**中软国际考题-笔试**

1. 单选题

1、表示键值对概念的接口是哪项？（D）

A. Set B. List C. Collection D. Map

2、下列描述错误的是（B）

A.Map是复合型数据容器B.Map按下标排列子元素

C.Map存储内容较为复杂D.Map也可以存储Map

3、下列描述正确的是（A）

1. Map允许存储null数据
2. Map与List无法互相转换
3. Map无法使用迭代器遍历。
4. Map由于其特点不属于集合框架之一

4、java中Map描述错误的是（D）

A.Map存储表格形式数据空间 B.Map采用键无序的方式存储值

C.Map可以与泛型同时声明产生 D.Map允许采用重复键存值

5、以下哪个不是Collection的子接口？D

A: List B: Set C: SortedSet D: Map

6、关于Map和List，下面说法正确的是( D )。

A．Map继承List B．List中可以保存Map或List

C．Map和List只能保存从数据库中取出的数据 D．Map的value可以是List或Map

7、以下能以键\_值对的方式存储对象的接口是（B）

A.java.util.Collection B.java.util.Map C.java.util.HashMap D.java.util.Set

8、在Java中，关于HashMap类的描述，以下错误的是( B )。

A.HashMap使用键/值得形式保存数据 B.HashMap 能够保证其中元素的顺序

C.HashMap允许将null用作键 D.HashMap允许将null用作值

9、以下说法正确的是( A )

A List和Set都是Collection接口的子接口

B HashSet和HashMap都实现自Map接口

C ArrayList LinkedList 都使用相同的方式存储数据

D TreeSet和TreeMap都实现自Map接口

10、选择有序的Map集合（**B**）

A. HashMap  **B. TreeMap** C. HashSet D. TreeSet

11、下列哪种创建Map集合的方式是正确的？（D）

A.Map m = newMap（）

B.Map m = new Map(init capacity,increment capacity)

C.Map m = new Map(new Colleciton())

D.Map是接口，所以不能实例化

12、在Java语言中，下面接口以键-值对的方式存储对象的是（B）

A.java.util.List

B. java.util.Map

C. java.util.Collection

D. java.util.Set

1. 多选题
2. 以下哪些是的Map集合？（A、B、C、D）

A、HashMap B. HashTable C. LinkedHashMap D、ConcurrentHashMap

2、下面哪些项描述不正确？（A、B）

A. Map存值方式比List更加先进 B. Map其值只允许存储基本数据类型

C. Map的泛型也是采用键值对规定 D. Map的size方法为获取集合长度

3、关于HashMap和Hashtable正确的说法有（AC）（多选）

a) 都实现了Map接口 b) Hashtable类不是同步的，而HashMap类是同步的

c) Hashtable 不允许null键或值 d) HashMap 不允许null键或值

4、对 Map 的用法，正确的有：（CD）

A: new java.util.Map().put("key" , "value") ; B: new java.util.SortedMap().put("key" , "value") ;

C: new java.util.HashMap().put( null , null ) ; D: new java.util.TreeMap().put( 0 , null ) ;

5、关于HashMap集合说法正确的是( B、D )

A.HashMap集合是单列集合 B.HashMap集合不允许存储重复键

C.HashMap集合不允许存储重复值 D.HashMap集合线程是不安全的

6、下面哪些不是Map的子类？（B、D）

A. HashMap B. HashSet C. Properties D. HashList

7、Java中以下哪些类的对象以 键-值的方式存储对象？（A、B）

A java.util.TreeMap B java.util.HashMap

C java.util.HashSet D java.util.List

8、选择出常用的Map集合（A、B、E、F）

A. HashMap B. LinkedHashMap C. LinkedList

D. ArrayList E. HashTable

F. ConcurrentHashMap

9、以下关于HashMap与Hashtable的说法中，正确的是（A,C,D）。

A. 迭代HashMap采用快速失败机制，而Hashtable不是；

B. Hashtable允许null值作为key和value，而HashMap不可以；

C. HashMap不是同步的，而Hashtable是同步的；

D. 两者都是用key-value方式获取数据

10. 以下关于Map的用法中，正确的有（C,D）

A. new.java.util.SortedMap().put(“key1”,”value1”);

B. new.java.util.Map().put(“key1”,”value1”);

C. new.java.util.HashMap().put(null , null);

D. new.java.util.TreeMap().put(0 , null);

1. 填空题（每空1分，共10分）
2. Hashtable是\_\_\_\_\_\_\_\_\_\_的子类，HashMap是\_\_\_\_\_\_\_\_\_一个实现类。 (Dictionary、AbstractMap)

