* 学习目标

* 能够理解Set接口概述

  * 无序，不可重复

* 能够通过阅读HashSet源码理解它底层实现就是HashMap的key集合

  * HashMap的key集合

  * 重写hashCode，equals‘

  * ...

* 能够通过阅读LinkedHashSet源码理解它底层实现就是LinkedHashMap的key集合

* 能够通过阅读TreeSet源码理解它底层实现就是TreeMap的key集合

* 能够掌握Collections常见的方法

    * addAll,shuffle,reverse,sort,binarySearch

    * ...(可变参数)

    * sort：（unicode编码）

* 能够掌握并发集合类之CopyOnWriteArrayList （ArrayList）

    * ArrayList

* 能够掌握并发集合类之ConcurrentHashMap （HashMap）

    * HashMap

* 能够掌握并发集合类之ConcurrentSkipListMap （TreeMap）

* 能够掌握并发集合类之 ConcurrentSkipListSet （TreeSet）

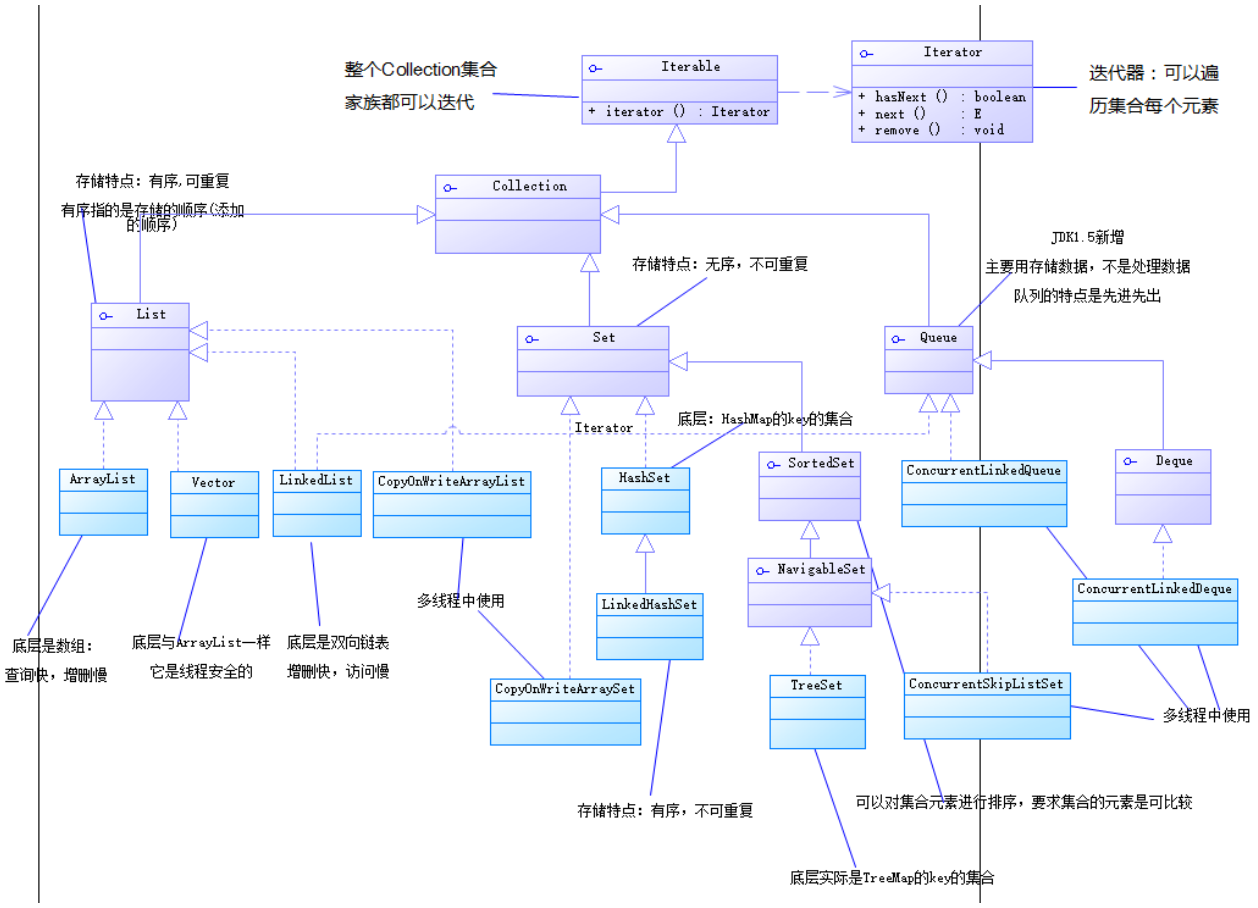
* 能够掌握并发集合类之CopyOnWriteArraySet （HashSet）

