链滴

# Java 集合

作者：Hefery

原文链接：https://ld246.com/article/1641884785915

来源网站：链滴

# 集合作用

集合：对象的容器，定义了对多个对象进行操作的常用方法

集合按照其存储结构可以分为两大类：单列集合java.util.Collection、双列集合java.util.Map

集合与数组的区别：

| | 长度 | 存储对象 | 存储数类型 |
|---|---|---|---|
| 数组 | 固定 | 基本数据类型、同类型的对象 | 定 |
| 集合 | 可变 | 不同类型的对象 | 定 |

# Collection

Collection是所有单列集合的父接口，在Collection中定义了单列集合(List和Set)通用的方法，这些法可用于操作所有的单列集合

| 数据类型 | 方法 | 说明 |
|---|---|---|
| boolean | add(E e) | 指定的元素加到集合中 |
| void | clear() | 清空所有元素 |
| boolean | remove(Object o) | 移指定元素 |
| boolean | contains(Object o) | 判是否包含指定的元素 |
| boolean | isEmpty() | 判断集合是为空 |
| int | size() | 返回集合的元素数 |
| Object[] | toArray() | 集合中的元存入数组 |

```
public class CollectionTest {
    public static void main(String[] args) {
        // 创建集合对象，可使用多态
        Collection<String> cell = new ArrayList<>();
        System.out.println(cell);  //重写了toString方法

        // 添加元素 add(E e)
        boolean b1 = cell.add("Hefery");
        System.out.println(b1);   //true,无需接收返回值
        System.out.println(cell); //[Hefery]

        cell.add("Hello");
        cell.add("World");
```

```
        // 移除元素 remove(Object o)
        boolean b2 = cell.remove("World");
        boolean b3 = cell.remove("abc");
        System.out.println(b2);  //true
        System.out.println(b3);  //false

        // 查询元素 contains(Object o)
        boolean b4 = cell.contains("Hefery");
        boolean b5 = cell.contains("Woorld");
        System.out.println(b4);  //true
        System.out.println(b5);  //false

        // 判空 isEmpty()
        boolean b6 = cell.isEmpty();
        System.out.println(b6);  //false

        // 元素个数 size()
        int b7 = cell.size();
        System.out.println(b7);  //2

        // 集合元素存入数组 toArray()
        Object[] arr = cell.toArray();
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }

        // 清空 clear()
        cell.clear();
        System.out.println(cell);  //[]
    }
}
```

# List

List特点：

1. 有序的 collection（也称为序列）
2. 带索引，有下标
3. 允许重复的元素

List的常用子类：

- ArrayList  底层结构是数组,底层查询快,增删慢
- LinkedList 底层结构是链表,增删快,查询慢
- Voctor  底层结构是数组,线程安全,增删慢,查询慢

## ArrayList

```
public class ArrayList<E> extends AbstractList<E> implements List<E>
```

底层：数组

特点：查询快，增删慢，线程不安全

必须开辟连续空间

## 重要方法

| 数据类型 | 方法 | 说明 |
| --- | --- | --- |
| boolean | add(E e) | 将指定的元素添加到此列表的尾部 |
| void | add(int index, E element) | 将指定的元素插入此列表中的指定位置 |
| E | remove(int index) | 移除此列表中指定位置上的元素 |
| E | set(int index, E element) | 用指定的元素替代此列表中指定位置上的元素 |
| void | clear() | 移除此列表中的所有元素 |
| Object[] | toArray() | 按适当顺序（从第一个到最后一个元素）返回包含此列表中所有元素的数组 |
| E | get(int index) | 返回此列表中指定位置上的元素 |
| boolean | isEmpty() | 如果此列表没有元素，则返回 true |
| int | size() | 返回此列表中的元素数 |
| boolean | contains(Object o) | 如果此列表中包含指定的元素，则返回 true |

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class listDemo {
    public static void main(String[] args) {
        //创建 List 对象(多态)
        List<String> list = new ArrayList<>();

        //添加元素
        list.add("a");
        list.add("b");
        list.add("c");
        list.add("a");
        System.out.println(list);

        //添加元素，在指定位置
        list.add(3,"d");
        System.out.println(list);
```

```
        //移除元素(位置)
        list.remove(3);
        System.out.println(list);

