**3.1、认识类集（理解）**

如果现在要想保存多个对象，肯定使用对象数组完成，但是对象数组本身有一个最大的问题在于其数据的长度，所以后来使用了链表完成了动态对象数组的开发，可是链表的开发难度实在是很大，而且如果一个链表要想真正去使用，只依靠之前所编写的还不够，还需要进行一些代码的调优。

而在JDK 1.2之后正式引入了类集的概念，类集是一种动态的对象数组，属于各个数据结构的实现类，在整个类集之中主要的组成是一些核心的操作接口：**Collection、List、Set、Map、Iterator、Enumeration**。

**3.2、单值保存的最大父接口：Collection（重点）**

所谓的单值保存指的是每一次操作只会保存一个对象，就好像之前的链表程序一样，每一次只保存了一个对象，在Collection接口之中定义了如下的一些操作方法。

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **方法名称** | **类型** | **描述** |
| **1** | **public boolean add(E e)** | **普通** | **数据增加** |
| 2 | public void clear() | 普通 | 清除数据 |
| 3 | public boolean contains(Object o) | 普通 | 查找数据是否存在 |
| 4 | public boolean isEmpty() | 普通 | 判断是否为空集合 |
| **5** | **public Iterator<E> iterator()** | **普通** | **为Iterator接口实例化** |
| 6 | public boolean remove(Object o) | 普通 | 删除数据 |
| 7 | public int size() | 普通 | 取得集合的个数 |
| 8 | public Object[] toArray() | 普通 | 将集合变为对象数组 |

在Collection接口之中一共定义了15个方法，在所有的方法之中，只有两个方法最为常用：add()、iterator()。不过从开发上讲，很少会去直接使用Collection，都会使用Collection的两个子接口：List、Set。

**3.3、允许重复的子接口：List（重点，80%）**

List是Collection的一个最为常用的子接口，首先这个接口的定义如下：

|  |
| --- |
| public interface List<E> extends Collection<E> |

但是List接口对Collection接口进行了大量的扩充，但是扩充之后的主要方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **方法名称** | **类型** | **描述** |
| 1 | public E get(int index) | 普通 | 取得指定索引位置上的数据 |
| 2 | public E set(int index, E element) | 普通 | 修改指定索引位置上的数据 |
| 3 | public ListIterator<E> listIterator() | 普通 | 为ListIterator接口实例化 |

但是以上的三个方法，只是针对于List接口起作用，而List接口有两个常用子类：ArrayList、Vector。

**3.3.1、新的子类：ArrayList，95%**

ArrayList是List子接口使用最多的一个子类，而这个类的定义如下：

|  |
| --- |
| public class ArrayList<E>  extends AbstractList<E>  implements **List<E>**, RandomAccess, Cloneable, Serializable |

按照面向对象的概念来讲，现在使用ArrayList主要的目的是为List接口实例化，所有的操作方法都以List接口为主。

**范例：**使用ArrayList进行List接口的功能验证

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  List<String> all = **new** ArrayList<String>() ;  all.add("Hello") ;  all.add("Hello") ; // 内容重复了  all.add("World") ;  **for** (**int** x = 0; x < all.size(); x++) {  String str = all.get(x) ; // get()方法只有List接口有  System.*out*.print(str + "、");  }  }  } |

