# day14

## 泛型

### 泛型在集合中使用(熟练掌握)

1. 泛型 : 广泛的类型, 如果定义一个类, 类中方法功能的参数列表或者返回值类型无法确定, 那么可以使用泛型来表示不确定的数据类型;

举例 : 集合是典型的泛型使用

ArrayList<E>--->带有泛型的类型, <E>称为泛型

解释 : ArrayList是个容器, 容器中可以存储任意引用类型数据, 而具体是什么类型的引用不确定, 那么需要存储的引用类型数据就可以使用泛型表示

1. 泛型类使用:
2. 在类名之后使用<>, <>中添加一个大写字母, 这个大写字母就表示泛型
3. 类上泛型在创建类对象时, 确定泛型的具体类型

举例 : ArrayList<Integer> list = new ArrayList<>();

list集合中的泛型具体类型为Integer, 以后list集合中只能存储Integer类型数据

1. 泛型的好处:
2. 提高代码安全性 : 将运行时会发生的类型转换错误提前到代码编译环节
3. 集合添加了泛型之后, 可以避免掉向下转型的麻烦, 代码简洁很多

注意 : 以后使用任何集合, 包括迭代器, 都添加泛型

1. 泛型使用注意事项:
2. 前后泛型一致
3. 从JDK1.7版本开始, 推出泛型推断; 后面的泛型可以不写, 默认推断成与前面一致的泛型类型

代码

|  |
| --- |
| package com.ujiuye.fanxing;  import java.util.ArrayList;  public class Demo01\_集合中泛型 {  public static void main(String[] args) {  // 1. 没有泛型  ArrayList list = new ArrayList();  list.add(1);  list.add("abc");    /\*for(int index = 0; index < list.size(); index++) {  // 1) 问题 : 需要向下转型, 很麻烦  Object obj = list.get(index);  // 2) 问题 : 集合中存储了不同引用类型, 导致获取数据转型时, 报错  Integer i = (Integer)obj;  System.out.println(i);  }\*/    // 2. 有泛型  // 泛型在创建集合对象时, 确定出具体类型  // list1集合中只能存储Integer类型数据  ArrayList<Integer> list1 = new ArrayList<>();  list1.add(12);  list1.add(67);    for(int index1 = 0; index1 < list1.size();index1++) {  Integer i = list1.get(index1);  System.out.println(i);  }  }  } |

### 带有泛型的类(了解)

1. 定义一个类时, 在类名后面添加泛型类型

public class 类名<泛型类型1, 泛型类型2...>{}

1. 说明:
2. 类名后面的<E>, 称为菱形泛型, 菱形泛型中写入一个大写字母, 比较常见E, K, V, T...
3. 类上泛型在整个类中作为一个已知数据类型进行使用
4. 当创建出一个类对象时, 确定泛型的具体类型(泛型可以是任意的引用数据类型)

代码

|  |
| --- |
| package com.ujiuye.fanxing;  import java.util.ArrayList;  public class FanXingClass<T> {  int i = 10;  ArrayList<T> list = new ArrayList<>();    public void addEle(T t) {  list.add(t);  System.out.println(list);  }  } |

### 带有泛型的方法(了解)

1. 定义一个方法时, 也可以在方法上添加泛型

修饰符<泛型类型> 返回值类型 方法名(参数列表){

}

1. 方法上泛型使用:
2. 非静态方法, 可以在方法上自己定义泛型, 也可以使用类上泛型
3. 静态方法, 不可以使用类上的泛型, 只能在方法中自己定义泛型使用

因为类上的泛型类型在创建对象同时确定, 而静态优先于对象存在于内存中, 导致静态方法可以使用时, 泛型类型还没有确定, 因此不能使用

1. 方法上的泛型, 只能在整个方法中使用

代码

|  |
| --- |
| package com.ujiuye.fanxing;  import java.util.Arrays;  public class FanXingMethod {  // 功能 : 实现任意类型数组的 index1和index2索引位置元素的互相替换  public static<W> void changeArrayEle(W[] arr, int index1, int index2) {  W temp = arr[index1];  arr[index1] = arr[index2];  arr[index2] = temp;  System.out.println(Arrays.toString(arr));  }  } |