2、Map分为两种接口类：\_\_\_\_\_\_\_\_\_、 \_\_\_\_\_\_\_\_\_ 。(HashMap，Hashtable)

3、你都知道哪些常用的Map集合?\_\_\_\_\_\_\_\_\_\_ 、\_\_\_\_\_\_\_\_\_\_ 、\_\_\_\_\_\_\_\_\_\_ 、\_\_\_\_\_\_\_\_\_\_ 等等。

（HashMap、HashTable、LinkedHashMap、ConcurrentHashMap）

4、java集合中一种以键-值的方式存储数据的接口是\_\_\_\_\_\_\_\_\_\_，实现此接口最常用的类是\_\_\_\_\_\_\_\_\_\_ 。(Map、HashMap)

5、HashTable \_\_\_\_\_\_\_\_（允许|不允许) null作为key,

6、HashMap基于hashing原理，我们通过\_\_\_\_\_和\_\_\_\_\_方法储存和获取对象（put(),get()）

7、写出四个Map集合中的方法：\_\_\_\_\_\_\_\_\_\_ 、\_\_\_\_\_\_\_\_\_、\_\_\_\_\_\_\_\_\_\_ 、\_\_\_\_\_\_\_\_\_。(put()、putAll()、clear()、remove)

8、常用的Map集合有 、 、 和 （HashMap、HashTable、LinkedHashMap、ConcurrentHashMap）。

1. 判断题
2. Map没有继承Collection接口，是用于保存具有映射关系的数据（key-vlaue）（ T ）
3. HashMap的链表结构设计是用来解决key的hash冲突问题的。（T）
4. **Map的键值对存储方式本身为纯无序。（ Y ）**
5. **使用HashMap要注意避免集合的扩容，它会很耗性能，根据元素的数量给它一个初始大小的值。（ T ）**
6. **HashMap不是线程安全的。（ T ）**
7. **性能最好的是ConcurrentHashMap。（ T ）**
8. HashMap是线程安全的。（ F ）
9. HashMap的键值都可以为NULL，HashTable不行。（ T ）
10. Map通过remove() 无参方法移除指定的对象。（F）
11. Map通过put()方法添加元素。（T）
12. Map通过isEmpty()方法清空所有的元素。（F）
13. HashMap存放的是有序的元素。（F）
14. HashMap可以通过for循环遍历输入所有的键值内容。（F）
15. HashTable可以通过get(Object key) 获取值得内容。（T）
16. TreeMap 是一个**有序的key-value集合**，它是通过红黑树实现的（正确）
17. HashMap不继承于Collection接口所以无法使用Iterator迭代器 迭代(*错误)*
18. Collection集合接口是Map集合的父接口。（ **F**）
19. HashMap不允许null键或值。（ F ）。
20. Hashtable是线程安全的，而HashMap是线程不安全的。（T）

五、简答题

1、Hashtable和HashMap的区别

答案：Hashtable和HashMap类有三个重要的不同之处。第一个不同主要是历史原因。Hashtable是基于陈旧的Dictionary类的，HashMap是Java 1.2引进的Map接口的一个实现。

　　也许最重要的不同是Hashtable的方法是同步的，而HashMap的方法不是。这就意味着，虽然你可以不用采取任何特殊的行为就可以在一个多线程的应用程序中用一个Hashtable，但你必须同样地为一个HashMap提供外同步。一个方便的方法就是利用Collections类的静态的synchronizedMap()方法，它创建一个线程安全的Map对象，并把它作为一个封装的对象来返回。这个对象的方法可以让你同步访问潜在的HashMap。这么做的结果就是当你不需要同步时，你不能切断Hashtable中的同步（比如在一个单线程的应用程序中），而且同步增加了很多处理费用。

　　第三点不同是，只有HashMap可以让你将空值作为一个表的条目的key或value。HashMap中只有一条记录可以是一个空的key，但任意数量的条目可以是空的value。这就是说，如果在表中没有发现搜索键，或者如果发现了搜索键，但它是一个空的值，那么get()将返回null。如果有必要，用containKey()方法来区别这两种情况。

