

第八章 集合

问题引入

我们之前学习过数组，数组特点如下：

- 长度固定：数组在创建时需要指定长度，并且长度在创建后不可改变
- 相同数据类型：例如int数组只能存储int类型的数据
- 连续内存分配：堆空间为数组开辟的内存是连续的，这也导致在插入和删除元素时需要元素整体移动，效率低下
- 随机访问：由于数组中的元素在内存中是连续存储的，并且可以通过索引来访问，因此可以通过索引直接访问数组中的任意元素，时间复杂度为 $O(1)$

如果我们要存储的多个元素值数据类型不一致，或个数不固定时，数组就无法完美的满足我们的要求，这个时候我们会使用Java中提供的集合框架。

1 集合概述

在Java中，集合（Collection）是一种用于存储和操作一组对象的数据结构。它提供了一组接口和类，用于处理和操作对象的集合。

集合框架（Collection Framework）是Java中用于表示和操作集合的一组类和接口。它位于 `java.util` 包中，并提供了一系列的接口和类，包括集合接口（Collection）、列表接口（List）、集合类（Set）、映射接口（Map）等。

集合框架的主要目标是**提供一种通用的方式来存储和操作对象的集合**，无论集合的具体实现方式如何，用户都可以使用统一的接口和方法来操作集合。

集合理解：

集合和数组都可以存储多个元素值，对比数组，我们来了解下集合：

- 数组的长度是固定的，**集合的长度是可变的**
- 数组中存储的是同一类型的元素，**集合中存储的数据可以是不同类型的**
- 数组中可以存放基本类型数据或者引用类型变量，**集合中只能存放引用类型变量**
- 数组是由JVM中现有的 `类型+[]` 组合而成的，除了一个 `length`属性，还有从Object中继承过来的方法 之外，数组对象就调用不到其他属性和方法了
- 集合框架由 `java.util` 包下多个接口和实现类组成，定义并实现了很多方法，功能强大

2 框架体系

集合框架主要有三个要素组成：

- **接口**

整个集合框架的上层结构，都是用接口进行组织的。接口中定义了集合中必须要有的基本方法。

通过接口还把集合划分成了几种不同的类型，每一种集合都有自己对应的接口。

- **实现类**

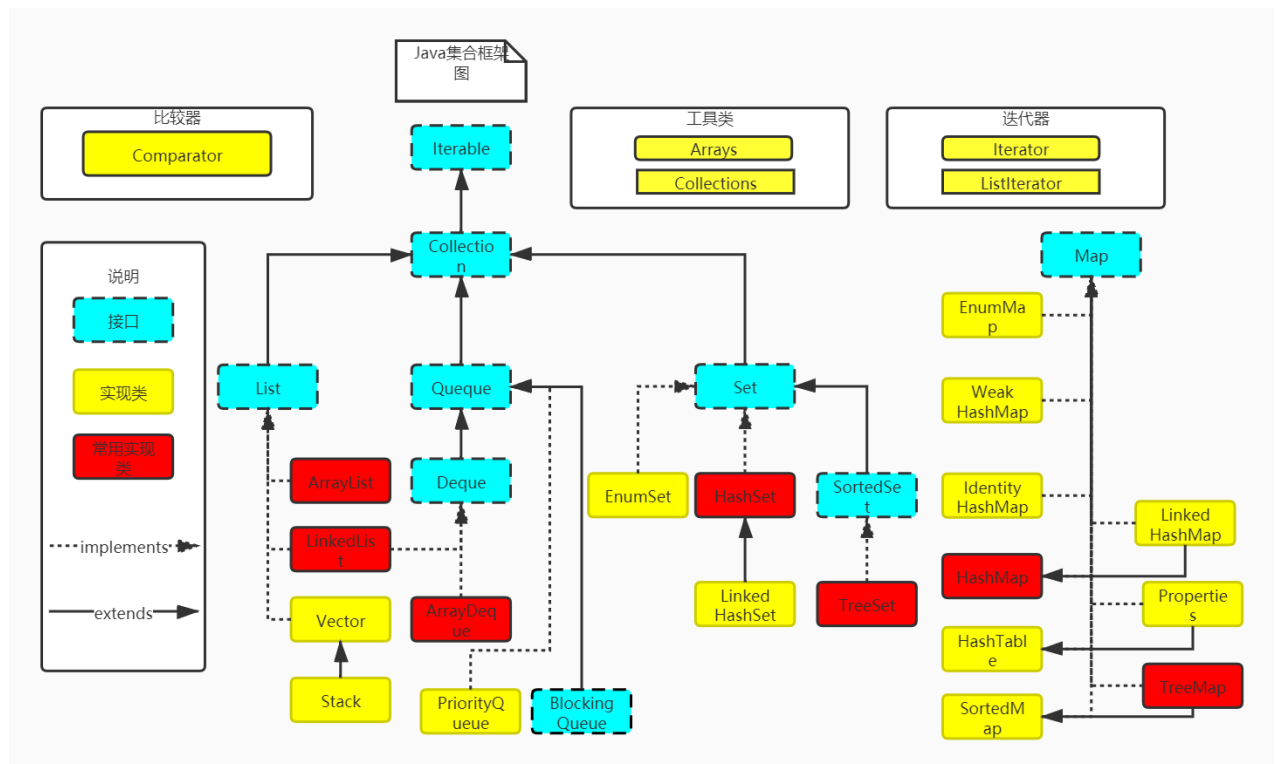
对于上层使用接口划分好的集合种类，每种集合的接口都会有对应的实现类。

每一种接口的实现类很可能有多个，每个的实现方式也会各有不同。

- **数据结构**

每个实现类都实现了接口中所定义的最基本的方法，例如对数据的存储、检索、操作等方法。但是不同的实现类，它们存储数据的方式不同，也就是使用的**数据结构**不同。

集合框架继承体系图：



集合分类：

- 单列集合（Single Column Collection）

根接口：`java.util.Collection`

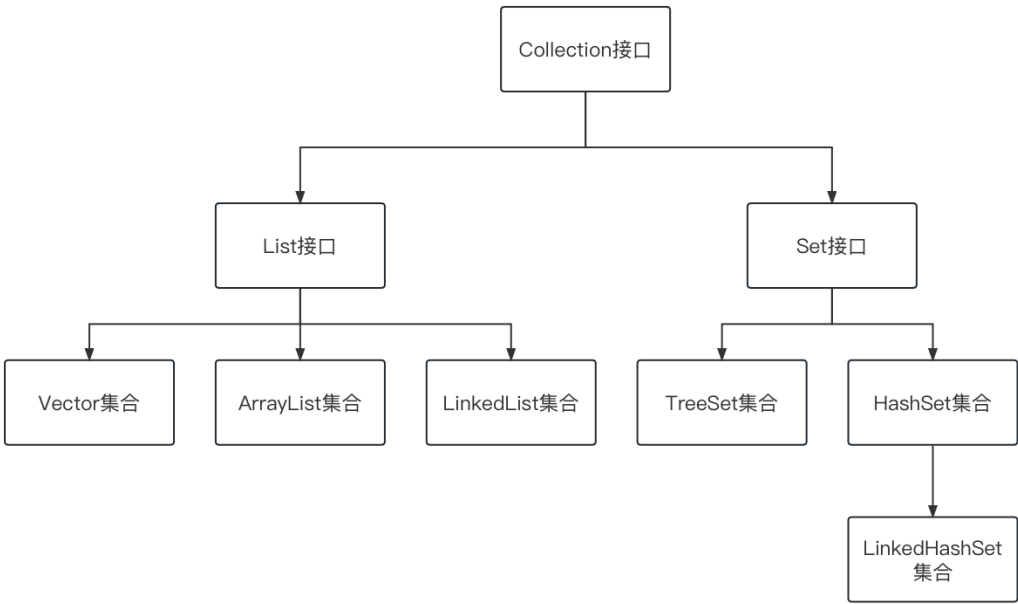
单列集合是指每个集合元素只包含一个单独的对象，它是集合框架中最简单的形式

- 多列集合（Multiple Column Collection）

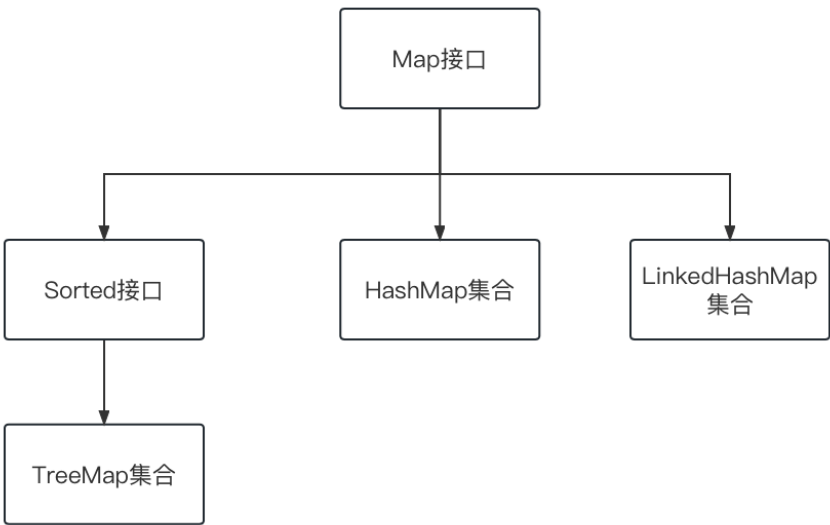
根接口：`java.util.Map`

多列集合是指每个集合元素由多个列（字段）组成，可以同时存储和操作多个相关的值

Collection接口结构图:



Map接口结构图如下:



注意事项:

- 图中列出了Java集合框架中的主要接口（并非全部），以及它们之间的继承关系

- 接口中定义了该种集合具有的主要方法
- 将来真正要使用的，是这些接口的实现类，每种实现类对接口的实现方式不同，底层所用的数据结构不同，其特点也不同

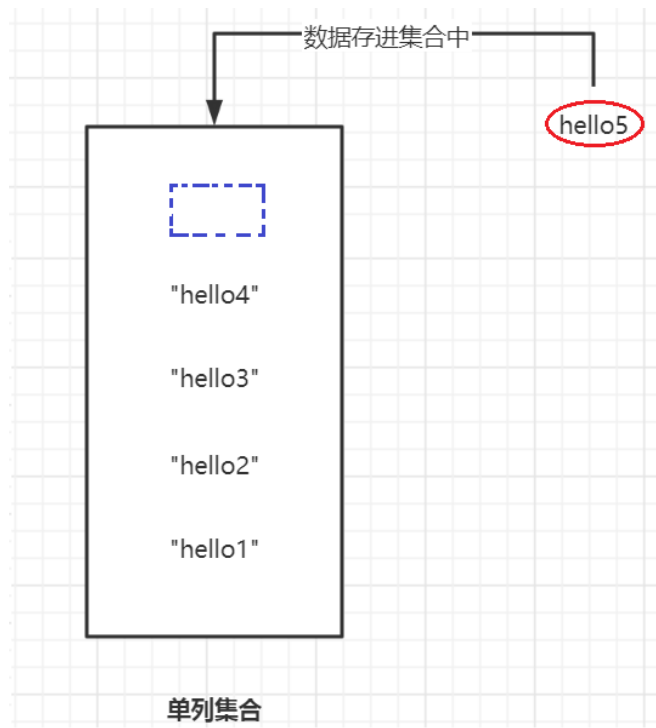
集合章节学习基本要求：

- 要求会用集合存储数据
- 可以从集合中取出数据
- 掌握每种集合的特点和应用场景

3 Collection

Collection接口是**单列**集合类的父接口，这种集合可以将数据一个一个的存放到集合中。它有两个重要的子接口，分别是 `java.util.List` 和 `java.util.Set`

如图：



Collection是父接口，其中定义了单列集合（List和Set）通用的一些方法，Collection接口的实现类，都可以使用这些方法。

1) Collection集合基础方法

```
1 package java.util;
2
3 public interface Collection<E> extends Iterable<E> {
4     //省略...
5
6     //向集合中添加元素
7     boolean add(E e)
8     //清空集合中所有的元素。
9     void clear()
10    //判断当前集合中是否包含给定的对象。
11    boolean contains(Object o)
12    //判断当前集合是否为空。
13    boolean isEmpty()
14    //把给定的对象，在当前集合中删除。
15    boolean remove(Object o)
16    //返回集合中元素的个数。
```

```
17     int          size()
18     //把集合中的元素，存储到数组中。
19     Object[]      toArray()
20 }
```

案例展示:

```
1  package com.briup.chap08.test;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5
6  public class Test03_CollectionBasic {
7      // 集合中 一般 也放 同一种类型数据
8      // contains remove
9      public static void main(String[] args) {
10         // 1.接口 引用 = new 实现类(实参);
11         // 暂时可把它当成固定写法
12         Collection coll = new ArrayList();
13         if (coll.isEmpty())
14             System.out.println("coll is Empty!");
15
16         // 2.任何引用类型都可以放入
17         // 自动扩容
18         coll.add("hello"); //String
19         Integer i = 12;
20         coll.add(i);
21         coll.add(2.3);      // Double
22         coll.add(1.2F);     // Float
23         coll.add('a');      // Character
24
25         int[] arr = { 1, 2, 3 };
26         coll.add(arr);      //int[]
27
28         // coll.add(new Student());
29     }
```

```

30         // 3.输出 coll.toString()
31         System.out.println(coll);
32         System.out.println("size: " + coll.size());
33         System.out.println("-----");
34
35         // 4.清空 coll
36         coll.clear();
37         System.out.println(coll);
38
39         // 5.判断是否为空
40         if (coll.isEmpty())
41             System.out.println("coll is Empty!");
42         else
43             System.out.println("coll is not empty!");
44     }
45 }

```

注意事项：黄色波浪线警告，后续我们处理，此处可以先不管。

```

6 public class Test03_CollectionBasic {
7     public static void main(String[] args) {
8         // 1.接口 引用 = new 实现类(实参);
9         Collection coll = new ArrayList();
10        if (coll.isEmpty())
11            System.out.println("coll is Empty!");
12
13        // 2.集合中可以放入任何引用类型，且能够自动扩容
14        coll.add("hello"); // String
15        Integer i = 12;
16        coll.add(i);        // Integer
17        coll.add(2.3);      // Double
18        coll.add(1.2F);     // Float
19        coll.add('a');      // Character
20    }
21 }

```

黄色波浪线表警告，不影响程序运行
后续案例我们会专门处理掉

2) 警告问题处理

集合接口引用指向实现类对象，固定书写格式：

接口类型<存储的数据类型> 接口引用名 = new 实现类<>(构造方法实参);

例：Collection<String> coll = new ArrayList<>();

注意：上述格式定义的集合对象，只能存储String类型的元素，如果存放其他引用类型元素，则编译报错。

集合中父类或实现类引用，指向具体类对象的写法也是如此

例： `ArrayList<Student> list = new ArrayList<>();`

注意，此时创建的list集合，只能存储Student类对象。

上述代码可以参考ArrayList构造方法源码：

```
1  package java.util;
2
3  //ArrayList也用到了<E>，表示其为泛型类【后续泛型章节会具体讨论，暂时了解即可】
4  public class ArrayList<E> extends AbstractList<E>
5      implements List<E>, RandomAccess, Cloneable,
6      java.io.Serializable
7  {
8
9      /** 无参构造器
10       * Constructs an empty list with an initial capacity of
11       * ten.
12       */
13     public ArrayList() {
14         this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
15     }
```

其中参数中的 **E** 代表范型，取自单词Element的首字母，目前大家可以理解为任何类型。

3) Collection 泛型参数 方法补充

```
1  package java.util;
2
3  public interface Collection<E> extends Iterable<E> {
4      //省略...
5
6      //把一个指定集合中的所有数据，添加到当前集合中
7      boolean    addAll(Collection<? extends E> c)
8      //判断当前集合中是否包含给定的集合的所有元素。
9      boolean    containsAll(Collection<?> c)
10     //把给定的集合中的所有元素，在当前集合中删除。
11     boolean    removeAll(Collection<?> c)
12     //判断两个集合中是否有相同的元素，如果有当前集合只保留相同元素，如
    果没有当前集合元素清空
13     boolean    retainAll(Collection<?> c)
14     //把集合中的元素，存储到数组中，并指定数组的类型
15     <T> T[]    toArray(T[] a)
16     //返回遍历这个集合的迭代器对象
17     Iterator<E> iterator()
18 }
```