* 能够掌握并发集合类之ConcurrentLinkedDeque （LinkedList）

---

* 能够理解Set接口概述

  * 特点：无序，不可重复



* 能够通过阅读HashSet源码理解它底层实现就是HashMap的key集合

```
* HashSet 核心代码
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    // ...
    private transient HashMap<E,Object> map;
    private static final Object PRESENT = new Object();
    // 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);
    }
    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }
    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
    public Iterator<E> iterator() {
        return map.keySet().iterator();
    }
    public boolean remove(Object o) {
        return map.remove(o)==PRESENT;
    }
    public boolean contains(Object o) {
        return map.containsKey(o);
    }
    public void clear() {
        map.clear();
    }
    public int size() {
        return map.size();
    }
    public boolean isEmpty() {
        return map.isEmpty();
    }
    // ...
}

* 总结
 * HashSet底层是HashMap
 * 调用add()添加元素，实际上是把该元素作为键添加到了HashMap中
 * Hashset实际上就是HashMap的键的集合
 * 因为Map中的键不允许重复,所以Set集合中的元素不重复
 * HashSet添加元素：建议元素重写equals，hashCode

 * 案例：
   创建一个ArayList的集合对象，该集合对象存储一批图书，该图书存在着重复元素
   定义一个方法清除重复元素。返回一个没有重复元素的集合对象。
```

```
 public static void main(String[] args) {
        List<Book> books=new ArrayList<Book>();
        books.add(new Book(1, "Java", 28.9));
        books.add(new Book(2, "html", 30.9));
        books.add(new Book(1, "Java", 28.9));
        books.add(new Book(1, "Java", 28.9));
        books.add(new Book(1, "Java", 28.9));
        Set<Book> books2 = getBooks(books);
        for(Book book:books2) {
            System.out.println(book);
        }
    }
//  定义一个方法清除重复元素。返回一个没有重复元素的集合对象
    public static Set<Book> getBooks(List<Book> books){
        HashSet<Book> set=new HashSet<Book>(books);
        return set;
    }
 结果:
 Book [id=1, name=Java, price=28.9]
 Book [id=2, name=html, price=30.9]
```

* 能够通过阅读LinkedHashSet源码理解它底层实现就是LinkedHashMap的key集合

```
public class LinkedHashSet<E>
    extends HashSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {
    public LinkedHashSet(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor, true);
    }
    public LinkedHashSet(int initialCapacity) {
        super(initialCapacity, .75f, true);
    }
    public LinkedHashSet() {
        super(16, .75f, true);
    }
    //...
}
 * 总结:
  * LinkedHashSet 继承 HashSet，调用HashSet三个构造器的方法
```

```
    * HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }
  * LinkedHashSet底层是LinkedHashMap
  * LinkedHashSet 是有序的

 * 例子演示
 * HashSet：是无序
  public static void main(String[] args) {
        Set<String> set=new HashSet<String>();
        set.add("刘备");
        set.add("关羽");
        set.add("张飞");
        for(String value:set) {
            System.out.printf("%s ",value);
        }
    }
  结果：
  关羽 张飞 刘备
  * LinkedHashSet 是有序的
  public static void main(String[] args) {
        Set<String> set=new LinkedHashSet<String>();
        set.add("刘备");
        set.add("关羽");
        set.add("张飞");
        for(String value:set) {
            System.out.printf("%s ",value);
        }
    }
  * 结果：
  刘备 关羽 张飞
```

* 能够通过阅读TreeSet源码理解它底层实现就是TreeMap的key集合

```
* TreeSet 核心源码
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
{
    private transient NavigableMap<E,Object> m;
```

```
    private static final Object PRESENT = new Object();
    TreeSet(NavigableMap<E,Object> m) {
        this.m = m;
    }
    public TreeSet() {
        this(new TreeMap<E,Object>());
    }
    public TreeSet(Comparator<? super E> comparator) {
        this(new TreeMap<>(comparator));
    }
    public TreeSet(Collection<? extends E> c) {
        this();
        addAll(c);
    }
    public TreeSet(SortedSet<E> s) {
        this(s.comparator());
        addAll(s);
    }
    public Iterator<E> iterator() {
        return m.navigableKeySet().iterator();
    }
     public int size() {
        return m.size();
    }
    public boolean isEmpty() {
        return m.isEmpty();
    }
    public boolean contains(Object o) {
        return m.containsKey(o);
    }
    public boolean add(E e) {
        return m.put(e, PRESENT)==null;
    }
    public boolean remove(Object o) {
        return m.remove(o)==PRESENT;
    }
    public void clear() {
        m.clear();
    }
   //..
```

```
}

 * TreeSet底层使用TreeMap的Key
 * 使用TreeMap前提
    * 要么Key的类实现了Comparable接口
    * 假如Key的类没有实现Comparable接口，就需要使用Comparator比较器

 * 例子
  * 存储一批员工对象 name 、 salary， 请根据薪水排序
  public static void main(String[] args) {
        Set<Emp> emps=new TreeSet<Emp>(new Comparator<Emp>() {
            @Override
            public int compare(Emp o1, Emp o2) {
                return (int) (o1.getSalary()-o2.getSalary());
            }
        });
        emps.add(new Emp(1001,"刘备",8500));
        emps.add(new Emp(1002,"关羽",18500));
        emps.add(new Emp(1003,"张飞",3500));

        for(Emp emp:emps) {
            System.out.println(emp);
        }
    }
  结果:
  Emp [empId=1003, name=张飞, salary=3500.0]
  Emp [empId=1001, name=刘备, salary=8500.0]
  Emp [empId=1002, name=关羽, salary=18500.0]
```