        //替换元素值
        list.set(3,"A");
        System.out.println(list);

        //遍历
        for (int i = 0; i < list.size(); i++) {
            String s = list.get(i);
            System.out.println(s);
        }

        for (String e:list) {
            System.out.println(e);
        }

        Iterator<String> it = list.iterator();
        while (it.hasNext()){
            String s0 = it.next();
            System.out.println(s0);
        }
    }
}
```

## 源码分析

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Clone
ble, java.io.Serializable {
    private static final long serialVersionUID = 8683452581122892189L;

    /** 默认容量 */
    private static final int DEFAULT_CAPACITY = 10;

    /** 数组元素 */
    private static final Object[] EMPTY_ELEMENTDATA = {};

    /** 默认空数组 */
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    /** 存放元素 */
    transient Object[] elementData; // non-private to simplify nested class access

    /** 元素个数 */
    private int size;

    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);
```

```
        }
    }

    /** 无参构造 */
    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    public boolean add(E e) {
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        elementData[size++] = e;
        return true;
    }
    private void ensureCapacityInternal(int minCapacity) {
        ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
    }
    private static int calculateCapacity(Object[] elementData, int minCapacity) {
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            return Math.max(DEFAULT_CAPACITY, minCapacity);
        }
        return minCapacity;
    }
    private void ensureExplicitCapacity(int minCapacity) {
        modCount++;
        // overflow-conscious code
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }
    /** 数组扩容：如果没有向集合中添加任何元素时，容量0，添加一个元素之后容量10，默认长度是
0，当超过长度时，按50%延长集合长度 */
    private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }

}
```

## LinkedList

底层：双向链表

特点：查询慢，增删快，线程不安全

无需开辟连续空间

### 重要方法

| 数据类型 | 方法 | 说明 |
| --- | --- | --- |
| boolean | add(E e) | 将指定的元素添加到此列表的尾部 |
| void | add(int index, E element) | 将指定的元素插入此列表中的指定位置 |
| void | addFirst(E e) | 将指定元素插入此列表的开头 |
| void | addLast(E e) | 将指定元素添加到此列表的结尾 |
| E | remove(int index) | 移除此列表中指定位置上的元素 |
| E | removeFirst() | 移除并返回此列表的第一个元素 |
| E | removeLast() | 移除并返回此列表的最后一个元素 |
| E | set(int index, E element) | 用指定的元素替代此列表中指定位置上的元素 |
| void | clear() | 移除此列表中的所有元素 |
| Object[] | toArray() | 按适当顺序（从第一个到最后一个元素）返回包含此列表中所有元素的数组 |
| E | get(int index) | 返回此列表中指定位置上的元素 |
| E | getFirst() | 返回此列表的第一个元素 |
| E | getLast() | 返回此列表的最后一个元素 |
| boolean | isEmpty() | 如果此列表没有元素，则返回 true |
| int | size() | 返回此列表中的元素数 |
| boolean | contains(Object o) | 如果此列表中包含指定的元素，则返回 true |
| E | pop() | 从此列表所表示的堆栈弹出一个元素，移除第一个元素 |
| void | push(E e) | 将元素推入此列表所表示的堆栈，移除最后一个元素 |

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class listDemo {
    public static void main(String[] args) {
        demo01();
    }
```

```
    private static void demo01() {
        //创建 LinkedList 对象(不能使用多态)
        LinkedList<String> linkedlist = new LinkedList<>();

        //添加
        linkedlist.add("a");
        linkedlist.add("b");
        linkedlist.add("c");

        linkedlist.addFirst("www.");
        linkedlist.addLast(".cn");

        linkedlist.add(1,"hefery");

        linkedlist.push("lalala");

        System.out.println(linkedlist);  //[lalala, www., hefery, a, b, c, .cn]

        //获取
        System.out.println(linkedlist.getFirst()); //lalala
        System.out.println(linkedlist.getLast());  //.cn
        System.out.println(linkedlist.get(2));     //hefery

        //移除
        linkedlist.removeFirst();
        linkedlist.removeLast();
        linkedlist.pop();
        System.out.println(linkedlist);  //[hefery, a, b, c]