泛型方法测试案例：

```
1  package com.briup.chap08.test;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5
6  //泛型方法测试
7  public class Test03_Element {
8      public static void main(String[] args) {
9          //1.实例化两个集合对象，专门存放String类型元素
10         // 集合实例化对象 固定写法
11         Collection<String> c1 = new ArrayList<>();
```

```

12         Collection<String> c2 = new ArrayList<>();
13
14         //2.分别往c1和c2集合中添加元素
15         String s1 = "hello";
16         String s2 = "world";
17         c1.add(s1);
18         c1.add(s2);
19
20         String s3 = "nihao";
21         String s4 = "hello";
22         String s5 = "okok";
23         c2.add(s3);
24         c2.add(s4);
25         c2.add(s5);
26
27         System.out.println("c1: " + c1);
28         System.out.println("c2: " + c2);
29
30         System.out.println("-----");
31
32         //3.将c2集合整体添加到c1中
33         c1.addAll(c2);
34         System.out.println("c1.size: " + c1.size());
35         System.out.println("after addAll(c2), c1: " + c1);
36
37         System.out.println("-----");
38
39         //4.判断是否包含指定元素
40         boolean f = c1.contains("hello");
41         System.out.println("contains hello: " + f);
42
43         //5.创建s6对象，判断集合中是否包含该对象
44         // 注意：s6的地址 和 "world"地址不一样
45         //      s6是堆中临时new出来的，"world"存在堆中的字符串常量
池中
46         String s6 = new String("world");

```

```

47      // 结果显示true，说明集合contains方法借助equals方法进行比较，而非 ==
48      f = c1.contains(s6);
49      System.out.println("contains(s6): " + f);
50
51      System.out.println("-----");
52
53      //6.判断是否包含c2对象
54      f = c1.containsAll(c2);
55      System.out.println("containsAll(c2): " + f);
56
57      System.out.println("-----");
58
59      //7.删除指定元素【底层借助equals比较，然后删除】
60      f = c1.remove(s6);
61      System.out.println("remove(s6): " + f);
62      System.out.println("after remove, c1: " + c1);
63
64      System.out.println("-----");
65
66      //8.删除c2整个集合【底层实现：遍历c2，逐个元素equals比较，然后删除】
67      f = c1.removeAll(c2);
68      System.out.println("removeAll(c2): " + f);
69      System.out.println("after remove, c1: " + c1);
70  }
71  }

```

4) 集合存放自定义类对象

定义Student类，创建多个对象放入集合中。测试集合的contains和remove方法。

注意：先不重写equals方法进行测试，再重写equals方法进行测试。

```

1  package com.briup.chap08.test;

```

```
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5
6 class Student {
7     private String name;
8     private int age;
9
10    public Student() {}
11    public Student(String name, int age) {
12        super();
13        this.name = name;
14        this.age = age;
15    }
16
17    //重写equals方法
18    // @Override
19    // public boolean equals(Object obj) {
20    //     if (this == obj)
21    //         return true;
22    //     if (obj == null)
23    //         return false;
24    //     if (getClass() != obj.getClass())
25    //         return false;
26    //     Student other = (Student) obj;
27    //     if (age != other.age)
28    //         return false;
29    //     if (name == null) {
30    //         if (other.name != null)
31    //             return false;
32    //     } else if (!name.equals(other.name))
33    //         return false;
34    //     return true;
35    // }
36
37    @Override
```

```

38     public String toString() {
39         return "Student [name=" + name + ", age=" + age + "]";
40     }
41 }
42
43 public class Test03_Student {
44     public static void main(String[] args) {
45         //准备学生对象，注意s1和s5两个对象的属性一模一样
46         Student s1 = new Student("zs",20);
47         Student s2 = new Student("ls",19);
48         Student s3 = new Student("ww",22);
49         Student s4 = new Student("tom",18);
50         Student s5 = new Student("zs",20);
51
52         //1.定义只能存储Student对象的集合
53         Collection<Student> coll = new ArrayList<>();
54
55         //2.往集合中添加元素
56         coll.add(s1);
57         coll.add(s2);
58         coll.add(s3);
59         coll.add(s4);
60
61         //3.输出集合元素个数，输出集合对象
62         System.out.println("coll.size: " + coll.size());
63         System.out.println("coll: " + coll);
64
65         System.out.println("-----");
66
67         //4.判断s5是否存在
68         boolean flag = coll.contains(s5);
69         System.out.println("contains(s5): " + flag);
70
71         //5.删除s5
72         flag = coll.remove(s5);
73         System.out.println("remove(s5): " + flag);

```

```
74         System.out.println("coll.size: " + coll.size());
75         System.out.println("coll: " + coll);
76     }
77 }
```

注意事项：集合中contains、remove等方法，底层借助元素对象的equals方法进行值比较，所以如果要用集合存放自定义类对象，注意重写自定义类的equals方法！

4 集合遍历

单列集合的遍历，一般有3种方法，我们由易到难分别介绍。

4.1 toArray

借助Collection接口中toArray()方法实现，方法原型为： `Object[] toArray()`；

遍历格式：

```
1  //将集合转化成数组
2  Object[] array = 集合引用.toArray();
3
4  //遍历数组
5  for (int i = 0; i < array.length; i++) {
6      System.out.println(array[i]);
7  }
```

案例展示：

定义一个集合，添加多个String字符串，然后遍历输出！

```
1  package com.briup.chap08.test;
```

```

2
3  import java.util.ArrayList;
4  import java.util.Collection;
5
6  public class Test041_ToArray {
7      public static void main(String[] args) {
8          //1.实例化集合对象，专门存放String元素
9          Collection<String> c1 = new ArrayList<>();
10
11          String s1 = "hello";
12          String s2 = "world";
13          String s3 = "nihao";
14
15          //2.往集合中添加元素
16          c1.add(s1);
17          c1.add(s2);
18          c1.add(s3);
19
20          //3.遍历集合第一种形式
21          Object[] array = c1.toArray();
22          for (int i = 0; i < array.length; i++) {
23              System.out.println(array[i]);
24          }
25
26          System.out.println("-----");
27      }
28  }

```

4.2 迭代器

迭代器是集合框架提供了一种遍历集合元素的方式。通过调用集合的 `iterator()` 方法可以获取一个迭代器对象，然后使用迭代器的 `hasNext()` 方法判断是否还有下一个元素，使用 `next()` 方法获取下一个元素。

遍历固定格式:

```
1  //1.获取迭代器对象
2  Iterator<集合元素类型> iterator = 集合对象.iterator();
3
4  //2.借助迭代器中hasNext()和next()方法完成遍历
5  while (iterator.hasNext()) {
6      //获取集合元素
7      集合元素类型 变量名 = iterator.next();
8
9      //对集合元素进行输出
10     System.out.println(变量名);
11 }
```

案例展示:

用迭代器遍历上述案例中集合元素。

```
1  package com.briup.chap08.test;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.Iterator;
6
7  public class Test042_Iterator {
8      public static void main(String[] args) {
9          //1.实例化集合对象，专门存放String元素
10         Collection<String> c1 = new ArrayList<>();
11
12         String s1 = "hello";
13         String s2 = "world";
14         String s3 = "nihao";
15
16         //2.往集合中添加元素
17         c1.add(s1);
```

```

18         c1.add(s2);
19         c1.add(s3);
20
21         //3.迭代器遍历
22         Iterator<String> it = c1.iterator();
23         while(it.hasNext()) {
24             String s = it.next();
25             System.out.println(s);
26         }
27     }
28 }

```

注意，这种迭代器方式获取集合中的每一个元素，是一种Collection集合及其子类型集合通用的方式

迭代器原理分析（补充内容，了解即可）：

- `java.lang.Iterable` 接口中，定义了获取迭代器的方法：

```

1 public interface Iterable {
2     Iterator iterator();
3 }

```

- `java.util.Collection` 接口继承了 `java.lang.Iterable` 接口

```

1 public interface Collection extends Iterable {
2     //...
3 }

```

所以，`Collection` 接口及其子接口中，都有一个获取迭代器对象的方法：
`Iterator iterator();`

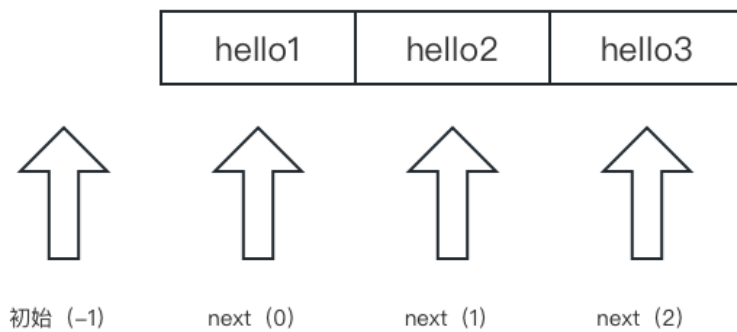
- `java.util.Iterator` 接口中，主要定义两个方法：

```
1 public interface Iterator {
2     //返回当前迭代器中是否还有下一个对象
3     boolean hasNext();
4
5     //获取迭代器中的下一个对象
6     Object next();
7 }
```

迭代器实现原理：

1. 获取迭代器对象：集合类实现了 `Iterable` 接口，该接口定义了一个 `iterator()` 方法，用于获取迭代器对象。迭代器对象是实现了 `Iterator` 接口的具体类的实例。
2. 迭代器位置初始化：在创建迭代器对象时，迭代器的位置通常初始化为集合的起始位置。不同的集合实现可能对位置初始化进行不同的处理。
3. 遍历集合元素：通过调用迭代器对象的 `hasNext()` 方法，可以判断集合中是否还有下一个元素。如果有下一个元素，可以通过 `next()` 方法获取下一个元素，并将迭代器的位置后移。
4. 迭代器状态管理：迭代器对象会记录当前迭代的状态，包括当前位置、遍历过程中的操作等。这些状态可以帮助迭代器在遍历过程中正确地访问和操作集合的元素。
5. 结束迭代：当集合中没有更多元素时，迭代器的 `hasNext()` 方法将返回 `false`，表示遍历结束。

迭代器next方法示意图：



注意事项:

- 迭代器的优点是提供了一种统一的遍历方式，对单列集合（直接或间接实现了Collection接口），都可以通过迭代器按顺序访问元素
- 迭代器隐藏了集合的具体实现细节，提供了一种抽象的访问接口，使代码更加灵活和可复用
- 迭代器是单向的，只能从前往后遍历，无法倒序遍历或在遍历过程中修改集合
- 如果在迭代过程中对集合进行修改（添加、删除元素），可能会导致迭代器抛出 `ConcurrentModificationException` 异常。因此，在使用迭代器遍历时，不要修改集合结构

4.3 foreach

除了使用迭代器遍历集合之外，JDK1.5及以后版本JDK，提供了**增强for循环**实现集合遍历，这种方式相对迭代器遍历更简单。

遍历格式:

```
1  for(集合元素类型 变量名 : 集合) {
2      //操作元素变量
3  }
```

每次循环，引用变量会指向集合中的一个元素对象，然后在循环体中对该元素对象进行操作

案例展示：

```
1  package com.briup.chap08.test;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5
6  public class Test043_Foreach {
7      public static void main(String[] args) {
8          //1.实例化集合对象，专门存放String元素
9          Collection<String> c1 = new ArrayList<>();
10
11          String s1 = "hello";
12          String s2 = "world";
13          String s3 = "nihao";
14
15          //2.往集合中添加元素
16          c1.add(s1);
17          c1.add(s2);
18          c1.add(s3);
19
20          //3.增强for循环遍历
21          // 快捷键：输入fore 后按下 Alt+/
22          for (String str : c1) {
23              System.out.println(str);
24          }
25      }
26  }
```

可以看出，使用foreach循环对集合进行遍历，会更加简单一些

同时，使用foreach循环也可以**遍历数组**：

```
1 public static void main(String[] args) {
2     int[] arr = {1,3,5,7,9};
3
4     //每次循环，使用变量i接收数组中的一个数据
5     for(int i : arr){
6         System.out.println(i);
7     }
8 }
```

注意，Collection及其子类型的集合，还有数组，都可以使用foreach循环进行遍历！

5 List接口

`java.util.List` 接口继承了 `Collection` 接口，是常用的一种集合类型。

`List` 集合具有 `Collection` 集合的特点之外，还具有自己的一些特点：

- List是一种**有序集合**

例如，向集合中存储的元素顺序是8、2、5。那么集合中就是按照这个顺序进行存储的

- List一种**带索引**的集合

可以通过元素的下标索引，精确查找对应的元素数据

- List集合**可以存放重复元素**

可以把相同的数据，在List集合中多次保存

5.1 继承体系

List接口继承了Collection接口，Collection接口继承了Iterable接口。

```
List<E> - java.util
└─ List<E>
   └─ Collection<E>
      └─ Iterable<T>
```

List接口源码：

```
1 package java.util;
2
3 public interface List<E> extends Collection<E> {
4     //省略...
5 }
```

List接口的实现类：

```
List<E> - java.util
└─ List<E>
   ├── AbstractList<E>
   ├── AbstractList<E>
   ├── ArrayList<E>
   ├── ArrayList<E>
   ├── CheckedList<E>
   ├── CopyOnWriteArrayList<E>
   ├── FixedList<E>
   ├── IdentityArrayList<E>
   ├── IdentityLinkedList<E>
   ├── LinkedList<E>
   ├── SubObservableList<E>
   ├── SynchronizedList<T>
   ├── SynchronizedList<E>
   ├── UnmodifiableList<E>
   ├── Vector<E>
   ├── Vector<E>
   └─ VetoableSubListDecorator<E>
```

重点掌握

注意，这些实现类中，都已经实现了List接口、Collection接口、Iterable接口中的方法，我们只要了解并能使用这些接口中的方法，就已经能够操作这些集合对象了（面向接口）。

额外的，我们还需要了解这些常用的接口实现类，分别都是什么特点，使用的什么数据结构，以及适合在什么样的场景下使用。

5.2 常用方法

```
1 //返回集合中指定位置的元素。
2 E      get(int index);
3 //用指定元素替换集合中指定位置的元素,并返回被替代的旧元素。
4 E      set(int index, E element);
5 //将指定的元素，添加到该集合中的指定位置上。
6 void    add(int index, E element);
7 //从指定位置开始，把另一个集合的所有元素添加进来
8 boolean addAll(int index, Collection<? extends E> c);
9 //移除列表中指定位置的元素，并返回被移除的元素。
10 E      remove(int index);
11 //查收指定元素在集合中的所有，从前往后查到的第一个元素（List集合可以重复存放数据）
12 int     indexOf(Object o);
13 //查收指定元素在集合中的所有，从后往前查到的第一个元素（List集合可以重复存放数据）
14 int     lastIndexOf(Object o);
15 //根据指定开始和结束位置，截取出集合中的一部分数据
16 List<E> subList(int fromIndex, int toIndex);
```

注意，除了这些方法之外，还有从父接口Collection中继承过来的方法

List集合的方法使用：


```

1  package com.briup.chap08.test;
2
3  import java.util.ArrayList;
4  import java.util.Iterator;
5  import java.util.List;
6
7  public class Test052_List {
8      public static void main(String[] args) {
9          //1.创建List集合对象
10         List<String> list = new ArrayList<>();
11
12         //2.添加元素，默认尾部添加
13         list.add("hello1");
14         list.add("hello2");
15         list.add("hello3");
16         list.add("hello1");
17
18         System.out.println(list);
19
20         //3.指定位置添加元素
21         // add(int index,String s)
22         list.add(1, "world");
23         System.out.println(list);
24
25         //3.删除索引位置为2的元素
26         //boolean f = list.remove(2);
27         //System.out.println("remove(2): " + f);
28         //System.out.println("after remove: " + list);
29
30         //4.修改指定位置元素
31         list.set(0, "briup");
32         System.out.println(list);
33
34         //5.借助get方法遍历集合
35         for (int i = 0; i < list.size(); i++) {
36             System.out.println(list.get(i));

```

```

37         }
38
39         System.out.println("-----");
40
41         //6.使用foreach遍历
42         for(Object obj : list){
43             System.out.println(obj);
44         }
45
46         System.out.println("-----");
47
48         //7.使用迭代器遍历
49         Iterator<String> it = list.iterator();
50         while(it.hasNext()){
51             String str = it.next();
52             System.out.println(str);
53         }
54     }
55 }

```

由上述案例可知，List集合的特点：有序可重复，并且可以使用下标索引进行访问

5.3 ArrayList

`java.util.ArrayList` 是最常用的一种List类型集合，`ArrayList` 类底层使用**数组**来实现数据的存储，所以它的**特点是：增删慢，查找快。**

