杨充 Lv3
2019年06月02日 阅读 238

关注

Java数据结构问题

目录介绍

- 3.0.2.2 TreeMap集合结构有何特点？使用场景是什么？将"aababcabcdabcde"打印成a(5)b(4)c(3)d(2)e(1)？
- 3.0.2.3 说一下HashSet集合特点？如何存储null值的？HashSet是如何去重操作？手写产生10个1-20之间的随机数要求随机数不能重复案例？

好消息

- 博客笔记大汇总【15年10月到至今】，包括Java基础及深入知识点，Android技术博客，Python学习笔记等等，还包括平时开发中遇到的bug汇总，当然也在工作之余收集了大量的面试题，长期更新维护并且修正，持续完善……开源的文件是markdown格式的！同时也开源了生活博客，从12年起，积累共计500篇[近100万字]，将会陆续发表到网上，转载请注明出处，谢谢！
- **链接地址：**github.com/yangchong21...
- 如果觉得好，可以star一下，谢谢！当然也欢迎提出建议，万事起于忽微，量变引起质变！**所有博客将陆续开源到GitHub！**

Java博客大汇总，所有笔记开源到GitHub上

- 01.Java基础[30篇]
- 02.面向对象[15篇]
- 03.数据结构[27篇]
- 04.IO流知识[11篇]
- 05.线程进程[9篇]
- 06.虚拟机[5篇]
- 07.类的加载[7篇]
- 08.反射原理[12篇]
- 09.Java并发[17篇]
- 10.Java异常[11篇]
- 11.枚举与注解[5篇]
- 12.设计模式[8篇]
- 13.Java深入[7篇]
- **阅读更多请点击：**Java博客汇总

3.0.0.1 在arrayList中System.arraycopy()和Arrays.copyOf()方法区别联系？System.arraycopy()和Arrays.copyOf()代码说明？

- System.arraycopy()和Arrays.copyOf()方法区别？
  - 比如下面add(int index, E element)方法就很巧妙的用到了arraycopy()方法让数组自己复制自己实现让index开始之后的所有成员后移一个位置:

```
     * 先调用 rangeCheckForAdd 对index进行界限检查；然后调用 ensureCapacityInternal 方法保证capaci
     * 再将从index开始之后的所有成员后移一个位置；将element插入index位置；最后size加1。
     */
    public void add(int index, E element) {
        rangeCheckForAdd(index);
        ensureCapacityInternal(size + 1);
        //arraycopy()方法实现数组自己复制自己
        //elementData:源数组;index:源数组中的起始位置;elementData：目标数组；index + 1：目标数组中的
        System.arraycopy(elementData, index, elementData, index + 1, size - index);
        elementData[index] = element;
        size++;
    }
```

- 如toArray()方法中用到了copyOf()方法

```
    /**
     *以正确的顺序（从第一个到最后一个元素）返回一个包含此列表中所有元素的数组。
     *返回的数组将是“安全的”，因为该列表不保留对它的引用。 （换句话说，这个方法必须分配一个新的数组）。
     *因此，调用者可以自由地修改返回的数组。 此方法充当基于阵列和基于集合的API之间的桥梁。
     */
    public Object[] toArray() {
    //elementData：要复制的数组；size：要复制的长度
        return Arrays.copyOf(elementData, size);
    }
```

- 两者联系与区别
  - 看了上面的两者源代码可以发现 `copyOf()` 内部调用了 `System.arraycopy()` 方法
  - 技术博客大总结
  - 区别：
    - 1.arraycopy()需要目标数组，将原数组拷贝到你自己定义的数组里，而且可以选择拷贝的起点和长度以及放入新数组中的位置
    - 2.copyOf()是系统自动在内部新建一个数组，并返回该数组。
- System.arraycopy()和Arrays.copyOf()代码说明？
  - 使用System.arraycopy()方法

```
    public static void main(String[] args) {
            // TODO Auto-generated method stub
            int[] a = new int[10];
            a[0] = 0;
            a[1] = 1;
```

```
        a[2]=99;
        for (int i = 0; i < a.length; i++) {
                System.out.println(a[i]);
        }
}

//结果:
//0 1 99 2 3 0 0 0 0 0
```