2、List、Map、Set三个接口，存取元素时，各有什么特点

答案：List 以特定次序来持有元素，可有重复元素。Set 无法拥有重复元素,内部排序。Map 保存key-value值，value可多值。

3、WeakHashMap的特点是什么？

答案：WeakHashMap 继承于AbstractMap，实现了Map接口。

和HashMap一样，WeakHashMap 也是一个散列表，它存储的内容也是键值对(key-value)映射，而且键和值都可以是null。

不过WeakHashMap的键是“弱键”。在 WeakHashMap 中，当某个键不再正常使用时，会被从WeakHashMap中被自动移除。更精确地说，对于一个给定的键，其映射的存在并不阻止垃圾回收器对该键的丢弃，这就使该键成为可终止的，被终止，然后被回收。某个键被终止时，它对应的键值对也就从映射中有效地移除了。

这个“弱键”的原理通过WeakReference和ReferenceQueue实现。 WeakHashMap的key是“弱键”，即是WeakReference类型的；ReferenceQueue是一个队列，它会保存被GC回收的“弱键”。实现步骤是：

(01) 新建WeakHashMap，将“键值对”添加到WeakHashMap中。

实际上，WeakHashMap是通过数组table保存Entry(键值对)；每一个Entry实际上是一个单向链表，即Entry是键值对链表。

(02) 当某“弱键”不再被其它对象引用，并被GC回收时。在GC回收该“弱键”时，这个“弱键”也同时会被添加到ReferenceQueue(queue)队列中。

(03) 当下一次需要操作WeakHashMap时，会先同步table和queue。table中保存了全部的键值对，而queue中保存被GC回收的键值对；同步它们，就是删除table中被GC回收的键值对。

4、HashMap中的get操作是什么原理？

答案：链表结构用于解决key的hash冲突，如果key equals一致将会覆盖值，不一致就会将值存储在key对应的链表中。先根据key的hashcode值找到对应的链表，再循环链表，根据key的hash是否相同且key的==或者equals比较操作找到对应的值。

5、请简述Map的主要特点（5分）

答案：1.以键值对形式存储数据

1. 不允许出现重复键
2. 元素存储顺序无序

6、使用HashMap有什么性能问题吗？（5分）

答案：

使用HashMap要注意避免集合的扩容，它会很耗性能，根据元素的数量给它一个初始大小的值。

7、 HashMap的数据结构是怎样的？默认大小是多少？内部是怎么扩容的？（15分）

答案：

HashMap是数组和链表组成的，默认大小为16，当hashmap中的元素个数超过数组大小\*loadFactor（默认值为0.75）时就会把数组的大小扩展为原来的两倍大小，然后重新计算每个元素在数组中的位置。

8、你都知道哪些常用的Map集合?  
HashMap、HashTable、LinkedHashMap、ConcurrentHashMap。

9、TreeMap与HashMap的区别和共同点？

（1）实现

TreeMap：SortMap接口，基于红黑树

HashMap：基于哈希散列表实现

（2）存储

TreeMap：默认按键的升序排序

HashMap：随机存储

（3）遍历

TreeMap：Iterator遍历是排序的

HashMap：Iterator遍历是随机的

（4）性能损耗

TreeMap：插入、删除

HashMap：基本无

（5）键值对

TreeMap：键、值都不能为null

HashMap：只允许键、值均为null

（6）安全

TreeMap：非并发安全Map

HashMap：非并发安全Map

（7）效率

TreeMap：低

HashMap：高

一般情况下我们选用HashMap，因为HashMap的键值对在取出时是随机的，其依据键的hashCode和键的equals方法存取数据，具有很快的访问速度，所以在Map中插入、删除及索引元素时其是效率最高的实现。而TreeMap的键值对在取出时是排过序的，所以效率会低点。

10、HashMap是线程安全的吗？线程安全的Map都有哪些？性能最好的是哪个？（5分）

答案：HashMap不是线程安全的。线程安全的有HashTable、ConcurrentHashMap、SynchronizedMap，性能最好的是ConcurrentHashMap。

11、 HashMap中的get操作是什么原理？

答案：先根据key的hashcode值找到对应的链表，再循环链表，根据key的hash是否相同且key的==或者equals比较操作找到对应的值。

12、 如果两个键的hashcode相同，你如何获取值对象？

答案：当我们调用get()方法，HashMap会使用键对象的hashcode找到bucket位置，找到bucket位置之后，会调用keys.equals()方法去找到链表中正确的节点，最终找到要找的值对象。