在日常的开发中，查询数据也是用的最多的功能，所以ArrayList是最常用的集合。

但是，如果项目中对性能要求较高，并且在集合中大量的数据做增删操作，那么 `ArrayList` 就不太适合了。

ArrayList的案例我们之前已经写过很多，此处不在额外演示。

ArrayList源码参考:

```
1  package java.util;
2
3  /**
4   * Resizable-array implementation of the <tt>List</tt>
5   * interface. Implements
6   * all optional list operations, and permits all elements,
7   * including
8   * <tt>null</tt>.
9   * @see      LinkedList
10  * @see      Vector
11  * @since    1.2
12  */
13 public class ArrayList<E> extends AbstractList<E>
14     implements List<E>, RandomAccess, Cloneable,
15     java.io.Serializable
16 {
17     //省略...
18
19     /**
20      * Default initial capacity.
21      */
22     private static final int DEFAULT_CAPACITY = 10;
23
24     /**
25      * Shared empty array instance used for empty instances.
26      */
27     private static final Object[] EMPTY_ELEMENTDATA = {};
28 }
```

5.4 LinkedList

`java.util.LinkedList` 底层采用的数据结构是双向链表，其特点是：增删快，查找慢

它的特点刚好和 `ArrayList` 相反，所以在代码中，需要对集合中的元素做大量的增删操作的时候，可以选择使用 `LinkedList`。

注意：这里描述的快和慢，需要在大量的数据操作下，才可以体现，如果数据量不大的话，集合每一种集合的操作几乎没有任何区别。

1) 特点验证

实例化 `ArrayList`、`LinkedList` 集合对象，放入 100000 个元素，测试两种集合插入、查询效率！

```
1  package com.briup.chap08.test;
2
3  import java.util.ArrayList;
4  import java.util.LinkedList;
5  import java.util.List;
6
7  public class Test054_LinkedList {
8      public static void main(String[] args) {
9          //操作集合的次数
10         final int NUM = 100000;
11
12         //1.实例化集合对象
13         List<String> list = new ArrayList<>();
14         //List<String> list = new LinkedList<>();
15
16         //2.开启计时，往集合种放入 100000 个元素
17         long start1 = System.currentTimeMillis();
18         for (int i = 0; i < NUM; i++) {
19             list.add(0, "hello"+i);
```

```

20         }
21         long end1 = System.currentTimeMillis();
22
23         //3.输出时长
24         System.out.println(list.getClass().getSimpleName()+"插入"+NUM+"条数据耗时"+(end1-start1)+"毫秒");
25
26         //4.开启计时，从集合种取 100000 个元素
27         long start2 = System.currentTimeMillis();
28         for(int i = 0; i < list.size(); i++){
29             list.get(i);
30         }
31         long end2 = System.currentTimeMillis();
32
33         //5.输出时长
34         System.out.println(list.getClass().getSimpleName()+"检索"+NUM+"条数据耗时"+(end2-start2)+"毫秒");
35
36     }
37 }
38
39 //运行效果：
40 //根据电脑的当前情况，每次运行的结果可能会有差异
41 //以下是我的电脑运行俩次的实验结果，第一次使用ArrayList，第二次使用
    LinkedList
42 ArrayList插入100000条数据耗时508毫秒
43 ArrayList检索100000条数据耗时2毫秒
44
45 LinkedList插入100000条数据耗时22毫秒
46 LinkedList检索100000条数据耗时17709毫秒

```

注意事项：

- `list.getClass().getSimpleName();`

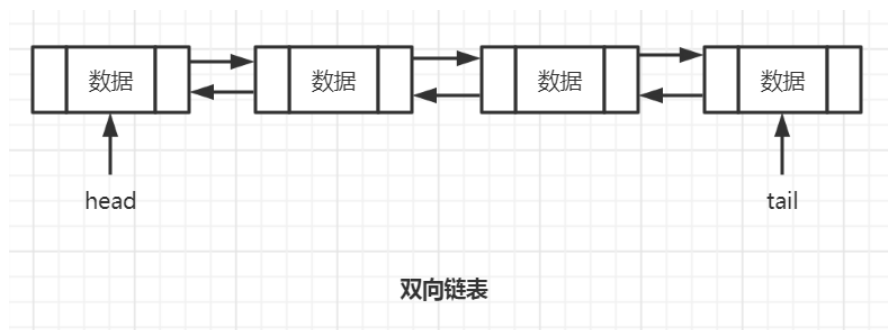
获取list引用指向对象所属类型名（不含包名），了解即可，后续反射会讲解。

- `System.currentTimeMillis();`

获取当前时刻的时间戳，**要求掌握**，后续还会用到。

2) 底层实现，此部分为补充内容（了解即可）

LinkedList底层借助双向链表实现，双向链表又由节点构成，每个节点由三部分构成：两个引用（分别指向前、后节点），一个数据域（存储集合元素），具体可参考下面图和源码理解。



LinkedList源码分析：

```
1  package java.util;
2
3  public class LinkedList<E>
4      extends AbstractSequentialList<E>
5      implements List<E>, Deque<E>, Cloneable,
6      java.io.Serializable
7  {
8      //省略...
9      //头节点
10     transient Node<E> first;
11     //尾节点
12     transient Node<E> last;
13
```

```

14      //静态内部类：Node节点类
15      private static class Node<E> {
16          E item; //数据域，存储数据
17          Node<E> next;    //指针域：指向后一个节点
18          Node<E> prev;    //指针域：指向前一个结点
19
20          Node(Node<E> prev, E element, Node<E> next) {
21              this.item = element;
22              this.next = next;
23              this.prev = prev;
24          }
25      }
26  }

```

3) 头尾节点操作方法

在 `LinkedList` 中，定义了一些操作头节点和尾节点的方法：

```

1  //将指定元素插入此列表的开头
2  void      addFirst(E e)
3  //将指定元素添加到此列表的结尾
4  void      addLast(E e)
5  //返回此列表的第一个元素
6  E          getFirst()
7  //返回此列表的最后一个元素
8  E          getLast()
9  //移除并返回此列表的第一个元素
10 E          removeFirst()
11 //移除并返回此列表的最后一个元素
12 E          removeLast()
13 //从此列表所表示的堆栈处弹出一个元素
14 E          pop()
15 //将元素推入此列表所表示的堆栈
16 void      push(E e)

```

案例展示:

创建一个LinkedList对象，通过节点方法往里面添加、更新、删除元素。

```
1  package com.briup.chap08.test;
2
3  import java.util.LinkedList;
4
5  public class Test054_Node {
6      public static void main(String[] args) {
7          //1.实例化集合对象
8          //注意，要测试LinkedList中的方法，必须用LinkedList引用指向
9          //LinkedList对象
10         LinkedList<String> list = new LinkedList<>();
11
12         String s1 = "hello";
13         String s2 = "world";
14         String s3 = "nihao";
15
16         //2.往集合中添加元素并输出
17         list.add(s1);
18         list.add(s2);
19         list.add(s3);
20         System.out.println("list: " + list);
21
22         System.out.println("-----");
23
24         //3.往头、尾节点添加元素
25         list.addFirst("first");
26         list.addLast("last");
27         System.out.println("list: " + list);
28
29         System.out.println("-----");
30
31         //4.获取头尾节点元素
```



```

31         System.out.println("getFirst: " + list.getFirst());
32         System.out.println("getLast: " + list.getLast());
33
34         System.out.println("-----");
35
36         //5.删除头尾节点元素
37         list.removeFirst();
38         list.removeLast();
39         System.out.println("list: " + list);
40     }
41 }
42
43 //程序运行效果:
44 list: [hello, world, nihao]
45 -----
46 list: [first, hello, world, nihao, last]
47 -----
48 getFirst: first
49 getLast: last
50 -----
51 list: [hello, world, nihao]

```

注意：要测试LinkedList中的方法，必须用LinkedList引用指向LinkedList对象！

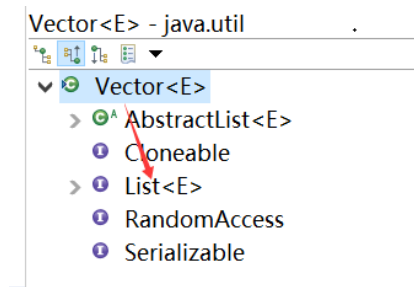
5.5 Vector

Vector是在JDK1.0引入的，它实现了List接口，属于Java集合框架的一部分，其**基于动态数组（Dynamic Array）实现，线程安全**，Vector在功能和使用方式上和ArrayList非常相似。

ArrayList是在JDK 1.2引入的，非线程安全，但单线程环境下性能更高效，是Vector的一个非线程安全的替代品。

关于线程安全，不需要过多关注，后面线程章节会具体学习。

Vector继承体系如下:



Vector部分源码:

```
1  package java.util;
2
3  /**
4   * The {@code Vector} class implements a growable array of
5   * objects. Like an array, it contains components that can be
6   * accessed using an integer index.
7   * @see LinkedList
8   * @since   JDK1.0
9   */
10 public class Vector<E>
11     extends AbstractList<E>
12     implements List<E>, RandomAccess, Cloneable,
13     java.io.Serializable
14 {
15     //省略
16
17     /**
18      * 底层借助数组存储数据
19      * The array buffer into which the components of the
20      * vector are
21      * stored.
22      */
23     protected Object[] elementData;
24 }
```

`Vector` 内部也是采用了数组来存储数据，但是 `Vector` 中的方法大多数都是线程安全的方法，所以在多线程并发访问的环境中，可以使用 `Vector` 来保证集合中元据操作的安全。

案例展示：

创建`Vector`集合，往里面添加元素，遍历输出，对比其和`ArrayList`的用法。

```
1  package com.briup.chap08.test;
2
3  import java.util.Enumeration;
4  import java.util.Vector;
5
6  //Vector线程安全但效率较低
7  //其早期提供的方法使用较为繁琐
8  public class Test055_Vector {
9      public static void main(String[] args) {
10         //1.实例化Vector对象
11         Vector<String> v = new Vector<>();
12
13         //2.往集合中添加元素
14         v.add("hello");
15         //早期添加元素，相对麻烦
16         v.addElement("123");
17         v.addElement("abc");
18
19         //3.遍历Vector
20         // 早期遍历方式，相对麻烦
21         Enumeration<String> elements = v.elements();
22         while(elements.hasMoreElements()) {
23             System.out.println(elements.nextElement());
24         }
25     }
```

结论：**我们今后主要使用ArrayList**。多线程环境需要保证线程安全的话，后期学习工具类，可以使 ArrayList变成线程安全。

5.6 List小结

集合类	特点	示例
ArrayList	内部使用数组实现，动态调整大小。读取快，插入删除慢	List list = new ArrayList<>();
LinkedList	内部使用双向链表实现，插入删除快，读取慢	List list = new LinkedList<>();
Vector	线程安全的动态数组实现，操作方法同步，多线程环境安全	List list = new Vector<>();

6 数据结构

6.1 概述

数据结构是计算机科学中研究数据组织、存储和操作的一门学科。它涉及了**如何组织和存储数据**以及如何设计和实现不同的数据操作算法和技术。常见的据结构有**线性数据结构**（含数组、链表、栈和队列等），**非线性数据结构**（树、图等）。

注意：不同的数据结构适用于不同的场景和问题，选择合适的数据结构可以提高算法的效率和性能。

Java集合框架中不同的实现类底层借助不同数据结构来存储输出，常见的数据结构有：

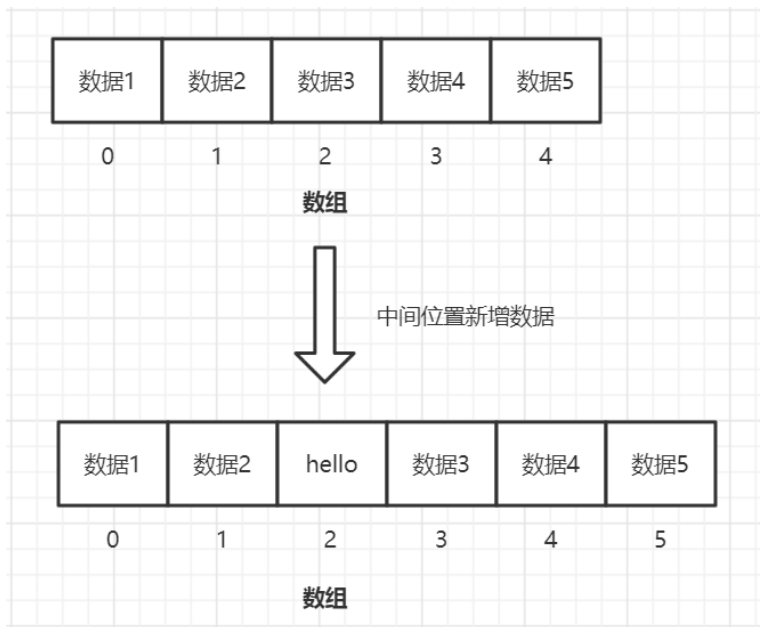
1. 数组（Array）：有序集合，可以包含重复的元素，常见实现类有ArrayList、Vector
2. 链表（LinkedList）：链表是一种动态数据结构，通过节点之间的链接来组织数据。常见的链表实现类是LinkedList
3. 集合（Set）：集合是不允许包含重复元素的无序集合。常见的集合实现类有HashSet、LinkedHashSet和TreeSet
4. 映射（Map）：映射是一种键值对的集合，每个键只能对应一个值。常见的映射实现类有HashMap、LinkedHashMap和TreeMap
5. 队列（Queue）：队列是一种先进先出（FIFO）的数据结构。常见的队列实现类有LinkedList和PriorityQueue
6. 栈（Stack）：栈是一种后进先出（LIFO）的数据结构。常见的栈实现类是Stack
7. 树（Tree）：树是一种具有分层结构的数据结构，常见的树实现类有BinaryTree和BinarySearchTree

6.2 数组

数组（array），内存中一块连续的空间，元素数据在其中按照下标索引依次存储，比较常用的数据结构。

其特点是：通过下标索引，可以快速访问指定位置的元素，但是在数组中间位置添加数据或者删除数据会比较慢，因为数组中间位置的添加和删除元素，为了元素数据能紧凑的排列在一起，那么就会引起其后面的元素移动位置。

所以，数组查询元素较快，中间位置的插入、删除元素较慢。



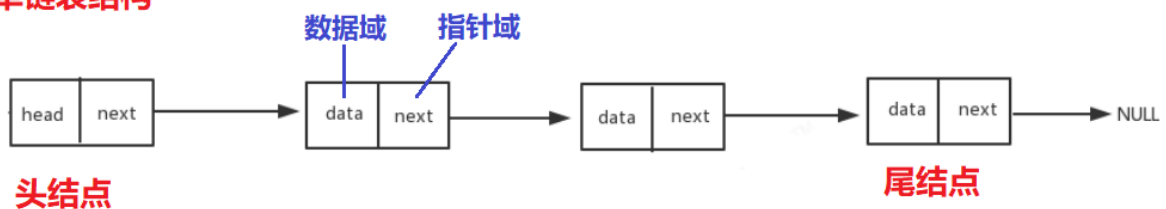
可以看出，数组中间添加数据后，之后的数据都要依次移动位置。同理，中间位置删除的时候也是这样

6.3 链表

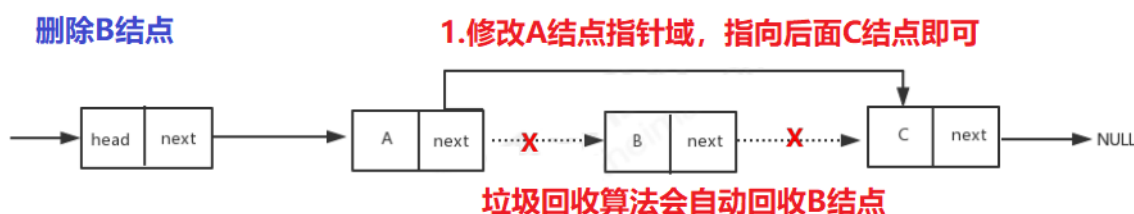
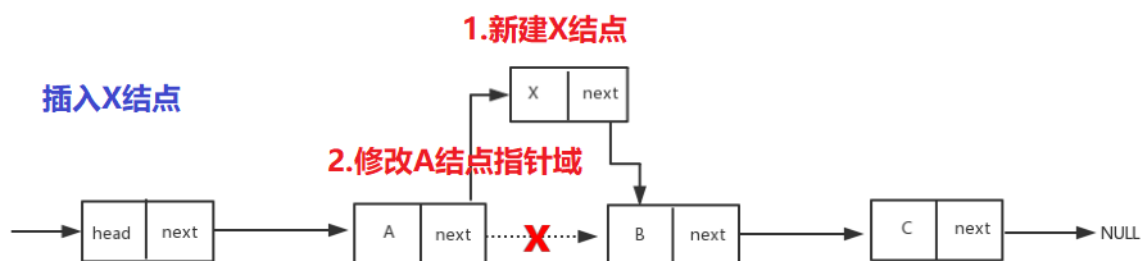
链表 (linked list)，是由一个一个node节点组成，每个node节点中包含两项数据：**指针域**、**数据域**。数据域存储了一个数据，指针域存储指向下一个node节点对象的引用（**单向链表**）。