在使用代码的时候可以发现，List集合之中即使存在了重复数据，也可以正常的保存，而且数据保存的顺序就是存入数据的顺序。

**范例：**使用List集合修改之前的程序

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.ArrayList;  **import** java.util.List;  **interface** Animal { // 动物  **public** String getName() ;  **public** **int** getAge() ;  }  **class** Zoo {  **private** List<Animal> animals = **new** ArrayList<Animal>() ; // 多个动物  **public** **void** add(Animal ani) { // 增加动物  **this**.animals.add(ani) ; // 增加动物  }  **public** **void** delete(Animal ani) {  **this**.animals.remove(ani) ; // 需要equals()  }  **public** List<Animal> search(String keyWord) {  List<Animal> result = **new** ArrayList<Animal>() ;  **for** (**int** x = 0 ; x < **this**.animals.size() ; x ++) {  Animal ani = **this**.animals.get(x) ;  **if** (ani.getName().contains(keyWord)) { // 满足  result.add(ani) ;  }  }  **return** result ;  }  }  **class** Dog **implements** Animal {  **private** String name ;  **private** **int** age ;  **public** Dog(String name,**int** age) {  **this**.name = name ;  **this**.age = age ;  }  **public** String getName() {  **return** **this**.name ;  }  **public** **int** getAge() {  **return** **this**.age ;  }  **public** **boolean** equals(Object obj) {  **if** (**this** == obj) {  **return** **true** ;  }  **if** (obj == **null**) {  **return** **false** ;  }  **if** (!(obj **instanceof** Dog)) {  **return** **false** ;  }  Dog dog = (Dog) obj ;  **if** (**this**.name.equals(dog.name)  && **this**.age == dog.age) {  **return** **true** ;  }  **return** **false** ;  }  **public** String toString() {  **return** "〖狗的信息〗名字：" + **this**.name + "，年龄：" + **this**.age ;  }  }  **class** Tiger **implements** Animal {  **private** String name ;  **private** **int** age ;  **public** Tiger(String name,**int** age) {  **this**.name = name ;  **this**.age = age ;  }  **public** String getName() {  **return** **this**.name ;  }  **public** **int** getAge() {  **return** **this**.age ;  }  **public** **boolean** equals(Object obj) {  **if** (**this** == obj) {  **return** **true** ;  }  **if** (obj == **null**) {  **return** **false** ;  }  **if** (!(obj **instanceof** Tiger)) {  **return** **false** ;  }  Tiger t = (Tiger) obj ;  **if** (**this**.name.equals(t.name)  && **this**.age == t.age) {  **return** **true** ;  }  **return** **false** ;  }  **public** String toString() {  **return** "〖老虎的信息〗名字：" + **this**.name + "，年龄：" + **this**.age ;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String args[]) {  Zoo zoo = **new** Zoo() ; // 动物园  zoo.add(**new** Dog("花狗",1)) ;  zoo.add(**new** Dog("黄狗",1)) ;  zoo.add(**new** Dog("黑狗",1)) ;  zoo.add(**new** Dog("斑点狗",1)) ;  zoo.add(**new** Tiger("斑点虎",2)) ;  zoo.add(**new** Tiger("黑虎",2)) ;  zoo.add(**new** Tiger("花虎",2)) ;  zoo.delete(**new** Dog("斑点狗",1)) ; // 删除  List<Animal> result = zoo.search("斑点") ;  **for** (**int** x = 0 ; x < result.size() ; x ++) {  System.*out*.println(result.get(x)) ;  }  }  } |

至少此时的程序不再需要自己去开发链表了，所有的链表的实现类都有了。

**3.3.2、旧的子类：Vector，5%**

Vector类是在JDK 1.0的时候就推出的一个最早的实现动态数组的操作类，实际上对于今天而言，有许多的类上依然还是在使用着Vector，不过从实际的开发来讲，现在设计的一些程序都是针对于接口的操作了。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.List;  **import** java.util.Vector;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  List<String> all = **new** Vector<String>() ;  all.add("Hello") ;  all.add("Hello") ; // 内容重复了  all.add("World") ;  **for** (**int** x = 0; x < all.size(); x++) {  String str = all.get(x) ; // get()方法只有List接口有  System.*out*.print(str + "、");  }  }  } |

