MyDate(1990,1,1)));  set.add(**new** Employee("李四", **new** MyDate(1992,2,2)));    **for** (Object object : set) {  System.***out***.println(object);  }  }  } |

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。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

### 12.4.2 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。

使用元素的自然顺序(Comparable)对元素进行排序，或者根据创建 set 时提供的 Comparator 进行排序，具体取决于使用的构造方法。

#### 1、自然排序

如果试图把一个对象添加到 TreeSet 时，则该对象的类必须实现 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：按照字符串Unicode编码值排序

|  |
| --- |
| @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：按照学生的学号排序

|  |
| --- |
| @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：学生类型未实现Comparable接口，单独指定Comparator比较器，按照学生的学号排序

|  |
| --- |
| @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。

## 12.5 Map集合

### 12.5.1 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。

### 12.5.2 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()

### 12.5.3 Map的遍历

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

Map的遍历，不能支持foreach

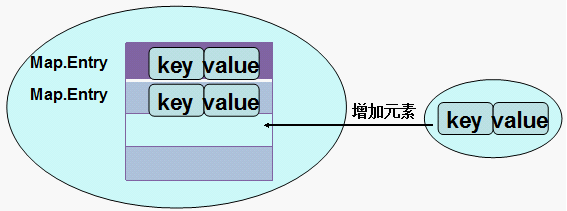
（1）分开遍历：单独遍历所有key

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);  }  }  } |

### 12.5.4 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。

Hashtable是线程安全的，任何非 null 对象都可以用作键或值。

HashMap是线程不安全的，并允许使用 null 值和 null 键。

示例代码：添加员工姓名为key，薪资为value

|  |
| --- |
| **package** com.atguigu.part04.map;  **import** java.util.HashMap;  **import** java.util.Set;  **public** **class** TestHashMap {  @SuppressWarnings("all")  **public** **static** **void** main(String[] args) {  HashMap map = **new** HashMap();  map.put("张三", 10000);  //key相同，新的value会覆盖原来的value  //因为String重写了hashCode和equals方法  map.put("张三", 12000);  map.put("李四", 14000);  //HashMap支持key和value为null值  String name = **null**;  Double salary = **null**;  map.put(name, salary);    Set entrySet = map.entrySet();  **for** (Object object : entrySet) {  System.***out***.println(object);  }  }  } |

#### 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）。

示例代码：添加员工姓名为key，薪资为value

|  |
| --- |
| **package** com.atguigu.part04.map;  **import** java.util.LinkedHashMap;  **import** java.util.Set;  **public** **class** TestLinkedHashMap {  @SuppressWarnings("all")  **public** **static** **void** main(String[] args) {  LinkedHashMap map = **new** LinkedHashMap();  map.put("张三", 10000);  //key相同，新的value会覆盖原来的value，这点和HashMap相同  //因为String重写了hashCode和equals方法  map.put("张三", 12000);  map.put("李四", 14000);    //遍历发现，可以体现添加顺序，这点和HashMap不同  Set entrySet = map.entrySet();  **for** (Object object : entrySet) {  System.***out***.println(object);  }  }  } |

#### 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。

代码示例：添加员工姓名为key，薪资为value

|  |
| --- |
| **package** com.atguigu.part04.map;  **import** java.util.Comparator;  **import** java.util.Set;  **import** java.util.TreeMap;  **import** org.junit.Test;  **public** **class** TestTreeMap {  @SuppressWarnings("all")  @Test  **public** **void** test1() {  TreeMap map = **new** TreeMap();  map.put("Jack", 11000);  map.put("Alice", 12000);  map.put("zhangsan", 13000);  map.put("baitao", 14000);  map.put("Lucy", 15000);    //String实现了Comparable接口，默认按照Unicode编码值排序  Set entrySet = map.entrySet();  **for** (Object object : entrySet) {  System.***out***.println(object);  }  }  @SuppressWarnings("all")  @Test  **public** **void** test2() {  //指定定制比较器Comparator，按照Unicode编码值排序，但是忽略大小写  TreeMap map = **new** TreeMap(**new** Comparator() {  @Override  **public** **int** compare(Object o1, Object o2) {  String s1 = (String) o1;  String s2 = (String) o2;  **return** s1.compareToIgnoreCase(s2);  }  });  map.put("Jack", 11000);  map.put("Alice", 12000);  map.put("zhangsan", 13000);  map.put("baitao", 14000);  map.put("Lucy", 15000);    Set entrySet = map.entrySet();  **for** (Object object : entrySet) {  System.***out***.println(object);  }  }  } |