13、 HashMap什么时候扩容？机制是什么？

答案：在put时，容量不够用的时候。因为每个元素都是一个单向链表，所以map里放的实际数量总是大于等于申请的空间。那么容量不够用就是实际存放的数量等于申请的空间大小的时候。

14 、HashMap、 HashTable与TreeMap区别

答案：Hashtable可以看做是线程安全版的HashMap，两者几乎“等价”（当然还是有很多不同）。Hashtable几乎在每个方法上都加上synchronized（同步锁），实现线程安全。

15、如何决定选用HashMap还是TreeMap

答案：对于在Map中插入，删除和定位元素这类操作，HashMap是最好的选择，然而假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的Collection的大小 也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的

16, HashMap如何工作

答案:HashMap以哈希算法方式工作，在put和get方法被调用时，使用hashCode（）和equals（）来配合：当我们使用put方法，HashMap使用key的hashCode（）散列在键 - 值存储对中找出索引。k-v条目Entry存储在LinkedList，因此如果有已经存在的Entry，它就使用equals（）方法来检查其相应的键key是否已经存在，如果是的，它覆盖原值，否则它创建一个新Enrty条目和存储这个键-值；当我们通过键key调用get方法，再次使用的hashCode（）找到数组中的索引，然后使用equals（）方法来找到正确的条目，并返回它的值。

HashMap的容量涉及加载因子，阈值调整大小。 HashMap的初始默认容量为32和负载系数为0.75。阈值是容量乘以负载因子，每当我们尝试添加一个条目，Map的大小如果大于阈值时，HashMap扩大Map内容到一个新的更大阵列容量。容量总是2的幂，所以，如果你知道你需要存储大量的键 - 值对，例如，在数据库中的数据缓存，初始化HashMap中正确的容量和负载因子是个好主意。

17、你用过HashMap吗？ 什么是HashMap？ 你为什么用到它？

HashMap实现了Map接口，Map接口对键值对进行映射。Map中不允许重复的键。Map接口有两个基本的实现，HashMap和TreeMap。TreeMap保存了对象的排列次序，而HashMap则不能。HashMap存储的是键值对，允许键和值为null。HashMap是非synchronized的，但collection框架提供方法能保证HashMap synchronized，这样多个线程同时访问HashMap时，能保证只有一个线程更改Map。

18、请简述HashMap 与 HashtTable 的区别（5分）

① HashMap允许key和value为null， 而HashTable不允许

② HashTable是线程同步的，而HashMap不是。所以HashMap适合单线程环境，HashTable适合多线程的环境。

③ 在java1.4中引入了LinkedHashMap，HashMap的一个子类，如果想要遍历顺序，和容易从HashMap转向LinkedHashMap。但HashTable不是这样的，HashTable的顺序是不可预知的。

④ HashMap提供对key的Set进行遍历，因此它支持fail-fast,但HashTable提供对key的Enumeration进行遍历，它不支持fail-fast。

19、如何决定选用HashMap还是TreeMap？

对于在Map中插入、删除和定位元素之类操作，HashMap是最好的选择，然而，如果需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于collection的大小，也许向HashMap中添加元素更快，将map换为TreeMap进行有序key遍历。

20、如何实现HashMap的同步？

答案：HashMap可以通过 Map m = Collections.synchromizedMap(new HashMap())来表达同步的效果。具体而言，该方法返回一个同步的Map，该Map封装了底层的HashMap的所有方法，使得底层的HashMap即使是在多线程的环境中也是安全的。

21. HashMap和Hashtable的区别（最少写出三条）？

答案：Java为数据结构中的映射定义了一个Java.util>Map，它有三个实现类：HashMap、Hashtable和TreeMap。Map用来存储键-值对的数据结构，在数组中通过数组下表来对其内容进行索引，而Map中，则是通过对象来进行索引，用来索引的对象叫做key，其对应的对象叫做value。

HashMap是一个对常见的Map，它根据键的HashCode值存储数据，根据键可以直接获取它的值，具有很快的访问速度。由于HashMap与Hashtable都采用hash方法进行索引，因此，二者具有许多相似之处，他们主要有如下的一些区别：