因为所有的操作都是针对于接口完成的，接口定义的方法不变，子类随便变。

**面试题：**请解释ArrayList和Vector的区别？

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **区别** | **ArrayList** | **Vector** |
| 1 | 推出时间 | JDK 1.2 | JDK 1.0 |
| 2 | 性能 | 采用异步处理方式，性能更高 | 采用同步处理方式，性能相对较低 |
| 3 | 安全性 | 非线程安全 | 线程安全 |
| 4 | 输出 | Iterator、ListIterator、foreach | Iterator、ListIterator、foreach、Enumeration |

从实际开发而言，几乎都是开发异步程序，所以首选的肯定是ArrayList子类。

**3.4、不允许重复的子接口：Set（重点），20%**

Set也是一个Collection较为常用的子接口，这个接口的定义如下：

|  |
| --- |
| public interface Set<E> extends Collection<E> |

在Collection接口定义了15个方法，但是Set子接口并不像List子接口那样对Collection接口进行了大量的扩充，而是完整的继承了下来，那么就证明了在Set子接口之中是肯定无法使用get()方法的。

那么在Set子接口之中常用的两个子类：HashSet、TreeSet，下面分别说明。

**3.4.1、散列存放的子类：HashSet，80%**

Hash（哈希）属于一种算法，这种算法的核心意义指的是找空保存算法，所以只要一看见hash第一反应就是说没有顺序的保存。

**范例：**观察Set接口使用

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.HashSet;  **import** java.util.Set;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Set<String> all = **new** HashSet<String>() ;  all.add("Hello") ;  all.add("Hello") ; // 内容重复了  all.add("World") ;  System.*out*.println(all);  }  } |

保存数据再输出之后可以发现，重复的数据没有了，并且其本身的保存也是没有任何顺序的。

**3.4.2、排序存放的子类：TreeSet，20%**

如果现在希望Set集合之中保存的数据有顺序，那么就通过TreeSet进行Set接口的实例化。

**范例：**使用TreeSet

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Set;  **import** java.util.TreeSet;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Set<String> all = **new** TreeSet<String>() ;  all.add("D") ;  all.add("A") ; // 内容重复了  all.add("B") ;  all.add("B") ;  all.add("C") ;  System.*out*.println(all);  }  } |

现在发现所有保存的数据没有重复且有序排列。

**3.4.3、关于TreeSet排序的说明（重点）**

通过之前的程序可以发现，使用TreeSet实例化Set接口之中，所有保存的数据都是有序的，那么在这种情况下，那么如果说使用的是一个自定义的类呢？

那么这个时候如果这个类对象要进行排序的话，则这个类必须实现Comparable接口，设置比较规则。但是在这种情况下有一点必须注意：一旦使用了Comparable之后，类之中的所有属性都必须写进排序规则。

**范例：**自定义类排序

|  |
| --- |
| **package** cn.mldn.demo;  **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 ;  }  @Override  **public** String toString() {  **return** "姓名：" + **this**.name + "，年龄：" + **this**.age + "\n" ;  }  @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** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Set<Person> all = **new** TreeSet<Person>() ;  all.add(**new** Person("张三",20)) ;  all.add(**new** Person("张三",20)) ; // 全部重复  all.add(**new** Person("李四",20)) ; // 年龄重复  all.add(**new** Person("王五",19)) ;  all.add(**new** Person("赵六",21)) ;  System.*out*.println(all);  }} |

TreeSet子类依靠Comparable中compareTo()方法的返回值是否为0来判断是否为重复元素。

**3.4.4、关于重复元素的说明**

那么TreeSet依靠Comparable进行重复元素判断，那么HashSet可以吗？发现以上的程序换为了HashSet之后，该有的重复还是有，因为从真正的意义上来讲，判断重复元素依靠的不是Comparable（只有排序的时候才依靠Comparable），所有的重复元素的判断依靠于Object类的两个方法：

· **hash码：**public int hashCode()；

· **对象比较：**public boolean equals(Object obj)。

在进行对象比较的过程之中，首先会先使用hashCode()与已保存在集合之中的对象的hashCode()进行比较，如果代码相同，则再使用equals()方法进行属性的依次判断，如果全部相同，则为相同元素。

