集合框架

集合和数组的相同点：
都是用来对多个数据进行存储的结构，简称java容器。

数组的缺点：
1.一旦初始化完成长度就确定了。
2.数组中的方法有限，对于添加，删除，修改等操作时非常不便，而且效率不高。
3.数组中没有现成的方法来获取数组中实际的元素个数。
4.数组中存储的数据是有序、可重复的，且数据类型要相同。

- Collection接口
  是一个单列集合，用来存储一个一个的对象。
  - List接口
    存储有序、可重复的数据。
    - ArrayList
      - jdk1.2
      - List接口的主要实现类，线程不安全，效率高
      - 底层
        - 底层使用Object[ ] elementDate 存储数据
          - 数组长度10
            - jdk7

```
public ArrayList() {
    this(10);
}
```

            - jdk8

```
public ArrayList() {
    super();
    this.elementData = EMPTY_ELEMENTDATA;
      //EMPTY_ELEMENTDATA = { }
}
```

          - 扩容1.5倍

```
grow()关键代码：
// oldCapacity  = 10
    int newCapacity = oldCapacity + (oldCapacity >> 1);
```

        - 源码分析

- jdk7

```
@Test
public void test1() {
    ArrayList arrayList = new ArrayList();  // 底层创建了长度为10的Object[]数组elementDate
    arrayList.add(123); // 等于 elementDate[0] = new Integer(123); --->自动装箱
/*          ...
 *          ...
 *      多次add()操作
 *          ...
 *          ...
 */
    arrayList.add(123); // 假设这次add(123)操作导致底层数组elementDate的容量不足，则扩容。默认情况下，数组长度扩容为原来的1.5倍，同时将旧数组中的数据复制到新数组中。
}
```

- jdk8

```
@Test
public void test2(){
    ArrayList arrayList = new ArrayList();  // 此时底层的Object[]数组elementDate被初始化为{},并没有创建长度为10的数组
    arrayList.add(123); //第一次进行add()操的时候才创建
}
```

- LinkedList
  - jdk1.2
  - 底层
    - 底层使用双向链表存储数据，对于频繁的插入删除操作，效率比ArrayList高
    - 源码分析

```
@Test
public void test2(){
    LinkedList linkedList = new LinkedList();// 此时内部声明了Node类型的first和last，默认值为null。
    linkedList.add(123); // 此时将调用linkLast()方法，将123封装到Node中
}

Node源码：
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;
    }
}
```

- Vector
  - jdk1.0
  - List接口的古老实现类，线程安全，效率低
  - 底层
    - 底层使用Object[ ] elementDate 存储数据
      - 数组长度10

        ```
        public Vector() {
            this(10);
        }
        ```

      - 扩容2倍

        ```
        grow()关键代码：
        // oldCapacity  = 10
        int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                         capacityIncrement : oldCapacity);
        ```

  - List的常用方法
    - void add(int index, Object obj)

      在index位置添加obj元素。

    - boolean addAll(int index, Collection coll)

      在index位置开始添加coll中的所有元素。

    - Object get(int index)

      获取指定index位置的元素。

    - int indexOf(Object obj)

      返回obj元素首次出现的位置。

    - int lastIndexOf(Object obj)

      返回obj元素末次出现的位置。

    - Object remove(int index)

      删除index位置的元素，并返回。

    - Object set(int index, Object obj)

      设置index位置的元素为obj。

    - List subList(int fromIndex, int toIndex)

      返回从fromIndex开始到toIndex结束的子集合。

- Set接口

  存储无序的，不可重复的数据。所以向Set添加数据时一定要重写其所在类的hashCode()方法和equals()方法。

  - 无序性和不可重复性

- 无序性

  存储的数据在底层数组中并非按照数组的索引顺序添加，而是根据数据的哈希值来判断数据在数组中的位置。

- 不可重复性

  保证添加的元素按照equals()判断时，不能返回true，即相同的元素只能添加一个。

- HashSet
  - jdk1.2
  - Set接口的主要实现类，线程不安全，可以存储null值
  - 底层
    - 实际上创建的是hashMap

      ```
      public HashSet() {
          map = new HashMap<>();
      }
      ```

    - 元素的添加过程

      假设向HashSet中添加元素a，会首先调用a所在类的hashCode()方法来计算a的哈希值，再通过某种算法(indexFor()方法：&操作)计算出a在HashSet底层数组中的存放位置，然后判断该位置上是否已经有元素：
      如果位置上没有其他元素：添加成功。①
      如果位置上有其他元素（或者是以链表形式存在的多个元素），则比较二者的哈希值：
      哈希值不同：添加成功。②
      哈希值相同，调用equals()方法：
      equals()方法返回true：添加失败。
      equals()方法返回false：添加成功。③

