Java 类集框架

最早的时候可以通过对象数组保存一组数据，但是慢慢的发现如果程序中都使用对象数组开发的话，本身会存在大小的限制问题，即：所有的数组的大小是不可改变的，但是从实际的开发来看，有很多的时候是根本就无法知道到底要开辟多少的数组空间，后来通过链表解决类此类问题，但是如果每一次的开发中都使用链表的话，肯定很麻烦，所以在 Java 中专门提供了一套动态对象数组的操作类——类集框架，在 Java 中类集框架实际上也就是对数据结构的 Java 实现。

在 Java 中类集框架里，为了操作方便提供了一系列的类集的操作接口，主要的操作接口有以下三个：

- Collection：存放单值的最大父接口

public interface Collection<E> extends Iterable<E>

- Map：是存放一对的内容

public interface Map<K,V>

- Iterator：输出作用

public interface Iterator<E>

在 JDK 1.5 之后这些接口中都增加了泛型的定义，最早的时候这三个接口中的内容都使用 Object 进行操作，但是很明显这样是会存在安全问题的，那么在 JDK 1.5 之后使用了泛型，那么这种安全性的问题就解决了，此时的类集真正的可以达到了以相同的类型或高度进行操作。

在以上的三个接口中 Collection 接口并不会被直接使用，而都使用它的两个子接口：List、Set。

No.	方法名称	类型	描述
1	public boolean add(E e)	普通	向集合中增加元素
2	public void clear()	普通	删除集合中的全部内容
3	public boolean contains(Object o)	普通	判断指定内容是否存在
4	public Iterator<E> iterator()	普通	为 Iterator 接口实例化
5	public boolean remove(Object o)	普通	从集合中删除元素
6	public int size()	普通	取得集合的大小
7	public Object[] toArray()	普通	将集合变为对象数组输出
8	public <T> T[] toArray(T[] a)	普通	将集合变为对象数组输出

允许重复的子接口：List

List 接口本身属于 Collection 的子接口，但是 List 子接口本身大量的扩充了 Collection 接口，主要的扩充方法如下：

No.	方法名称	类型	描述
1	public void add(int index,E element)	普通	在指定的位置上增加内容
2	**public E get(int index)**	**普通**	**取得指定位置上的内容**
3	public E set(int index,E element)	普通	修改指定位置的内容
4	public ListIterator<E> listIterator()	普通	为 ListIterator 接口实例化
5	public E remove(int index)	普通	删除指定位置上的内容

既然要使用接口，那么就一定要依靠子类进行父接口的实例化

List 接口子类 ArrayList

ArrayList 子类是在进行 List 接口操作中使用最多的一个子类，那么此类定义如下：

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

那么下面通过代码来观察基本的使用。

```
package cn.edu.tjpu.cs.listdemo;
import java.util.ArrayList;
import java.util.List;
public class ArrayListDemo01 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>(); //  实例化 List 接口
        all.add("hello"); //  向集合中增加内容
        all.add("world"); //  向集合中增加内容
        all.add("!!!"); //  向集合中增加内容
        for (int x = 0; x < all.size(); x++) {
            System.out.println(all.get(x)) ;
        }
    }
}
```

本程序的操作代码的形式与之前的链表操作非常的类似，所以，类集的主要功能就是增加和取出数据。

以上的操作功能由于 get()方法只是 List 接口才有的，那么以后这种操作只能适合于 List 接口，如果现在接收对象的不是 List 了，而是 Collection 呢？那么如果要想输出，则必须将所有的集合变成对象数组完成。

```
package cn.edu.tjpu.cs.listdemo;
import java.util.ArrayList;
import java.util.Collection;
public class ArrayListDemo02 {
    public static void main(String[] args) {
        Collection<String> all = new ArrayList<String>(); //  实例化 List 接口
        all.add("hello"); //  向集合中增加内容
        all.add("world"); //  向集合中增加内容
        all.add("!!!"); //  向集合中增加内容
        all.remove("!!!"); //  从集合中删除指定对象
```

```
        Object obj[] = all.toArray(); //  将所有的内容变为对象数组
        for (int x = 0; x < obj.length; x++) {
            String str = (String) obj[x];
            System.out.print(str + " 、 ");
        }
    }
}
```