那么为了保证每一个对象的hashCode()不一样，需要设计一组数学公式才可以，会吗？

**范例：**重复元素

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.HashSet;  **import** java.util.Set;  **class** Person {  **private** String name ;  **private** **int** age ;  **public** Person(String name,**int** age) {  **this**.name = name ;  **this**.age = age ;  }  @Override  **public** String toString() {  **return** "姓名：" + **this**.name + "，年龄：" + **this**.age + "\n" ;  }  @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + age;  result = prime \* result + ((name == **null**) ? 0 : name.hashCode());  **return** result;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj)  **return** **true**;  **if** (obj == **null**)  **return** **false**;  **if** (getClass() != obj.getClass())  **return** **false**;  Person other = (Person) obj;  **if** (age != other.age)  **return** **false**;  **if** (name == **null**) {  **if** (other.name != **null**)  **return** **false**;  } **else** **if** (!name.equals(other.name))  **return** **false**;  **return** **true**;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Set<Person> all = **new** HashSet<Person>() ;  all.add(**new** Person("张三",20)) ;  all.add(**new** Person("张三",20)) ; // 全部重复  all.add(**new** Person("李四",20)) ; // 年龄重复  all.add(**new** Person("王五",19)) ;  all.add(**new** Person("赵六",21)) ;  System.*out*.println(all);  }  } |

至此，Object类之中的全部方法就讲解完成了。

**3.5、集合的输出操作（重点）**

在之前所介绍的都属于单值集合的基本操作，可是对于集合有一个最为重要的问题就是如何进行集合内容的输出操作，而这个问题在Java的类集框架之中给出了四种输出方式：Iterator、ListIterator、Enumeration、foreach。

**3.5.1、迭代输出：Iterator（核心），95%**

Iterator是最为常用的集合输出接口，在这个接口中一共定义了三个方法，但只有两个有真正用处：

**· 判断是否有下一个元素：public boolean hasNext()；**

**· 取得下一个元素：public E next()。**

在Iterator接口之中存在了一个remove()方法，但是这个方法真没用。而且在之前学习的Scanner也是Iterator的子类。

但是如何取得Iterator接口的实例化对象呢？这一操作在Collection接口就已经明确定义了，因为Collection继承了一个Iterable接口，在这个接口下定义了一个方法：public Iterator<T> iterator()，取得Iterator接口的实例化对象，但是与之前在IO操作部分学习的一样，OutputStream实现了Closeable和Flushable两个接口，但是这两个接口属于新的接口，和这两个接口一样，Iterable接口也是在JDK 1.5的时候出现的，那么基本上也不会去关心这两个接口，因为Collection接口中也已经明确定义了iterator()方法。

**范例：**使用Iterator输出集合数据

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.ArrayList;  **import** java.util.Iterator;  **import** java.util.List;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  List<String> all = **new** ArrayList<String>();  all.add("Hello");  all.add("Hello"); // 内容重复了  all.add("World");  Iterator<String> iter = all.iterator();  **while** (iter.hasNext()) { // 判断是否有下一个元素  String str = iter.next() ;  System.*out*.print(str + "、");  }  }  } |

**以后只要是见到了集合的输出操作，永远都使用Iterator接口完成。**

**3.5.2、双向迭代输出：ListIterator（了解），0.09%**

Iterator可以完成的是由前向后的单向输出操作，如果现在希望可以完成由前向后，和由后向前输出的话，那么就可以利用ListIterator接口完成，此接口是Iterator的子接口，在ListIterator接口主要使用以下两个扩充方法：

· 判断是否有前一个元素：public boolean hasPrevious()；

· 取出前一个元素：public E previous()。

但是如果要想取得ListIterator接口的实例化对象，Collection没有这样的方法支持，这个方法在List接口之中存在：public ListIterator<E> listIterator()。