#### 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);  } |

## 12.6 深入源码分析

### 12.6.1 Set的源码分析

Set的内部实现其实是一个Map。即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。

部分源代码摘要：

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

但是，咱们存到Set中只有一个元素，又是怎么变成(key,value)的呢？

以HashSet为例：

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

原来是，把添加到Set中的元素作为内部实现map的key，然后用一个常量对象PRESENT对象，作为value。

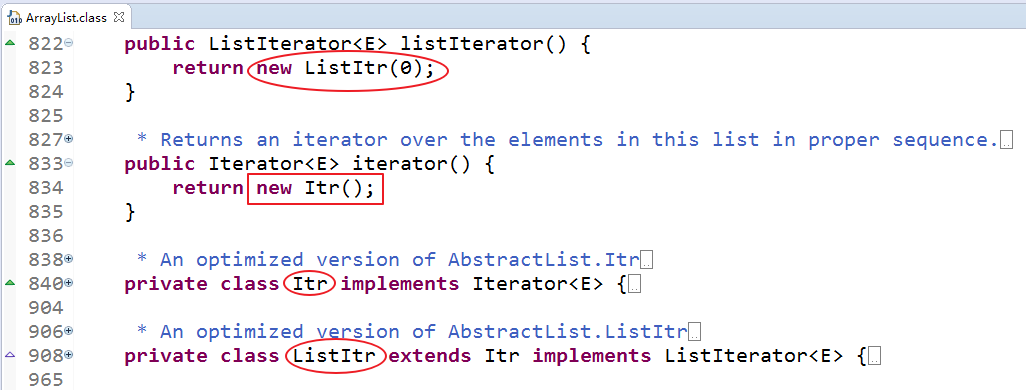
这是因为Set的元素不可重复和Map的key不可重复有相同特点。Map有一个方法keySet()可以返回所有key。

### 12.6.2 Iterator的源码分析

概念：迭代器（iterator）有时又称为游标（cursor），提供一种方法访问一个容器（container）对象中各个元素，而又不需要暴露该对象的内部细节。

作用：是为容器遍历而生，用以方便的实现对容器内元素的遍历操作。类似于“公交车上的售票员”、“飞机上的空姐”，他们只为关注当前“容器”中的乘客，可以“访问”每一位乘客。例如：公交车上的售票员走到每个座位之前，看看是否有乘客，如果有检查该乘客，如果该乘客未购票，可以请他下车，也只能请刚刚检查过的乘客下车。每一类集合都有自己的迭代器，迭代器是为对应的集合服务的，因此迭代器都是作为对应集合的内部类定义的。