不允许重复的子接口：Set

List 接口中的内容是允许重复的，但是如果现在要求集合中的内容不允许重复的话，则就可以使用 Set 子接口完成，Set 接口并不像 List 接口那样对 Collection 接口进行了大量的扩充，而与 Collection 接口的定义是完全一样的。

与 List 接口一样，如果要想使用 Set 接口则一定也要通过子类进行对象的实例化，常用的两个子类：HashSet、TreeSet。

HashSet 本身是 Set 的子类，此类的定义如下：

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

与 ArrayList 类的定义结构是非常类似的，也是继承了一个抽象类，而且实现了接口。

```
package cn.edu.tjpu.cs.setdemo;
import java.util.HashSet;
import java.util.Set;
public class HashSetDemo {
    public static void main(String[] args) {
        Set<String> all = new HashSet<String>();
        all.add("hello");
        all.add("hello"); //  重复设置
        all.add("world");
        all.add("!!!");
        System.out.println(all) ;
    }
}
```

在 Set 接口中不允许有重复的元素出现，而且发现与 List 接口不同的是，List 采用的是顺序的方式加入的元素，而 Set 中的内容并没有任何的顺序，属于散列存放的。

TreeSet 子类的内容是允许进行排序的，那么下面就使用这个子类完成一个任意类型的排序操作。

如果多个对象要想进行排序，则无论在何种情况下都必须使用 Comparable 接口完成，用于指定排序的规则。但是在进行排序的时候实际上每一个类中的属性最好都进行判断。

```
package cn.edu.tjpu.cs.setdemo.sort;
import java.util.Set;
import java.util.TreeSet;
class Person implements Comparable<Person> {
```

```
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String toString() {
        return " 姓名： " + this.name + " ， 年龄： " + this.age;
    }
    @Override
    public int compareTo(Person o) {
        if (this.age < o.age) {
            return 1;
        } else if (this.age > o.age) {
            return -1;
        } else {
            return this.name.compareTo(o.name);
        }
    }
}
public class SortDemo {
    public static void main(String[] args) {
        Set<Person> all = new TreeSet<Person>();
        all.add(new Person(" 张三 ", 20));
        all.add(new Person(" 李四 ", 20));
        all.add(new Person(" 李四 ", 20));
        all.add(new Person(" 王五 ", 19));
        System.out.println(all);
    }
}
```

集合输出迭代器 Iterator

Iterator 本身是一个专门用于输出的操作接口，其接口定义了三种方法：

No.	方法名称	类型	描述
1	public boolean hasNext()	普通	判断是否有下一个元素
2	public E next()	普通	取出当前元素
3	public void remove()	普通	删除当前内容

在 Collection 接口中已经定义了 iterator()方法，可以为 Iterator 接口进行实例化操作。

```
package cn.edu.tjpu.cs.printdemo;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
```

```
public class IteratorDemo {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();
        all.add("hello");
        all.add("world");
        Iterator<String> iter = all.iterator();
        while (iter.hasNext()) { //  指针向下移动，判断是否有内容
            String str = iter.next();
            System.out.print(str + "  、  ");
        }
    }
}
```