**范例：**执行双向迭代

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.ArrayList;  **import** java.util.List;  **import** java.util.ListIterator;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  List<String> all = **new** ArrayList<String>();  all.add("Hello");  all.add("Hello"); // 内容重复了  all.add("World");  ListIterator<String> iter = all.listIterator();  System.*out*.print("由前向后输出：");  **while** (iter.hasNext()) { // 判断是否有下一个元素  String str = iter.next() ;  System.*out*.print(str + "、");  }  System.*out*.print("\n由后向前输出：");  **while** (iter.hasPrevious()) {  String str = iter.previous() ;  System.*out*.print(str + "、");  }  }  } |

但是对于由后向前的输出操作，在进行之前一定要首先发生由前向后的输出。由于此输出接口只有List可以使用，所以在开发之中几乎不会出现。

**3.5.3、废弃的接口：Enumeration（重点），4.9%**

Enumeration是一个最早的输出接口，最早称为枚举输出，在JDK 1.0的时候就已经推出了，并且在JDK 1.5的时候将其功能进行了扩充，主要就是增加了泛型，在Enumeration接口里面只定义了两个方法：

· **判断是否有下一个元素：**public boolean hasMoreElements()；

· **取得当前元素：**public E nextElement()；

不过要想取得Enumeration的实例化对象，不能依靠Collection接口了，只能够依靠Vector类完成，在Vector子类之中定义了如下一个方法：public **Enumeration<E>** elements()。

**范例：**使用Enumeration进行输出

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Enumeration;  **import** java.util.Vector;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Vector<String> all = **new** Vector<String>();  all.add("Hello");  all.add("Hello"); // 内容重复了  all.add("World");  Enumeration<String> enu = all.elements();  **while** (enu.hasMoreElements()) {  String str = enu.nextElement();  System.*out*.print(str + "、");  }  }  } |

从开发而言，首先考虑绝对不是Enumeration，考虑的肯定是Iterator，只有在必须使用的时候才用它。

**3.5.4、JDK 1.5的支持：foreach（理解），0.01%**

对于foreach输出除了可以进行数组内容的输出之外，也可以针对于集合完成输出。

**范例：**使用foreach

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  List<String> all = **new** ArrayList<String>();  all.add("Hello");  all.add("Hello"); // 内容重复了  all.add("World");  **for** (String str : all) {  System.*out*.print(str + "、");  }  }  } |

使用foreach并不是一个被广泛认可的操作代码形式。

**3.6、偶对象保存：Map接口（重点）**

偶对象指的是一对对象，即：两个对象要同时保存。这两个对象是按照了“key =value”的形式进行定义的，即：可以通过key找到对应的value数据，就好象电话号码本一样，例如，电话号码本之中保存了如下的信息：

· key = 张三，value = 123456；

· key = 李四，value = 234567；

现在如果要想找到张三的电话，那么肯定根据张三的key，取得对应的value，，而如果现在要想找王五的电话，由于没有王五这个key，所以返回的结果就是null。

Map就是实现这样一种操作的数据结构，这个接口之中定义的主要操作方法如下。

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **方法名称** | **类型** | **描述** |
| **1** | **public V put(K key, V value)** | **普通** | **向集合之中保存数据** |
| **2** | **public V get(Object key)** | **普通** | **通过指定的key取得对应的value** |
| 3 | public Set<K> keySet() | 普通 | 将Map中的所有key以Set集合的方式返回 |
| **4** | **public Set<Map.Entry<K,V>> entrySet()** | **普通** | **将Map集合变为Set集合** |

在Map接口之中有两个常用的子类：HashMap、Hashtable。

**3.6.1、新的子类：HashMap，95%**

HashMap是Map接口之中使用最多的一个子类，这个子类的定义如下：

|  |
| --- |
| public class HashMap<K,V>  extends AbstractMap<K,V>  implements **Map<K,V>**, Cloneable, Serializable |

