

## 集合

**1) 可以动态保存任意多个对象，使用比较方便！**  
**2) 提供了一系列方便的操作对象的方法：add、remove、set、get等**  
**3) 使用集合添加,删除新元素的示意代码- 简洁了**

1. 集合主要是两组(单列集合 ， 双列集合)
2. Collection 接口有两个重要的子接口 List Set ， 他们的实现子类都是单列集合
3. Map 接口的实现子类 是双列集合，存放的 K-V、

Collection 接口：

**1) collection实现子类可以存放多个元素，每个元素可以是Object**  
**2) 有些Collection的实现类，可以存放重复的元素，有些不可以**  
**3) 有些Collection的实现类，有些是有序的(List)，有些不是有序(Set)**  
**4) Collection接口没有直接的实现子类，是通过它的子接口Set 和 List 来实现的**

迭代器：

1. iterator 对象成为迭代器。只要用于遍历 collection 集合之中的元素
2. 所有实现了 collection 接口的集合都有一个 iterator () 方法，用以返回一个实现了 iterator 接口的对象，即返回一个迭代器。
3. iterator 仅用于遍历集合，本身不存放对象。

1. 先得到 col 对应的 迭代器

```
Iterator iterator = col.iterator();
```

2. 使用 while 循环遍历

```
while (iterator.hasNext()) {判断是否还有数据
```

返回下一个元素，类型是 Object

```
Object obj = iterator.next();下移并且将下移以后集合位置上的元素返回
```

```
System.out.println("obj=" + obj);
```

```
}
```

for 循环增强：

增强for循环，可以代替iterator迭代器，特点：增强for就是简化版的iterator，本质一样。只能用于遍历集合或数组。

➤ 基本语法

```
for(元素类型 元素名 : 集合名或数组名) {  
    访问元素  
}
```

List 接口：

1. List 集合类中元素有序(即添加顺序和取出顺序一致)、且可重复
2. List 集合中的每个元素都有其对应的顺序索引，即支持索引

常用方法：

void add(int index, Object ele):在 index 位置插入 ele 元素

boolean addAll(int index, Collection eles):从 index 位置开始将 eles 中的所有元素添加进来

Object get(int index):获取指定 index 位置的元素

int lastIndexOf(Object obj):返回 obj 在当前集合中末次出现的位置

int indexOf(Object obj):返回 obj 在集合中首次出现的位置

Object remove(int index):移除指定 index 位置的元素，并返回此元素

Object set(int index, Object ele):设置指定 index 位置的元素为 ele , 相当于是替换。

List subList(int fromIndex, int toIndex):返回从 fromIndex 到 toIndex 位置的子集合

List 三种遍历方式：

1. 使用遍历器
2. 使用增强 for 循环
3. 使用普通 for 循环

ArrayList：

**ArrayListDetail.java**

- 1) permits all elements, including null , ArrayList 可以加入null,并且多个
- 2) ArrayList 是由数组来实现数据存储的[后面老师解读源码]
- 3) ArrayList 基本等同于Vector , 除了 ArrayList是线程不安全(执行效率高) 看源码. 在多线程情况下，不建议使用ArrayList

### ArrayListSource.java , 先说结论 , 在分析源码(示意图)

- 1) ArrayList中维护了一个Object类型的数组elementData. [debug 看源码]  
`transient Object[] elementData;` //transient 表示瞬间,短暂的, 表示该属性不会被序列化
- 2) 当创建ArrayList对象时, 如果使用的是无参构造器, 则初始elementData容量为0, 第1次添加, 则扩容elementData为10, 如需要再次扩容, 则扩容elementData为1.5倍。
- 3) 如果使用的是指定大小的构造器, 则初始elementData容量为指定大小, 如果需要扩容, 则直接扩容elementData为1.5倍。

老师建议: 自己去debug 一把我们的ArrayList的创建和扩容的流程.