    }
}
```

## 源码分析

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

    /** 集合大小 */
    transient int size = 0;

    /** 链表头结点 */
    transient Node<E> first;

    /** 链表尾节点 */
    transient Node<E> last;

    /** 空参构造 */
    public LinkedList() {
    }

    /** 节点结构 */
    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;
        }
    }

    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++;
    }

    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++;
    }

}
```

## Voctor

Vector 类可以实现可增长的对象数组，线程安全

底层：数组

特点：查询快，增删慢，线程安全

### 重要方法

| 数据类型 | 方法 | 说明 |
| --- | --- | --- |
| boolean | add(E e) | 将指定的元素添加到此列表的尾部 |
| void | add(int index, E element) | 将指定的元素插入此列表中的指定位置 |
| void | addElement(E obj) | 将指 |

的组件添加到此向量的末尾，将其大小增加 1

| | | |
|---|---|---|
| E | remove(int index) | 移除此列表中指定位置上的元素 |
| E | removeElement() | 从此向量中除变量的第一个（索引最小的）匹配项 |
| E | removeLast() | 移除并返回此列表的最后一个元素 |
| E | set(int index, E element) | 用定的元素替代此列表中指定位置上的元素 |
| void | clear() | 移除此列表中的所元素 |
| Object[] | toArray() | 按适当顺序从第一个到最后一个元素）返回包含此列表中所有元素的数组 |
| E | get(int index) | 返回此列表中定位置上的元素 |
| boolean | isEmpty() | 如果此列表没有元素，则返回 true |
| int | size() | 返回此列表中的元素数 |
| boolean | contains(Object o) | 如此列表中包含指定的元素，则返回 true |
| Enumeration<E> | elements() | 回此向量的组件的枚举，遍历 |

# Set

## HashSet

底层：哈希表，基于 HashMap 实现的

特点：不包含重复元素，无索引，无序

### 重要方法

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class setDemo {
    public static void main(String[] args) {
        Set<Integer> set = new HashSet<>();

        //添加
        set.add(1);
        set.add(2);
        set.add(3);
        set.add(1);
```

```
        //遍历
        Iterator<Integer> it = set.iterator();
        while (it.hasNext()){
            Integer n = it.next();
            System.out.println(n);
        }

        for (Integer i : set) {
            System.out.println(i);
        }
    }
}

//存储自定义类型元素，需重写 hashCode() 和 equals()
import java.util.HashSet;
/*
    重写 hashCode() 和 equals() 保证同名同龄的人只能存储一次
 */
public class hashCodeDemo {
    public static void main(String[] args) {
        HashSet<Person> set = new HashSet<>();

        Person p1 = new Person("Hefery",22);
        Person p2 = new Person("Hefery",22);
        Person p3 = new Person("Hefery",18);
        System.out.println(p1.hashCode());  //未重写:356573597     重写后：-1830346093
        System.out.println(p2.hashCode());  //未重写:1735600054    重写后：-1830346093
        System.out.println(p1==p2);         //未重写:false         重写后：false     比较地址
        System.out.println(p1.equals(p2));  //未重写:false         重写后：true

        set.add(p1);
        set.add(p2);
        set.add(p3);

        System.out.println(set);
        //未重写：[Person{name='Hefery', age=22}, Person{name='Hefery', age=22}, Person{nam
='Hefery', age=18}]
        //重写后：[Person{name='Hefery', age=22}, Person{name='Hefery', age=18}]
    }
}
```

## 源码分析

## LinkedHashSet

具有可预知迭代顺序的 Set 接口的哈希表和链接列表实现

底层：哈希表(链表+红黑树) + 链表(记录元素存储顺序)

```
import java.util.HashSet;
import java.util.LinkedHashSet;

public class linkedhashsetDemo {
    public static void main(String[] args) {
```

```
        HashSet<String> set = new HashSet<>();

        set.add("www");
        set.add("abc");
        set.add("abc");
        set.add("cn");

        System.out.println(set);  //无序，不允许重复

        LinkedHashSet<String> linked = new LinkedHashSet<>();

        linked.add("www");
        linked.add("abc");
        linked.add("abc");
        linked.add("cn");