下面就直接通过HashMap演示Map接口中各个主要方法的作用。

**范例：**验证Map方法

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.HashMap;  **import** java.util.Map;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Map<Integer, String> map = **new** HashMap<Integer, String>();  map.put(3, "张三");  map.put(**null**, "无名氏");  map.put(3, "李四"); // key重复，value会被新内容覆盖  map.put(1, "王五");  map.put(0, "赵六");  System.*out*.println(map.get(3));  System.*out*.println(map.get(**null**));  }  } |

通过这一代码可以发现，Map和Collection在操作上的不同：

**· Collection接口设置完的内容目的是为了输出；**

**· Map接口设置完内容的目的是为了查找。**

**范例：**取得全部的key，全部的key通过Set集合返回

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.HashMap;  **import** java.util.Iterator;  **import** java.util.Map;  **import** java.util.Set;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Map<Integer, String> map = **new** HashMap<Integer, String>();  map.put(3, "张三");  map.put(**null**, "无名氏");  map.put(3, "李四"); // key重复，value会被新内容覆盖  map.put(1, "王五");  map.put(0, "赵六");  Set<Integer> set = map.keySet() ;// 取得全部的key  Iterator<Integer> iter = set.iterator() ;  **while** (iter.hasNext()) {  Integer key = iter.next() ;  System.*out*.println(key + " --> " + map.get(key));  }  }  } |

**3.6.2、旧的子类：Hashtable，5%**

Hashtable是在JDK 1.0的时候推出的一个数据结构类，在JDK 1.2之后，让Hashtable实现了一个Map接口，所以用户在使用的时候依然采用子类为接口实例化的方法进行，那么只要接口的方法不变，实际上不管使用那一个子类都一样。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Hashtable;  **import** java.util.Map;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Map<Integer, String> map = **new** Hashtable<Integer, String>();  map.put(3, "张三");  map.put(3, "李四"); // key重复，value会被新内容覆盖  map.put(1, "王五");  map.put(0, "赵六");  System.*out*.println(map.get(3));  System.*out*.println(map.get(10));  }  } |

这个时候在使用Hashtable子类的时候，里面的数据不能有null。

**面试题：**请解释HashMap和Hashtable的区别？

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **区别** | **HashMap** | **Hashtable** |
| 1 | 推出时间 | JDK 1.2 | JDK 1.0 |
| 2 | 性能 | 采用异步处理方式，性能更高 | 采用同步处理方式，性能相对较低 |
| 3 | 安全性 | 非线程安全 | 线程安全 |
| 4 | 设置null | 允许将key或value设置为null | 不允许出现null，否则出现空指向异常 |

**3.6.3、关于Map集合的输出问题（核心）**

对于集合操作，在之前就一直强调：只要是集合的输出都使用Iterator完成，但是对于现在的Map集合就麻烦了，因为Map接口之中并没有提供像Collection接口那样的iterator()方法，所以如何使用Iterator输出Map集合呢？

如果要想真正的去思考Map接口通过Iterator输出，那么首先需要来观察一个Map.Entry接口，此接口定义如下：

|  |
| --- |
| public **static** interface Map**.**Entry<K,V> |

很明显，这是一个在Map接口之中使用static定义的一个内部接口。而且通过Map接口的定义也可以发现此内部接口的存在。

而在Map.Entry这个内部接口之中还存在有以下的两个常用方法：

· 取得当前的key：public K getKey()；

· 取得当前的value：public V getValue()。

下面通过一个图形来对比一下Collection和Map接口保存的数据形式。

通过以上的对比可以发现，在Map集合和Collection集合之中保存的最大区别：Collection直接保存的是要操作对象，而Map集合是将保存的key和value变成了一个Map.Entry对象，通过这个对象包装了key和value后保存的，所以根据这一特征，就可以给出**Map使用Iterator输出的操作步骤：**