Map 接口

Collection 接口操作的时候每次都会向集合中增加一个元素，但是如果现在增加的元素是一对的话，则就可以使用 Map 接口完成功能，Map 接口的定义如下：

public interface Map<K,V>

里面需要同时指定两个泛型，主要的原因，Map 中的所有保存数据都是按照“key -> value”的形式存放的，例如：以电话号码本为例：

- 张三：123456
- 李四：234567
- 王五：345678

以上的数据每次保存的时候都是按照一对的形式存放的，如果现在要找到张三的电话，很明显张三是一个 key，而他的电话就是一个 value。

在 Map 接口中有以下几个常用方法：

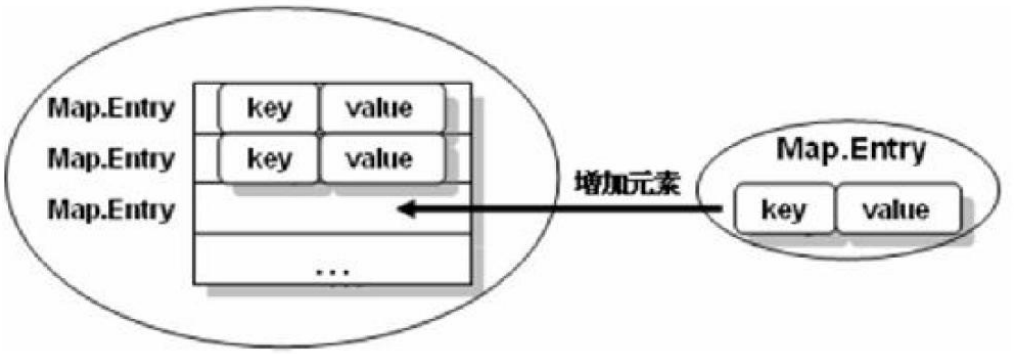
No.	方法名称	类型	描述
1	public V put(K key,V value)	普通	向集合中增加元素
2	public V get(Object key)	普通	根据 key 取得 value
3	public Set<K> keySet()	普通	取出所有的 key
4	public Collection<V> values()	普通	取出所有的 value
5	public V remove(Object key)	普通	删除一个指定的 key
6	public Set<Map.Entry<K,V>> entrySet()	普通	将所有的集合变为 Set 集合

需要说明的是，在 Map 接口中还定义了一个内部接口 —— Map.Entry。

public static interface Map.Entry<K,V>

Entry 是在 Map 接口中使用的 static 定义的内部接口，所以就是一个外部接口。

Map与Map.Entry



子类：HashMap

如果要使用 Map 接口的话，可以使用 HashMap 子类为接口进行实例化操作。

```
package cn.edu.tjpu.cs.mapdemo;
import java.util.HashMap;
import java.util.Map;
public class HashMapDemo01 {
    public static void main(String[] args) {
        Map<String, String> all = new HashMap<String, String>();
        all.put("BJ", "BeiJing");
        all.put("NJ", "NanJing");
        String value = all.get("BJ"); //  根据 key 查询出 value
        System.out.println(value);
        System.out.println(all.get("TJ"));
    }
}
```

在 Map 的操作中，可以发现，是根据 key 找到其对应的 value，如果找不到，则内容为 null。

而且现在由于使用的是 HashMap 子类，所以里面的 key 允许一个为 null。

```
package cn.edu.tjpu.cs.mapdemo;
import java.util.HashMap;
import java.util.Map;
public class HashMapDemo02 {
    public static void main(String[] args) {
        Map<String, String> all = new HashMap<String, String>();
        all.put("BJ", "BeiJing");
        all.put("NJ", "NanJing");
        all.put(null, "NULL");
        System.out.println(all.get(null));
    }
}
```

现在所有的内容都有了，下面可以通过 keySet()方法取出所有的 key 集合。

```
package cn.edu.tjpu.cs.mapdemo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class HashMapDemo03 {
	public static void main(String[] args) {
		Map<String, String> all = new HashMap<String, String>();
		all.put("BJ", "BeiJing");
		all.put("NJ", "NanJing");
		all.put(null, "NULL");
		Set<String> set = all.keySet();
		Iterator<String> iter = set.iterator();
		while (iter.hasNext()) {
			String key = iter.next() ;
			System.out.println(key + " --> " + all.get(key)) ;
		}
	}
}
```

Map 集合的输出

按照最正统的做法，所有的 Map 集合的内容都要依靠 Iterator 输出，以上虽然是完成了输出，但是完成的不标准，Map 集合本身并不能直接为 Iterator 实例化，如果此时非要使用 Iterator 输出 Map 集合中内容的话，则要采用如下的步骤：