        System.out.println(linked);  //有序，不允许重复
    }
}
```

### TreeSet

底层数据结构：红黑树

特点：

基于排列顺序实现元素不重复

实现了 Sortedset接口，对集合元素自动排序

元素对象的类型必须实现 Comparable接口，指定排序规则

通过 CompareTo方法确定是否为重复元素

## Collections

public class Collections extends Object

### 常用方法

| 数据类型 | 方法 | 说明 |
| --- | --- | --- |
| static <T> boolean | addAll(Collection<? super T> c, T... elements) | 将所有指定元素添加到指定 collection 中 |
| static void | shuffle(List<?> list) | 用默认随机源对指定列表进行置换 |
| static <T extends Comparable<? super T>> <br/>void | sort(List<T> list) | 根据元素的自然顺序 对指定列表按升序进行排序 |
| static <T> void | sort(List<T> list, Comparator<? super T> c) | 据指定比较器产生的顺序对指定列表进行排序 |

```
@Data
public class Person implements Comparable<Person>{
    private String name;
```

```
    private int age;
    //重写排序规则
    @Override
    public int compareTo(Person o) {
        return  this.getAge() - o.getAge(); //年龄升序
        //return  o.getAge() - this.getAge(); //年龄降序
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Person person = (Person) o;

        if (age != person.age) return false;
        return name != null ? name.equals(person.name) : person.name == null;
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + age;
        return result;
    }
}

import java.util.ArrayList;
import java.util.Collections;
/*
    注意：sort()使用：在被排序的集合实现必须重写接口中的 compareTo 定义的规则
    compareTo：
        升序：this.参数 - o.参数
        降序：o.参数 - this.参数
 */

public class CollectionsDemo {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();

        //添加多个元素
        Collections.addAll(list,"a","b","c","d","e");
        System.out.println(list);  //[a, b, c, d, e]

        //打乱元素顺序
        Collections.shuffle(list);
        System.out.println(list);  //[e, b, d, a, c]

        System.out.println("================================");

        ArrayList<Integer> in = new ArrayList<>();
        Collections.addAll(in,12,15,14,13,16);
        System.out.println(in);  //[12, 15, 14, 13, 16]
```

```
        //升序
        Collections.sort(in);
        System.out.println(in);  //[12, 13, 14, 15, 16]

        System.out.println("================================");

        ArrayList<String> str = new ArrayList<>();
        Collections.addAll(str,"lalala","hefery","hahaha","biubiubiu");
        System.out.println(str);  //[lalala, hefery, hahaha, biubiubiu]
        Collections.sort(str);
        System.out.println(str);  //[biubiubiu, hahaha, hefery, lalala]

        System.out.println("================================");

        ArrayList<Person> p = new ArrayList<>();
        p.add(new Person("lalala",12));
        p.add(new Person("hahaha",8));
        p.add(new Person("wawawa",32));
        System.out.println(p); //[Person{name='lalala', age=12}, Person{name='hahaha', age=8},
erson{name='wawawa', age=32}]

        //重写 compareTo 方法，根据 age 升序
        Collections.sort(p); //[Person{name='hahaha', age=8}, Person{name='lalala', age=12}, Per
on{name='wawawa', age=32}]

        System.out.println(p);
    }
}
```

## Comparator和Comparable

Comparable：自己this与别人o比较，自己需要实现 Comparable 接口，重写比较规则 compareTo 方法

Comparator：找第三方仲裁

```
public class CollectionsDemo {
    public static void main(String[] args) {
        ArrayList<Person> p = new ArrayList<>();
        p.add(new Person("lalala",12));
        p.add(new Person("bhahaha",8));
        p.add(new Person("ahihihi",8));
        p.add(new Person("wawawa",32));
        System.out.println(p);
        //[Person{name='lalala', age=12}, Person{name='hahaha', age=8}, Person{name='wawa
a', age=32}]

        //重写 compareTo 方法，根据 age 升序
        //Collections.sort(p);
        //[Person{name='hahaha', age=8}, Person{name='lalala', age=12}, Person{name='wawa
a', age=32}]