**· 使用Map接口中的entrySet()方法，将Map集合变为Set集合；**

**· 取得了Set接口实例之后就可以利用iterator()方法取得Iterator的实例化对象；**

**· 使用Iterator迭代找到每一个Map.Entry对象，并进行key和value的分离。**

**范例：**使用Iterator输出Map集合

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Hashtable;  **import** java.util.Iterator;  **import** java.util.Map;  **import** java.util.Set;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Map<Integer, String> map = **new** Hashtable<Integer, String>();  map.put(3, "张三");  map.put(3, "李四"); // key重复，value会被新内容覆盖  map.put(1, "王五");  map.put(0, "赵六");  Set<Map.Entry<Integer, String>> set = map.entrySet();  Iterator<Map.Entry<Integer, String>> iter = set.iterator();  **while** (iter.hasNext()) {  Map.Entry<Integer, String> me = iter.next();  System.*out*.println(me.getKey() + "，" + me.getValue());  }  }  } |

这种代码在日后的所有开发之中一定会出现，所以必须会。

**面试题：**现在在一个List集合之中保存了多个String对象，要求将这个List集合变为Set集合，而后再将这个Set集合之中的全部数据保存在Map集合的value里面，而Map集合的key使用UUID生成，最后将Map中的数据进行迭代输出。

**相关说明：**

1、 在Collection接口之中存在一个增加一组集合的方法：public boolean addAll(Collection<? extends E> c)；

2、 UUID是一种算法，在Java中有指定的类，这个类可以根据时间戳生成一个几乎不会重复的字符串；

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.ArrayList;  **import** java.util.HashMap;  **import** java.util.HashSet;  **import** java.util.Iterator;  **import** java.util.List;  **import** java.util.Map;  **import** java.util.Set;  **import** java.util.UUID;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  List<String> list = **new** ArrayList<String>();  list.add("Hello");  list.add("Hello");  list.add("World");  Set<String> set = **new** HashSet<String>();  set.addAll(list); // 将List集合加到Set之中  Map<UUID, String> map = **new** HashMap<UUID, String>();  Iterator<String> iter = set.iterator();  **while** (iter.hasNext()) {  map.put(UUID.*randomUUID*(), iter.next());  } // 数据保存到Map集合  Iterator<Map.Entry<UUID, String>> iterMap = map.entrySet().iterator();  **while** (iterMap.hasNext()) {  Map.Entry<UUID, String> me = iterMap.next();  System.*out*.println(me.getKey() + " --> " + me.getValue());  }  }  } |

就是把几个集合互相折腾了一番。

**3.6.4、关于Map中保存key的说明**

通过程序可以发现，之前的Map集合之中都是使用了系统类作为了Map的key，那么实际上用户也可以使用自定义的类作为key出现，可是如果要想作为key的类必须注意一点：因为key属于查找操作，所以要想找到符合的key，那么作为key所在的类就必须覆写Object类之中的两个方法：hashCode()、equals()。

**范例：**自定义类作为key

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.HashMap;  **import** java.util.Map;  **class** Person {  **private** String name;  **public** Person(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "姓名：" + **this**.name;  }  @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + ((name == **null**) ? 0 : name.hashCode());  **return** result;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj)  **return** **true**;  **if** (obj == **null**)  **return** **false**;  **if** (getClass() != obj.getClass())  **return** **false**;  Person other = (Person) obj;  **if** (name == **null**) {  **if** (other.name != **null**)  **return** **false**;  } **else** **if** (!name.equals(other.name))  **return** **false**;  **return** **true**;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Map<Person, String> map = **new** HashMap<>();  map.put(**new** Person("张三"), **new** String("zs"));  System.*out*.println(map.get(**new** Person("张三")));  }  } |

