Java 基础：集合入门（0）

集合框架

Java 中的集合框架大类可分为 Collection 和 Map；两者的区别：

- Collection 是单列集合；Map 是双列集合；
- Collection 中只有 Set 系列要求元素唯一；Map 中键需要唯一，值可以重复；
- Collection 的数据结构是针对元素的；Map 的数据结构是针对键的。

泛型

在说两大集合体系之前先说说泛型，因为在后面的集合中都会用到；

所谓的泛型就是：类型的参数化

泛型是类型的一部分，类名+泛型是一个整体

如果有泛型，不使用时，参数的类型会自动提升成 Object 类型，如果再取出来的话就需要向下强转，就可能发生类型转化异常(ClassCaseException)；不加泛型就不能在编译期限定向集合中添加元素的类型，导致后期的处理麻烦。

加了泛型和不加泛型的区别：

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Test {
    public static void main(String[] args) {
        // 不加泛型，添加和遍历
        List list = new ArrayList<>();
        list.add(1);
        list.add("123");
        list.add("hello");
        Iterator it = list.iterator();
        while(it.hasNext()){
            // 没有添加泛型，这里只能使用 Object 接收
            Object obj = it.next();
            System.out.println(obj);
        }
    }
```

```
}


import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Test {
    public static void main(String[] args) {
        // 加泛型，添加和遍历
        List<String> list = new ArrayList<String>();
        list.add("123");
        list.add("hello");
        Iterator<String> it = list.iterator();
        while(it.hasNext()){
            // 因为添加了泛型，就说明集合中装的全部都是 String 类型的数据
            // 所以这里用 String 类型接收，就不会发生异常，并且可以使用 String 的方法
            String str = it.next();
            System.out.println(str.length());
        }
    }
}
```

自定义带泛型的类:

```
public class Test {
    // 自定义一个带有一个参数的泛型类,可以向传入什么类型就传入什么类型
    public static void main(String[] args) {
        // 进行测试，传入一个 String 对象
        Person<String> perStr = new Person<String>();
        perStr.setT("我是字符串");
        String str = perStr.getT();
        System.out.println(str);
        // 进行测试，传入一个 Integer 对象
        Person<Integer> perInt = new Person<Integer>();
        perInt.setT(100);
        Integer intVal = perInt.getT();
        System.out.println(intVal);
    }
}
//自定义一个带有一个参数的泛型类
class Person<T>{
    private T t;
```

```
    void setT(T t){
        this.t = t;
    }
    T getT(){
        return t;
    }
}
```

实现带有泛型的接口类型：

实现接口的同时，指定了接口中的泛型类型. (定义类时确定)；

```
public class GenericImpl1 implements GenericInter<String> {}
```

实现接口时，没有指定接口中的泛型类型.此时，需要将该接口的实现类定义为泛型类.接口的类型需要在创建实现类对象时才能真正确定其类型. (始终不确定类型，直到创建对象时确定类型);

```
public class GenericImpl2<T> implements GenericInter<T> {}
```

泛型的通配符(?)：

上限限定：比如定义方法的时候出现，public void getFunc(List<? extends Animal> an)，那么表示这里的参数可以传入 Animal，或者 Animal 的子类；

下限限定: 比如定义方法的时候出现，public void getFunc(Set<? super Animal> an)，那么表示这里的参数可以传入 Animal，或者 Animal 的父类。

使用泛型的注意点:

1、泛型不支持基本数据类型

2、泛型不支持继承，必须保持前后一致 (比如这样是错误的：List<Object> list = new ArrayList<String>();

Collection 体系

collection 包括两大体系，List 和 Set

List 的特点：

存取有序，有索引，可以根据索引来进行取值，元素可以重复

Set 的特点：

存取无序，元素不可以重复

List

下面有 ArrayList，LinkedList，Vector(已过时)

集合的的最大目的就是为了存取；List 集合的特点就是存取有序，可以存储重复的元素，可以用下标进行元素的操作。

ArrayList

底层是使用数组实现，所以查询速度快，增删速度慢。

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Test {
    // 使用 ArrayList 进行添加和遍历
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("接口 1");
        list.add("接口 2");
        list.add("接口 3");
        // 第一种遍历方式,使用迭代器
        Iterator<String> it = list.iterator();
        while(it.hasNext()){
            String next = it.next();
            System.out.println(next);
        }
        System.out.println("-------------------");
        // 第二种遍历方式，使用 foreach
        for (String str : list){
            System.out.println(str);
        }
    }
}
```