### 泛型通配符(了解)

1. ? : 就表示泛型通配符, ? 表示可以是任意类型的泛型

[containsAll](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/util/Collection.html" \l "containsAll(java.util.Collection))([Collection](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/util/Collection.html" \o "java.util 中的接口)<?> c) : 问方法调用集合,是否完全包含参数集合c中的所有元素; 不不包含返回false

1. ? extends [E](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/util/Collection.html" \o "Collection 中的类型参数) : 提供的泛型类型 ? , 需要是泛型E类型本身或者是E类型的子类类型
2. ? super E : 提供的泛型类型 ? , 需要是泛型E类型本身或者是E类型的父类类型

代码

|  |
| --- |
| package com.ujiuye.fanxing;  import java.util.ArrayList;  public class Demo02\_泛型通配符 {  public static void main(String[] args) {  ArrayList<Integer> list = new ArrayList<>();  list.add(12);  list.add(15);    ArrayList<String> list2 = new ArrayList<>();  list2.add("a");    // list2集合中存储的是什么类型数据都可以  // containsAll(Collection<?> c)  System.out.println(list.containsAll(list2));// fasle    // ? extends E : E表示方法调用集合的泛型类型, ?表示参数需要提供的泛型类型  // 要求 : 提供的类型必须是E类型本身或者是E类型的子类类型  // addAll(Collection<? extends E> c)  // list.addAll(list2);  }  } |

## Set集合

### Set集合概述

1. Set : 来自于java.util包, 是Collection的一个子接口
2. 特点:
3. 元素存取无序(存入集合的元素与从集合中取出元素的顺序不保证一致)
4. 没有索引
5. 不存储重复元素
6. Set接口不能实例化对象, 因此找到实现类HashSet, 哈希表实现方式

Set<String> set = new HashSet<>();// 接口多态

1. Set集合中没有特殊方法功能,使用Collection中的共有方法即可

代码

|  |
| --- |
| package com.ujiuye.set;  import java.util.HashSet;  import java.util.Set;  public class SetDefine {  public static void main(String[] args) {  Set<String> set = new HashSet<>();  set.add("hello");  set.add("a");  set.add("world");  set.add("a");  set.add("hello");    System.out.println(set);  }  } |

### Set集合的遍历方式

1. toArray() : 将Set集合中的元素放置到一个Object[]数组中; 集合转数组, 通过遍历数组中的元素相当于在遍历集合中的数据
2. toArray(T[] a) : 通过参数提供一个T[]数组, T表示泛型, 与要遍历的集合泛型类型一致; 将集合中的数据同步到参数T[]中, 返回值类型也是T[], 通过遍历T[]相当于遍历集合中元素内容; T[]数组在遍历过程中, 不需要向下转型

如果T[]数组长度大于等于集合长度, 返回提供的参数T[]

如果参数T[]长度小于集合长度, 方法自动提供一个新的T[]进行返回

1. 迭代器遍历 iterator()
2. 增强for : forEach

for(元素数据类型 变量名 : 需要遍历的数组或者集合){

// 变量名就表示集合或者数组中的每一个元素数据

}

注意:

