集合框架

- java集合大家族导图
- java集合详解
- 为了方便多个对象进行操作，要存储多个对象，就不能是一个基本的变量，而应该是一个容器类型的变量,此时的数组为对象数组。而对象数组又不能适应变化的需求，因为数组的长度是固定的，为此提出了集合框架。
- 整个集合框架就围绕一组**标准接口**而设计。你可以直接使用这些接口的标准实现，(++诸如：LinkedList, HashSet, 和 TreeSet 等,除此之外你也可以通过这些接口实现自己的集合++)
- 集合框架被设计成要满足以下几个目标。
  - 该框架必须是高性能的。基本集合（==动态数组，链表，树，哈希表==）的实现也必须是高效的。
  - 该框架允许不同类型的集合，以类似的方式工作，具有高度的互操作性。
  - 对一个集合的扩展和适应必须是简单的。

java 集合框架图

- 从上面的集合框架图可以看到，Java集合框架主要包括两种类型的容器
  - 一种是集合（Collection），存储一个元素集合
  - 另一种是图（Map），存储键/值对映射。
- Collection 接口又有 3 种子类型:List、Set和Queue，
- 再下面是一些抽象类，最后是具体实现类

集合框架体系

- 使用的数据类型都是引用类型或者包装类型
- 集合框架是一个用来代表和操纵集合的统一架构。所有的集合框架都包含如下内容：
  - **接口**：是代表集合的抽象数据类型。例如 Collection、List、Set、Map 等。之所以定义多个接口，是为了以不同的方式操作集合对象
  - **实现（类）**：是集合接口的具体实现。从本质上讲，它们是可重复使用的数据结构，例如：ArrayList、LinkedList、HashSet、HashMap。
  - **算法**：是实现集合接口的对象里的方法执行的一些有用的计算，例如：搜索和排序。这些算法被称为多态，那是因为相同的方法可以在相似的接口上有着不同的实现

- 集合导图
- 除了集合，该框架也定义了几个 Map 接口和类。Map 里存储的是键/值对。java集合框架位于java.util包中,存储的时候都是以object类存储，因此在从集合中取出的数据要进行强制类型转换。
- List 有三个儿子，我们到底使用谁呢?视需求而定，要安全吗?

    - 要：Vector(即使安全，也不用，因为有替代品)
    - 不要：ArrayList 或者LinkedList
        - 查询多：ArrayList
        - 增删多：LinkedList
- Collections类可以对集合进行排序、查找和替换操作:
    - sort():排序
        - binarySearch()：
        - 查找max()\min():查找最大\最小值

集合接口的描述

- **Collection接口**:Collection 是最基本的集合接口，一个 Collection 代表一组 Object，即 Collection 的元素,
    - Java不提供直接继承自Collection的类，只提供继承于的子接口(如List和set)。
    - Collection接口存储一组不唯一，无序的对象。
- **Lsit接口**:List接口是collection的子接口
    - Lsit接口是一个有序的 Collection，使用此接口能够精确的控制每个元素插入的位置，能够通过索引(元素在List中位置，类似于数组的下标)来访问List中的元素，第一个元素的索引为 0，而且允许有相同的元素。
    - List 接口存储一组不唯一，有序（插入顺序）的对象。
- **Set接口**：
    - Set 具有与 Collection 完全一样的接口，只是行为上不同，Set 不保存重复的元素。
    - Set接口存储一组唯一，无序的对象。
- **SortedSet**: ==继承==于Set保存有序的集合。
- **Map**: Map 接口存储一组键值对象，提供key（键）到value（值）的映射。
- **Map.Entry**:描述在一个Map中的一个元素（键/值对）。是一个Map的内部类。
- ** SortedMap**: ==继承==于 Map，使 Key 保持在升序排列。
- **Enumeration(被迭代器取代)**: 这是一个传统的接口和定义的方法，通过它可以枚举（一次获得一个）对象集合中的元素。

Set和List的区别

- Set 接口实例存储的是无序的，不重复的数据。List 接口实例存储的是有序的，可以重复的元素。
- Set检索效率低下，删除和插入效率高，插入和删除不会引起元素位置改变 <==实现类有HashSet,TreeSet==>。
- List和数组类似，可以动态增长，根据实际存储的数据的长度自动增长List的长度。查找元素效率高，插入删除效率低，因为会引起其他元素位置改变 <实现类有==ArrayList,LinkedList,Vector==> (List默认长度为10，自动扩容第一次到15，第二次加7,即增加当前长度的50%)

集合的实现类

- **AbstractCollection**:实现了大部分的集合接口。
- **AbstractList**:继承于AbstractCollection 并且实现了大部分List接口。