        /*Collections.sort(p, new Comparator<Person>() {
            @Override
```

```
            public int compare(Person o1, Person o2) {
                //return o1.getAge() - o2.getAge();  //age升序
                return o2.getAge() - o1.getAge();   //age降序
            }
        });*/

        Collections.sort(p, new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                int result = o1.getAge() - o2.getAge();
                //age相同，比较姓的首字母
                if( result > 0 ){
                    result = o1.getName().charAt(0) - o2.getName().charAt(0);
                }
                return result;
            }
        });

        System.out.println(p);
    }
}
```

# Map

## HashMap<K,V>

底层：数组+链表(Java8)、数组+链表+红黑树(Java8及之后)

特点：无序，存储取出元素顺序可能不一致

### 重要方法

```
public class mapDemo {
    public static void main(String[] args) {
        //show01();
        //show02();
        //show03();
        show04();
    }

    private static void show04() {
        /*
            boolean containsKey(Object key)
              如果此映射包含指定键的映射关系，则返回 true
         */
        Map<String,Integer> map4 = new HashMap<>();

        map4.put("赵丽颖",168);
        map4.put("林志玲",178);
        map4.put("刘洋",188);

        System.out.println(map4);
```

```
    boolean v1 = map4.containsKey("赵丽颖");
    System.out.println("v1:" + v1);  //v1:true
    boolean v2 = map4.containsKey("赵颖");
    System.out.println("v1:" + v2);  //v1:false
}

private static void show03() {
    /*
        V get(Object key)
          返回指定键所映射的值;
          如果此映射不包含该键的映射关系，则返回 null
     */
    Map<String,Integer> map3 = new HashMap<>();

    map3.put("赵丽颖",168);
    map3.put("林志玲",178);
    map3.put("刘洋",188);

    System.out.println(map3);

    Integer v1 = map3.get("刘洋");
    System.out.println("v1: " + v1);
}

private static void show02() {
    /*
        V remove(Object key) :
          返回值：V
             Key 存在，返回被删除值
             Key不存在，返回null
     */
    Map<String,Integer> map2 = new HashMap<>();

    map2.put("赵丽颖",168);
    map2.put("林志玲",178);
    map2.put("刘洋",188);

    System.out.println(map2);

    Integer v1 = map2.remove("林志玲");
    System.out.println("v1:" + v1);  //v1:178
    System.out.println(map2);

    Integer v2 = map2.remove("林志颖");
    System.out.println("v1:" + v2);  //v1:null
}

private static void show01() {
    /*
       put(K key, V value):
          返回值：V value
             存储键值对时，Key 不重复，返回值 value 为 null
             存储键值对时，Key 重复，使用新的 value 替换 map 中的 value
     */
```

```
        Map<String,String> map1 = new HashMap<>();

        //测试 Key
        String v1 = map1.put("hefery", "wahaha1");
        System.out.println("v1: " + v1);  //v1: null

        String v2 = map1.put("hefery", "wahaha2");
        System.out.println("v2: " + v2);  //v2: wahaha1

        System.out.println(map1);  //{hefery=wahaha2}

        //测试 value
        map1.put("张杰","谢娜");
        map1.put("李晨","范冰冰");
        map1.put("陈乔恩","范冰冰");

        System.out.println(map1);  //{陈乔恩=范冰冰, hefery=wahaha2, 李晨=范冰冰, 张杰=谢娜}
    }
}

/*
    Map 集合遍历:
        1.键找值
        2.键值对
 */
public class mapDemo {
    public static void main(String[] args) {
        /*Map<String,Integer> map = new HashMap<>();

        map.put("赵丽颖",168);
        map.put("林志玲",178);
        map.put("刘洋",188);

        //1.将所有 Key 取出，存入 Set 集合
        Set<String> set = map.keySet();

        //2.使用迭代器遍历
        Iterator<String> it = set.iterator();
        while (it.hasNext()){
            //3.通过 key 找到 value
            String key = it.next();
            Integer value = map.get(key);
            System.out.println(key + " = " + value);
        }

        //2.foreach遍历
        for (String key : set) {
            //3.通过 key 找到 value
            String value = it.next();
            System.out.println(key + " = " + value);
        }

        for (String key : map.keySet()) {
            String value = it.next();
```

```
            System.out.println(key + " = " + value);
        }*/

        System.out.println("==========================================
======");