      对于添加成功的②③情况：元素a与已经在指定索引位置上的数据以链表的方式存储。并且：
      jdk7中：元素a放在数组中，指向原有的元素。
      jdk8中：原有的元素放在数组中，指向元素a
  - LinkedHashSet
    - jdk1.4
    - HashSet的子类，对于频繁的遍历操作，效率比HashSet高
    - 底层

      LinkedHashSet在遍历其内部数据时可以根据添加的顺序遍历，原因是在添加数据的同时为每个数据维护了两个引用，记录了此数据的前一个和后一个元素。

- 实际上创建的是LinkedHashMap

```
LinkedHashSet构造器：
public LinkedHashSet() {
    super(16, .75f, true);
}

super是HashSet的构造器：
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
    map = new LinkedHashMap<>(initialCapacity, loadFactor);
}
```

- TreeSet
  - jdk1.2
  - 两种排序方法

    可以按照添加对象的指定属性进行排序。
    - 自然排序
      - Comparable接口
    - 定制排序
      - Comparator接口
  - 底层
    - 实际上创建的是TreeMap

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

- Collection接口的方法
  - add(Object obj)

    将元素obj添加到集合中。
  - size()

    获取添加元素的个数。
  - addAll(Collection coll)

    将coll中的元素添加到当前集合中。
  - clear()

    清空当前集合元素。
  - isEmpty()

    判断集合是否为空。
  - contains(Object obj)

    判断当前集合中是否包含元素obj。
  - containsAll(Collection coll)

    判断形参coll中的元素是否都存在于当前集合中，存在的返回true，不存在的返回false。

- remove(Object obj)

  从当前集合中移除元素obj，成功返回true，失败返回false。

- removeAll(Collection coll)

  从当前集合中移除coll中所有元素，成功返回true，失败返回false。

- retainAll(Collection coll)

  求两个集合的交集，并返回给当前集合。

- equals(Object obj)

  判断两个集合的元素是否都相等。

- hashCode()

  返回当前对象的哈希值。

- toArray()

  将集合转为数组。

- iterator()

  返回Iterator接口的实例，用于遍历集合。

  - Iterator接口

    - 每调用一次Iterator()方法都会得到一个全新的迭代器对象
    - 两个方法

      - next()

        调用该方法时先将指针下移，在返回其元素。

      - hasNext()

        判断是否由下一个元素。

    - 遍历集合的三种方法（以ArrayList为例）

- 普通for循环

```
//以数组为例
@Test
public void test1(){
    String[] s = new String[]{"aa","bb","cc","dd","ee"};
    for (int i = 0; i < s.length; i++) {
        System.out.println(s[i]);
    }
}

@Test
//以ArrayList为例
public void test2(){
    Collection collection = new ArrayList();
    collection.add(1);
    collection.add("a");
    collection.add(3);
    collection.add("b");
    collection.add(5);

    Object[] objects = collection.toArray();

    for (int i = 0; i < objects.length; i++) {
        System.out.println(objects[i]);
    }
}
```

- 增强for循环

```
@Test
//以数组为例
public void test1(){
    String[] s = new String[]{"aa","bb","cc","dd","ee"};
    for (String ss :
            s) {
        System.out.println(ss);
    }
}

@Test
//以ArrayList为例
public void test2(){
    Collection collection = new ArrayList();
    collection.add(1);
    collection.add("a");
    collection.add(3);
    collection.add("b");
    collection.add(5);

    for (Object o :
            collection) {
        System.out.println(o);
    }
}
```

- 迭代器Iterator

```
@Test
//以ArrayList为例
public void test(){
    Collection collection = new ArrayList();
    collection.add(1);
    collection.add("a");
    collection.add(3);
    collection.add("b");
    collection.add(5);

    Iterator iterator = collection.iterator();

    while(iterator.hasNext()){
        System.out.println(iterator.next());
    }
}
```