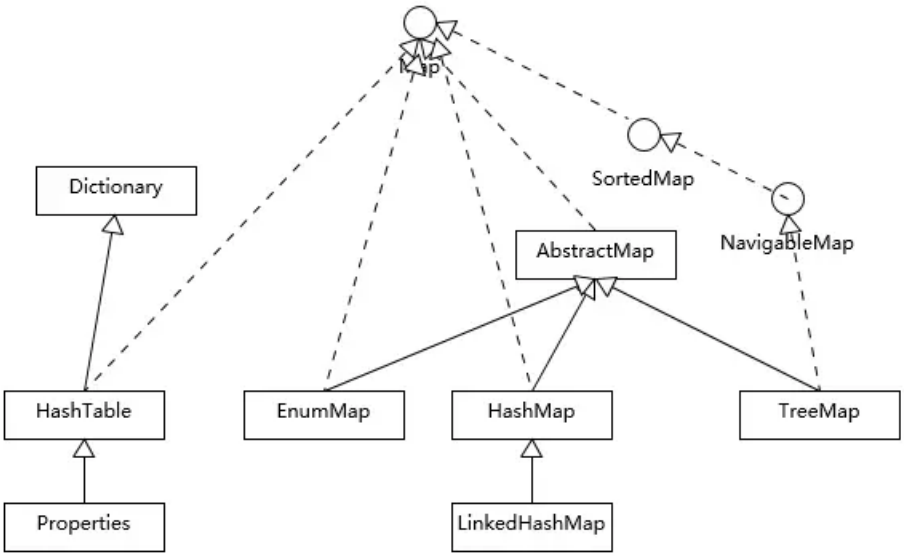
- 使用Arrays.copyOf()方法。技术博客大总结

```
public static void main(String[] args) {
        int[] a = new int[3];
        a[0] = 0;
        a[1] = 1;
        a[2] = 2;
        int[] b = Arrays.copyOf(a, 10);
        System.out.println("b.length"+b.length);
        //结果:
    //10
}
```

- 得出结论
  - `arraycopy()` 需要目标数组，将原数组拷贝到你自己定义的数组里或者原数组，而且可以选择拷贝的起点和长度以及放入新数组中的位置 `copyOf()` 是系统自动在内部新建一个数组，并返回该数组。

3.0.0.3 Collection集合和Map集合的区别？Map集合的特点？说下Map集合整体结构？

- Collection集合和Map集合的区别
  - Map集合由两列组成(双列集合)，而Collection集合由一列组成(单列集合)；Map集合是夫妻对，Collection孤狼
  - Collection集合中的Set集合可以保证元素的唯一性，而Map集合中的键是唯一的
  - Collection集合的数据结构是对存储的元素是有效的,而Map集合的数据结构只和键有关系,和值没有关系。
- Map集合的特点
  - 将键映射到值的对象
  - 一个映射不能包含重复的键
  - 每个键最多只能映射到一个值
- 说下Map集合整体结构？



- 整体结构介绍
  - HashMap 等其他 Map 实现则是都扩展了AbstractMap，里面包含了通用方法抽象。不同Map的用途，从类图结构就能体现出来，设计目的已经体现在不同接口上。
  - Hashtable 比较特别，作为类似 Vector、Stack 的早期集合相关类型，它是扩展了Dictionary 类的，类结构上与 HashMap 之类明显不同。
  - 大部分使用 Map 的场景，通常就是放入、访问或者删除，而对顺序没有特别要求，HashMap 在这种情况下基本是最好的选择。HashMap的性能表现非常依赖于哈希码的有效性，请务必掌握hashCode 和 equals 的一些基本约定，比如：博客
    - equals 相等，hashCode 一定要相等。
    - 重写了 hashCode 也要重写 equals。
    - hashCode 需要保持一致性，状态改变返回的哈希值仍然要一致。
    - equals 的对称、反射、传递等特性。

3.0.0.4 Java集合框架中有哪些类？都有什么特点？集合框架用到Collection接口，这个接口有何特点？

- 可将Java集合框架大致可分为Set、List、Queue 和Map四种体系
  - Set：代表无序、不可重复的集合，常见的类如HashSet、TreeSet
  - List：代表有序、可重复的集合，常见的类如动态数组ArrayList、双向链表LinkedList、数组Vector
  - Map：代表具有映射关系的集合，常见的类如HashMap、LinkedHashMap、TreeMap
  - Queue：代表一种队列集合

- ~~ArrayList：基于动态数组实现，支持随机访问。~~
- Vector：和 ArrayList 类似，但它是线程安全的。
- LinkedList：基于双向链表实现，只能顺序访问，但是可以快速地在链表中间插入和删除元素。不仅如此，LinkedList 还可以用作栈、队列和双向队列。