- **AbstractSequentialList**:继承于 AbstractList，提供了对数据元素的链式访问而不是随机访问。
- **LinkedList**: 该类实现了List接口，允许有null（空）元素
  - 主要用于创建链表数据结构，该类没有同步方法，如果多个线程同时访问一个List，则必须自己实现访问同步，解决方法就是在创建List时候构造一个同步的List。

```
List list=Collections.synchronizedList(newLinkedList(...));
```

- **ArrayList**: 该类也是实现了List的接口，++实现了可变大小的数组++，随机访问和遍历元素时，提供更好的性能。==该类也是非同步的,在多线程的情况下不要使用==。ArrayList 增长当前长度的50%，插入删除效率低。
- **AbstractSet**： 继承于AbstractCollection并且实现了大部分Set接口。
- **HashSet**: 该类实现了Set接口，不允许出现重复元素，不保证集合中元素的顺序，==允许包含值为null的元素，但最多只能一个==。
- **LinkedHashSet**: 具有可预知迭代顺序的Set接口的哈希表和链接列表实现。
- **TreeSet**: 该类实现了Set接口，可以实现排序等功能。
- **AbstractMap**: 实现了大部分的Map接口
- **HashMap**: HashMap 是一个散列表，它存储的内容是键值对(key-value)映射。该类实现了Map接口，根据键的HashCode值存储数据，具有很快的访问速度，==最多允许一条记录的键为null，不支持线程同步==。
- **TreeMap**: 继承了AbstractMap，并且使用一颗树。
- **WeakHashMap**: 继承AbstractMap类，使用弱密钥的哈希表。
- **LinkedHashMap**:继承于HashMap，使用元素的自然顺序对元素进行排序.
- **IdentityHashMap**:继承AbstractMap类，比较文档时使用引用相等。

集合相关的使用方法

![](https://ws1.sinaimg.cn/large/d8441690ly1ft1sz2jbgdj20dx0g5jwp.jpg

)

- 实现一个类的对象之间比较大小，该类要实现Comparable接口 重写compareTo()方法

方法名	说明
boolean add(Object o)	在列表的末尾顺序添加元素，起始索引位置从0开始
void add(int index,Object o)	在指定的索引位置添加元素。索引位置必须介于0和列表中元素个数之间
int size()	返回列表中的元素个数
Object get(int index)	返回指定索引位置处的元素。取出的元素是Object类型，使用前需要进行强制类型转换
boolean contains(Object o)	判断列表中是否存在指定元素
boolean remove(Object o)	从列表中删除元素
Object remove(int index)	从列表中删除指定位置元素，起始索引位置从0开始

Collection接口常用通用方法还有：clear()、isEmpty()、iterator()、toArray()

方法名	说明
void addFirst(Object o)	在列表的首部添加元素
void addLast(Object o)	在列表的末尾添加元素
Object getFirst()	返回列表中的第一个元素
Object getLast()	返回列表中的最后一个元素
Object removeFirst()	删除并返回列表中的第一个元素
Object removeLast()	删除并返回列表中的最后一个元素

类的描述

- **Vector**:该类和ArrayList非常相似，但是该类是同步的，可以用在多线程的情况，该类允许设置默认的增长长度，默认扩容方式为原来的2倍。
- **Stack**:栈是Vector的一个子类，它实现了一个标准的后进先出的栈。
- **Dictionary**: Dictionary 类是一个抽象类，用来存储键/值对，作用和Map类相似
- **Hashtable**:Hashtable 是 Dictionary(字典) 类的子类，位于 java.util 包中。
- **Properties**:Properties 继承于 Hashtable，表示一个持久的属性集，属性列表中每个键及其对应值都是一个字符串。
- **BitSet**:一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。

集合算法

- 集合框架定义了几种算法，可用于集合和映射。这些算法被定义为集合类的静态方法。
- 集合定义三个静态的变量：EMPTY_SET，EMPTY_LIST，EMPTY_MAP的。这些变量都不可改变。
- 在尝试比较不兼容的类型时，一些方法能够抛出 ClassCastException异常。当试图修改一个

> 不可修改的集合时，抛出UnsupportedOperationException异常。

如何使用迭代器(Iterator和ListIterator是接口)

> - 一般遍历数组都是采用for循环或者增强for，这两个方法也可以用在集合框架，但是还有一种方法是采用==迭代器遍历集合框架==，它是一个对象，实现了Iterator 接口或ListIterator 接口。
> - **迭代器**，使你能够通过循环来得到或删除集合的元素
> - ListIterator 继承了Iterator，==以允许双向遍历列表和修改元素==。