1. 增强for底层是迭代器实现原理
2. 增强for在进行集合元素迭代时, 也会发生并发修改异常

代码

|  |
| --- |
| package com.ujiuye.set;  import java.util.HashSet;  import java.util.Iterator;  import java.util.Set;  public class Demo01\_Set集合遍历方法 {  public static void main(String[] args) {  Set<Integer> set = new HashSet<>();// 19 -8 12  set.add(12);  set.add(19);  set.add(12);  set.add(-8);  bianLiSet4(set);  }    // 1. toArray() : 集合转数组遍历  public static void bianLiSet1(Set<Integer> set) {  Object[] objArr = set.toArray();  for(int index = 0; index < objArr.length; index++) {  Integer i = (Integer)objArr[index];  System.out.println(i);  }  }    // 2. toArray(T[]) : 集合转数组  public static void bianLiSet2(Set<Integer> set) {  Integer[] arr = new Integer[set.size()];  set.toArray(arr);    for(int index = 0; index < arr.length; index++) {  Integer i = arr[index];  System.out.println(i+"--");  }  }    // 3. 迭代器遍历  public static void bianLiSet3(Set<Integer> set) {  Iterator<Integer> it = set.iterator();  while(it.hasNext()) {  Integer i = it.next();  System.out.println(i + "~~~");  }  }    // 4. forEach遍历  public static void bianLiSet4(Set<Integer> set) {  Set<Integer> set2 = new HashSet<>();  for(Integer i : set) {  System.out.println(i+">>");  if(i == -8) {  set2.add(888);  }  }  set.addAll(set2);  System.out.println(set);  }  } |

### HashSet保证元素唯一原理

1. HashSet中存储JDK提供的类型, 直接保证元素唯一 : Integer, String...(因为JDK提供的类型都重写过hashCode和equals方法)
2. HashSet存储自定义类型, 想要按照类型成员变量进行去重复, 失败

定义出一个Person类, 有两个属性 : 姓名, 年龄; 将Person类型对象存储在HashSet集合中, 要求 : 如果姓名和年龄都一致, 认为这是重复Person类型数据, 去掉重复数据

1. 分析为什么自定义类型没有根据成员变量进行去重复:
2. 因为调用了add方法功能, 向HashSet集合中添加自定义类型数据, 于是查看add方法怎么实现 : a 调用了hashCode() b: 调用了equals()方法
3. 结论 : 自定义类型通过alt + shift + s , 重写hashCode()和equals()两个方法之后,可以根据对象的成员变量进行去重复
4. hashCode和equals方法

两个方法都是顶层父类Object中方法功能, 被所有类继承使用, 也可以重写

1. equals() :