- Map接口

  双列集合，用来存储一对(key,value)的对象。

  - HashMap
    - jdk1.2
    - Map的主要实现类，线程不安全，效率高，可以存储null的key和value
    - 底层
      - key和value
        - key

          无序的，不可重复的，使用Set来存储所有的key。
          要求：要重写key所在类要重写equals()和hashCode()方法，若是TreeMap则要重写CompareTo()方法。

        - value

          无序，可重复的，使用Collection存储所有的value。

      - 源码分析
        - jdk7与jdk8的异同
          - 底层结构
            - 都涉及到的几个常量

              DEFAULT_INITIAL_CAPACITY // HashMap的默认容量：16
              DEFAULT_LOAD_FACTOR // 默认加载因子：0.75
              threshold // 扩容的临界值：0.75 * 16 = 12
              TREEIFY_THRESHOLD // 链表中的元素数量大于该值，转为红黑树：8
              MIN_TREEIFY_CAPACITY // Node被树化时要求最小的数组容量：64

            - jdk7
              - Entry[]

                Entry[]

- 子主题 2
- 默认容量16

  在初始化时就设置容量为16，加载因子为0.75,并计算扩容临界值。

  ```
  public HashMap() {
      this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR);
  }
  ```

  HashMap(int initialCapacity, float loadFactor)计算临界值代码：

  ```
  this.loadFactor = loadFactor;
  threshold = (int)Math.min(capacity * loadFactor,
  MAXIMUM_CAPACITY + 1);
  ```

- 元素的添加过程

  ```
  HashMap hashMap = new HashMap();    // 实例化后底层会创建一个长度为16的数组Entry[] table
  /*          ...
   *          ...
   *      多次put()操作
   *          ...
   *          ...
   */
  hashMap.put(key1,value1);  // 此时会首先调用key1所在类的hashCode()方法计算其哈希值，再经过某种算法(indexFor()方法：&操作)计算后得到Entry在数组中的存放位置,然后判断该位置上是否已经有元素：
      如果位置上数据为空：key1-value1添加成功。①
      如果位置上数据不为空（意味着此位置上有一个或多个数据以链表形式存在），则比较key1和已存在数据的哈希值：
          key1的哈希值与所有已存在数据的哈希值都不同：添加成功。②
          key1的哈希值与其中某一个数据(key2,value2)的哈希值相同，调用key1.equals(key2)方法：
              equals()方法返回true：添加失败。
              equals()方法返回false：添加成功。③

  对于情况②③，此时的key1-value1和原来的数据以链表的形式存储。
  ```

- 扩容2倍

  jdk7在添加元素时会在put()方法中调用addEntry()方法，此方法中会判断当前的数组中存在的元素个数是否超过了扩容临界值12，并且存放的位置非空。满足条件则调用resize()方法，将容量扩大为原来的两倍，并将原有数据复制到新的数组中。

  ```
  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);
      }
  ```

- 链表

- jdk8
  - Node[]

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
```

    - 初始化

      jdk8中初始化HashMap时并没有直接创建一个长度为16的数组，而是先将加载因子loadFactor先赋值为DEFAULT_LOAD_FACTOR = 0.75。

```
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields
defaulted
}
```

    - 默认容量16

      在首次调用put()方法时调用其中的putVal()方法，将默认容量设置为16，并且计算扩容的临界值。

      put()方法：

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

      putVal()方法关键代码：

```
newCap = DEFAULT_INITIAL_CAPACITY;  // 默认容量16
newThr = (int)(DEFAULT_LOAD_FACTOR *
DEFAULT_INITIAL_CAPACITY);  // 加载因子 * 默认容量 = 12
```

    - 元素的添加过程

      和jdk7中类似。

    - 扩容2倍

      jdk8在添加元素时会在put()方法中调用putVal()方法，此方法中会判断当前的数组中存在的元素个数是否超过了扩容临界值12，满足条件则调用resize()方法，将容量扩大为原来的两倍，并将原有数据复制到新的数组中。

      putVal()方法中判断是否超过扩容值代码：