如果是**双向链表**的话，指针域会存储2个引用，一个指向前一个node节点对象，另一个指向了下一个node节点对象。

单链表结构



单链表插入、删除节点：



总结：

插入和删除 不需要像数组那样 整体移动元素，所以效率高

查询第n个元素，必须从头结点开始逐个结点往后遍历，所以效率低

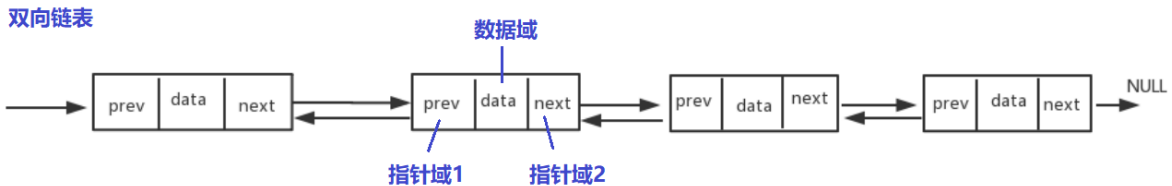
链表特点：

- 查找元素慢，因为需要通过连接的节点，依次向后查找指定元素（没有直接的下标索引）
- 新增和删除元素较快，例如删除，只需要让当前node节点中的引用指向另一个节点对象即可，原来的指向的node节点就相当于删除了

可以看出，只需要将数据2节点中的引用，指向数据4的节点对象即可

head表示链表的头部，tail表示链表的尾部

思考，是否能根据单向链表的特点，想象出双向链表的特点？



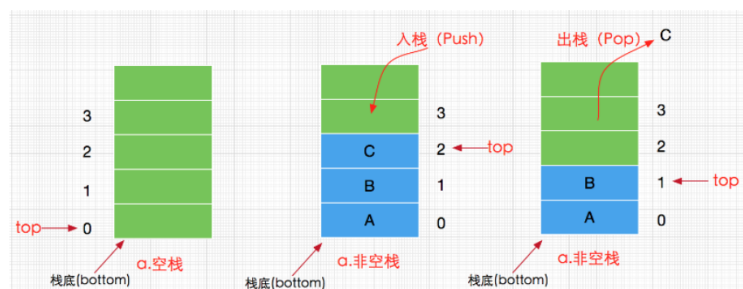
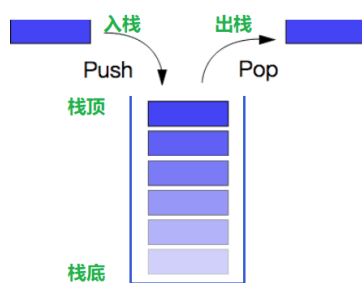
6.4 栈

栈 (stack)，又称堆栈，仅允许在栈的一端进行插入和删除操作，并且不允许在其他任何位置进行操作。

其特点是：**先进后出，最先存进去的元素，最后才能取出来。**

例如，薯片存在薯片桶中，我们当前只能取出最上面的一个薯片，而最早存放到薯片桶的薯片，反而是我们最后吃到的一片。

栈的存储结构及操作特点



注意1，入栈也称为压栈，把数据存入到栈的顶端位置

注意2，出栈也称为弹栈，把栈顶位置的数据取出

思考，JVM中的栈区中，为什么把main方法标注在最低端位置？

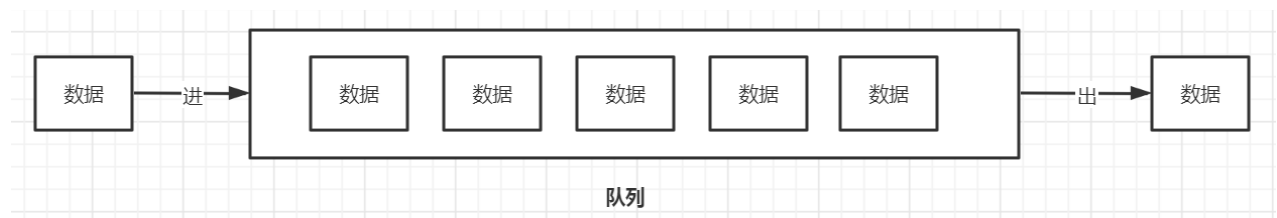
6.5 队列

队列（queue），仅允许在队列的一端进行插入，而在队列的另一端进行删除。

其特点是：**先进先出，最先存进去的元素，可以最先取出来。**

例如，火车穿过山洞的时候，第一节车厢先进去山洞的一端，并且这节车厢优先从山洞的另一端出来，后面的车厢依次从一端进入并另一端出来。

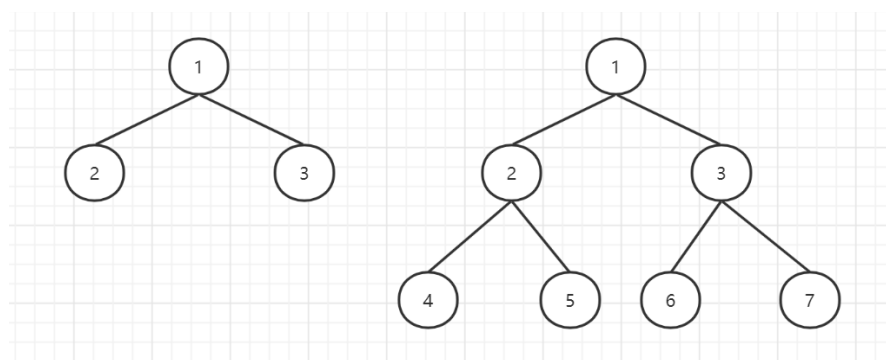
队列的入口、出口分别在队列的俩端：



6.6 红黑树

二叉树（Binary tree）是树形结构的一个重要类型。二叉树特点是每个结点最多只能有两棵子树，且有左右之分。

二叉树顶上的叫根结点，两边被称作“左子树”和“右子树”。



二叉树中有一种叫做红黑树（Red/Black Tree），它最早被称为**平衡二叉B树**（symmetric binary B-trees），后来被称为红黑树。

红黑树是一种特殊化的平衡二叉树，它可以在进行插入和删除的时候，如果左右子树的高度相差较大，那么就通过特定操作（左旋、右旋）保持二叉查找树的平衡（动态平衡），从而获得较高的查找性能。

红黑树的每一个节点的左子树的所有数据都比自己小，而右子树的所有数据都比自己大，并且左右子树的高度近似。

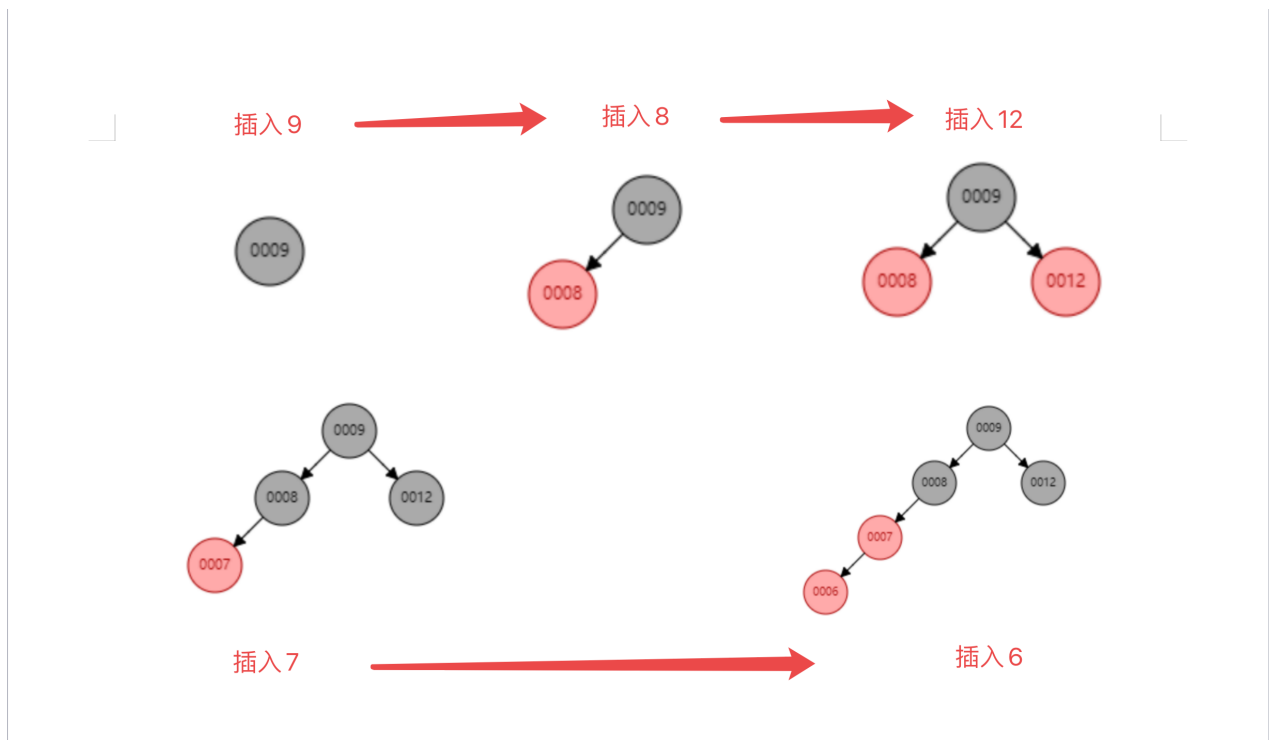
红黑树的约束:

1. 根节点必须是黑色
2. 其他节点可以是红色的或者黑色
3. 叶子节点（特指null节点）是黑色的
4. 每个红色节点的子节点都是黑色的
5. 任何一个节点到其每一个叶子节点的所有路径上黑色节点数相同

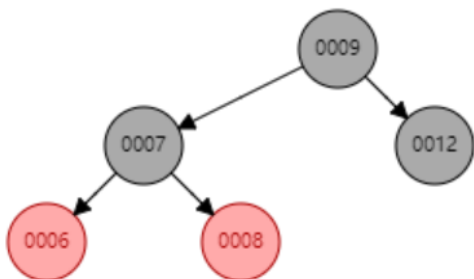
注意，红黑树的指定颜色的目的，是利用颜色值作为二叉树的平衡对称性的检查

例如，从空树开始演示一个案例：

数字插入顺序为 9、8、12、7、6，对于一个节点来说，新数据如果小于本节点，会被放在左节点的位置，反之则放在右节点的位置



当插入数字6的时候，对于红黑树来说整个结构失去平衡，需要通过自旋来调整，最后结果如下：



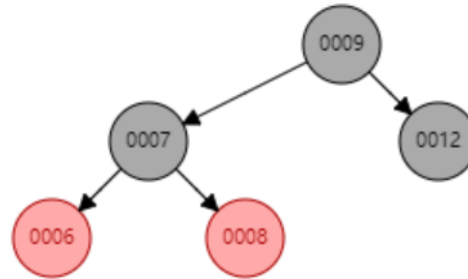
红黑树在线演示

可以通过在线工具，进行节点的添加，查看红黑树的动态调整的**动画效果**，建议使用chrome浏览器打开：

← → ↻ 🏠 cs.usfca.edu/~galles/visualization/RedBlack.html

应用 我的书签 必应 百度 Google springboot springcloud Spring

Insert Delete Find Print ☐ Show Null Leaves

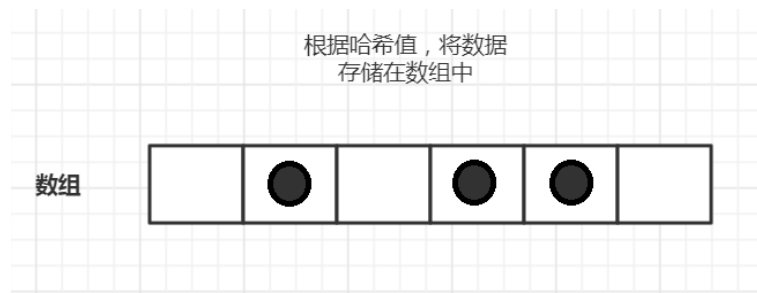


6.7 哈希表

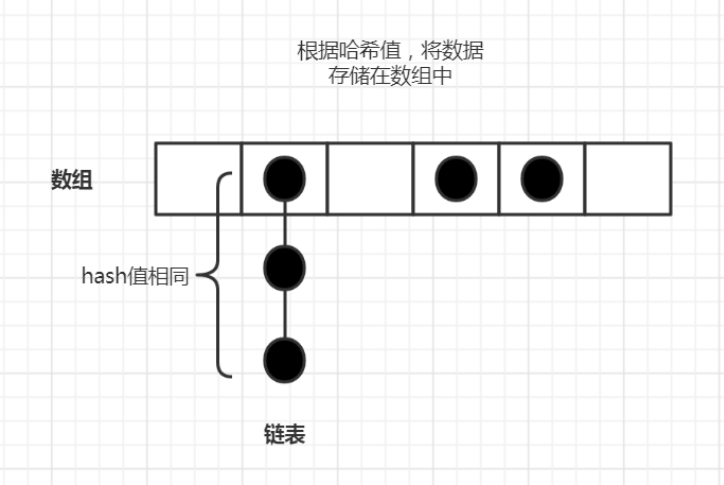
java中的哈希表 (hash) , **在JDK1.8之前是采用数组+链表进行实现**，根据数据的哈希值，把数据存在数组中，但是当前哈希值冲突的时候，再使用链表进行存储，那么在数组中，同一hash值的数据都存在一个链表里。