Vector:

### 1) Vector类的定义说明

```
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, Serializable
```

2) Vector底层也是一个对象数组, `protected Object[] elementData;`

3) Vector 是线程同步的, 即线程安全, Vector类的操作方法带有**synchronized**

```
public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    return elementData(index);
}
```

4) 在开发中, 需要线程同步安全时, 考虑使用Vector

Vector 与 arraylist

	底层结构	版本	线程安全 (同步) 效率	扩容倍数
ArrayList	可变数组	jdk1.2	不安全, 效率高	如果有参构造1.5倍 如果是无参 1.第一次10 2.从第二次开始按1.5扩
Vector	可变数组 Object[]	jdk1.0	安全, 效率不高	如果是无参, 默认10 , 满后, 就按2倍扩容  如果指定大小, 则每次直接按2倍扩容.

Linkedlist:

1. 底层实现了双向链表和双端队列
2. 可以添加任意元素, 元素可以重复, 包括 null
3. 线程不安全

Linkedlist 底层操作机制:



1. 底层维护了双向链表
2. 维护了两个属性 first 和 last 分别指向首节点和尾节点
3. 每个节点 (node 对象) 之中有 prev, next, item 三个属性, prev 指向前一个对象, next 指向后一个对象, 实现双向链表
4. 其添加删除效率较高

Arraylist 和 linkedlist

1. 改查操作较多, 选择 arraylist
2. 增删操作较多, 选择 linkedlist

Set 接口:

1. 无序 (添加和取出顺序不同), 没有索引
2. 不允许重复元素, 最多只能包含一个 null
3. 注意: 取出的顺序的顺序虽然不是添加的顺序, 但是他的固定

常用方法: 和 List 接口一样, Set 接口也是 Collection 的子接口, 因此, 常用方法和 Collection 接口一样.

遍历方法:

1. 使用遍历器
2. 使用增强 for 循环

Hashset:



**HashSet java**

- 1) HashSet实现了Set接口
- 2) HashSet实际上是HashMap, 看下源码. (图)

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

- 3) 可以存放null值, 但是只能有一个null
- 4) HashSet不保证元素是有序的, 取决于hash后, 再确定索引的结果. (即, 不保证存放元素的顺序和取出顺序一致)
- 5) 不能有重复元素/对象. 在前面Set 接口使用已经讲过

扩容机制:

1. HashSet 底层是 HashMap, 第一次添加时, table 数组扩容到 16, 临界值 (threshold) 是  $16 * \text{加载因子} (\text{loadFactor}) = 12$ , 如果 table 数组使

用到了临界值 12, 就会扩容到  $16 * 2 = 32$ , 新的临界值就是  $32 * 0.75 = 24$ , 依次类推。

2. 在 Java8 中, 如果一条链表的元素个数到达 TREEIFY\_THRESHOLD(默认是 8), 并且 table 的大小  $\geq$  MIN\_TREEIFY\_CAPACITY(默认 64), 就会进行树化(红黑树), 否则仍然采用数组扩容机制

3. 当我们向 HashSet 增加一个元素,  $\rightarrow$  Node  $\rightarrow$  加入 table, 就算是增加了一个 size++。

添加元素:



Linkedhashset:

1. 是 HashSet 的子类
2. 底层是 LinkedHashMap, 维护了一个数组和双向链表
3. 根据元素的 hashCode 确定元素的存储位置, 使用链表维护元素的次序, 使得元素看起来是以插入顺序保存
4. 不允许添加重复元素

Map 接口:

1. Map 与 Collection 并列存在。用于保存具有映射关系的数据:Key-Value(双列元素)
2. Map 中的 key 和 value 可以是任何引用类型的数据, 会封装到 HashMap\$Node 对象中
3. Map 中的 key 不允许重复, 原因和 HashSet 一样, 前面分析过源码。
4. Map 中的 value 可以重复
5. Map 的 key 可以为 null, value 也可以为 null, 注意 key 为 null, 只能有一个, value 为 null, 可以多个