- **2. Set**

  - TreeSet：基于红黑树实现，支持有序性操作，例如根据一个范围查找元素的操作。但是查找效率不如 HashSet，HashSet 查找的时间复杂度为 O(1)，TreeSet 则为 O(logN)。
  - HashSet：基于哈希表实现，支持快速查找，但不支持有序性操作。并且失去了元素的插入顺序信息，也就是说使用 Iterator 遍历 HashSet 得到的结果是不确定的。
  - LinkedHashSet：具有 HashSet 的查找效率，且内部使用双向链表维护元素的插入顺序。

- **3.Map**

  - TreeMap：基于红黑树实现。
  - HashMap：基于哈希表实现。
  - HashTable：和 HashMap 类似，但它是线程安全的，这意味着同一时刻多个线程可以同时写入 HashTable 并且不会导致数据不一致。它是遗留类，不应该去使用它。现在可以使用 ConcurrentHashMap 来支持线程安全，并且 ConcurrentHashMap 的效率会更高，因为 ConcurrentHashMap 引入了分段锁。博客
  - LinkedHashMap：使用双向链表来维护元素的顺序，顺序为插入顺序或者最近最少使用（LRU）顺序。

- **4. Queue**

  - LinkedList：可以用它来实现双向队列。
  - PriorityQueue：基于堆结构实现，可以用它来实现优先队列。

- Java带有一组接口和类，使得操作成组的对象更为容易，这就是集合框架

  - 集合框架主要用到的是Collection接口，Collection是将其他对象组织到一起的一个对象，提供了一种方法来存储、访问和操作其元素
  - List、Set和Queue是Collection的三个主要的子接口。此外，还有一个Map接口用于存储键值对

Collection	Collection是最基本的集合接口，一个 Collection 代表一组Object，Java不提供直接继承自Collection的类，只提供继承于它的子接口
List	List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，能够通过索引来访问List中的元素，而且允许有相同的元素
Set	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
Queue	Queue通过先进先出的方式来存储元素，即当获取元素时，最先获得的元素是最先添加的元素，依次递推
SortedSet	继承于Set保存有序的集合
Map	将唯一的键映射到值
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

3.0.0.5 ArrayList添加元素时如何扩容？如何添加元素到指定位置，该操作复制是深拷贝还是浅拷贝？

- ArrayList添加元素时如何扩容
  - 通过add方法添加元素，其操作是将指定的元素追加到此列表的末尾。博客
  - 它的实现其实最核心的内容就是 `ensureCapacityInternal` 。这个函数其实就是**自动扩容机制的核心**。依次来看一下他的具体实现。

```
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    ensureExplicitCapacity(minCapacity);
}
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
private void grow(int minCapacity) {
```

```
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        // 如果扩为1.5倍还不满足需求，直接扩为需求值
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

  - 当增加数据的时候，如果ArrayList的大小已经不满足需求时，那么就将数组变为原长度的1.5倍，之后的操作就是把老的数组拷到新的数组里面。例如，默认的数组大小是10，也就是说当我们 `add` 10个元素之后，再进行一次add时，就会发生自动扩容，数组长度由10变为了15具体情况如下所示。
- 如何添加元素到指定位置，该操作拷贝是深拷贝还是浅拷贝？
  - 在指定索引处添加一个元素，先对index进行界限检查，；然后调用 ensureCapacityInternal 方法保证capacity足够大，再将从index开始之后的所有成员后移一个位置；将element插入index位置；最后size加1。博客
  - 可以看出它比add(index)方法还要多一个System.arraycopy。arraycopy()这个实现数组之间复制的方法一定要看一下，下面就用到了arraycopy()方法实现数组自己复制自己。该System.arraycopy()拷贝确实是浅拷贝，不会进行递归拷贝，所以产生的结果是基本数据类型是值拷贝，对象只是引用拷贝。
  - arraycopy()需要目标数组，将原数组拷贝到你自己定义的数组里，而且可以选择拷贝的起点和长度以及放入新数组中的位置

```
public void add(int index, E element) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

    ensureCapacityInternal(size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

- 测试一下System.arraycopy()是深拷贝还是浅拷贝？

```
public static void main(String[] args) {
    ArrayList<String> listStr = new ArrayList<>();
    for(int i = 0 ; i < 3 ;i++){
```

```
    ArrayList<String> listStrCopy = (ArrayList<String>) listStr.clone();
    //修改clone后对象的值
    listStrCopy.remove(2);
    listStrCopy.add(100+"");
    for (int i = 0; i < listStr.size(); i++) {
        System.out.println(listStr.get(i).toString());
        System.out.println(listStrCopy.get(i).toString());
    }
}
```