a : Object源代码中, 比较两个对象的地址值

b : 重写之后, 比较两个对象成员变量数值是否相等

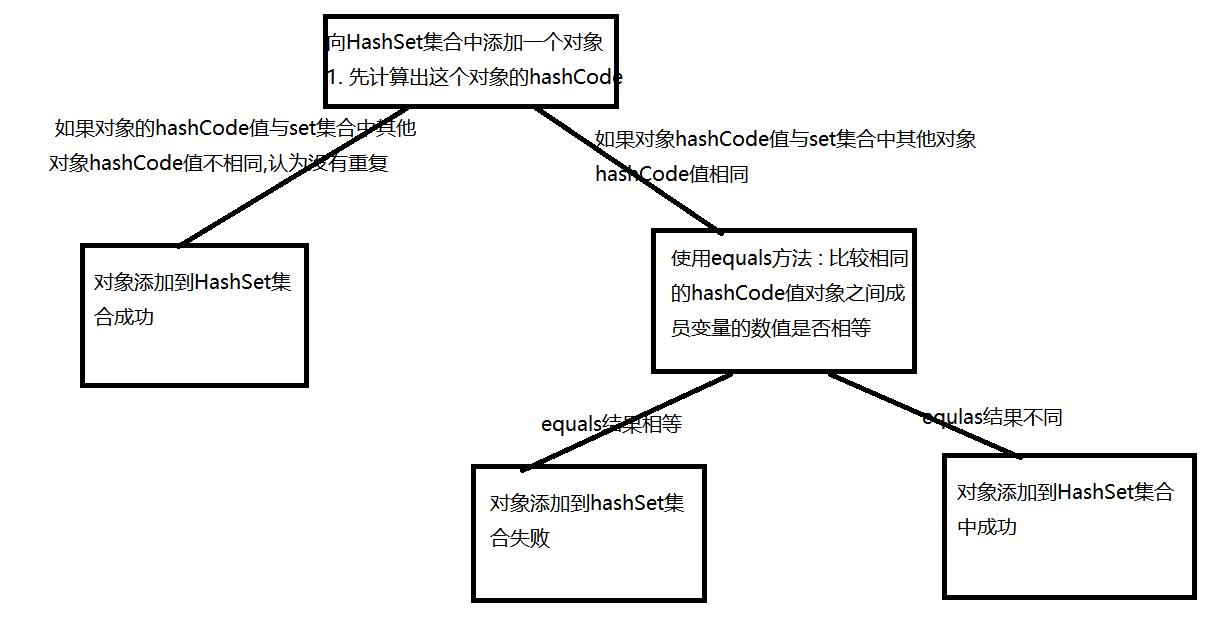
1. hashCode(): 通过将该对象的内存地址转换成一个整数来实现的

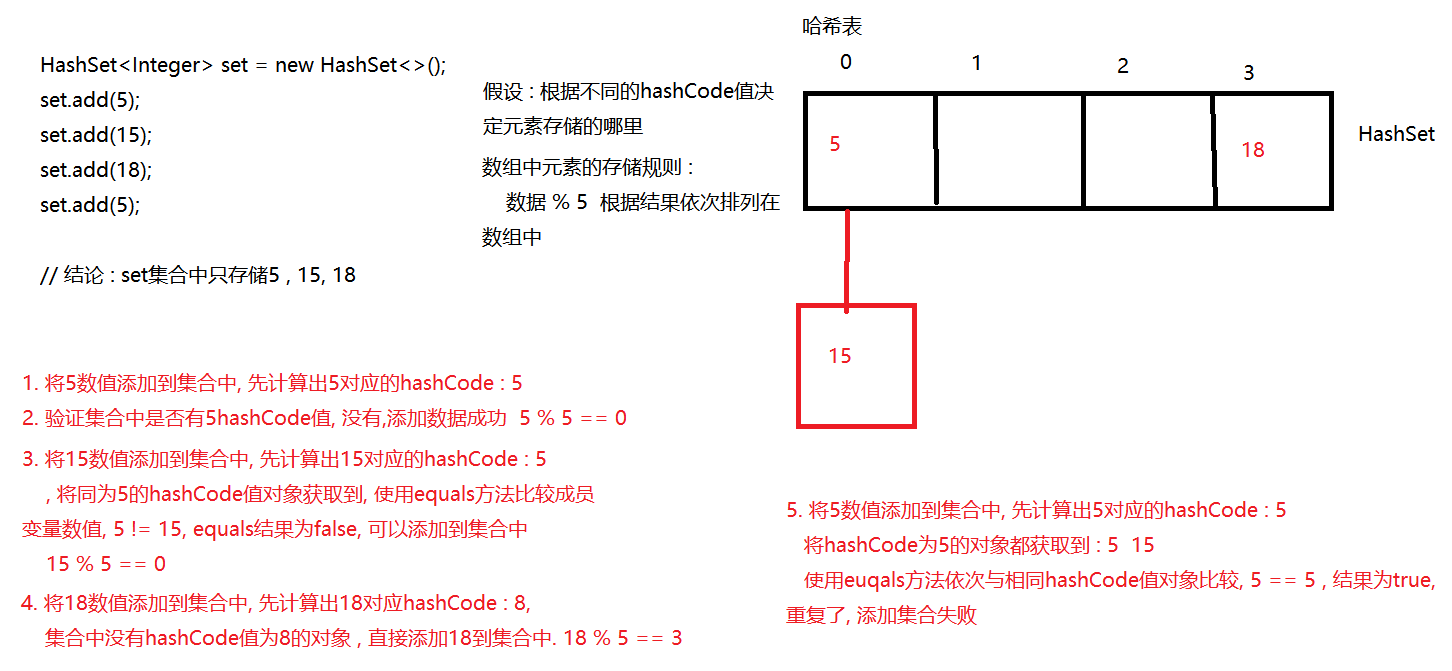
a : Object 类源代码中,定义的 hashCode 方法会针对不同的对象地址返回不同的整数

b : 重写之后, 得到的是对象成员变量数值转换成整数的结果

注意 :