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

        map.put("赵丽颖",168);
        map.put("林志玲",178);
        map.put("刘洋",188);

        //1.使用 Map 集合中的 entrySet() 方法，把集合中的多个 Entry 对象取出，存入 Set 集合
        Set<Map.Entry<String,Integer>> set = map.entrySet();

        //2.迭代器遍历
        Iterator<Map.Entry<String, Integer>> it = set.iterator();
        while (it.hasNext()){
            Map.Entry<String, Integer> entry = it.next();
            //3.通过 getKey()、getValue() 得到 key、value 值
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println(key + " = " + value);
        }

        //foreach遍历
        for (Map.Entry<String, Integer> entry : set ) {
            //3.通过 getKey()、getValue() 得到 key、value 值
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println(key + " = " + value);
        }
    }
}
```

存储自定义类型的数据

```
@Data
public class Person{
    private String name;
    private int age;
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Person person = (Person) o;

        if (age != person.age) return false;
        return name != null ? name.equals(person.name) : person.name == null;
    }

    @Override
    public int hashCode() {
```

```
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + age;
        return result;
    }
}

/*
    Map 集合包装 key 唯一:
        作为 key 元素，必须重写 hSHcODE() 和 equlas() 方法
        作为 value 值，可重复
 */
public class mapDemo {
    public static void main(String[] args) {
        show();
    }

    private static void show() {
        //创建 HashMap 集合
        HashMap<String,Person> map = new HashMap<>();
        HashMap<Person,String> map1 = new HashMap<>();

        //添加元素
        map.put("北京",new Person("张三",13));
        map.put("上海",new Person("李四",20));
        map.put("深圳",new Person("王五",10));
        map.put("北京",new Person("麻子",13));

        map1.put(new Person("张三",13),"北京");
        map1.put(new Person("李四",20),"上海");
        map1.put(new Person("王五",10),"深圳");
        map1.put(new Person("张三",13),"南京");

        //遍历
        Set<String> set = map.keySet();
        for (String key : set){
            Person value = map.get(key);
            System.out.println(key + " = " + value);
        }
        //麻子将张三替换

        //使用 entrySet() 和 foreach 遍历
        Set<Map.Entry<Person, String>> set1 = map1.entrySet();
        for (Map.Entry<Person, String> entry : set1) {
            Person key = entry.getKey();
            String value = entry.getValue();
            System.out.println(key + " = " + value);
        }

    }
}
```

## 源码分析

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, S
```

```
rializable {

    private static final long serialVersionUID = 362498820763181265L;

    /** 默认初识容量 */
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;  // aka 16

    /** 最大容量 */
    static final int MAXIMUM_CAPACITY = 1 << 30;

    /** 加载因子 */
    static final float DEFAULT_LOAD_FACTOR = 0.75f;

    /** 当链表长度大于8，并且数组长度大于64，链表转化为红黑树 */
    static final int TREEIFY_THRESHOLD = 8;

    /** 当链表长度小于6，红黑树转化为链表 */
    static final int UNTREEIFY_THRESHOLD = 6;

    /** 转化为红黑树的最小数组长度 */
    static final int MIN_TREEIFY_CAPACITY = 64;

    /** 元素个数 */
    transient int size;

    /** 键值对 */
    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;
        }
    }

    /** 存放键值对Node的数组 */
    transient Node<K,V>[] table;

    /** 无参构造 */
    public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }


    public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }

    static final int hash(Object key) {
```

```
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }

    /** HashMap刚创建时， table是nu11，为了节省空间，当添加第一个元素时， table容量调整为1
*/
    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);
        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;
                }
            }
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }

    /** 当元素个数大于阈值（16*8.75=12）时，会进行扩容，扩容后大小为原来的2倍。目的是减少
整元素的个数
        jdk1.8：当每个链表长度大于8，并且元素个数大于等于64时会调整为红黑树，目的提高执行效率
        当链表长度小于6时，调整成链表
        jdk1.8以前，链表头插入，jdk1.8以后链表尾插入*/
    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
}
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;
@SuppressWarnings({"rawtypes","unchecked"})
Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
table = newTab;
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;
}

}
```