1)HashMap 没有排序，允许一个 null 键和多个 null 值 , 而Hashtable 不允许；

2)HashMap 把 Hashtable 的 contains 方 法 去 掉 了 ， 改 成containsvalue 和 containsKey, 因为 contains 方法容易让人引起误解；

3)Hashtable 继承自 Dictionary 类，HashMap 是 Java1.2 引进的Map 接口的实现；

4)Hash值得使用不用，Hashtable直接使用对象的hashCode。

5)Hashtable 的方法是 Synchronized ，而 HashMap 不是，在多个线程访问 Hashtable 时，不需要自己为它的方法实现同步， 而HashMap 就必须为之提供外同步。 Hashtable 和 HashMap 采用 的hash/rehash 算法大致一样，所以性能不会有很大的差异

六、程序实现题

1、字符统计：定义一个字符串，统计字符串中每个字符的个数。如“a,d,b,d,a”

使用HashMap统计每个字符的个数，其中key保存字符，value用来记录字符的数量

（10分）

|  |
| --- |
| public class Test {    public static void main(String[] args) {  System.out.println("请输入随机字符串");  Scanner it=new Scanner(System.in);  String it0=it.next();  Map<String,Integer> map=new HashMap<String,Integer>();  for(int i=0;i<it0.length();i++){  // System.out.println(it0.substring(i, i+1));  if(map.get(it0.substring(i, i+1))==null){  map.put(it0.substring(i, i+1), 1);  }else{  map.put(it0.substring(i, i+1), map.get(it0.substring(i, i+1))+1);  }  }  System.out.println("请查看指定字符串重复次数");  String it1=it.next();  System.out.println(map.get(it1)+"次");    }  } |

# 2、试写出学员类，实现使用Map接收该学员，从而实现HashMap的排序。（10分）

|  |
| --- |
| public class Student {  private String name;  private int age;  public Student(String name, int age) {  this.name = name;  this.age = 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;  }  @Override  public String toString() {  return "Student{" +  "name='" + name + '\'' +  ", age=" + age +  '}';  }  } |

3、选择某种Map集合保存学号从1到15的学员的学号（键）和姓名（值），学号用字符串表示，输入的时候要以学号乱序的方式存入Map集合，然后按照学号从大到小的顺序将Map集合中的元素输出打印。需要自定义Map集合的比较器Comparator，因字符串对象的大小比较是按字典序，而非对应的数值。

要求：必须使用Map集合的内部排序机制进行排序，不能在外部排序。

（10分）

|  |
| --- |
| import java.util.\*;  public class Test2 {  public static void main(String[] args) {  TreeMap<String,String> map=new TreeMap<String,String>();  map.put("05","张五");  map.put("03","张三");  map.put("12","张十二");  map.put("04","张四");  map.put("06","张六");  map.put("01","张一");  map.put("11","张十一");  map.put("08","张八");  map.put("15","张十五");  map.put("09","张九");  map.put("02","张二");  map.put("10","张十");  map.put("14","张十四");  map.put("13","张十三");  map.put("07","张七");  Set keySet=map.keySet();  Iterator it=keySet.iterator();  while(it.hasNext()) {  Object key=it.next();  Object value=map.get(key);  System.out.println(key+" "+value);  }  }  class MapKeyComparator implements Comparator{  public int compare(Object o1, Object o2) {  String a=(String)o1;  String b=(String)o2;  return b.compareTo(a);  }  }  } |

4、已知某学校的教学课程内容安排如下，完成下列要求：

（10分）

|  |
| --- |
| ① 使用一个Map，以老师的名字作为键，以老师教授的课程名作为值，表示下述（图片）课程安排。（4分）  ② 将教师Lucy，教授课程更改为CoreJava。（2分）  ③ 遍历Map，输出所有的老师及老师教授的课程。（2分）  ④ 利用Map，输出所有教授课程为JSP的老师。（3分） |

5、现有一段文字，请算出这段文字每个字及标点符号出现的次数，并保存到HashMap集合中

并显示打印(显示时按文字多少 从高到底显示)