* 能够掌握Collections常见的方法

    * addAll,shuffle,reverse,sort,binarySearch

```
* 可变参数
public class Test1 {

    public static void main(String[] args) {
//      printArgs(1, 2, 3, 4, 5);
        int[] arrs= {1,2,3,4,5};
```

```
        printArgs1(arrs);
        printArgs1(arrs);
    }

    public static void printArgs(int a,int b,int c,int d, int e) {
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
        System.out.println(e);
    }

    public static void printArgs(int[] arrs) {
        for (int i = 0; i < arrs.length; i++) {
            System.out.println(arrs[i]);
        }
    }

    // 可变参数：jdk1.5
    public static void printArgs1(int... arrs) {
        for (int i = 0; i < arrs.length; i++) {
            System.out.println(arrs[i]);
        }
    }

}

*
package com.lg.test3;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Test2 {
    public static void main(String[] args) {
        List<String> list=new ArrayList<String>();
        // A B C D E
        list.add("刘备");//   \u5218\u5907
```

```
        List<String> subList=new ArrayList<String>();
        subList.toArray();
        subList.add("赵云");// \u8d75\u4e91
        subList.add("黄忠");// \u9ec4\u5fe0
        list.addAll(subList);
        //可变参数...           \u5173\u7fbd
        Collections.addAll(list,"关羽","张飞");
//      Collections.addAll(list,subList.toArray(new String[0]));
        System.out.println(list);
        // [关羽, 刘备, 张飞, 赵云, 黄忠]
        // \u5173\u7fbd  \u5218\u5907 \u5f20\u98de \u8d75\u4e91 \u9ec4\u5fe0
        Collections.shuffle(list);
        System.out.println(list);
        int index = Collections.binarySearch(list, "刘备");
        System.out.println(index);

        Collections.reverse(list);
        System.out.println(list);

        Collections.sort(list,new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                return o1.compareTo(o2);
            }
        });
        System.out.println(list);// Unicode 编码

        index = Collections.binarySearch(list, "刘备");
        System.out.println(index);

        Collections.reverse(list);
        System.out.println(list);

        index = Collections.binarySearch(list, "刘备");
        System.out.println(index);
    }
}

* String compareTo重新认识
import java.util.ArrayList;
```

```java
import java.util.Collections;
import java.util.List;

public class Test3 {
    public static void main(String[] args) {
        List<String> list=new ArrayList<String>();
        list.add("李白");// \u674e\u767d
        list.add("杜甫");// \u675c\u752b
        list.add("白居易");// \u767d\u5c45\u6613
        list.add("孟浩然");// \u5b5f\u6d69\u7136
        list.add("屈原");//  \u5c48\u539f
        // 孟浩然: \u5b5f\u6d69\u7136  23391
        //          5b5f  6d69  7136
        // 屈原: \u5c48\u539f         23624
        // 李白: \u674e\u767d         26446
        // 杜甫: \u675c\u752b         26460
        // 白居易: \u767d\u5c45\u6613 30333
        // 假如是
        Collections.sort(list);
        System.out.println(list);


        String str="中国";//\u4e2d\u56fd
        String name="白居易";
        char[] chs = name.toCharArray();
        for (int i = 0; i < chs.length; i++) {
//          System.out.println(chs[i]);
            System.out.println(Integer.valueOf(chs[i]));
        }
//      Set<String> set=new TreeSet<String>();
//      set.add("李白");// L
//      set.add("杜甫");// D
//      set.add("白居易");// B
//      set.add("孟浩然");// M
//      set.add("屈原");// Q
//      System.out.println(set);
        二分查找();
    }

    public static void 二分查找() {
```

```
127
128     }
129 }
130
```