三种遍历集合的方法：

> - 第三种方法是采用迭代器的方法，该方法可以不用担心在遍历的过程中会超出集合的长度。

```
import java.util.*;

public class Test{
 public static void main(String[] args) {
     List<String> list=new ArrayList<String>();
     list.add("Hello");
     list.add("World");
     list.add("HAHAHAHA");
     //第一种遍历方法使用foreach遍历List
     for (String str : list) {  //也可以改写for(int i=0;i<list.size();i++)这种形式
        System.out.println(str);
     }
     //第二种遍历，把链表变为数组相关的内容进行遍历
     String[] strArray=new String[list.size()];
     list.toArray(strArray);
     for(int i=0;i<strArray.length;i++) //这里也可以改写为  foreach(String str:strArray)这种形式
     {
        System.out.println(strArray[i]);
     }
    //第三种遍历 使用迭代器进行相关遍历

     Iterator<String> ite=list.iterator();
     while(ite.hasNext())//判断下一个元素之后有值
     {
         System.out.println(ite.next());
     }
 }
}
```

遍历map

```
import java.util.*;

public class Test{
```

```
    public static void main(String[] args) {
     Map<String, String> map = new HashMap<String, String>();
     map.put("1", "value1");
     map.put("2", "value2");
     map.put("3", "value3");
     //第一种：普遍使用，二次取值
     System.out.println("通过Map.keySet遍历key和value：");
     for (String key : map.keySet()) {
      System.out.println("key= "+ key + " and value= " + map.get(key));
     }
     //第二种
     System.out.println("通过Map.entrySet使用iterator遍历key和value：");
     Iterator<Map.Entry<String, String>> it = map.entrySet().iterator();
     while (it.hasNext()) {
      Map.Entry<String, String> entry = it.next();
      System.out.println("key= " + entry.getKey() + " and value= " +
entry.getValue());
     }
     //第三种：推荐，尤其是容量大时
     System.out.println("通过Map.entrySet遍历key和value");
     for (Map.Entry<String, String> entry : map.entrySet()) {
      System.out.println("key= " + entry.getKey() + " and value= " +
entry.getValue());
     }
     //第四种
     System.out.println("通过Map.values()遍历所有的value，但不能遍历key");
     for (String v : map.values()) {
      System.out.println("value= " + v);
     }
    }
}
```

如何使用比较器

> - TreeSet和TreeMap的按照排序顺序来存储元素. 然而，这是通过比较器来精确定义按照什么样的排序顺序,这个接口可以让我们以不同的方式来排序一个集合。
> - 使用 Java Comparator: 这里通过实例列出Comparator接口提供的所有方法

集合与数组的区别

> - 长度区别
>   - 数组的长度固定
>   - 集合长度可变
> - 内容不同
>   - 数组存储的是同一种类型的元素
>   - 集合是一个对象，可容纳其他对象的引用。集合接口声明对每一种类型的集合可以执行的操作。
> - 元素的数据类型问题

- 数组可以存储基本数据类型，也可以存储引用数据类型
- 集合只能存储引用类型。

集合框架并发修改异常的产生原因及解决方案

- 问题：有一集合，需要判断里面有无某个元素，如果有，添加一个新元素，请用代码实现。
  - 结果：ConcurrentModificationException： 当方法检测到对象的并发修改，但不允许这种修改时，抛出此异常。
  - 产生的原因： ==迭代器是依赖与集合而存在的==，在判断成功后，集合中新添加了元素，而迭代器却不知道，所以报错了，==这个错误叫并发修改异常==,其实这个问题描述的是：迭代器遍历元素的时候，通过集合是不能修改元素的。
- 如何解决呢?
  - 迭代器迭代元素，迭代器修改元素 而Iterator 迭代器却没有添加元素功能，所以我们使用其子接口ListIterator，结果是元素添加在刚才迭代的元素之后。
  - 集合遍历元素，集合修改元素(普通for循环)；结果是新添加元素在最后添加的。

set集合去重原理

```
package org.u2.Day0319;
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;
public class demo {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("请输入数字：");
        int n = input.nextInt();
        Set set = new HashSet();
        int a = n / 2;
        for (int i = 0; i <=a ; i++) {
            //异或去重
            int q =i ^ (n - i);
            set.add(q);
        }
        System.err.println(set.size());
    }
}
```

总结：

- Java集合框架为程序员提供了预先包装的数据结构和算法来操纵他们。
- 集合是一个对象，可容纳其他对象的引用。集合接口声明对每一种类型的集合可以执行的操作。
- 集合框架的类和接口均在java.util包中。
- 任何对象加入集合类后，自动转变为Object类型，所以在取出的时候，需要进行强制类型转换。