LinkedList

基于链表结构实现的，所以查询速度慢，增删速度快，提供了特殊的方法，对头尾的元素操作（进行增删查）。

使用 LinkedList 来实现栈和队列；栈是先进后出，而队列是先进先出。

```
import java.util.LinkedList;
/**
 * 利用 LinkedList 来模拟栈
 * 栈的特点：先进后出
 */
 class MyStack {
    private LinkedList<String> linkList = new LinkedList<String>();
```

```
    // 压栈
    public void push(String str){
        linkList.addFirst(str);
    }
    // 出栈
    public String pop(){
        return linkList.removeFirst();
    }
    // 查看
    public String peek(){
        return linkList.peek();
    }
    // 判断是否为空
    public boolean isEmpty(){
        return linkList.isEmpty();
    }
}

public class Test {
    public static void main(String[] args) {
        // 测试栈
        StackTest stack = new StackTest();
        stack.push("我是第 1 个进去的");
        stack.push("我是第 2 个进去的");
        stack.push("我是第 3 个进去的");
        stack.push("我是第 4 个进去的");
        stack.push("我是第 5 个进去的");
        // 取出
        while (!stack.isEmpty()){
            String pop = stack.pop();
            System.out.println(pop);
        }
        // 打印结果
        /*我是第 5 个进去的
        我是第 4 个进去的
        我是第 3 个进去的
        我是第 2 个进去的
        我是第 1 个进去的*/
    }
}
```

LinkedList 实现 Queue:

```
import java.util.LinkedList;
/**
 * 利用 linkedList 来实现队列
 * 队列：先进先出
 */
class QueueTest {
    private LinkedList<String> link = new LinkedList<String>();
    // 放入
    public void put(String str){
        link.addFirst(str);
    }
    // 获取
    public String get(){
        return link.removeLast();
    }
    // 判断是否为空
    public boolean isEmpty(){
        return link.isEmpty();
    }
}

public class Test {
    public static void main(String[] args) {
        // 测试队列
        QueueTest queue = new QueueTest();
        queue.put("我是第 1 个进入队列的");
        queue.put("我是第 2 个进入队列的");
        queue.put("我是第 3 个进入队列的");
        queue.put("我是第 4 个进入队列的");
        // 遍历队列
        while (!queue.isEmpty()){
            String str = queue.get();
            System.out.println(str);
        }
        // 打印结果
        /*我是第 1 个进入队列的
        我是第 2 个进入队列的
        我是第 3 个进入队列的
```

```
        我是第 4 个进入队列的*/
    }
}
```

Vector

因为已经过时，被 ArrayList 取代了；它还有一种迭代器通过 vector.elements()获取，判断是否有元素和取元素的方法为：hasMoreElements()，nextElement()。

```
import java.util.Enumeration;
import java.util.Vector;
public class Test {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<String>();
        vector.add("搜索");
        vector.add("vector");
        vector.add("list");
        Enumeration<String> elements = vector.elements();
        while (elements.hasMoreElements()){
            String nextElement = elements.nextElement();
            System.out.println(nextElement);
        }
    }
}
```

Set

Set 集合的特点：元素不重复，存取无序，无下标

Set 集合下面有：HashSet，LinkedHashSet，TreeSet

HashSet

HashSet 存储字符串:

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
public class Test {
    public static void main(String[] args) {
        // 利用 HashSet 来存取
        Set<String> set = new HashSet<String>();
        set.add("我的天");
```

```
        set.add("我是重复的");
        set.add("我是重复的");
        set.add("welcome");
        // 遍历 第一种方式 迭代器
        Iterator<String> it = set.iterator();
        while(it.hasNext()){
            String str = it.next();
            System.out.println(str);
        }
        System.out.println("--------------");
        for (String str : set){
            System.out.println(str);
        }
        // 打印结果，重复的已经去掉了
        /*我的天
        welcome
        我是重复的
        --------------
        我的天
        welcome
        我是重复的*/
    }
```

那哈希表是怎么来保证元素的唯一性的呢，哈希表是通过 hashCode 和 equals 方法来共同保证的。

哈希表的存储数据过程（哈希表底层也维护了一个数组）：

根据存储的元素计算出 hashCode 值，然后根据计算得出的 hashCode 值和数组的长度进行计算出存储的下标；如果下标的位置无元素，那么直接存储；如果有元素，那么使用要存入的元素和该元素进行 equals 方法，如果结果为真，则已经有相同的元素了，所以直接不存；如果结果假，那么进行存储，以链表的形式存储。

演示 HashSet 来存储自定义对象：