（10分）

|  |
| --- |
| public class Test {  public static void main(String[] args) {  String message=”一段文字…..”;  Map<String,Integer> map=new HashMap<String,Integer>();  For(int i=0;i<message.length;i++){  String c=message.charAt(i)+””;  Integer count=map.get(c)  Map.put(c,count==null?1:count+1)  }  ArrayList<Entry<Character,Integer>>list =new ArrayLIst<Entry<Character,Integer>>map.entrySet  Collections.sort(list,new Comparator<Entry<String,Integer>>(){  Public int compare(Entyr<String,Integer>01,Entry<String,Integer o2>){  Return o2.getvalue()-o1.getValue();  })  for(Entry<String,Integer>entyr:list){  System.out.println(entry.getkey+”:”+entry.getValue())  }  })  } |

6、对于一个字符串，请设计一个高效算法，找到第一次重复出现的字符。   
给定一个字符串(不一定全为字母)A及它的长度n。请返回第一个重复出现的字符。保证字符串中有重复字符，字符串的长度小于等于500。用HashMap存储字符串（10分）

|  |
| --- |
| **package** a;  **import** java.util.ArrayList;  **import** java.util.LinkedHashMap;  **import** java.util.List;  **import** java.util.Map;  **import** java.util.Map.Entry;  /\*\*利用了哈希存储，将给定字符串以键值对形式存储在哈希Map中，key就是每一个字符，value就是每个字符出现的次数。存好后再按顺序遍历Map，找出第一次出现次数不为1的即可。返回。  注意：基于hash的存储是很高效的一种方式，存储键值对方便查找，可以根据key查找value,或者根据value查找key，或者查找特殊的字符等。\*/  **public** **class** ab {  **private** **static** **void** findFirstRepeat(String A, **int** n) {  //新插入的key-value给value赋初始值都为1  **int** sum = 1;  //用LinkedHashMap来存储键值对  //HashMap插入和输出顺序不一样  //但LinkedHashMap插入和输出顺序相同，即先存的先显示，也就是尾插  //HashSet要求不能存重复元素，但HashMap和LinkedHashMap可以存重复元素。  Map<String, Integer> hashMap = **new** LinkedHashMap  <String, Integer>();  **for** (**int** i = 0; i < n; i++) {  //将main中传过来的String类型参数一个字符一个字符的转化为char类型，方便处理。  **char** item = A.charAt(i);  //再将每一个char类型的字符转化为String类型。这一步操作和上一步操作作用是分离出每一个字符。  String item2 = String.*valueOf*(item);  String key = item2;  // 判断是否包含指定的键值  **boolean** contains = hashMap.containsKey(key);  **if** (contains) { // 如果条件为真  sum++;  hashMap.put(item2, sum);// 输出信息  } **else** {  sum = 1;  hashMap.put(item2, sum);  }  }  //输出存进哈希表中的所有键值对  System.***out***.println(hashMap);  //以下是通过给定的value值反查key的过程  //将所有的键值对存在一个entrySet中  //遍历entrySet,若找到value等于给定的value值，则将其对应key值存在keyList中。  List<String> keyList = **new** ArrayList<String>();  **for** (Entry<String, Integer> e : hashMap.entrySet()) {  **if** (e.getValue().equals(2)) {  keyList.add(e.getKey());  }  }  //最后遍历keyList，输出第一次出现的key即可。  **for** (String i : keyList) {  System.***out***.println(i);  **break**;  }  }  **public** **static** **void** main(String[] args) {  String A = "qywyer2333333333tdd";  **int** n = A.length();  *findFirstRepeat*(A, n);  }  } |

7、获取字符串中每一个字母出现的次数。比如输入字符串"aababcabcdabcde",结果为：a(5)b(4)c(3)d(2)e(1) 提示：使用TreeMap，将输入的字符串的每个字符作为键到集合中去找值，看返回值。

（10分）

|  |
| --- |
| public class TreeMapDemo {  public static void main(String[] args) {  // 定义一个字符串(可以改进为键盘录入)  Scanner sc = new Scanner(System.in);  System.out.println("请输入一个字符串：");  String line = sc.nextLine();  // 定义一个TreeMap集合  TreeMap<Character, Integer> tm = new TreeMap<Character, Integer>();  //把字符串转换为字符数组  char[] chs = line.toCharArray();  //遍历字符数组，得到每一个字符  for (char ch : chs) {  //拿刚才得到的字符作为键到集合中去找值，看返回值  Integer i = tm.get(ch);  //是null:说明该键不存在，就把该字符作为键，1作为值存储  if (i == null) {  tm.put(ch, 1);  } else {  i++;  tm.put(ch, i);  }  }  //定义字符串缓冲区变量  StringBuffer sb = new StringBuffer();  //遍历集合，得到键和值，进行按照要求拼接  Set<Character> set = tm.keySet();  for (Character key : set) {  Integer value = tm.get(key);  sb.append(key).append("(").append(value).append(")");  }  //把字符串缓冲区转换为字符串输出  String result = sb.toString();  System.out.println("result:" + result);  }  } |