1. 同一对象, 计算出相同的hashCode值
2. equals比较结果为true的对象, 计算出相同的hashCode值
3. equals比较结果为false的对象, 尽量计算出不同的hashCode值
4. HashSet保证元素唯一的原理





结论 : 自定义类型通过alt + shift + s , 重写hashCode()和equals()两个方法之后,可以根据对象的成员变量在HashSet集合中去重复存储

代码

|  |
| --- |
| package com.ujiuye.set;  public class Person {  private String name;  private int age;  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getAge() {  return age;  }  public void setAge(int age) {  this.age = age;  }  public Person(String name, int age) {  super();  this.name = name;  this.age = age;  }  public Person() {  super();  }  @Override  public String toString() {  return "Person [name=" + name + ", age=" + age + "]";  }    @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;  }  } |

|  |
| --- |
| package com.ujiuye.set;  import java.util.HashSet;  public class Demo02\_HashSet存储自定义类型 {  public static void main(String[] args) {  HashSet<Person> set = new HashSet<>();  set.add(new Person("张三",20));  set.add(new Person("李四",19));  set.add(new Person("张三",20));  set.add(new Person("王五",18));  // [Person [name=王五, age=18], Person [name=李四, age=19], Person [name=张三, age=20]]  System.out.println(set);    /\* Person p = new Person("张三", 20);  Person p1 = new Person("张三", 20);  int first = p.hashCode();  int second = p1.hashCode();  System.out.println(first);  System.out.println(second);\*/  }  } |

### LinkedHashSet

1. LinkedHashSet是HashSet的一个子类 : 所有功能包括保证元素唯一原理, 都是与HashSet一致, 唯一不同就在于, LinkedHashSet底层维护了一个双向链表结果, 能保证元素的存入和取出的顺序是一致的

代码

|  |
| --- |
| package com.ujiuye.set;  import java.util.HashSet;  import java.util.LinkedHashSet;  public class LinkedHashSetDemo {  public static void main(String[] args) {  HashSet<Integer> set = new HashSet<>();  set.add(88);  set.add(6);  set.add(16);  set.add(6);    System.out.println(set);//[16, 6, 88]    LinkedHashSet<Integer> set1 = new LinkedHashSet<>();  set1.add(88);  set1.add(6);  set1.add(16);  set1.add(6);    System.out.println(set1);// [88, 6, 16]  }  } |

## Map

### Map集合的概述

1. Map<K,V>集合 : 来自于java.uitl包, 双列集合的父接口
2. 对比理解 : map 表示地图含义, 地图上的每一个点都对应现实生活中的一个地理位置, 一对一的对应关系映射成Map集合
3. Map<K,V>: 称为键值对映射关系
4. --key, 表示键