```
public class Person {
    // 属性
    private String name;
    private int age;
    // 构造方法
    public Person() {
        super();
    }
    public Person(String name, int age) {
```

```
        super();
        this.name = name;
        this.age = age;
    }
    // 要让哈希表存储不重复的元素，就必须重写 hasCode 和 equals 方法
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + age;
        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;
        Person other = (Person) obj;
        if (age != other.age)
            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 "Person [name=" + name + ", age=" + age + "]";
    }
    // getter & setter
    ...
}

import java.util.HashSet;
```

```
import java.util.Set;
public class Test {
    public static void main(String[] args) {
        // 利用 HashSet 来存取自定义对象 Person
        Set<Person> set = new HashSet<Person>();
        set.add(new Person("cy", 12));
        set.add(new Person("李四", 13));
        set.add(new Person("王五", 22));
        set.add(new Person("cy", 12));
        // 遍历
        for (Person p : set){
            System.out.println(p);
        }
        // 结果：向集合中存储两个 cy 对象，但是集合中就成功存储了一个
        /*Person [name=王五, age=22]
        Person [name=李四, age=13]
        Person [name=cy, age=12]*/
    }
}
```

所以在向 HashSet 集合中存储自定义对象时，为了保证 set 集合的唯一性，那么必须重写 hashCode 和 equals 方法。

LinkedHashSet

基于链表和哈希表共同实现的，所以具有存取有序，元素唯一

```
import java.util.LinkedHashSet;
public class Test {
    public static void main(String[] args) {
        // 利用 LinkedHashSet 来存取自定义对象 Person
        LinkedHashSet<Person> set = new LinkedHashSet<Person>();
        set.add(new Person("cy", 12));
        set.add(new Person("李四", 13));
        set.add(new Person("王五", 22));
        set.add(new Person("cy", 12));
        // 遍历
        for (Person p : set){
            System.out.println(p);
        }
        // 结果：向集合中存储两个 cy 对象，但是集合中就成功存储了一个，
```

```
        // 并且存进的顺序，和取出来的顺序是一致的
        /*Person [name=cy, age=12]
        Person [name=李四, age=13]
        Person [name=王五, age=22]*/
    }
}
```

TreeSet

特点：存取无序，元素唯一，可以进行排序（排序是在添加的时候进行排序）。

TreeSet 是基于二叉树的数据结构，二叉树的：一个节点下不能多余两个节点。

二叉树的存储过程：

如果是第一个元素，那么直接存入，作为根节点，下一个元素进来是会跟节点比较，如果大于节点放右边的，小于节点放左边；等于节点就不存储。后面的元素进来会依次比较，直到有位置存储为止。

TreeSet 集合存储 String 对象

```
import java.util.TreeSet;
public class Test {
    public static void main(String[] args) {
        TreeSet<String> treeSet = new TreeSet<String>();
        treeSet.add("abc");
        treeSet.add("zbc");
        treeSet.add("cbc");
        treeSet.add("xbc");
        for (String str : treeSet){
            System.out.println(str);
        }
        // 结果：取出来的结果是经过排序的
        /*
        abc
        cbc
        xbc
        zbc*/
    }
}
```

TreeSet 保证元素的唯一性是有两种方式：

1、自定义对象实现 Comparable 接口，重写 comparaTo 方法，该方法返回 0 表示相等，小于 0 表示准备存入的元素比被比较的元素小，否则大于 0；

2、在创建 TreeSet 的时候向构造器中传入比较器 Comparator 接口实现类对象，实现

Comparator 接口重写 compara 方法。

如果向 TreeSet 存入自定义对象时，自定义类没有实现 Comparable 接口，或者没有传入 Comparator 比较器时，会出现 ClassCastException 异常。

演示用两种方式来存储自定义对象

```
public class Person implements Comparable<Person>{
    // 属性
    private String name;
    private int age;
    // 构造方法
    public Person() {
        super();
    }
    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    // 要让哈希表存储不重复的元素，就必须重写 hasCode 和 equals 方法
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + age;
        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;
        Person other = (Person) obj;
        if (age != other.age)
            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 "Person [name=" + name + ", age=" + age + "]";
    }
    // getter & setter
   ...
    @Override
    public int compareTo(Person o) {
        int result = this.age - o.age;
        if (result == 0){
            return this.name.compareTo(o.name);
        }
        return result;
    }
}