* 能够掌握并发集合类之CopyOnWriteArrayList （ ArrayList ）

```
* 多个线程同时操作并且遍历集合
 * ArrayList
// 多个线程同时操作并且遍历集合
    public static List<String> list=new ArrayList<String>();
    public static void main(String[] args) {
        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    list.add(String.valueOf(i));
                }
                for (String v:list) {
                    System.out.printf("%s ",v);
                }
                System.out.println();
            }
        }).start();

        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 10; i < 20; i++) {
                    list.add(String.valueOf(i));
                }
                for (String v:list) {
                    System.out.printf("%s ",v);
                }
                System.out.println();
            }
        }).start();
```

```
    }
结果：
 出现了异常：java.util.ConcurrentModificationException

* CopyOnWriteArrayList
 * public static List<String> list=new CopyOnWriteArrayList<String>();
某次的结果：
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

```

* 能够掌握并发集合类之ConcurrentHashMap （HashMap）

```
* 多个线程同时操作并且遍历集合
  * HashMap
  public static Map<String,String> map=new HashMap<String,String>();
    public static void main(String[] args) {
        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    map.put(String.valueOf(i), String.valueOf(i));
                }
                Set<String> keys = map.keySet();
                for(String key:keys) {
                    System.out.printf("%s ",key);
                }
                System.out.println();
            }
        }).start();

        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 10; i < 20; i++) {
                    map.put(String.valueOf(i), String.valueOf(i));
                }
                Set<String> keys = map.keySet();
```

```
                for(String key:keys) {
                    System.out.printf("%s ",key);
                }
                System.out.println();
            }
        }).start();
    }
结果：
 出现了异常：java.util.ConcurrentModificationException

* ConcurrentHashMap
 public static Map<String,String> map=new ConcurrentHashMap<String,String>();
* 某次运行结果：
 0 11 12 13 14 15 16 17 18 1 19 0 1 2 3 4 5 6 7 8 9 10 13
 2 14 3 15 4 16 5 17 6 18 7 19 8 9 10 （结果不确定）
```

* 能够掌握并发集合类之ConcurrentSkipListMap（TreeMap）

```
* TreeMap（数要大点才能测试异常）
 测试：
 public static Map<String,String> map=new TreeMap<String,String>();
    public static void main(String[] args) {
        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    map.put(String.valueOf(i), String.valueOf(i));
                }
                Set<String> keys = map.keySet();
                for(String key:keys) {
                    System.out.printf("%s ",key);
                }
                System.out.println();
            }
        }).start();

        new Thread(new Runnable() {
```

```
21
22              @Override
23              public void run() {
24                  for (int i = 10; i < 2000; i++) {
25                      map.put(String.valueOf(i), String.valueOf(i));
26                  }
27                  Set<String> keys = map.keySet();
28                  for(String key:keys) {
29                      System.out.printf("%s ",key);
30                  }
31                  System.out.println();
32              }
33          }).start();
34      }
35
36  结果:
37      出现异常：java.util.ConcurrentModificationException
38
39  * ConcurrentSkipListMap
40    public static Map<String,String> map=
41    new ConcurrentSkipListMap<String,String>();
42    * 使用ConcurrentSkipListMap替换
```

* 能够掌握并发集合类之 ConcurrentSkipListSet （ TreeSet ）

* 能够掌握并发集合类之CopyOnWriteArraySet （ HashSet ）

```
1   * 测试HashSet
2  public static Set<String> set=new HashSet<String>();
3      public static void main(String[] args) {
4          new Thread(new Runnable() {
5
6              @Override
7              public void run() {
8                  for (int i = 0; i < 10; i++) {
9                      set.add(String.valueOf(i));
10                 }
11                 for (String v:set) {
12                     System.out.printf("%s ",v);
```

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

        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 10; i < 20; i++) {
                    set.add(String.valueOf(i));
                }
                for (String v:set) {
                    System.out.printf("%s ",v);
                }
                System.out.println();
            }
        }).start();
    }

结果:
    出现异常：java.util.ConcurrentModificationException、
 * CopyOnWriteArraySet
  * public static Set<String> set=new CopyOnWriteArraySet<String>();
```

* 能够掌握并发集合类之ConcurrentLinkedDeque（LinkedList）

```
 * LinkedList做双向队列来使用
 public static Deque<String> deque=new LinkedList<String>();
    public static void main(String[] args) {
        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    deque.add(String.valueOf(i));
                }
                for (String v:deque) {
                    System.out.printf("%s ",v);
```

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

        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 10; i < 20; i++) {
                    deque.add(String.valueOf(i));
                }
                for (String v:deque) {
                    System.out.printf("%s ",v);
                }
                System.out.println();
            }
        }).start();
    }
* 出现异常：java.util.ConcurrentModificationException

* ConcurrentLinkedDeque
    * public static Deque<String> deque=new ConcurrentLinkedDeque<String>();

* ConcurrentLinkedQueue 这个是单向
```