8、自行实现一个简单HashMap

|  |
| --- |
| public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>,Serializable,Cloneable {  private static final long serialVersionUID = 4340114131866878570L;  private LinkedList<HashMapEntry<K,V>>[] table;  private static final int initCapacity = 16;  private static final double loadFactor = 0.75;  private transient int modCount = 0;  @SuppressWarnings("unchecked")  public HashMap(){  this.table = new LinkedList[initCapacity];  }  @SuppressWarnings("unchecked")  public HashMap(int init){  int capacity = 1;  while (capacity<init){  capacity <<= 1;  }  this.table = new LinkedList[capacity];  }  public HashMap(Map<? extends K,? extends V> map){  this(map.size());  this.putAll(map);  }  private int hash(Object o){  return hash(o,table.length);  }  private int hash(Object o,int length){  // Math.abs(o==null?0:o.hashCode()) % length  int h = (o==null?0:o.hashCode());  h ^= (h >>> 20) ^ (h >>> 12);  h = h ^ (h >>> 7) ^ (h >>> 4);  return h &(length-1);  }  private void checkIfNeedResize(LinkedList<HashMapEntry<K,V>> currentList){  //检查是否需要扩容，如果是，则扩容为2\*table.length  if(currentList.size() >= loadFactor \* table.length){  resize(2\*table.length);  }  }  @SuppressWarnings("unchecked")  private void resize(int newLength){  modCount++;  //hashmap扩容，newLength一般是原table的2倍。  LinkedList<HashMapEntry<K,V>>[] newTable = new LinkedList[newLength];  //遍历原table中每个元素，重新计算hash值，放到newTable中.  for (LinkedList<HashMapEntry<K, V>> list : table) {  if (list != null) {  for (HashMapEntry<K, V> entry : list) {  int hash = hash(entry.key, newLength);  LinkedList<HashMapEntry<K, V>> newList = newTable[hash];  if (newList == null) {  newTable[hash] = newList = new LinkedList<>();  }  newList.add(entry);  }  }  }  this.table = newTable;  }  @Override  public Set<Entry<K, V>> entrySet() {  final class EntryItr implements Iterator<Entry<K, V>> {  private int expectedMod = modCount;  private Iterator<HashMapEntry<K, V>> iterator;  private int index;  EntryItr(){  index = 0;  }  @Override  public boolean hasNext() {  checkMod();  LinkedList<HashMapEntry<K, V>> list = null;  while (list == null && index < table.length) {  list = table[index++];  }  if (list == null) {  return false;  }  iterator = list.iterator();  return iterator.hasNext() || hasNext();  }  @Override  public Entry<K, V> next() {  checkMod();  if(iterator==null){  throw new IllegalStateException();  }  return iterator.next();  }  public void remove() {  checkMod();  if(iterator==null){  throw new IllegalStateException();  }  iterator.remove();  }  private void checkMod(){  if(modCount!=expectedMod){  throw new ConcurrentModificationException();  }  }  }  final class EntrySet extends AbstractSet<Entry<K,V>> {  @Override  public Iterator<Entry<K, V>> iterator() {  return new EntryItr();  }  @Override  public void clear() {  HashMap.this.clear();  }  @Override  public int size() {  return HashMap.this.size();  }  @Override  public boolean contains(Object o) {  return HashMap.this.containsKey(o);  }  @Override  public boolean remove(Object o) {  if(o instanceof Map.Entry){  Map.Entry<?,?> e = (Map.Entry<?,?>) o;  Object key = e.getKey();  Object value = e.getValue();  Object rVal = get(key);  if(!objectEquals(rVal,value)||(rVal == null && !containsKey(key))){  return false;  }  HashMap.this.remove(key);  return true;  }  return false;  }  }  return new EntrySet();  }  @Override  public Object clone(){  try {  HashMap map = (HashMap) super.clone();  map.table = new LinkedList[this.table.length];  for(int i=0;i<this.table.length;i++){  LinkedList<HashMapEntry<K,V>> list = this.table[i];  if(list!=null){  map.table[i] = (LinkedList<HashMapEntry>) list.clone();  }  }  map.modCount = 0;  return map;  } catch (CloneNotSupportedException e) {  throw new RuntimeException(e);  }  }  @Override  public int size() {  int sum = 0;  for (LinkedList<HashMapEntry<K, V>> list : table) {  if(list!