import java.util.TreeSet;
public class Test {
    public static void main(String[] args) {
        // 利用 TreeSet 来存储自定义类 Person 对象
        TreeSet<Person> treeSet = new TreeSet<Person>();
        // Person 类实现了 Comparable 接口，并且重写 compareTo 方法
        // 比较规则是先按照 年龄排序，年龄相等的情况按照年龄排序
        treeSet.add(new Person("张山 1", 20));
        treeSet.add(new Person("张山 2", 16));
        treeSet.add(new Person("张山 3", 13));
        treeSet.add(new Person("张山 4", 17));
        treeSet.add(new Person("张山 5", 20));
        for (Person p : treeSet){
            System.out.println(p);
        }
        // 结果：按照 compareTo 方法内的逻辑来排序的
        /*
        Person [name=张山 3, age=13]
```

```
        Person [name=张山2, age=16]
        Person [name=张山4, age=17]
        Person [name=张山1, age=20]
        Person [name=张山5, age=20]
         */
    }
}
```

另一种方式：使用比较器 Comparator

```
public class Person{
    // 属性
    private String name;
    private int age;
    // 构造方法
    public Person() {
        super();
    }
    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    // 要让哈希表存储不重复的元素，就必须重写 hasCode 和 equals 方法
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + age;
        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;
        Person other = (Person) obj;
```

```
        if (age != other.age)
            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 "Person [name=" + name + ", age=" + age + "]";
    }
    // getter & setter
   ...
}

import java.util.Comparator;
import java.util.TreeSet;
public class Test {
    public static void main(String[] args) {
        // 利用 TreeSet 来存储自定义类 Person 对象
        // 创建 TreeSet 对象的时候传入 Comparator 比较器，使用匿名内部类的方式
        // 比较规则是先按照 年龄排序，年龄相等的情况按照年龄排序
        TreeSet<Person> treeSet = new TreeSet<Person>(new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                if (o1 == o2){
                    return 0;
                }
                int result = o1.getAge() - o2.getAge();
                if (result == 0){
                    return o1.getName().compareTo(o2.getName());
                }
                return result;
            }
        });
        treeSet.add(new Person("张山 1", 20));
        treeSet.add(new Person("张山 2", 16));
        treeSet.add(new Person("张山 3", 13));
```

```
        treeSet.add(new Person("张山4", 17));
        treeSet.add(new Person("张山5", 20));
        for (Person p : treeSet){
            System.out.println(p);
        }
        // 结果：按照compara方法内的逻辑来排序的
        /*
        Person [name=张山3, age=13]
        Person [name=张山2, age=16]
        Person [name=张山4, age=17]
        Person [name=张山1, age=20]
        Person [name=张山5, age=20]
         */
    }
}
```

Collection 体系总结

- List：存取有序，元素有索引，元素可以重复。
- ArrayList：数组结构，查询快，增删慢，线程不安全，因此效率高。
- Vector：数据结构，查询快，增删慢，线程安全，因此效率低。
- LinkedList：链表结构，查询慢，增删快，线程不安全，因此效率低。

```
addFirst()    removeFirst()    getFirst()
```

- Set：存取无序，元素无索引，元素不可以重复。
- HashSet：存储无序，元素无索引，元素不可以重复，底层是哈希表。哈希表底层依赖 hashCode 和 equals 方法保证元素唯一性。当存储元素的时候,先根据 hashCode + 数组长度 计算出一个索引,判断索引位置是否有元素。如果没有元素,直接存储,如果有元素,先判断 equals 方法,比较两个元素是否相同,不同则存储,相同则舍弃。自定义对象存储的元素一定要实现 hashCode 和 equals。
- LinkedHashSet：存储有序，元素不可以重复。
- TreeSet：存储无序，但是可以排序（自然排序），元素不可以重复。

有两种排序方式：

自然排序 :元素必须实现 Comparable 接口、实现 CompareTo 方法。

比较器排序 :我们需要自定义类,实现 Comparetor 接口,这个类就是比较器实现 compare 方法。

然后在创建 TreeSet 的时候,把比较器对象作为参数传递给 TreeSet。

Map

Map 是一个双列集合，其中保存的是键值对，键要求保持唯一性，值可以重复。键值是一一对应的，一个键只能对应一个值。

Map 的特点：是存取无序，键不可重复。

Map 在存储的时候，将键值传入 Entry，然后存储 Entry 对象。

HashMap

基于哈希表结构实现的，所以存储自定义对象作为键时，必须重写 hasCode 和 equals 方法。存取无序的。

演示 HashMap 以自定义对象作为键：

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map.Entry;
import java.util.Set;
public class Test {
    public static void main(String[] args) {
        // 利用 HashMap 存储，自定义对象 Person 作为键
        // 为了保证键的唯一性，必须重写 hashCode 和 equals 方法
        HashMap<Person,String> map = new HashMap<Person,String>();
        map.put(new Person("cy", 12), "JAVA");
        map.put(new Person("李四", 13), "IOS");
        map.put(new Person("小花", 22), "JS");
        map.put(new Person("sihai", 32), "PHP");
        map.put(new Person("cy", 12), "C++");
        Set<Entry<Person, String>> entrySet = map.entrySet();
        Iterator<Entry<Person, String>> it = entrySet.iterator();
        while (it.hasNext()){
            Entry<Person, String> entry = it.next();
            System.out.println(entry.getKey() + "---" + entry.getValue());
        }
        // 结果：存入的时候添加了两个 cy，如果 Map 中键相同的时候，当后面的值会覆盖掉前面的值
        /*
        Person [name=李四, age=13]---IOS
        Person [name=cy, age=12]---C++
        Person [name=sihai, age=32]---PHP
        Person [name=小花, age=22]---JS
```

```
        */
    }
}
```