示例：只摘取关键代码ArrayList的内部迭代器Itr。



### 12.6.3 ArrayList的源码分析

以下源码分析基于JDK1.8源码：

#### 初始化

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

#### 添加元素

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

#### 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。

### 12.6.4 LinkedList的源码分析

以下源码分析基于JDK1.8源码：

#### 1、初始化

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

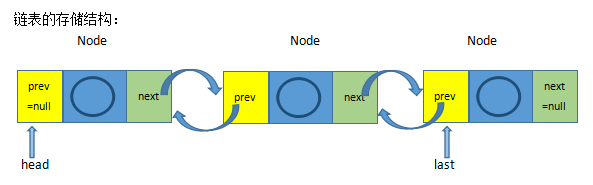
并未开辟任何类似于数组一样的存储空间，那么链表是如何存储元素的呢？

#### 2、Node类型

存储到链表中的元素会被封装为一个Node类型的结点。并且链表只需记录第一个结点的位置和最后一个结点的位置。然后每一个结点，前后连接，就可以串起来变成一整个链表。

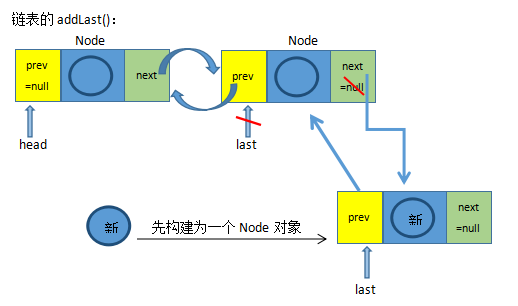


|  |
| --- |
| **transient** Node<E> first;//指向链表的第一个结点  **transient** Node<E> last;//指向链表的最后一个结点    //LinkedList中有一个内部类Node类型  **private** **static** **class** Node<E> {  E item;  Node<E> next;  Node<E> prev;  Node(Node<E> prev, E element, Node<E> next) {  **this**.item = element;  **this**.next = next;  **this**.prev = prev;  }  } |



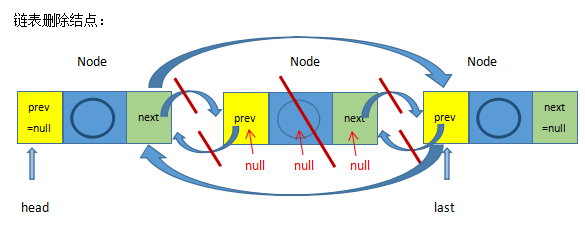
#### 3、其中一个添加元素的方法：add(E e)

|  |
| --- |
| **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++;  } |



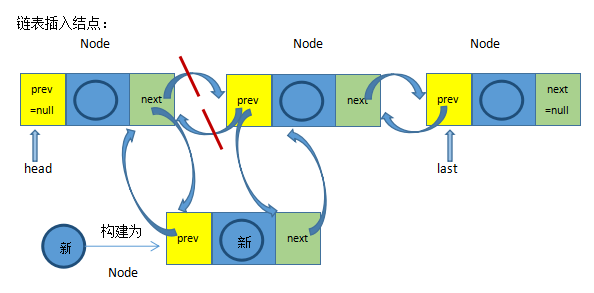
#### 4、其中一个删除方法：remove（Object obj）

|  |
| --- |
| **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

对于频繁的插入或删除元素的操作，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。

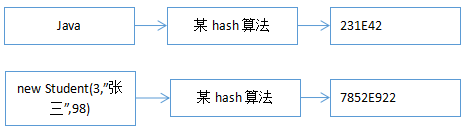
此类虽然提供按照索引查找与操作的方法，但是效率不高，如果需要按索引操作，那么建议使用动态数组。

### 12.6.5 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

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

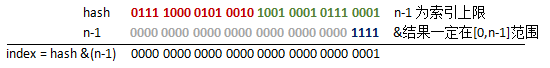
|  |  |
| --- | --- |
| JDK1.7 | **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {  **if** ((size >= threshold) && (**null** != table[bucketIndex])) {  **resize(2 \* table.length);**  hash = (**null** != key) ? hash(key) : 0;  bucketIndex = *indexFor*(hash, table.length);  }  createEntry(hash,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

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。



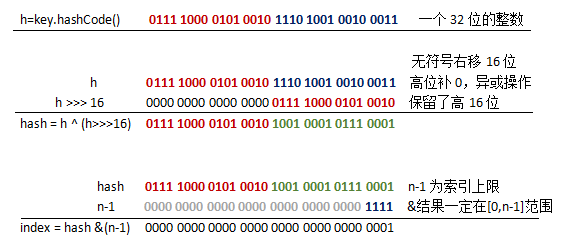
|  |  |
| --- | --- |
| JDK1.7 | **static** **int** indexFor(**int** h, **int** length) {  // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";  **return** **h & (length-1)**;  } |
| JDK1.8 | **final** V putVal(**int** hash, K key, V value, **boolean** onlyIfAbsent,  **boolean**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