但是这种程序也只是作为学习之中的概念出现，而在实际的开发之中，永远都是String作为key，因为最方便。

**3.7、Stack类（理解）**

Stack是栈，栈是一种先进后出的数据结构，Stack类的定义如下：

|  |
| --- |
| public class Stack<E> extends Vector<E> |

可以发现Stack类属于Vector的子类，但是使用的时候却不使用Vector类定义的方法，而使用Stack类自己的方法：

· 入栈操作：public E push(E item)；

· 出栈操作：public E pop()；

**范例：**观察栈的基本操作

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Stack;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Stack<String> all = **new** Stack<>();  all.add("A");  all.add("B");  all.add("C");  System.*out*.println(all.pop());  System.*out*.println(all.pop());  System.*out*.println(all.pop());  System.*out*.println(all.pop()); // 没数据了，出现EmptyStackException  }  } |

对于栈这一概念在自己编写的代码之中使用不多，不过以后的学习都会出现栈的概念，例如：在Android开发之中，多个Activity之间的互相调用和返回就是利用了栈。

**3.8、Collections类（了解）**

Collections是专门提供的一个集合的工具类，并没有实现Collection接口，但是在这个类之中，有许多的操作方法，可以方便的进行集合的操作（根本没用）。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.List;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  List<String> all = **new** ArrayList<String>();  Collections.*addAll*(all, "A", "B", "C");  System.*out*.println(all);  Collections.*reverse*(all) ;  System.*out*.println(all);  }  } |

**面试题：**请解释Collection和Collections的区别？

· Collection是一个接口，用于定义集合操作的标准；

· Collections是一个工具类，可以操作任意的集合对象。

**3.9、属性操作类：Properties（理解）**

属性一般都是指的是针对于字符串数据，并且所有的字符串数据都会按照“key = value”的形式保存，属性操作类主要是针对于属性文件完成的。Properties类本身是Hashtable的子类：

|  |
| --- |
| public class Properties extends Hashtable<Object,Object> |

但是在使用方法上操作的并不是由Map接口定义的方法，使用Properties自己的方法：

· 设置属性：public Object setProperty(String key, String value)；

· 取得属性：public String getProperty(String key)，如果没有指定的key返回null；

· 取得属性：public String getProperty(String key, String defaultValue)，如果没有指定的key，返回默认值。

**范例：**观察属性的设置和取得

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Properties;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Properties pros = **new** Properties();  pros.setProperty("BJ", " BeiJing ");  pros.setProperty("SH", "上海");  System.*out*.println(pros.getProperty("BJ"));  System.*out*.println(pros.getProperty("TJ"));  System.*out*.println(pros.getProperty("TJ", "没有发现"));  }  } |

但是使用Properties类最方便的特点是可以直接将这些属性以OutputStream的方式或InputStream的方式输出或读取：

· 向输出流中输出属性：public void store(OutputStream out, String comments) throws IOException；

· 从输入流中读取属性：public void load(InputStream inStream) throws IOException。

**范例：**将属性保存到文件之中，一般保存属性文件的后缀都是“\*.properties”

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.util.Properties;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Properties pros = **new** Properties();  pros.setProperty("BJ", " BeiJing ");  pros.setProperty("SH", "上海");  pros.store(**new** FileOutputStream(**new** File("D:" + File.*separator*  + "area.properties")), "Area Info");  }  } |

**范例：**通过属性文件读取内容

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.util.Properties;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Properties pros = **new** Properties();  pros.load(**new** FileInputStream(**new** File("D:" + File.*separator*  + "area.properties")));  System.*out*.println(pros.getProperty("BJ"));  System.*out*.println(pros.getProperty("TJ"));  System.*out*.println(pros.getProperty("TJ", "没有发现"));  }  } |

在日后的开发之中，一些框架会帮助用户自动的编写读取属性的操作，所以用户以后最关心的只是修改属性文件的工作，这一点在日后的Struts、Spring中都会见到。

**4、总结**

1、 Collection负责输出、Map负责查找；

2、 集合的输出就使用Iterator完成。