1、将所有的 Map 集合通过 entrySet()方法变成 Set 集合，里面的每一个元素都是 Map.Entry 的实例；

2、利用 Set 接口中提供的 iterator()方法为 Iterator 接口实例化；

3、通过迭代，并且利用 Map.Entry 接口完成 key 与 value 的分离。

```
package cn.edu.tjpu.cs.mapdemo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class MapPrint {
	public static void main(String[] args) {
		Map<String, String> all = new HashMap<String, String>();
		all.put("BJ", "BeiJing");
		all.put("NJ", "NanJing");
		all.put(null, "NULL");
		Set<Map.Entry<String, String>> set = all.entrySet();
		Iterator<Map.Entry<String, String>> iter = set.iterator();
		while (iter.hasNext()) {
```

```
            Map.Entry<String, String> me = iter.next();
            System.out.println(me.getKey() + " --> " + me.getValue());
        }
    }
}
```

有序的存放：TreeMap

HashMap 子类中的 key 都属于无序存放的，如果现在希望有序（按 key 排序）则可以使用 TreeMap 类完成，但是需
要注意的是，由于此类需要按照 key 进行排序，而且 key 本身也是对象，那么对象所在的类就必须实现 Comparable 接口。

```
package cn.edu.tjpu.cs.mapdemo;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
public class TreeMapDemo {
public static void main(String[] args) {
        Map<String, String> all = new TreeMap<String, String>();
        all.put("BJ", "BeiJing");
        all.put("NJ", "NanJing");
        Set<Map.Entry<String, String>> set = all.entrySet();
        Iterator<Map.Entry<String, String>> iter = set.iterator();
        while (iter.hasNext()) {
            Map.Entry<String, String> me = iter.next();
            System.out.println(me.getKey() + " --> " + me.getValue());
        }
    }
}
```

集合类总结

接口名称	实现	描述/特长	缺点
List	ArrayList, LinkedList	元素按照插入顺序排列	查询速度慢，在任意位置插入删除元素也很慢
Set	HashSet, TreeSet	一组各不相同的元素，查询速度快	没有索引；不能随机访问任意元素
Map	HashMap TreeMap	一组“键”和“值”的关联	通用性不好；不能反向从值获得对应的键

集合类应用：实现一对多关系

一个人有多本书，要求通过程序描述。

```
package cn.edu.tjpu.cs.demo01;
import java.util.ArrayList;
import java.util.List;
public class Person {
	private String name;
	private int age;
	private List<Book> books;
	public Person() {
		super();
		this.books = new ArrayList<Book>();
	}
	public Person(String name, int age) {
		this();
		this.name = name;
		this.age = age;
	}
	public List<Book> getBooks() {
		return this.books;
	}
	@Override
	public String toString() {
		return " 姓名： " + this.name + " ，年龄： " + this.age;
	}
}
```

因为现在根本就无法明确的知道一个人有多少本书。

```
package cn.edu.tjpu.cs.demo01;
public class Book {
	private String title;
	private Person person;
	public Book(String title) {
		this.title = title;
	}
	public Person getPerson() {
		return person;
	}
	public void setPerson(Person person) {
		this.person = person;
}
```

```
    @Override
    public String toString() {
        return "  书名：   " + this.title;
    }
}
```

在主方法中设置两者的关系。

```
package cn.edu.tjpu.cs.demo01;
import java.util.Iterator;
public class Test {
    public static void main(String[] args) {
        Person per = new Person("  张三  ", 20);
        Book b1 = new Book("Java");
        Book b2 = new Book("WEB");
        per.getBooks().add(b1);
        per.getBooks().add(b2);
        b1.setPerson(per);
        b2.setPerson(per);
        System.out.println(per);
        Iterator<Book> iter = per.getBooks().iterator();
        while (iter.hasNext()) {
            System.out.println("\t|- " + iter.next()) ;
        }
    }
}
```

集合类应用：实现多对多关系

一个学生可以参加多门课程，一门课程有多个学生参加，现在要求通过一个学生可以找到他所参加的全部课程，也可以通过一门课程找到参加本课程的所有学生。具体程序自行实现。

```
package cn.edu.tjpu.cs.demo02;
import java.util.List;
public class Course {
    private List<Student> students ;
}
package cn.edu.tjpu.cs.demo02;
import java.util.List;
public class Student {
    private List<Course> courses ;
}
```