V---value, 表示值

4. Map集合使用特点:

1) 整个Map集合中, Key值不重复(唯一), value值可以重复

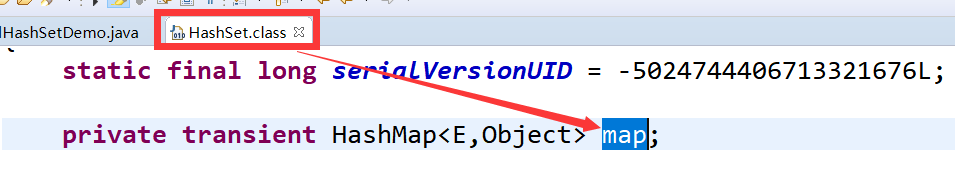
2) 一个key值对应一个value值

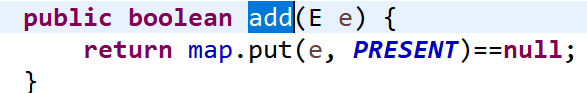
3) 通常都会通过唯一的key值操作其对应的value值

4) Map集合元素存取无序, 没有索引

Map接口, 不能实例化对象, 找到一个实现类 : HashMap<K,V> 哈希表结构

1. HashSet保证元素唯一的原理, 与HashMap中key的值保证元素唯一的原理一致, 因为HashSet中add方法, 就是使用了HashMap中的put方法实现的





### Map中的常用方法

1. put(K key, V value) : 返回值结果是V--->value

a : 如果添加的key值在Map集合中不存在, put方法表示新增; 添加键值对数据到Map集合中

b : 如果添加的key值在Map集合中存在, put方法表示修改, 表示修改value值

1. clear() : 清空Map集合中所有键值对元素
2. containsKey(Object key) : 验证Map集合键中是否包含指定参数key的值, 包含返回true.否则false
3. containsValue(Object value): 验证Map集合值中是否包含指定参数value的值包含返回true.否则false
4. get(Object key) : 通过Map集合中唯一的key值,获取到其对应的value值
5. isEmpty() : 验证Map集合中是否还有键值对元素, 如果没有返回true; 有返回false
6. remove(Object key) : 通过Map集合中唯一的key值, 定位到键值对, 删除键值对, 返回被删除的value值
7. size() : 获取到Map集合中的键值对数量, 返回int类型结果