#### 3、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：



那么，JDK1.8为什么要保留高16位呢？

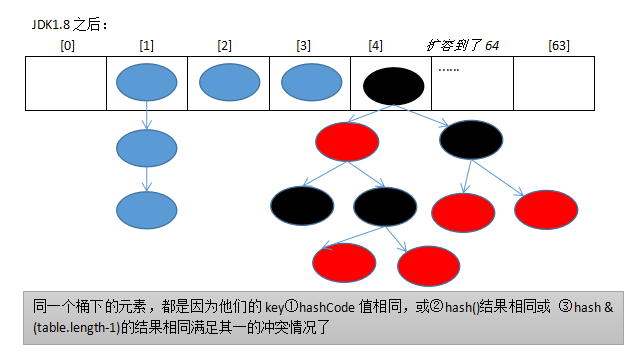
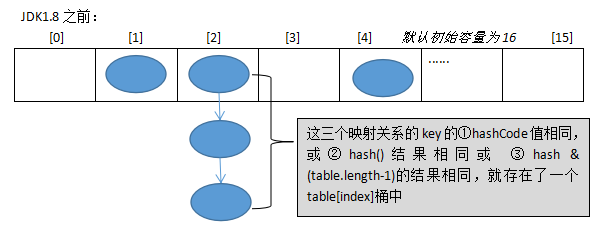
因为一个HashMap的table数组一般不会特别大，至少在不断扩容之前，那么table.length-1的大部分高位都是0，直接用hashCode和table.length-1进行&运算的话，就会导致总是只有最低的几位是有效的，那么就算你的hashCode()实现的再好也难以避免发生碰撞，这时保留高16位的意义就体现出来了。它对hashcode的低位添加了随机性并且混合了高位的部分特征，显著减少了碰撞冲突的发生。

#### 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。



#### 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;

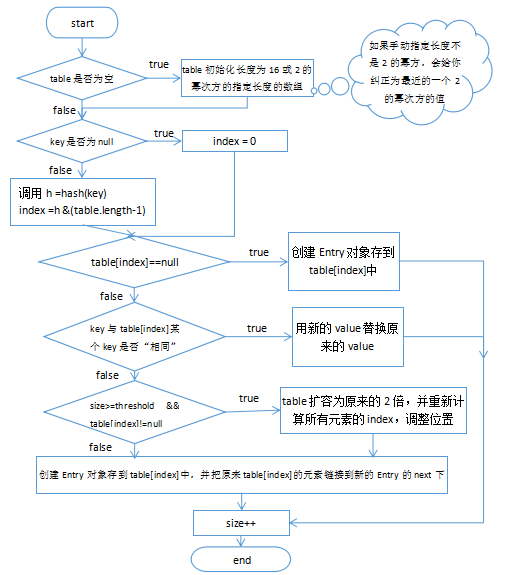
③阈值

**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;

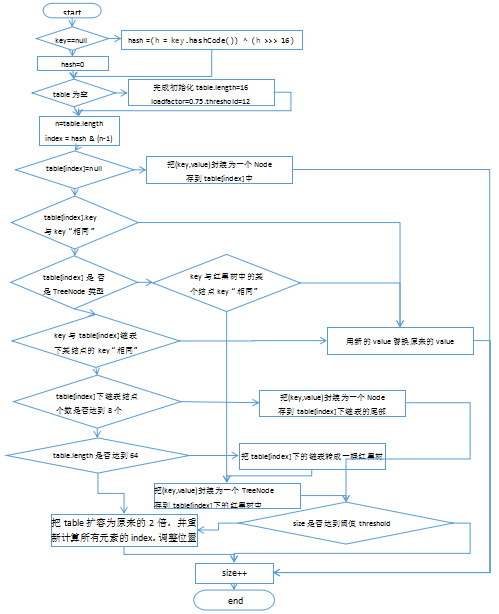
当某个链表的结点个数达到8时，还要检查table的长度是否达到64，如果没有达到，先扩容解决冲突问题

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  } |

## 12.7 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、要求用集合实现简易字符串压缩算法，其中连续出现2次以上（含2次）的字母转换为字母和出现的次数。

例如：AAAABCCDEEEEE，压缩之后A4BC2DE5

例如：AABAABCCDEEEEEF，压缩之后A2BA2BC2DE5F

2、随机产生10个[0,100)的整数，放到集合中，要求从大到小输出

3、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 |

4、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