## LinkedHashMap

public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>

1.底层：哈希表 + 链表

2.有序，存储取出元素顺序一致

```
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.LinkedHashSet;

public class LinkedHashMapDemo {
    public static void main(String[] args) {
        HashMap<String,String> map = new HashMap<>();

        map.put("a","a");
        map.put("c","c");
        map.put("b","b");
        map.put("a","d");

        System.out.println(map); //key 值不允许重复，无序 {a=d, b=b, c=c}

        System.out.println("==========================================");

        LinkedHashMap<String,String> linked = new LinkedHashMap<>();
        linked.put("a","a");
        linked.put("c","c");
        linked.put("b","b");
        linked.put("a","d");

        System.out.println(linked); //key 值不允许重复，有序 {a=d, c=c, b=b}
    }
```

```
}

public class LinkedHashMapDemo {
    public static void main(String[] args) {
        LinkedHashMap<String, String> map = new LinkedHashMap<String, String>();
        map.put("邓超", "孙俪");
        map.put("李晨", "范冰冰");
        map.put("刘德华", "朱丽倩");
        Set<Entry<String, String>> entrySet = map.entrySet();
        for (Entry<String, String> entry : entrySet) {
            System.out.println(entry.getKey() + "  " + entry.getValue());
        }
    }
}
```

# ConcurrentHashMap<K,V>

所有操作都是线程安全的，但获取操作不必锁定，并且不支持以某种防止所有访问的方式锁定整个表

此类可以通过程序完全与 Hashtable 进行互操作，这取决于其线程安全，而与其同步细节无关

## 构造函数

| 构造函数 | 说明 |
| --- | --- |
| ConcurrentHashMap() | 创建一个带有默认初始量 (16)、加载因子 (0.75) 和 concurrencyLevel (16) 的新的空映射 |
| ConcurrentHashMap(int initialCapacity) | 创建个带有指定初始容量、默认加载因子 (0.75) 和 concurrencyLevel (16) 的新的空映射 |
| ConcurrentHashMap(int initialCapacity, float loadFactor) | 建一个带有指定初始容量、加载因子和默认 concurrencyLevel (16) 的新的空映射 |
| ConcurrentHashMap(Map m) | 创建一个带有指初始容量、加载因子和并发级别的新的空映射 |
| ConcurrentHashMap(Map<? extents K, ? extents V> m) | 造一个与给定映射具有相同映射关系的新映射 |

## 重要方法

| 数据类型 | 方法 | 说明 |
| --- | --- | --- |
| V | put(K key, V value) | 将指定键射到此表中的指定值 |
| V | get(Object key) | 返回指定键映射到的值，如果此映射不包含该键的映射关系，则返回 null |
| void | clear() | 从该映射中移除所映射关系 |
| boolean | contains(Object value) | 种遗留方法，测试此表中是否有一些与指定值存在映射关系的键 |
| boolean | containsKey(Object key) | |

试指定对象是否为此表中的键

boolean containsValue(Object value)
果此映射将一个或多个键映射到指定值，则返回 true

Enumeration<V> elements()
回此表中值的枚举

Set<Map.Entry<K,V>> entrySet()
回此映射所包含的映射关系的 Set 视图

boolean isEmpty() 如果此映射
包含键-值映射关系，则返回 true

Enumeration<K> keys() 返回
表中键的枚举

Set<K> keySet() 返回此映射中
含的键的 Set 视图

void putAll(Map<? extends K,? extends V> m)
指定映射中所有映射关系复制到此映射中

V putIfAbsent(K key, V value)
果指定键已经不再与某个值相关联，则将它与给定值关联

V remove(Object key) 从此映
中移除键（及其相应的值）

boolean remove(Object key, Object value)
有目前将键的条目映射到给定值时，才移除该键的条目

V replace(K key, V value) 只有
前将键的条目映射到某一值时，才替换该键的条目

boolean replace(K key, V oldValue, V newValue)
有目前将键的条目映射到给定值时，才替换该键的条目

int size() 返回此映射中的键-值
射关系数

Collection<V> values() 返回此
射中包含的值的 Collection 视图