=null){  sum+=list.size();  }  }  return sum;  }  @Override  public V get(Object key) {  int h = hash(key);  LinkedList<HashMapEntry<K,V>> list = table[h];  if(list==null){  return null;  }  for (HashMapEntry<K, V> entry : list) {  if(objectEquals(entry.key,key)){  return entry.value;  }  }  return null;  }  @Override  public V put(K key, V value) {  modCount++;  int h = hash(key);  LinkedList<HashMapEntry<K,V>> list = table[h];  if(list == null){  table[h] = (list = new LinkedList<>());  }  for (HashMapEntry<K, V> entry : list){  if(objectEquals(entry.key,key)){  V old = entry.value;  entry.value = value;  return old;  }  }  HashMapEntry<K, V> entry = new HashMapEntry<K, V>(key,value);  list.add(entry);  checkIfNeedResize(list);//检查扩容。  return null;  }  @Override  public V remove(Object key) {  modCount++;  int h = hash(key);  LinkedList<HashMapEntry<K,V>> list = table[h];  if(list==null){  return null;  }  Iterator<HashMapEntry<K,V>> iterator = list.iterator();  for(int i=0;iterator.hasNext();i++){  HashMapEntry<K,V> entry = iterator.next();  if(objectEquals(entry.key,key)){  iterator.remove();  return entry.value;  }  }  return null;  }  @Override  public boolean remove(Object key, Object value) {  modCount++;  int h = hash(key);  LinkedList<HashMapEntry<K,V>> list = table[h];  if(list==null){  return false;  }  Iterator<HashMapEntry<K,V>> iterator = list.iterator();  for(int i=0;iterator.hasNext();i++){  HashMapEntry<K,V> entry = iterator.next();  if(objectEquals(entry.key,key) && objectEquals(entry.value,value)){  iterator.remove();  return true;  }  }  return false;  }  @Override  public void clear() {  modCount++;  for(int i=0;i<table.length;i++){  table[i] = null;  }  }  protected boolean objectEquals(Object a, Object b) {  return (a == b) || (a != null && a.equals(b));  }    private static class HashMapEntry<K,V> implements Entry<K,V>,Serializable{  private static final long serialVersionUID = 3549069266974728431L;  final K key;  V value;  HashMapEntry(K key,V value){  this.key = key;  this.value = value;  }  @Override  public K getKey() {  return key;  }  @Override  public V getValue() {  return value;  }  @Override  public V setValue(V value) {  V old = this.value;  this.value = value;  return old;  }  @Override  public String toString() {  return key+"="+value;  }  @Override  public boolean equals(Object o) {  if (this == o) return true;  if (o == null || getClass() != o.getClass()) return false;  HashMapEntry<?, ?> that = (HashMapEntry<?, ?>) o;  if (key != null ? !key.equals(that.key) : that.key != null) return false;  return value != null ? value.equals(that.value) : that.value == null;  }  @Override  public int hashCode() {  int result = key != null ? key.hashCode() : 0;  result = 31 \* result + (value != null ? value.hashCode() : 0);  return result;  }  }  void debug(){  System.out.print(Arrays.toString(table));  }  public static void main(String[] args) {  HashMap<String,Integer> map = new HashMap<>();  map.put("100",100);  map.put("abc",10000);  System.out.println(map);  System.out.println(map.get("abc"));  map.put("abc",40);  System.out.println(map.get("abc"));  map.put("001",1);  System.out.println(map);  HashMap<String,Integer> map2 = new HashMap<>(map);  map2.put("222",2);  System.out.println(map2);  System.out.println(map2.size());  Set<Entry<String,Integer>> set = map2.entrySet();  set.remove(new HashMapEntry<>("abc",40));  System.out.println(set);  System.out.println(map2);  HashMap<String,Integer> clone = (HashMap<String, Integer>) map2.clone();  clone.put("clone",1);  clone.remove("222",2);  System.out.println(clone);  System.out.println(map2);  }  } |