代码

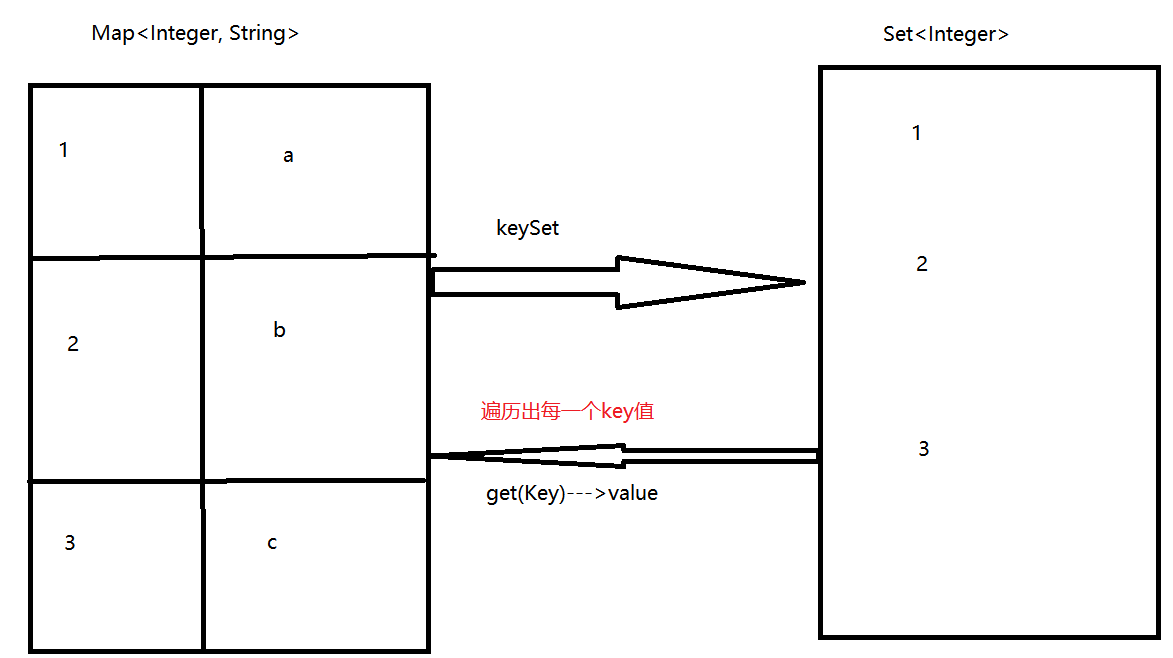
|  |
| --- |
| package com.ujiuye.map;  import java.util.HashMap;  import java.util.Map;  public class Demo01\_Map中方法 {  public static void main(String[] args) {  Map<Integer, String> map = new HashMap<>();  //1.如果添加的key值在Map集合中不存在, put方法表示新增; 添加键值对数据到Map集合中  map.put(2, "b");  map.put(1, "a");  map.put(3, "c");    System.out.println(map);//{1=a, 2=b, 3=c}    //2.如果添加的key值在Map集合中存在, put方法表示修改, 表示修改value值  map.put(2, "hello");  System.out.println(map);//{1=a, 2=hello, 3=c}      //3.containsKey(Object key) : 验证Map集合键中是否包含指定参数key的值, 包含返回true.否则false  System.out.println(map.containsKey(3)); // true    //4.containsValue(Object value): 验证Map集合值中是否包含指定参数value的值包含返回true.否则false  System.out.println(map.containsValue("b"));// false    //5.get(Object key) : 通过Map集合中唯一的key值,获取到其对应的value值  System.out.println(map.get(2));// hello    //6.remove(Object key) : 通过Map集合中唯一的key值, 定位到键值对, 删除键值对, 返回被删除的value值  String value = map.remove(3);  System.out.println(value);// c  System.out.println(map);// {1=a, 2=hello}    //7.size() : 获取到Map集合中的键值对数量, 返回int类型结果  System.out.println(map.size() + "----");// 2    //8.clear() : 清空Map集合中所有键值对元素  map.clear();  System.out.println(map);//{}    //9.isEmpty() : 验证Map集合中是否还有键值对元素, 如果没有返回true; 有返回false  System.out.println(map.isEmpty());// true  System.out.println(map.size());// 0  }  } |

### Map集合的第一种遍历

1. Map集合中有方法 :

keySet(): 表示将Map集合中的所有key值获取到, 放置到一个Set集合中

1. 获取到包含有所有key的Set集合之后, 遍历这个Set集合, 获取到每一个key值;
2. 通过get(key)--->value



代码

|  |
| --- |
| package com.ujiuye.map;  import java.util.HashMap;  import java.util.Map;  import java.util.Set;  public class Demo02\_Map第一种遍历KeySet {  public static void main(String[] args) {  Map<Integer, String> map = new HashMap<>();  map.put(2, "b");  map.put(1, "a");  map.put(3, "c");  // 1. 获取到Map集合中的所有Key值  Set<Integer> setKey = map.keySet();  // 2. 遍历setKey获取到每一个key  for(Integer key : setKey) {  // 3. 使用Map中的get(key)  String value = map.get(key);  System.out.println(key + "---" + value);  }  }  } |

### Map集合第二种遍历

1. Map集合中有方法功能:

[entrySet](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/util/Map.html" \l "entrySet())() : 将Map集合中的键值对关系,放置到一个Set集合中, Set集合中包含的是一对一对的元素; 每一对元素在Map集合中使用Entry类型进行表示, 返回值类型Set<Map.Entry<K,V>>