注意，之前学习过Object中hashCode方法的作用，hash值的特点以及和对象之间的关系

例如，



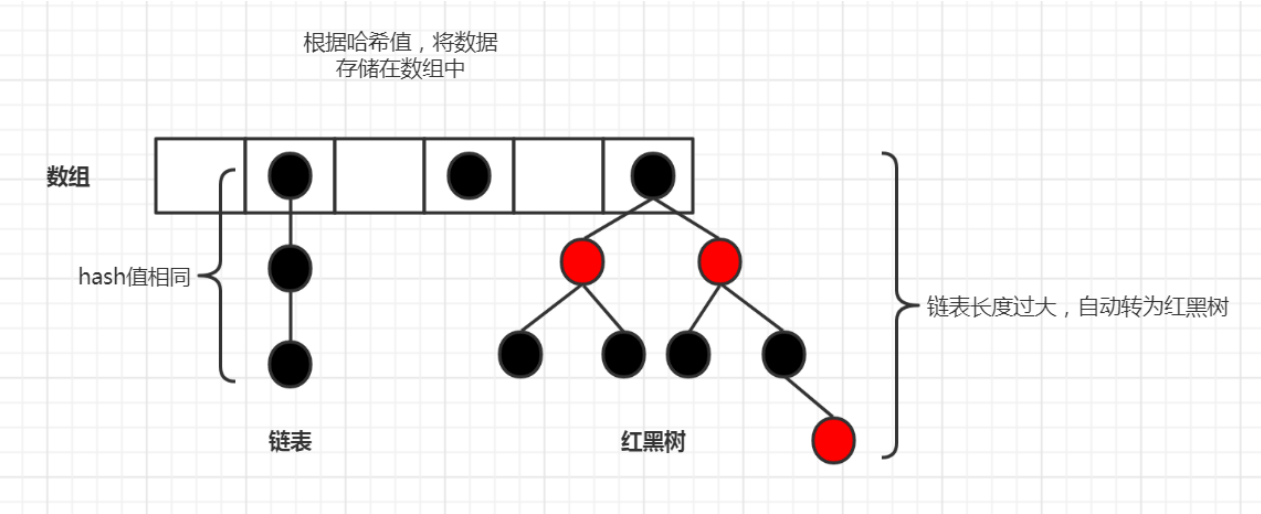
例如，如果数据的哈希值相同，在数组使用使用链表存储哈希值相同的几个数据



可以看出，当链表中元素过多，即hash值相等的元素较多时，查找的效率会变低

JDK1.8中，哈希表存储采用数组+链表+红黑树进行实现，当链表长度超过阈值（8）时，将链表转换为红黑树，这样可以大大提高查找的性能。

例如，



7 Set集合

7.1 Set概述

`java.util.Set` 接口继承了 `Collection` 接口，是常用的一种集合类型。

相对于之前学习的List集合，Set集合特点如下：

除了具有 `Collection` 集合的特点，还具有自己的一些特点：

- Set是一种**无序**的集合
- Set是一种**不带下标索引**的集合
- Set是一种**不能存放重复数据**的集合

Set接口继承体系：

Set<E> - java.util

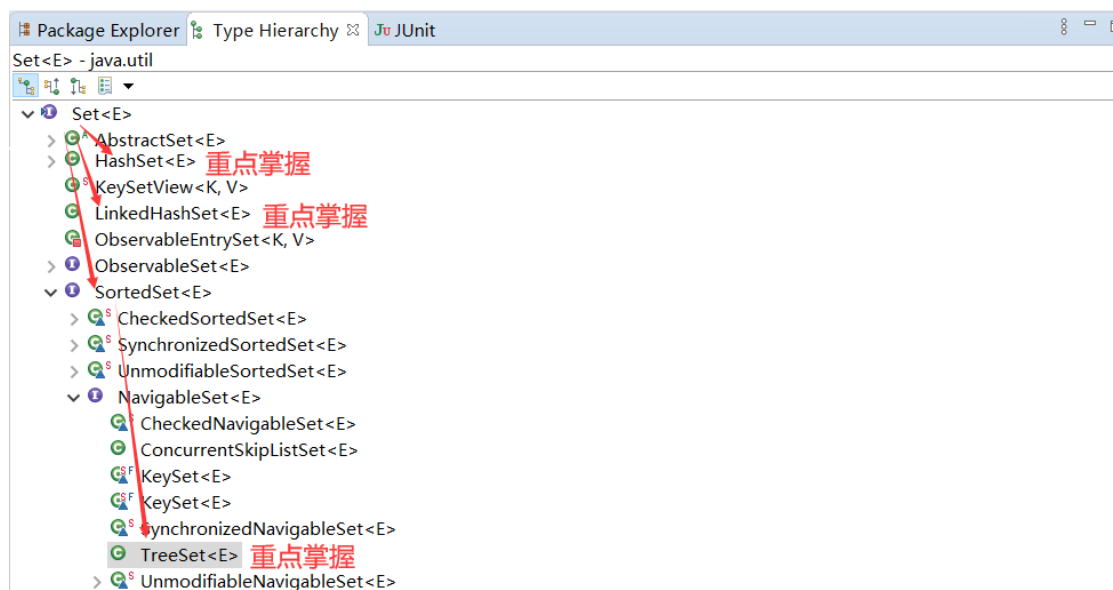
- ▼ Set<E>
 - ▼ Collection<E>
 - Iterable<T>

Set接口方法：



Set集合方法，基本都是从Collection继承，新增了equals()和hashCode()

Set集合实现类：



重点学习的Set实现类：

- HashSet 底层借助哈希表实现
- TreeSet 底层借助二叉树实现

注意，TreeSet是Set接口的子接口SortedSet的实现类

基础案例：

实例化一个Set集合，往里面添加元素并输出，注意观察集合特点（无序、不重复）。

```
1 package com.briup.chap08.test;  
2  
3 import java.util.HashSet;  
4 import java.util.Iterator;  
5 import java.util.Set;  
6  
7 public class Test071_SetBasic {
```

```

8      public static void main(String[] args) {
9          //1.实例化Set集合，指向HashSet实现类对象
10         Set<String> set = new HashSet<>();
11
12         set.add("hello1");
13         set.add("hello2");
14         set.add("hello3");
15         set.add("hello4");
16         set.add("hello5"); //添加失败 重复元素
17         set.add("hello5"); //添加失败 重复元素
18
19         //加强for循环遍历
20         for(String obj : set){
21             System.out.println(obj);
22         }
23
24         System.out.println("-----");
25
26         //迭代器遍历
27         Iterator<String> it = set.iterator();
28         while(it.hasNext()){
29             Object obj = it.next();
30             System.out.println(obj);
31         }
32     }
33 }
34
35 //输出结果：
36 hello1
37 hello4
38 hello5
39 hello2
40 hello3
41 -----
42 省略...

```


通过以上代码和运行结果，可以看出Set类型集合的特点：无序、不可重复

7.2 HashSet

`java.util.HashSet` 是Set接口的实现类，它使用哈希表（Hash Table）作为其底层数据结构来存储数据。

HashSet特点：

- 无序性：HashSet中的元素的存储顺序与插入顺序无关

HashSet使用哈希表来存储数据，哈希表根据元素的哈希值来确定元素的存储位置，而哈希值是根据元素的内容计算得到的，与插入顺序无关。

- 唯一性：HashSet中不允许重复的元素，即每个元素都是唯一的
- 允许null元素：HashSet允许存储null元素，但只能存储一个null元素，HashSet中不允许重复元素

- 高效性：HashSet的插入、删除和查找操作的时间复杂度都是O(1)

哈希表通过将元素的哈希值映射到数组的索引来实现快速的插入、删除和查找操作。

基础案例1：

实例化HashSet对象，往里面插入多个字符串，验证HashSet特点。

```
1 package com.briup.chap08.test;
2
3 import java.util.HashSet;
4 import java.util.Set;
5
6 public class Test072_Basic {
```

```

7      public static void main(String[] args) {
8          //1.实例化HashSet对象
9          Set<String> set = new HashSet<>();
10
11         //2.往集合中添加元素
12         set.add("hello");
13         set.add("world");
14         set.add("nihao");
15         set.add("hello");
16         set.add(null);
17         set.add(null);
18         System.out.println("size: " + set.size());
19
20         //3.遍历
21         for (String str : set) {
22             System.out.println(str);
23         }
24     }
25 }
26
27 //运行结果:
28 size: 4
29 null
30 world
31 nihao
32 hello

```

根据结果可知，HashSet无序、唯一、null值可存在。

基础案例2:

实例化HashSet对象，往里面存入多个自定义类Student对象，观察结果。

自定义Student类:

```
1  package com.briup.chap08.bean;
2
3  public class Student {
4      private String name;
5      private int age;
6
7      public Student() {}
8      public Student(String name, int age) {
9          super();
10         this.name = name;
11         this.age = age;
12     }
13
14     public String getName() {
15         return name;
16     }
17     public void setName(String name) {
18         this.name = name;
19     }
20     public int getAge() {
21         return age;
22     }
23     public void setAge(int age) {
24         this.age = age;
25     }
26
27     @Override
28     public String toString() {
29         return "Student [name=" + name + ", age=" + age + "]";
30     }
31 }
```

测试类:

```
1  package com.briup.chap08.test;
2
```

```

3  import java.util.HashSet;
4  import java.util.Set;
5  import com.briup.chap08.bean.Student;
6
7  public class Test072_Student {
8      public static void main(String[] args) {
9          //1.实例化HashSet对象
10         Set<Student> set = new HashSet<>();
11
12         //2.往集合中添加元素
13         set.add(new Student("zs",20));
14         set.add(new Student("zs",20));
15         set.add(new Student("zs",20));
16         set.add(new Student("zs",20));
17
18         System.out.println("size: " + set.size());
19
20         //3.遍历
21         for (Student stu : set) {
22             System.out.println(stu);
23         }
24     }
25 }
26
27 //运行结果:
28 size: 4
29 Student [name=zs, age=20]
30 Student [name=zs, age=20]
31 Student [name=zs, age=20]
32 Student [name=zs, age=20]

```

从结果可知，HashSet认为4个数据成员一模一样的Student对象是不同的对象，成功将它们添加到了集合中。这明显是不合理的，思考为什么？

元素插入过程：

- 当向HashSet中插入元素时，先**获取元素的hashCode()值**，再检查HashSet中是否存在相同哈希值的元素，如果元素哈希值唯一，则直接插入元素
- 如果存在相同哈希值的元素，会**调用元素的equals()方法来进一步判断元素是否是相同**。如果相同，则不会插入重复元素；如果不同，则插入

hashCode() 方法复习：

- `hashCode()` 方法是 `Object`类 中的一个方法，它返回一个 `int` 类型的哈希码值
- 通常情况下，`hashCode()` 方法会根据对象的属性值计算一个哈希码值（重写自定义类中 `hashCode`方法）
- 如果两个对象的hashCode值不同，则两个对象的属性值肯定不同
- 如果两个对象的hashCode值相同，则两个对象的属性值可能相同，也可能不同

基础案例3：

在案例2的基础上，重写Student类的hashCode和equals方法，再次运行程序观察结果。

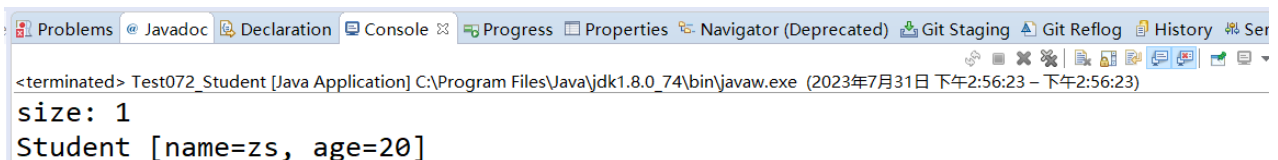
```
1 package com.briup.chap08.bean;
2
3 public class Student {
4     private String name;
5     private int age;
6
7     //省略，跟之前的一样...
8
9     //重写hashCode和equals方法， Alt+Shift+S 自动生成即可
10    // 自动生成的hashCode，根据类属性值计算得到哈希码
```

```

11     @Override
12     public int hashCode() {
13         final int prime = 31;
14         int result = 1;
15         result = prime * result + age;
16         result = prime * result + ((name == null) ? 0 :
name.hashCode());
17         return result;
18     }
19
20     @Override
21     public boolean equals(Object obj) {
22         if (this == obj)
23             return true;
24         if (obj == null)
25             return false;
26         if (getClass() != obj.getClass())
27             return false;
28         Student other = (Student) obj;
29         if (age != other.age)
30             return false;
31         if (name == null) {
32             if (other.name != null)
33                 return false;
34         } else if (!name.equals(other.name))
35             return false;
36         return true;
37     }
38 }
39

```

重新运行程序，结果如下：



```

<terminated> Test072_Student [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月31日 下午2:56:23 - 下午2:56:23)
size: 1
Student [name=zs, age=20]

```

结论：如果要往HashSet集合存储自定义类对象，那么一定要重写自定义类中的hashCode方法和equals方法。

7.3 TreeSet

TreeSet是SortedSet（Set接口的子接口）的实现类，它基于红黑树（Red-Black Tree）数据结构实现。

TreeSet特点：

- 有序性：插入的元素会自动排序，每次插入、删除或查询操作后，TreeSet会自动调整元素的顺序，以保持有序性。
- 唯一性：TreeSet中不允许重复的元素，即集合中的元素是唯一的。如果尝试插入一个已经存在的元素，该操作将被忽略。
- 动态性：TreeSet是一个动态集合，可以根据需要随时添加、删除和修改元素。插入和删除操作的时间复杂度为 $O(\log n)$ ，其中 n 是集合中的元素数量。
- 高效性：由于TreeSet底层采用红黑树数据结构，它具有快速的查找和插入性能。对于有序集合的操作，TreeSet通常比HashSet更高效。

入门案例1：

准备一个TreeSet集合，放入多个Integer元素，观察是否自动进行排序。

```
1 package com.briup.chap08.test;
2
3 import java.util.Set;
4 import java.util.TreeSet;
5
```

```

6  public class Test073_Integer {
7      public static void main(String[] args) {
8          //1.实例化集合对象
9          Set<Integer> set = new TreeSet<>();
10
11         //2.添加元素
12         set.add(3);
13         set.add(5);
14         set.add(1);
15         set.add(7);
16
17         //3.遍历
18         for(Integer obj : set) {
19             System.out.println(obj);
20         }
21     }
22 }
23
24 //输出结果:
25 1
26 3
27 5
28 7

```

可以看出，存进去的数字已经安从小到大进行了排序

入门案例2:

准备一个TreeSet集合，放入多个自定义类Person对象，观察是否自动进行排序。

```

1  package com.briup.chap08.test;
2
3  import java.util.Set;

```



```

4  import java.util.TreeSet;
5
6  class Person {
7      private String name;
8      private int age;
9
10     public Person() {}
11     public Person(String name, int age) {
12         this.name = name;
13         this.age = age;
14     }
15
16     public String getName() {
17         return name;
18     }
19     public void setName(String name) {
20         this.name = name;
21     }
22     public int getAge() {
23         return age;
24     }
25     public void setAge(int age) {
26         this.age = age;
27     }
28
29     @Override
30     public String toString() {
31         return "Person [name=" + name + ", age=" + age + "]";
32     }
33 }
34
35 public class Test073_Person {
36     public static void main(String[] args) {
37         //1.实例化TreeSet
38         Set<Person> set = new TreeSet<>();
39

```

```

40         //2.添加元素
41         set.add(new Person("zs",21));
42         set.add(new Person("ls",20));
43         set.add(new Person("tom",19));
44
45         //3.遍历集合
46         for (Person person : set) {
47             System.out.println(person);
48         }
49     }
50 }
51
52 //程序运行，提示异常，具体如下：
53 Exception in thread "main" java.lang.ClassCastException:
    com.briup.chap08.test.Person cannot be cast to
    java.lang.Comparable
54     at java.util.TreeMap.compare(TreeMap.java:1290)
55     at java.util.TreeMap.put(TreeMap.java:538)
56     at java.util.TreeSet.add(TreeSet.java:255)
57     at
    com.briup.chap08.test.Test073_Person.main(Test073_Person.java:
    41)

```

问题分析：

根据异常提示可知错误原因是：`Person` 无法转化成 `Comparable`，从而导致 `ClassCastException` 异常。

为什么会这样呢？

`TreeSet`是一个有序集合，存储数据时，一定要指定元素的排序规则，有两种指定的方式，具体如下：

TreeSet排序规则：

- 自然排序（元素所属类型要实现 `java.lang.Comparable` 接口）
- 比较器排序

上述2个案例中，Integer存入TreeSet没有报错，是因为Integer类已经实现自然排序，而Person类既没有实现自然排序，也没有额外指定比较器排序规则。

```
52 public final class Integer extends Number implements Comparable<Integer> {  
53     /**  
54      * A constant holding the minimum value an {@code int} can  
55      * have, -2<sup>31</sup>.  
56      */
```

1) TreeSet：自然排序

如果一个类，实现了 `java.lang.Comparable` 接口，并重写了 `compareTo` 方法，那么这个类的对象就是可以比较大小的。

```
1 public interface Comparable<T> {  
2     public int compareTo(T o);  
3 }
```

compareTo方法说明：

```
int result = this.属性.compareTo(o.属性);
```

- result的值大于0，说明this比o大
- result的值小于0，说明this比o小
- result的值等于0，说明this和o相等

元素插入过程：

当向TreeSet插入元素时，TreeSet会使用元素的 `compareTo()` 方法来比较元素之间的大小关系。根据比较结果，TreeSet会将元素插入到合适的位置，以保持有序性。如果两个元素相等（`compareTo()` 方法返回0），TreeSet会认为这是重复的元素，只会保留一个。

案例展示：

使用自然排序（**先按name升序，name相同则按age降序**）解决上述案例问题。

基础Person类：

```
1  package com.briup.chap08.bean;
2
3  //定义Person类，实现自然排序
4  public class Person implements Comparable<Person> {
5      private String name;
6      private int age;
7
8      public Person() {}
9      public Person(String name, int age) {
10         this.name = name;
11         this.age = age;
12     }
13
14     public String getName() {
15         return name;
16     }
17     public void setName(String name) {
18         this.name = name;
19     }
20     public int getAge() {
21         return age;
22     }
23     public void setAge(int age) {
```

```

24         this.age = age;
25     }
26
27     @Override
28     public String toString() {
29         return "Person [name=" + name + ", age=" + age + "]";
30     }
31
32     //重写方法，指定比较规则：先按name升序，name相同则按age降序
33     @Override
34     public int compareTo(Person o) {
35         //注意：字符串比较需要使用compareTo方法
36         int r = name.compareTo(o.name);
37         if(r == 0) {
38             r = o.age - this.age;
39         }
40         return r;
41     }
42 }