```
if (++size > threshold)
    resize();
```

      resize()扩容关键代码：

```
else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
         oldCap >= DEFAULT_INITIAL_CAPACITY)
    newThr = oldThr << 1; // double threshold
```

  - 链表

- 红黑树

  如果链表上存在的元素大于8，且数组的长度大于64时，将链表转换为红黑树结构。

  ```
  if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for
  1st,TREEIFY_THRESHOLD  = 8
      treeifyBin(tab, hash);

  treeifyBin()关键源码：  // MIN_TREEIFY_CAPACITY = 64
  if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
      resize(); // 数组长度没达到64，则仅进行扩容
  ```

- LinkedHashMap
  - jdk1.4
  - 底层

    为每个数据维护了两个Entry类型的引用before和after，分别记录了此数据的前一个和后一个元素。

    ```
    static class Entry<K,V> extends HashMap.Node<K,V> {
        Entry<K,V> before, after;
        Entry(int hash, K key, V value, Node<K,V> next) {
            super(hash, key, value, next);
        }
    }
    ```

- TreeMap
  - jdk1.2
  - 两种排序方法

    向TreeMap中添加的key-value，要求key必须是由同一个类创建的对象，且按照key来进行排序。

    - 自然排序
      - Comparable接口

        实现Comparable接口并重写compareTo方法。

    - 定制排序

- Comparator接口

```
@Test
public void test2() {
    new TreeMap(new Comparator() {
        @Override
        public int compare(Object o1, Object o2) {
            if (o1 instanceof Xxx && o2 instanceof Xxx){
                Xxx x1 = (Xxx)o1;
                Xxx x2 = (Xxx)o2;
                /*

                ...比较操作...

                 */
            }
        }
    })
}
```

- HashTable
  - jdk1.0
  - Map的古老实现类，线程安全，效率低，不能存储null的key和value
  - Properties
    - jdk1.0
    - 常用来处理配置文件，其key和value都是String类型
- Map接口的常用方法
  - 增、删、改操作
    - Object put(Object key，Object value)

      将指定key-value添加(或修改)到指定map中。
    - void putAll(Map m)

      将m中的所有key-value放到当前map中。
    - Object remove(Object key)

      移除指定key的key-value对，并返回其value。
    - void clear()

      清空当前map中的数据。
  - 元素查询操作
    - Object get(Object key)

      获取指定key对应的value。
    - boolean containsKey(Object key)

      返回是否包含指定key。

- boolean containsValue(Object value)
  返回是否包含指定value。
- int size()
  返回map中key-value对的个数。
- boolean isEmpty()
  判断当前map是否为空。
- boolean equals(Object obj)
  判断当前map和参数对象obj是否相等。

- 元视图操作
  - Set keySet()
    返回所有key构成的集合。
  - Collection values()
    返回所有value构成的集合。
  - Set entrySet()
    返回所有key-value对构成的集合。

- 是操作Collection和Map的工具类
- 方法
  - 排序
    - reverse(List list)
      反转list中的元素顺序。
    - shuffle(List list)
      对list集合元素随机排序。
    - sort(List list)
      根据元素的自然顺序来对指定list集合的元素按照升序排序。
    - sort(List list， Comparator)
      根据指定的Comparator顺序来对指定list集合的元素进行排序。
    - swap(List list, int i, int j)
      将指定list中的i,j位置上的元素进行互换。
  - 查找、替换
    - Object max(Collection coll)
      根据元素的自然顺序,返回coll中最大元素。
    - Object max(Collection coll, Comparator)
      根据指定的Comparator顺序,返回coll中最大元素。
    - Object min(Collection coll)
      根据元素的自然顺序,返回coll中最小元素。

- Object min(Collection coll, Comparator)

  根据指定的Comparator顺序,返回coll中最小元素。

- int frequency(Collection coll, Object obj)

  返回集合coll中元素obj的出现次数。

- void copy(List dest, List src)

  将src中的内容复制到dest中。

- boolean replaceAll(List list, Object oldVal, Object newVal)

  使用newVal替换list中的oldVal。

- 同步控制

  - synchronizedXxx()

    解决Collection,List,Set和Map的线程不安全问题。

    ```
    @Test
    public void test1() {
        ArrayList arrayList = new ArrayList;    // 此时的arrayList是线程不安全的
        List list = Collections.synchronizedList(arrayList);    // 此时的list是线程安全的
    }
    ```

- 把键值对中(key-value)变为了(key-null)