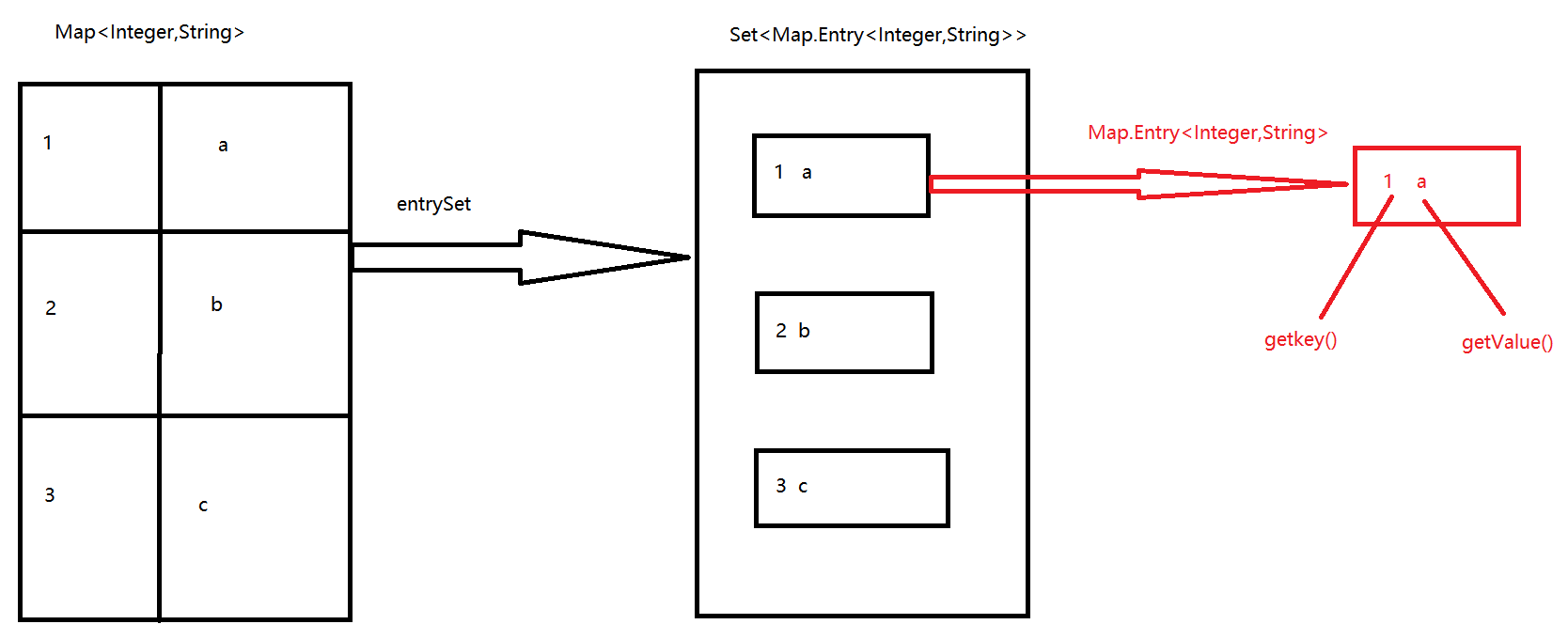
Map接口中有内部接口Entry <K,V>: 一个Entry <K,V>类型表示一对元素

1. 当获取到了很多对数据 Entry<K,V>

Entry中有方法 :

getKey() : 获取到一对数据中的key键值

getValue() : 获取到一对数据中的value值



代码

|  |
| --- |
| package com.ujiuye.map;  import java.util.HashMap;  import java.util.Map;  import java.util.Set;  public class Demo03\_Map第二种遍历EntrySet {  public static void main(String[] args) {  Map<Integer, String> map = new HashMap<>();  map.put(2, "b");  map.put(1, "a");  map.put(3, "c");  map.put(88, "world");    // 1. 获取到Map集合中的所有键值对数据  Set<Map.Entry<Integer, String>> set= map.entrySet();  // 2. 遍历set集合, 获取到每一对键值对关系  for(Map.Entry<Integer, String> entry : set) {  // 3. 获取到一对entry中的每一个key和value  Integer key = entry.getKey();  String value = entry.getValue();  System.out.println(key + "---" + value);  }  }  } |