```

测试类：

```

1  package com.briup.chap08.test;
2
3  import java.util.Set;
4  import java.util.TreeSet;
5  import com.briup.chap08.bean.Person;
6
7  //自然排序测试
8  public class Test073_Comparable {
9      public static void main(String[] args) {
10         //1.实例化TreeSet
11         Set<Person> set = new TreeSet<>();
12

```

```

13      //2.添加元素
14      set.add(new Person("zs",21));
15      set.add(new Person("ww",20));
16      set.add(new Person("zs",21));
17      set.add(new Person("tom",19));
18      set.add(new Person("tom",23));
19      set.add(new Person("jack",20));
20
21      //3.遍历集合
22      for (Person person : set) {
23          System.out.println(person);
24      }
25  }
26  }
27
28  //输出结果:
29  Person [name=jack, age=20]
30  Person [name=tom, age=23]
31  Person [name=tom, age=19]
32  Person [name=ww, age=20]
33  Person [name=zs, age=21]

```

补充内容1: Integer类中重写compare源码

```

1215 public int compareTo(Integer anotherInteger) {
1216     return compare(this.value, anotherInteger.value);
1217 }
1218
1233 public static int compare(int x, int y) {
1234     return (x < y) ? -1 : ((x == y) ? 0 : 1);
1235 }

```

可以看出, Integer的俩个对象比较大小, 其实就是比较Integer对象对应的int值的大小

注意, compareTo方法返回值代表的含义 (三种情况: 正数、负数、零)

补充内容2：整形、浮点型、字符串自然排序规则

对于整形、浮点型元素，它们会按照从小到大的顺序进行排序。对于字符串类型的元素，它们会按照字典顺序进行排序。

注意事项：compareTo方法的放回结果，只关心正数、负数、零，不关心具体的值是多少

2) TreeSet：比较器排序

思考：如果上述案例中Person类不是自定义类，而是第三方提供好的（不可以修改源码），那么如何实现排序规则的指定？

我们可以可以使用比较器（Comparator）来自定义排序规则。

比较器排序步骤：

- 创建一个实现了Comparator接口的比较器类，并重写其中的 `compare()` 方法

该方法定义了元素之间的比较规则。在 `compare()` 方法中，我们可以根据元素的属性进行比较，并返回负整数、零或正整数，来表示元素之间的大小关系。

- 创建TreeSet对象时，将比较器对象作为参数传递给构造函数，这样，TreeSet会根据比较器来进行排序。

TreeSet类构造器补充：

```
1 //实例化TreeSet类对象时，需要额外传入一个比较器类对象
2 public TreeSet(Comparator<? super E> comparator) {
3     this(new TreeMap<>(comparator));
4 }
```

java.util.Comparator 接口源码：

```
1  package java.util;
2
3  @FunctionalInterface
4  public interface Comparator<T> {
5      /**
6       * Compares its two arguments for order.  Returns a
7       * negative integer,
8       * zero, or a positive integer as the first argument is
9       * less than, equal
10      * to, or greater than the second.<p>
11      */
12      int compare(T o1, T o2);
13  }
```

compare方法说明：

```
int result = compare(o1, o2);
```

- result的值大于0，表示升序
- result的值小于0，表示降序
- result的值等于0，表示元素相等，不能插入

注意，这里和自然排序的规则是一样的，只关心正数、负数、零，不关心具体的值是多少

案例说明：

使用比较器排序，对上述案例中Person进行排序，要求**先按age升序，age相同则按name降序**。

基础类需要注释掉自然排序接口：

```
1  package com.briup.chap08.bean;
2
3  //定义Person类，实现自然排序
4  public class Person /* implements Comparable<Person> */ {
5      private String name;
6      private int age;
7
8      public Person() {}
9      public Person(String name, int age) {
10         this.name = name;
11         this.age = age;
12     }
13
14     public String getName() {
15         return name;
16     }
17     public void setName(String name) {
18         this.name = name;
19     }
20     public int getAge() {
21         return age;
22     }
23     public void setAge(int age) {
24         this.age = age;
25     }
26
27     @Override
28     public String toString() {
29         return "Person [name=" + name + ", age=" + age + "]";
30     }
31
32     //重写方法，指定比较规则
33     // @Override
34     // public int compareTo(Person o) {
```

```

35 //      //先按name升序，name相同则按age降序
36 //      int r = name.compareTo(o.name);
37 //
38 //      if(r == 0) {
39 //          r = o.age - this.age;
40 //      }
41 //
42 //      return r;
43 //  }
44 }

```

测试类:

```

1  package com.briup.chap08.test;
2
3  import java.util.Comparator;
4  import java.util.Set;
5  import java.util.TreeSet;
6  import com.briup.chap08.bean.Person;
7
8  //比较器排序测试
9  public class Test073_Comparator {
10      public static void main(String[] args) {
11
12          //1.准备自定义比较器对象
13          // 匿名内部类形式
14          Comparator<Person> comp = new Comparator<Person>() {
15              //重写比较算法：先按age升序，age相同则按name降序
16              @Override
17              public int compare(Person o1, Person o2) {
18                  int r = o1.getAge() - o2.getAge();
19                  if(r == 0) {
20                      //注意：字符串比较需要使用compareTo方法
21                      r = o2.getName().compareTo(o1.getName());
22                  }
23                  return r;

```

```

24         }
25     };
26
27     //2.实例化TreeSet,传入自定义比较器对象
28     Set<Person> set = new TreeSet<>(comp);
29
30     //3.添加元素
31     set.add(new Person("zs",21));
32     set.add(new Person("ww",20));
33     set.add(new Person("zs",21));
34     set.add(new Person("tom",19));
35     set.add(new Person("tom",23));
36     set.add(new Person("jack",20));
37
38     //4.遍历集合
39     for (Person person : set) {
40         System.out.println(person);
41     }
42 }
43 }
44
45 //输出结果:
46 Person [name=tom, age=19]
47 Person [name=ww, age=20]
48 Person [name=jack, age=20]
49 Person [name=zs, age=21]
50 Person [name=tom, age=23]

```

注意：如果同时使用自然排序和比较器排序，比较器排序将覆盖自然排序。

请自行验证：放开Person类自然排序相关注释代码，再运行上述测试类，观察结果输出。

自然排序：

- 使用元素类实现Comparable接口，并重写其中的 `compareTo()` 方法
- 元素会按照其自身的比较规则进行排序
- 自然排序是默认的排序方式，可以直接使用TreeSet或Collections.sort()方法进行排序
- 只能有一种自然排序方式

比较器排序：

- 使用Comparator对象来定义元素之间的比较规则
- 可以自定义排序规则，不依赖于元素类的实现Comparable接口
- 需要创建一个实现Comparator接口的比较器类，并重写其中的 `compare()` 方法
- 可以有多个比较器，根据需要选择不同的比较器进行排序
- 可以通过传入Comparator对象给TreeSet或Collections.sort()方法来进行比较器排序

自然排序和比较器排序都有自己的应用场景。自然排序适用于元素类已经实现了Comparable接口，并且希望按照元素自身的比较规则进行排序的情况。比较器排序适用于需要自定义排序规则，或者元素类没有实现Comparable接口的情况。

7.4 LinkedHashSet

`LinkedHashSet` 是 `HashSet` 的一个子类，具有 `HashSet` 的高效性能和唯一性特性，并且**保持了元素的插入顺序**，其底层基于哈希表和链表实现。

案例展示：

实例化一个LinkedHashSet集合对象，存入多个String字符串，观察是否唯一及顺序存储。

```
1  package com.briup.chap08.test;
2
3  import java.util.Iterator;
4  import java.util.LinkedHashSet;
5  import java.util.Set;
6
7  public class Test074_LinkedHashSet {
8      public static void main(String[] args) {
9          // 1.实例化LinkedHashSet
10         Set<String> set = new LinkedHashSet<>();
11
12         // 2.添加元素
13         set.add("bbb");
14         set.add("aaa");
15         set.add("abc");
16         set.add("bbc");
17         set.add("abc");
18
19         // 3.迭代器遍历
20         Iterator<String> it = set.iterator();
21         while (it.hasNext()) {
22             System.out.println(it.next());
23         }
24     }
25 }
26
27 //运行结果:
28 bbb
29 aaa
30 abc
31 bbc
```

7.5 Set小结

集合类	特点	示例
HashSet	基于哈希表实现，无序集合，不允许重复元素。	Set set = new HashSet<>();
TreeSet	基于红黑树实现，有序集合，不允许重复元素。	Set set = new TreeSet<>();
LinkedHashSet	基于哈希表和链表实现，按插入顺序排序，不允许重复元素。	Set set = new LinkedHashSet<>();

8 Map集合

很多时候，我们会遇到成对出现的数据，例如，姓名和电话，身份证和人，IP和域名等等，这种成对出现，并且一一对应的数据关系，叫做映射。

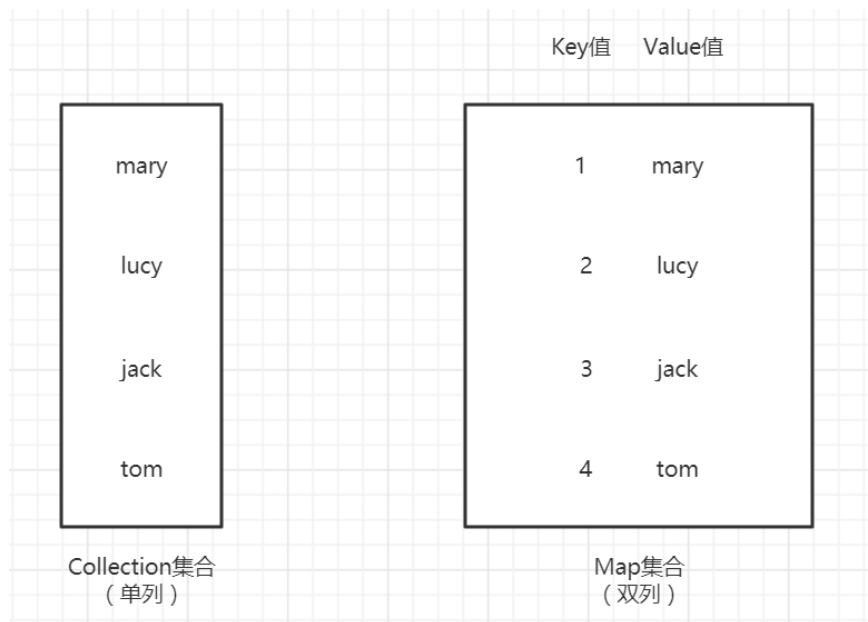
`java.util.Map<K, V>` 接口，就是专门处理这种映射关系数据的集合类型。

Map集合是一种用于存储键值对（key-value）映射关系的集合类。它提供了一种快速查找和访问数据的方式，其中每个键都是唯一的，而值可以重复。

8.1 Map概述

Collection接口为单列集合的根接口，Map接口为双列集合的根接口。

Map集合与Collection集合，存储数据的形式不同：



Map集合特点:

- 存储元素时，必须以key-value (键值对) 的方式进行
- 键唯一性：Map集合中的键是唯一的，每个键只能对应一个值
- 可重复值：Map集合中的值可以重复，不同的键可以关联相同的值
- 高效的查找和访问：通过给定键key值（唯一），可以快速获取与之对应的value值

Map集合内部使用哈希表或红黑树等数据结构来实现高效的查找和访问

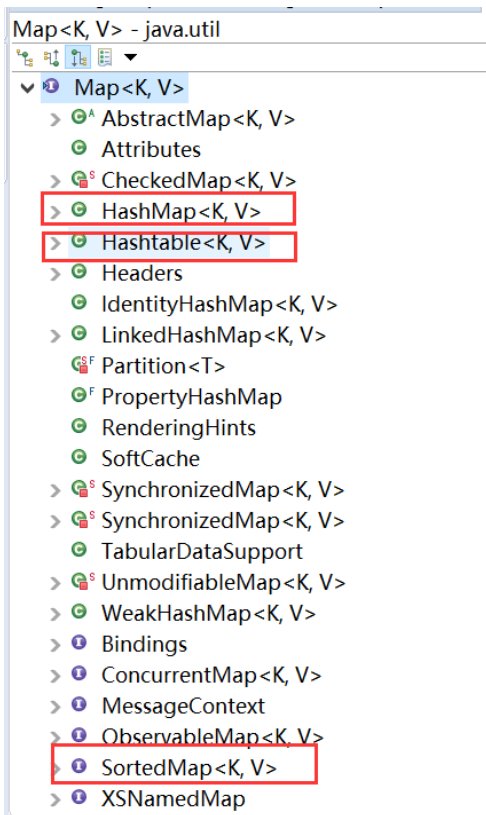
Map接口常用方法 (注意泛型K代表Key, 范型V代表Value) :

```
1 //把key-value存到当前Map集合中
2 V put(K key, V value)
3 //把指定map中的所有key-value, 存到当前Map集合中
4 void putAll(Map<? extends K, ? extends V> m)
5 //当前Map集合中是否包含指定的key值
6 boolean containsKey(Object key)
7 //当前Map集合中是否包含指定的value值
8 boolean containsValue(Object value)
```

```
9    //清空当前Map集合中的所有数据
10   void                clear()
11   //在当前Map集合中，通过指定的key值，获取对应的value
12   V                    get(Object key)
13   //在当前Map集合中，移除指定key及其对应的value
14   V                    remove(Object key)
15   //返回当前Map集合中的元素个数（一对key-value，算一个元素数据）
16   int                 size()
17   //判断当前Map集合是否为空
18   boolean             isEmpty()
19   //返回Map集合中所有的key值
20   Set<K>               keySet()
21   //返回Map集合中所有的value值
22   Collection<V>       values()
23   //把Map集合中的key-value封装成Entry类型对象，再存放到set集合中，并
   返回
24   Set<Map.Entry<K,V>> entrySet()
```

Map集合实现类：

Java提供的Map集合实现类，常见的包括HashMap、TreeMap、LinkedHashMap等。它们在内部实现和性能方面有所不同，可以根据具体需求选择适合的实现类。



- **HashMap**，底层借助哈希表实现，元素的存取顺序不能保证一致。由于要保证键的唯一、不重复，需要重写键所属类的hashCode()方法、equals()方法（重要，最常用）
- **Hashtable**：和之前List集合中的**Vector**的功能类似，可以在多线程环境中，保证集合中的数据的操作安全（线程安全）
- **LinkedHashMap**：该类是**HashMap**的子类，存储数据采用的**哈希表结构+链表结构**。通过链表结构可以保证元素的存取顺序一致（存入顺序就是取出顺序）
- **TreeMap**：该类是**Map**接口的子接口**SortedMap**下面的实现类，和**TreeSet**类似，它可以对key值进行排序，同时构造器也可以接收一个比较器对象作为参数。支持key值的自然排序和比较器排序两种方式（支持key排序）

基本方法使用案例：

```
1 package com.briup.chap08.test;
```

```

2
3  import java.util.HashMap;
4  import java.util.Map;
5  import java.util.Map.Entry;
6  import java.util.Set;
7
8  public class Test081_MapBasic {
9      //双列集合 存放 id-name
10     public static void main(String[] args) {
11         //1.创建HashMap集合对象, 并添加元素
12         Map<Integer,String> map = new HashMap<>();
13
14         //注意, 使用put方法添加键值对
15         map.put(1, "zs");
16         map.put(2, "ls");
17         map.put(4, "rose");
18         map.put(3, "jack");
19         map.put(2, "lucy"); //lucy 会把 ls覆盖掉
20
21         //2.输出集合元素个数
22         //System.out.println(map);
23         System.out.println("size: "+map.size());
24
25         //3.判断key和value是否存在
26         System.out.println("key 2: " + map.containsKey(2));
27         System.out.println("value ls: " +
28             map.containsValue("ls"));
29
30         //4.根据key获取value
31         String name = map.get(3);
32         System.out.println("3: " + name);
33
34         //5.根据key删除键值对
35         map.remove(3);
36         System.out.println(map);
37     }