LinkedHashMap

用法跟 HashMap 基本一致，它是基于链表和哈希表结构的所以具有存取有序，键不重复的特性。

演示利用 LinkedHashMap 存储，注意存的顺序和遍历出来的顺序是一致的:

```
import java.util.LinkedHashMap;
import java.util.Map.Entry;
public class Test {
    public static void main(String[] args) {
        // 利用 LinkedHashMap 存储，自定义对象 Person 作为键
        // 为了保证键的唯一性，必须重写 hashCode 和 equals 方法
        LinkedHashMap<Person,String> map = new LinkedHashMap<Person,String>();
        map.put(new Person("cy", 12), "JAVA");
        map.put(new Person("李四", 13), "IOS");
        map.put(new Person("小花", 22), "JS");
        map.put(new Person("sihai", 32), "PHP");
        map.put(new Person("cy", 12), "C++");
        // foreach 遍历
        for (Entry<Person,String> entry : map.entrySet()){
            System.out.println(entry.getKey()+"==="+entry.getValue());
        }
        // 结果：存入的时候添加了两个 cy，如果 Map 中键相同的时候，当后面的值会覆盖掉前面的值
        // 注意：LinkedHashMap 的特点就是存取有序，取出来的顺序就是和存入的顺序保持一致
        /*
        Person [name=cy, age=12]===C++
        Person [name=李四, age=13]===IOS
        Person [name=小花, age=22]===JS
        Person [name=sihai, age=32]===PHP
        */
    }
}
```

TreeMap

给 TreeMap 集合中保存自定义对象，自定义对象作为 TreeMap 集合的 key 值。由于

TreeMap 底层使用的二叉树，其中存放进去的所有数据都需要排序，要排序，就要求对象具备比较功能。对象所属的类需要实现 Comparable 接口。或者给 TreeMap 集合传递一个 Comparator 接口对象。

利用 TreeMap 存入自定义对象作为键：

```
import java.util.Comparator;
import java.util.Map.Entry;
import java.util.TreeMap;
public class Test {
    public static void main(String[] args) {
        // 利用 TreeMap 存储，自定义对象 Person 作为键
        // 自定义对象实现 Comparable 接口或者传入 Comparator 比较器
        TreeMap<Person,String> map = new TreeMap<Person,String>(new
Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                if (o1 == o2){
                    return 0;
                }
                int result = o1.getAge() - o2.getAge();
                if (result == 0){
                    return o1.getName().compareTo(o2.getName());
                }
                return result;
            }
        });
        map.put(new Person("cy", 12), "JAVA");
        map.put(new Person("李四", 50), "IOS");
        map.put(new Person("小花", 32), "JS");
        map.put(new Person("sihai", 32), "PHP");
        map.put(new Person("cy", 12), "C++");
        // foreach 遍历
        for (Entry<Person,String> entry : map.entrySet()){
            System.out.println(entry.getKey()+"==="+entry.getValue());
        }
        // 结果：存入的时候添加了两个 cy，如果 Map 中键相同的时候，当后面的值会覆盖掉前面
的值
        // 注意：TreeMap 取出来的顺序是经过排序的，是根据 compara 方法排序的
        /*
        Person [name=cy, age=12]===C++
        Person [name=小花, age=32]===JS
```

```
        Person [name=sihai, age=32]===PHP
        Person [name=李四, age=50]===IOS
        */
    }
}
```