6. 常用 String 类作为 Map 的 key
7. key 和 value 之间存在单向一对一关系, 即通过指定的 key 总能找到对应的 value
8. map 存放 k-v 在 hashmap\$node 之中, 因为 node 实现了 entry 接口。

遍历方法:

1. 先取出 所有的 Key , 通过 Key 取出对应的 Value
2. 把所有的 values 取出
3. 通过 EntrySet 来获取 k-v

HashMap:

- 1) Map接口的常用实现类: HashMap、Hashtable和Properties。
- 2) HashMap是 Map 接口使用频率最高的实现类。
- 3) HashMap 是以 key-val 对的方式来存储数据(HashMap\$Node类型) [案例 Entry ]
- 4) key 不能重复, 但是值可以重复, 允许使用null键和null值。
- 5) 如果添加相同的key, 则会覆盖原来的key-val , 等同于修改。(key不会替换, val会替换)
- 6) 与HashSet一样, 不保证映射的顺序, 因为底层是以hash表的方式来存储的。(jdk8的 hashMap 底层 数组+链表+红黑树)
- 7) HashMap没有实现同步, 因此是线程不安全的, 方法没有做同步互斥的操作, 没有 synchronized

- 1) HashMap底层维护了Node类型的数组table, 默认为null
- 2) 当创建对象时, 将加载因子(loadfactor)初始化为0.75.
- 3) 当添加key-val时, 通过key的哈希值得到在table的索引。然后判断该索引处是否有元素, 如果没有元素直接添加。如果该索引处有元素, 继续判断该元素的key和准备加入的key是否相等, 如果相等, 则直接替换val; 如果不相等需要判断是树结构还是链表结构, 做出相应处理。如果添加时发现容量不够, 则需要扩容。
- 4) 第1次添加, 则需要扩容table容量为16, 临界值(threshold)为12 (16\*0.75)
- 5) 以后再扩容, 则需要扩容table容量为原来的2倍(32), 临界值为原来的2倍, 即24, 依次类推。
- 6) 在Java8中, 如果一条链表的元素个数超过 TREEIFY\_THRESHOLD(默认是 8), 并且 table的大小 >= MIN\_TREEIFY\_CAPACITY(默认64), 就会进行树化(红黑树)

Hashtable:

1. 存放的元素是键值对, k-v
2. 键和值都不可为 null, 否则会抛出异常
3. 使用方法和 hashmap 基本相同
4. 线程安全, hashmap 线程不安全

Properties 类:

1. Properties 继承 Hashtable
2. 可以通过 k-v 存放数据, 当然 key 和 value 不能为 null

3. 还可以用于 `xx.properties` 文件之中，加载数据到 `properties` 类对象，进行读取与修改

4. `xx.properties` 可以作为配置文件

`Treeset`:

1. 当我们使用无参构造器，创建 `TreeSet` 时，仍然是无序的
2. 使用 `TreeSet` 提供的一个构造器，可以传入一个比较器(匿名内部类)，并指定排序规则

`Collection` 工具类:

**1) Collections 是一个操作 Set、List 和 Map 等集合的工具类**  
**2) Collections 中提供了一系列静态的方法对集合元素进行排序、查询和修改等操作**

方法:

`reverse(List)`: 反转 `List` 中元素的顺序

`shuffle(List)`: 对 `List` 集合元素进行随机排序

`sort(List)`: 根据元素的自然顺序对指定 `List` 集合元素按升序排序

`sort(List, Comparator)`: 根据指定的 `Comparator` 产生的顺序对 `List` 集合元素进行排序

`swap(List, int, int)`: 将指定 `list` 集合中的 `i` 处元素和 `j` 处元素进行交换

`Object max(Collection)`: 根据元素的自然顺序，返回给定集合中的最大元素

`Object max(Collection, Comparator)`: 根据 `Comparator` 指定的顺序，返回给定集合中的最大元素

`int frequency(Collection, Object)`: 返回指定集合中指定元素的出现次数

`boolean replaceAll(List list, Object oldVal, Object newVal)`: 使用新值替换 `List` 对象的所有旧值