```

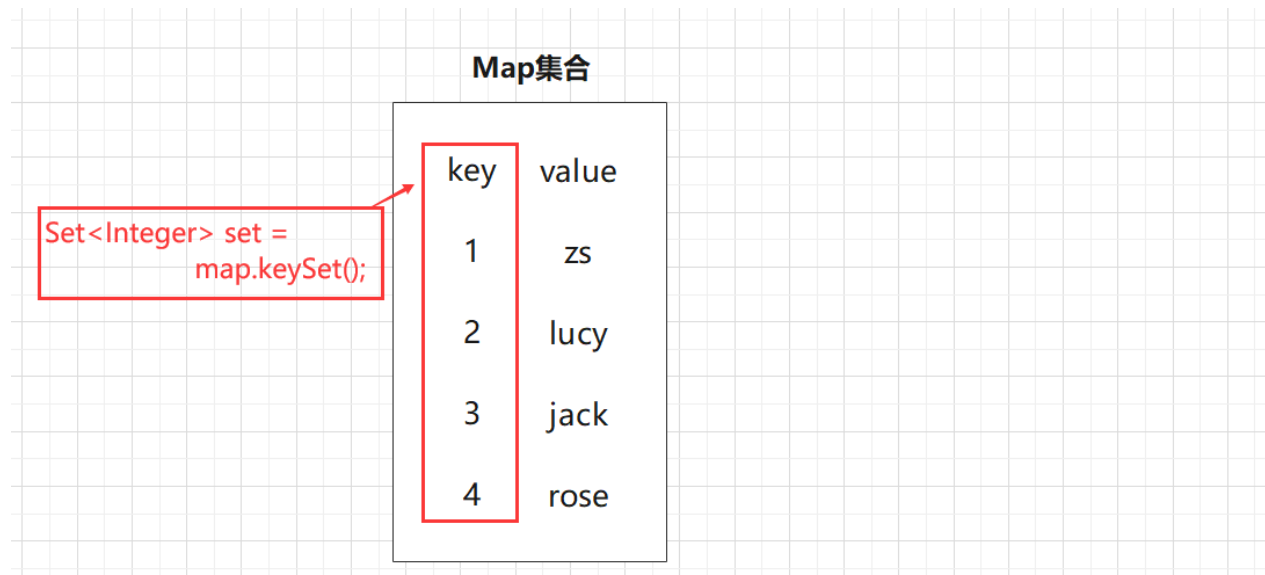
```
37 }
38
39 //运行结果:
40 size: 4
41 key 2: true
42 value 1s: false
43 3: jack
44 {1=zs, 2=lucy, 4=rose}
```

8.2 Map遍历

Map集合提供了2种遍历方式。

1) 第一种遍历思路

借助Map中的keySet方法，获取一个Set集合对象，内部包含了Map集合中所有的key，进而遍历Set集合获取每一个key值，再根据key获取对应的value。



keySet遍历案例：

对前面案例中的map集合对象进行遍历。

```
1 package com.briup.chap08.test;
2
```

```

3  import java.util.HashMap;
4  import java.util.Map;
5  import java.util.Set;
6
7  public class Test082_Each {
8      //双列集合 存放 id-name
9      public static void main(String[] args) {
10         //1.创建HashMap集合对象, 并添加元素
11         Map<Integer,String> map = new HashMap<>();
12
13         map.put(1, "zs");
14         map.put(2, "ls");
15         map.put(4, "rose");
16         map.put(3, "jack");
17         map.put(2, "lucy"); //lucy 会把 ls覆盖掉
18
19         //2.第一种遍历方法
20         // 先获取所有key,再根据key获取value
21         Set<Integer> set = map.keySet();
22         for (Integer k : set) {
23             //借助key获取对应的value值
24             String v = map.get(k);
25             System.out.println("id: " + k + " name: " + v);
26         }
27     }
28 }
29
30 //运行结果:
31 id: 1 name: zs
32 id: 2 name: lucy
33 id: 3 name: jack
34 id: 4 name: rose

```

2) 第二种遍历思路

借助**Map中的entrySet方法**，获取一个Set对象，内部包含了Map集合中所有的键值对，然后对键值对进行拆分，得到key和value进行输出。

Map 接口源码分析：

```
1  package java.util;
2
3  public interface Map<K,V> {
4      //省略...
5
6      //获取map集合种所有的键值对
7      Set<Map.Entry<K, V>> entrySet();
8
9
10     //Map接口的内部接口，类似内部类
11     interface Entry<K,V> {
12         //Returns the key corresponding to this entry.
13         K getKey();
14
15         //Returns the value corresponding to this entry.
16         V getValue();
17
18         //省略...
19     }
20
21     //省略...
22 }
```

Map 接口 **entrySet()** 方法解析：将 **Map** 集合中的每一组key-value（键值对）都封装成一个 **Entry** 类型对象，并且把这些个 **Entry** 对象存放到Set集合中，并返回。

entrySet遍历案例：

对前面案例中的map集合对象进行遍历。

```
1  package com.briup.chap08.test;
2
3  import java.util.HashMap;
4  import java.util.Map;
5  import java.util.Set;
6  import java.util.Map.Entry;
7
8  public class Test082_Each {
9      //双列集合 存放 id-name
10     public static void main(String[] args) {
11         //1.创建HashMap集合对象，并添加元素
12         Map<Integer,String> map = new HashMap<>();
13
14         map.put(1, "zs");
15         map.put(2, "ls");
16         map.put(4, "rose");
17         map.put(3, "jack");
18         map.put(2, "lucy"); //lucy 会把 ls覆盖掉
19
20         //2.第二种遍历
21         // 获取所有的key-value键值对，得到一个Set集合
22         Set<Entry<Integer,String>> entrySet = map.entrySet();
23         // 遍历Set集合
24         for (Entry<Integer, String> entry : entrySet) {
25             // 拆分键值对中的key和value
26             Integer key = entry.getKey();
27             String value = entry.getValue();
28             System.out.println("id: " + key + " name: " +
value);
29         }
30     }
31 }
32
33 //运行结果:
```

```
34 id: 1 name: zs
35 id: 2 name: lucy
36 id: 3 name: jack
37 id: 4 name: rose
```

8.3 HashMap

HashMap底层借助哈希表实现，元素的存取顺序不能保证一致。

HashMap存储的键值对时，如果键类型为自定义类，那么一般需要**重写键所属类的hashCode()和equals()方法（重要，最常用）**。

HashMap特点：

- 键唯一
- 值可重复
- 无序性
- 线程不安全
- 键和值的允许使用null【重点记忆】

案例展示：

实例化 `HashMap<Student, String>` 对象，添加键值对，测试常用方法，最后2种方法遍历。

注意：Student类在之前案例中已经提供好！

```
1 package com.briup.chap08.test;
2
3 import java.util.HashMap;
4 import java.util.Map;
```

```

5  import java.util.Map.Entry;
6  import java.util.Set;
7
8  import com.briup.chap08.bean.Student;
9
10 public class Test083_HashMap {
11     public static void main(String[] args) {
12         //1.实例化HashMap对象，其中key类型为自定义Student
13         Map<Student, String> map = new HashMap<>();
14
15         //2.往集合中添加元素
16         // map中插入键值对，调用key所属类的hashCode和equals方法进行
        判断是否重复
17         map.put(new Student("zs", 78), "010");
18         map.put(new Student("rose", 82), "005");
19         map.put(new Student("lucy", 70), "009");
20         map.put(new Student("lucy", 70), "019"); //相同key，只能
        保留一项，"019"会覆盖"009"
21         map.put(new Student("ww", 67), "002");
22
23         //注意：HashMap中key和value都可以为null
24         map.put(new Student("tom", 86), null);
25         map.put(null, "002");
26
27         //3.基本方法测试
28         // 获取长度
29         System.out.println("size: " + map.size());
30         // 判断key是否存在
31         // 借助 key所属类的hashCode和equals方法完成
32         System.out.println("Student(ww,67)是否存在: " +
        map.containsKey(new Student("ww", 67)));
33         // 判断value是否存在
34         // 借助value所属类型的 equals方法
35         System.out.println("是否存在 009: " +
        map.containsValue("009"));
36         // 根据key删除，返回键对应的值

```



```

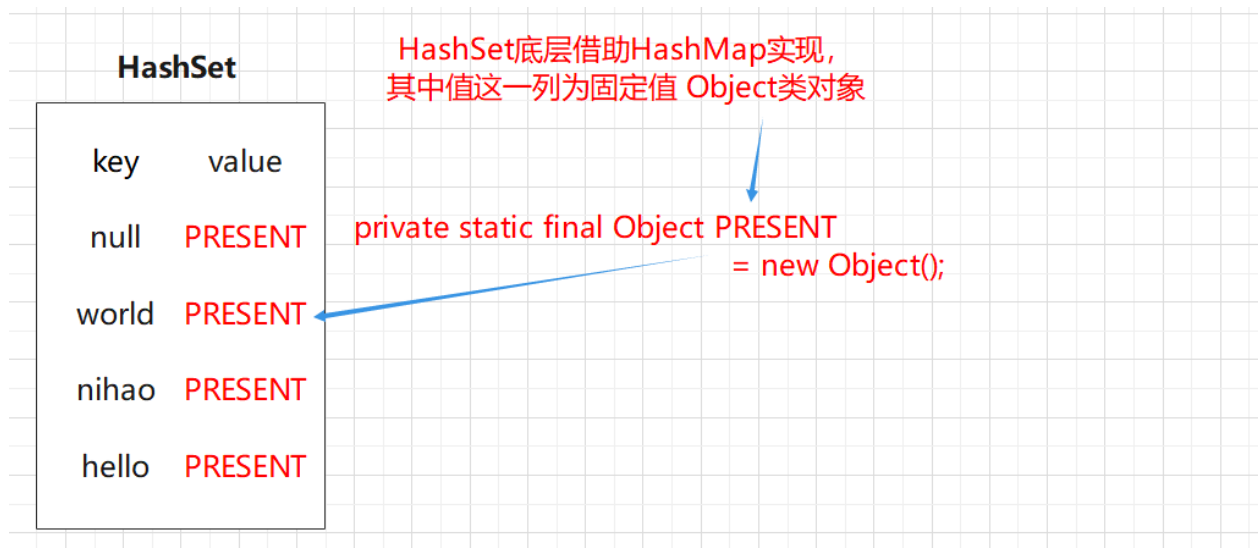
37         String value = map.remove(new Student("lucy", 70));
38         System.out.println("remove(Student(lucy, 70)): " +
value);
39
40         System.out.println("-----");
41
42         // 4.第一种遍历方法
43         Set<Student> keySet = map.keySet();
44         for (Student key : keySet) {
45             System.out.println(key + ": " + map.get(key));
46         }
47
48         System.out.println("-----");
49
50         // 第二种方式遍历
51         Set<Entry<Student, String>> entrySet = map.entrySet();
52         for (Entry<Student, String> entry : entrySet) {
53             System.out.println(entry.getKey() + ": " +
entry.getValue());
54         }
55     }
56 }
57
58 //运行结果:
59 size: 6
60 Student(ww,67)是否存在: true
61 是否存在 009: false
62 remove(Student(lucy, 70)): 019
63 -----
64 null: 002
65 Student [name=rose, age=82]: 005
66 Student [name=zs, age=78]: 010
67 Student [name=tom, age=86]: null
68 Student [name=ww, age=67]: 002
69 -----
70 省略...

```

结论：key类型如果为自定义类型，重写其hashCode和equals方法！

- HashMap中add(key, value)时，需要判断key是否存在（先hashCode再equals）
- HashMap中containsKey(key)时，同样要借助hashCode和equals方法
- HashMap中remove(key)时，同样要借助hashCode和equals方法

补充内容：HashSet底层借助HashMap实现功能



源码分析：

```
1 package java.util;
2
3 public class HashSet<E>
4     extends AbstractSet<E>
5     implements Set<E>, Cloneable, java.io.Serializable
6 {
7     private transient HashMap<E, Object> map;
8
9     // Dummy value to associate with an Object in the backing
    Map
```

```

10     private static final Object PRESENT = new Object();
11
12     /**
13      * Constructs a new, empty set; the backing
14      * <tt>HashMap</tt> instance has
15      * default initial capacity (16) and load factor (0.75).
16      */
17     public HashSet() {
18         map = new HashMap<>();
19     }
20
21     //添加元素，value值固定为Object类对象PRESENT
22     public boolean add(E e) {
23         return map.put(e, PRESENT) == null;
24     }
25
26     //省略...

```

8.4 Hashtable

Hashtable是Java中早期的哈希表实现，它实现了Map接口，并提供了键值对的存储和访问功能。

```

127 * @see      TreeMap
128 * @since    JDK1.0
129 */
130 public class Hashtable<K,V>
131     extends Dictionary<K,V>
132     implements Map<K,V>, Cloneable, java.io.Serializable {
133

```

Hashtable特点:

- JDK1.0提供，接口方法较为复杂，后期实现了Map接口
- 线程安全：Hashtable是线程安全的，相对于HashMap性能稍低

- **键和值都不能为null**：如果尝试使用null作为键或值，将会抛出NullPointerException
- 哈希表实现：和HashMap一样，内部使用哈希表数据结构来存储键值对

案例展示：

实例化Hashtable<Student, String>对象，添加元素并遍历。

```
1  package com.briup.chap08.test;
2
3  import java.util.Enumeration;
4  import java.util.Hashtable;
5
6  import com.briup.chap08.bean.Student;
7
8  public class Test084_Hashtable {
9      public static void main(String[] args) {
10         // 1.实例化HashMap对象，其中key类型为自定义Student
11         Hashtable<Student, String> map = new Hashtable<>();
12
13         // 2.往集合中添加元素
14         // map中插入键值对，调用key所属类的hashCode和equals方法进行
15         判断是否重复
16         map.put(new Student("zs", 78), "010");
17         map.put(new Student("rose", 82), "005");
18         map.put(new Student("lucy", 70), "009");
19         map.put(new Student("lucy", 70), "019"); // 相同key，只
20         能保留一项，"019"会覆盖"009"
21         map.put(new Student("ww", 67), "002");
22
23         // 注意：Hashtable中key和value不能为null，否则抛出
24         NullPointerException
25         //map.put(new Student("tom", 86), null);
26         //map.put(null, "002");
```

```

24
25         // 3.遍历, Hashtable早期提供的方法较为繁琐
26         Enumeration<Student> keys = map.keys();
27         while(keys.hasMoreElements()) {
28             Student key = keys.nextElement();
29             String value = map.get(key);
30
31             System.out.println(key + ": " + value);
32         }
33     }
34 }
35
36 //运行输出:
37 Student [name=lucy, age=70]: 019
38 Student [name=zs, age=78]: 010
39 Student [name=ww, age=67]: 002
40 Student [name=rose, age=82]: 005

```

Hashtable小结:

- 单线程环境下建议使用HashMap, 性能更好
- 多线程环境下, 建议使用 HashMap + Collections (后期补充)

8.5 TreeMap

TreeMap是有序映射实现类, 它实现了SortedMap接口, 基于红黑树数据结构来存储键值对。

TreeMap特点:

- 键的排序: TreeMap中的键是按照自然顺序或自定义比较器进行排序的

- 红黑树实现：TreeMap内部使用红黑树这种自平衡二叉搜索树数据结构来存储键值对
- 键唯一，值可重复
- 线程不安全，如果在多线程环境下使用TreeMap，应该使用Collections工具类处理
- 初始容量：TreeMap没有初始容量的概念，它会根据插入的键值对动态地调整红黑树的大小

TreeMap 自然排序案例：

实例化TreeMap<Integer,String>对象，添加元素并遍历，观察程序运行结果。

```
1  package com.briup.chap08.test;
2
3  import java.util.Map;
4  import java.util.Set;
5  import java.util.TreeMap;
6
7  public class Test085_TreeMap {
8      public static void main(String[] args) {
9          Map<Integer,String> map = new TreeMap<>();
10         map.put(4,"mary");
11         map.put(2,"jack");
12         map.put(1,"tom");
13         map.put(3,"lucy");
14
15         Set<Integer> keys = map.keySet();
16         for(Integer key : keys){
17             System.out.println(key + ": " + map.get(key));
18         }
19     }
20 }
21
```

```
22 //运行结果:
23 1: tom
24 2: jack
25 3: lucy
26 4: mary
```

TreeMap 比较器排序案例:

创建TreeMap<Student, Strin>集合对象, 额外指定比较器, 要求按名字降序, 如果名字相同则按年龄升序。

注意: Student类前面案例已经提供!

```
1 //注意导入自定义Student类
2 import com.briup.chap08.bean.Student;
3
4 //比较器排序
5 public static void main(String[] args) {
6     //1.额外提供比较器对象
7     Comparator<Student> comp = new Comparator<Student>() {
8         @Override
9         public int compare(Student o1, Student o2) {
10             //先按照名字降序
11             int r = o2.getName().compareTo(o1.getName());
12             //如果年龄一样, 再按照年龄升序
13             if(r == 0)
14                 r = o1.getAge() - o2.getAge();
15             return r;
16         }
17     };
18
19     //2.实例化TreeMap对象
20     Map<Student, String> map = new TreeMap<>(comp);
21
22     //3.添加键值对
```

```

23     map.put(new Student("zs", 78), "010");
24     map.put(new Student("rose", 82), "005");
25     map.put(new Student("lucy", 79), "009");
26     map.put(new Student("lucy", 79), "009");
27     map.put(new Student("tom", 68), "019");
28     map.put(new Student("tom", 86), "012");
29     map.put(new Student("ww", 67), "002");
30     //key不能为null, 否则出 NullPointerException
31     //map.put(null, "002");
32
33     //4.遍历
34     Set<Entry<Student, String>> entrySet = map.entrySet();
35     for (Entry<Student, String> entry : entrySet) {
36         System.out.println(entry.getKey() + ": " +
37             entry.getValue());
38     }
39
40     //运行结果:
41     Student [name=zs, age=78]: 010
42     Student [name=ww, age=67]: 002
43     Student [name=tom, age=68]: 019
44     Student [name=tom, age=86]: 012
45     Student [name=rose, age=82]: 005
46     Student [name=lucy, age=79]: 009

```

从结果可知，已经按照自定义比较器规则实现了排序。

思考，如果要求重复的元素也能够放入，如何实现？

修改比较器中返回值即可：当`r==0`，即元素属性一样时，返回非0值即可。

```

1     //1.额外提供比较器对象
2     Comparator<Student> comp = new Comparator<Student>() {
3         @Override

```



```

4      public int compare(Student o1, Student o2) {
5          //先按照名字降序
6          int r = o2.getName().compareTo(o1.getName());
7          //如果年龄一样，再按照年龄升序
8          if(r == 0)
9              r = o1.getAge() - o2.getAge();
10
11          //当r==0，即元素属性一样时，返回非0值即可
12          return (r == 0) ? 1 : r;
13      }
14  };

```

TreeMap小结:

TreeMap底层借助红黑树实现，它提供了高效的有序映射功能，可以用于范围查找、排序和遍历等操作。但红黑树的平衡操作会带来额外的开销，相比于HashMap等实现类，TreeMap在插入和删除操作上可能稍慢。

因此，在选择使用TreeMap时，需要根据具体需求权衡性能和有序性的需求。

8.6 LinkedHashMap

LinkedHashMap是HashMap的一个子类，底层在哈希表的基础上，通过维护一个双向链表来保持键值对的有序性，可以保证存取次序一致。

LinkedHashMap 基础案例:

创建LinkedHashMap<String, Student>对象，添加元素，观察是否存取次序一致。

```

1  package com.briup.chap08.test;
2
3  import java.util.LinkedHashMap;
4  import java.util.Map;
5  import com.briup.chap08.bean.Student;
6
7  // LinkedHashMap可以保证 存取次序一致
8  public class Test086_LinkedHashMap {
9      public static void main(String[] args) {
10         //1.实例化LinkedHashMap类对象
11         Map<String, Student> map = new LinkedHashMap<>();
12
13         //2.添加元素
14         map.put("010", new Student("zs", 78));
15         map.put("005", new Student("rose", 82));
16         map.put("009", new Student("lucy", 70));
17         map.put("019", new Student("lucy", 70));    //value可以
重复
18         map.put("002", null);
19         map.put(null, new Student("ww", 67));
20
21         //3.遍历
22         for (String key : map.keySet()) {
23             System.out.println(key + " " + map.get(key));
24         }
25     }
26 }
27
28 //结果输出:
29 010 Student [name=zs, age=78]
30 005 Student [name=rose, age=82]
31 009 Student [name=lucy, age=70]
32 019 Student [name=lucy, age=70]
33 002 null
34 null Student [name=ww, age=67]

```

可以看出，数据存入Map中的顺序，就是存储的顺序，也是取出的顺序！

8.7 Map小结

集合类	特点	示例
HashMap	基于哈希表实现，无序键值对集合，键和值均可为null。	<pre>Map<String, Integer> map = new HashMap<>();</pre>
TreeMap	基于红黑树实现，按键有序排序，键不可为null。	<pre>Map<String, Integer> map = new TreeMap<>();</pre>
LinkedHashMap	基于哈希表和双向链表实现，按插入顺序排序，键和值均可为null。	<pre>Map<String, Integer> map = new LinkedHashMap<>();</pre>
Hashtable	基于哈希表实现，键值对存储，线程安全，键无序且不允许重复， 键和值都不能为null。	<pre>Map<String, Integer> map = new Hashtable<>();</pre>

9 Collections

数组工具类是 `java.util.Arrays`，可以专门来操作数组对象，提供静态方法，可直接调用。

集合工具类是 `java.util.Collections`，专门来操作集合对象，里面都是静态方法，可以直接调用。

注意，`Collection` 是单列集合根接口，`Collections` 是集合工具类，两者不同！

1) `Collections` 常用方法:

```
1  package java.util;
2
3  public class Collections {
4      // 类外不能实例化对象，工具类主要调用static方法
5      private Collections() {
6      }
7
8      //填充元素值
9      public static <T> void fill(List<? super T> list, T obj) {
10         //省略...
11     }
12
13     //获取最大值
14     public static <T extends Object & Comparable<? super T>> T
max(Collection<? extends T> coll) {
15         Iterator<? extends T> i = coll.iterator();
16         T candidate = i.next();
17
18         while (i.hasNext()) {
19             T next = i.next();
20             if (next.compareTo(candidate) > 0)
21                 candidate = next;
22         }
23         return candidate;
24     }
25
26     //排序
27     @SuppressWarnings("unchecked")
```

```

28     public static <T extends Comparable<? super T>> void
        sort(List<T> list) {
29         list.sort(null);
30     }
31
32     //省略 ...
33 }

```

2) 具体方法讲解

- fill方法，使用指定元素替换指定列表中的所有元素

```

1  List<Integer> list = new ArrayList<>();
2  list.add(1);
3  list.add(2);
4  list.add(3);
5
6  Collections.fill(list, 20);
7
8  for(Integer o : list) {
9      System.out.println(o);
10 }

```

- max方法，根据元素的自然顺序，返回给定集合的最大元素

```

1  List<Integer> list = new ArrayList<>();
2  list.add(1);
3  list.add(9);
4  list.add(3);
5  System.out.println(Collections.max(list));

```

- min方法，根据元素的自然顺序，返回给定集合的最小元素
- reverse方法，反转集合中的元素

```
1 List<Integer> list = new ArrayList<>();
2 list.add(1);
3 list.add(9);
4 list.add(3);
5
6 Collections.reverse(list);
7
8 for(Integer o : list) {
9     System.out.println(o);
10 }
```

- sort方法，根据元素的自然顺序，对指定列表按升序进行排序

```
1 List<Integer> list = new ArrayList<>();
2 list.add(1);
3 list.add(9);
4 list.add(3);
5
6 //如果需要，也可以在第二个参数位置传一个比较器对象
7 //Collections.sort(list,c);
8 Collections.sort(list);
9
10 for(Integer o : list) {
11     System.out.println(o);
12 }
```

- shuffle方法，使用默认随机源对指定列表进行置换

```

1  List<Integer> list = new ArrayList<>();
2  list.add(1);
3  list.add(9);
4  list.add(3);
5
6  Collections.shuffle(list);
7
8  for(Integer o : list) {
9      System.out.println(o);
10 }

```

- addAll方法，往集合中添加一些元素

```

1  List<Integer> list = new ArrayList<>();
2
3  //注意，addAll的第二个参数，是可变参数
4  Collections.addAll(list, 1, 3, 5, 7);
5
6  for(Integer o : list) {
7      System.out.println(o);
8  }

```

- synchronized相关方法，将非线程安全的集合转为线程安全（了解，后续多线程再学习）
 - synchronizedCollection，把非线程安全的Collection类型集合，转为线程安全的集合
 - synchronizedList，把非线程安全的List类型集合，转为线程安全的集合
 - synchronizedSet，把非线程安全的Set类型集合，转为线程安全的集合
 - synchronizedMap，把非线程安全的Map类型集合，转为线程安全的集合

案例展示：

准备一个集合，往里面添加元素，然后使用工具类上述方法对其进行操作，观察效果。

```
1  package com.briup.chap08.test;
2
3  import java.util.ArrayList;
4  import java.util.Collections;
5  import java.util.Comparator;
6  import java.util.List;
7
8  public class Test09_Collections {
9      public static void main(String[] args) {
10         //1.准备集合并添加元素
11         //List<Integer> list = Arrays.asList(5,3,4,2,2,7);
12         List<Integer> list = new ArrayList<>();
13
14         //addAll 往集合中添加多个元素
15         Collections.addAll(list,5,3,4,2,2,7);
16         System.out.println(list);
17
18         //2.max 获取最大值【默认采用自然排序】
19         Integer max = Collections.max(list);
20         System.out.println("max: " + max);
21
22         //3.min 获取集合最小值
23         Integer min = Collections.min(list);
24         System.out.println("min: " + min);
25
26         //4.reverse 反转
27         Collections.reverse(list);
28         System.out.println(list);
29
30         //4.sort 排序【按照 传入的排序算法 进行排序】
31         Comparator<Integer> comp = new Comparator<Integer>() {
32             @Override
33             public int compare(Integer o1, Integer o2) {
```



```

34             // 逆序
35             return o2-o1;
36         }
37     };
38     //注意：排序不会删除集合中的元素
39     Collections.sort(list,comp);
40     System.out.println("逆序： " + list);
41
42     //5.shuffle 随机打乱
43     Collections.shuffle(list);
44     System.out.println("打乱： " + list);
45
46     //6.sort 排序【默认自然排序】
47     Collections.sort(list);
48     System.out.println("自然排序： " + list);
49
50     //7.fill 填充
51     Collections.fill(list, 20);
52     System.out.println("after fill: " + list);
53
54     //8.将ArrayList【线程不安全】的集合 转换成 线程安全集合
55     //List<Integer> list2 =
56     Collections.synchronizedList(list);
57 }

```

注意： Collections.sort排序不会删除集合中的元素。

10 综合案例

设计实现一个斗地主游戏，该游戏可以生成牌、洗牌、发牌，三名玩家可以看自己手里的牌，最后也能看到三张底牌。

具体要求:

- 生成54张扑克牌 (13个数字*4种花色, 再加大小王2张)
- 洗牌: 将54张牌随机打乱
- 准备三个玩家, 三人交替摸牌, 最后三张底牌不能摸
- 查看三人各自手中的牌和底牌, 要求按牌的大小顺序展示

10.1 分析

- 思考: 需要几个集合, 分别保存什么样的数据?
花色信息、点数信息、组合好的牌放在什么地方
- 思考: 如何比较每张牌的大小, 相同数值的牌如何比较大小?
给牌添加对应的权重值
- 思考: 洗牌需要用什么方法才能将集合中的元素顺序打乱?
- 思考: 如何发牌, 有什么注意事项?

10.2 实现

```
1  package com.briup.chap08.test;
2
3  import java.util.ArrayList;
4  import java.util.Collections;
5  import java.util.Comparator;
6  import java.util.HashMap;
7  import java.util.List;
8  import java.util.Map;
9
10 //封装扑克牌类
11 class Card {
```

```

12      // 花色
13      String color;
14      // 点数
15      String number;
16      // 权重（比较大小用的）
17      int weight;
18
19      public Card(String color, String number, int weight) {
20          this.color = color;
21          this.number = number;
22          this.weight = weight;
23      }
24
25      //获取权重值
26      public int getWeight() {
27          return weight;
28      }
29
30      // 重写toString方法，方便输出牌的信息
31      @Override
32      public String toString() {
33          return color + number;
34      }
35  }
36
37  public class Test10_Game {
38      //      牌面值      权重值
39      private static Map<String, Integer> map;
40
41      static {
42          map = new HashMap<>();
43          map.put("3", 1);
44          map.put("4", 2);
45          map.put("5", 3);
46          map.put("6", 4);
47          map.put("7", 5);

```

```

48         map.put("8", 6);
49         map.put("9", 7);
50         map.put("10", 8);
51         map.put("J", 9);
52         map.put("Q", 10);
53         map.put("K", 11);
54         map.put("A", 12);
55         map.put("2", 13);
56         map.put("小王", 14);
57         map.put("大王", 15);
58     }
59
60     //根据牌面值 获取 对应权重值
61     public static int getWeight(String number) {
62         return map.get(number);
63     }
64
65     public static void main(String[] args) {
66         //1.生成54张牌
67         // 准备三个List集合，分别存放花色信息、点数信息、以及组合好
        的54张牌
68         List<String> colors = new ArrayList<>();
69         List<String> numbers = new ArrayList<>();
70         // 提前准备 集合 存储所有牌[54张]
71         List<Card> cards = new ArrayList<>();
72
73         // 向colors集合中添加4种花色
74         Collections.addAll(colors, "♠", "♥", "♣", "♦");
75         // 向numbers集合中添加13种点数
76         Collections.addAll(numbers, "A", "2", "3", "4", "5",
        "6", "7", "8", "9", "10", "J", "Q", "K");
77
78         //循环嵌套 生成所有 Card牌【大小王除外】
79         for (int i = 0; i < colors.size(); i++) {
80             // 获取 花色
81             String color = colors.get(i);

```

```

82         for (int j = 0; j < numbers.size(); j++) {
83             // 获取 牌面值
84             String number = numbers.get(j);
85             // 获取 权重值
86             int weight = getWeight(number);
87
88             //根据 花色 牌面值 权重值 构建 扑克牌对象
89             Card c = new Card(color, number, weight);
90             cards.add(c);
91         }
92     }
93
94     //循环结束，可以生成 4*13==52张牌
95     //生成 大小王
96     cards.add(new Card("", "小王", getWeight("小王")));
97     cards.add(new Card("", "大王", getWeight("大王")));
98
99     //2.将牌随机打乱
100    Collections.shuffle(cards);
101
102    //3.发牌，准备4个集合
103    List<Card> player1 = new ArrayList<>();
104    List<Card> player2 = new ArrayList<>();
105    List<Card> player3 = new ArrayList<>();
106    List<Card> dipai = new ArrayList<>();
107
108    Card c = null;
109    for (int i = 0; i < cards.size(); i++) {
110        //记录当前要发的牌
111        c = cards.get(i);
112        //最后三张牌 index: 51 52 53
113        if (i >= cards.size() - 3) {
114            dipai.add(c);
115        } else if (i % 3 == 0) {
116            player1.add(c);
117        } else if (i % 3 == 1) {

```

```

118         player2.add(c);
119     } else if (i % 3 == 2) {
120         player3.add(c);
121     }
122 }
123
124 //4.看牌【先排序 再看牌】
125 Comparator<Card> comp = new Comparator<Card>() {
126     @Override
127     public int compare(Card o1, Card o2) {
128         //升序
129         return o1.getWeight() - o2.getWeight();
130     }
131 };
132
133 //注意：对已经存在list进行排序，不会删减list元素
134 Collections.sort(player1, comp);
135 Collections.sort(player2, comp);
136 Collections.sort(player3, comp);
137 Collections.sort(dipai, comp);
138
139 //准备map集合，存放玩家名称和手中的牌
140 Map<String,List<Card>> map = new HashMap<>();
141 map.put("player1", player1);
142 map.put("player2", player2);
143 map.put("player3", player3);
144 map.put("dipai", dipai);
145
146 //遍历map集合，输出3个玩家的牌以及底牌
147 for(String key : map.keySet()) {
148     System.out.println(key + ": " + map.get(key));
149 }
150 }
151 }
152
153 //运行结果：每次输出结果可能不同

```

```
154  dipai: [♣6, ♦6, ♠9]
155  player1: [♦3, ♥4, ♠4, ♥5, ♠6, ♠8, ♥8, ♣9, ♥9, ♥J, ♣J, ♦J, ♠Q,
    ♠K, ♦K, ♣K, 小王]
156  player2: [♠3, ♦4, ♣5, ♥6, ♠7, ♣7, ♦7, ♣8, ♠9, ♦10, ♦Q, ♥Q, ♥K,
    ♥A, ♣2, ♥2, 大王]
157  player3: [♥3, ♣3, ♣4, ♠5, ♦5, ♥7, ♦8, ♥10, ♠10, ♣10, ♠J, ♣Q,
    ♠A, ♦A, ♣A, ♦2, ♠2]
```