实验结果，可以看出修改对原始数据没有改变，是复制了值

```
0
0
1
1
2
100
```

3.0.0.6 如何理解ArrayList的扩容消耗？Arrays.asList方法后的List可以扩容吗？ArrayList如何序列化？

- 如何理解ArrayList的扩容消耗
  - 由于ArrayList使用elementData = Arrays.copyOf(elementData, newCapacity);进行扩容，而每次都会重新创建一个newLength长度的数组，所以扩容的空间复杂度为O(n),时间复杂度为O(n)

```
public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType) {
    T[] copy = ((Object)newType == (Object)Object[].class)
        ? (T[]) new Object[newLength]
        : (T[]) Array.newInstance(newType.getComponentType(), newLength);
    System.arraycopy(original, 0, copy, 0,
                     Math.min(original.length, newLength));
    return copy;
}
```

- Arrays.asList方法后的List可以扩容吗？
  - 不能，asList返回的List为只读的。其原因为：asList方法返回的ArrayList是Arrays的一个内部类，并且没有实现add，remove等操作
- List怎么实现排序？
  - 实现排序，可以使用自定义排序：list.sort(new Comparator(){...})
  - 或者使用Collections进行快速排序：Collections.sort(list)
- ArrayList如何序列化？

- 技术博客大总结
- 保存元素的数组 elementData 使用 transient 修饰，该关键字声明数组默认不会被序列化。

```
transient Object[] elementData; // non-private to simplify nested class access
```

- ArrayList 实现了 writeObject() 和 readObject() 来控制只序列化数组中有元素填充那部分内容。

```
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    elementData = EMPTY_ELEMENTDATA;
    s.defaultReadObject();
    s.readInt(); // ignored
    if (size > 0) {
        ensureCapacityInternal(size);
        Object[] a = elementData;
        for (int i=0; i<size; i++) {
            a[i] = s.readObject();
        }
    }
}

private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    int expectedModCount = modCount;
    s.defaultWriteObject();
    s.writeInt(size);
    for (int i=0; i<size; i++) {
        s.writeObject(elementData[i]);
    }
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}
```

- 序列化时需要使用 ObjectOutputStream 的 writeObject() 将对象转换为字节流并输出。而 writeObject() 方法在传入的对象存在 writeObject() 的时候会去反射调用该对象的 writeObject() 来实现序列化。反序列化使用的是 ObjectInputStream 的 readObject() 方法，原理类似。

```
ArrayList list = new ArrayList();
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(file));
oos.writeObject(list);
```

ArrayList有何不同？

- **读写机制**
  - ArrayList在执行插入元素是超过当前数组预定义的最大值时，数组需要扩容，扩容过程需要调用底层System.arraycopy()方法进行大量的数组复制操作；在删除元素时并不会减少数组的容量（如果需要缩小数组容量，可以调用trimToSize()方法）；在查找元素时要遍历数组，对于非null的元素采取equals的方式寻找。
  - LinkedList在插入元素时，须创建一个新的Entry对象，并更新相应元素的前后元素的引用；在查找元素时，需遍历链表；在删除元素时，要遍历链表，找到要删除的元素，然后从链表上将此元素删除即可。
  - Vector与ArrayList仅在插入元素时容量扩充机制不一致。对于Vector，默认创建一个大小为10的Object数组，并将capacityIncrement设置为0；当插入元素数组大小不够时，如果capacityIncrement大于0，则将Object数组的大小扩大为现有size+capacityIncrement；如果capacityIncrement<=0,则将Object数组的大小扩大为现有大小的2倍。博客
- **读写效率**
  - ArrayList对元素的增加和删除都会引起数组的内存分配空间动态发生变化。因此，对其进行插入和删除速度较慢，但检索速度很快。
  - LinkedList由于基于链表方式存放数据，增加和删除元素的速度较快，但是检索速度较慢。
- **什么是CopyOnWriteArrayList，它与ArrayList有何不同？**
  - CopyOnWriteArrayList是ArrayList的一个线程安全的变体，其中所有可变操作（add、set等等）都是通过对底层数组进行一次新的复制来实现的。相比较于ArrayList它的写操作要慢一些，因为它需要实例的快照。
  - CopyOnWriteArrayList中写操作需要大面积复制数组，所以性能肯定很差，但是读操作因为操作的对象和写操作不是同一个对象，读之间也不需要加锁，读和写之间的同步处理只是在写完后通过一个简单的"="将引用指向新的数组对象上来，这个几乎不需要时间，这样读操作就很快很安全，适合在多线程里使用，绝对不会发生ConcurrentModificationException ，因此CopyOnWriteArrayList适合使用在读操作远远大于写操作的场景里，比如缓存。
  - 技术博客大总结

3.0.0.8 如何理解Java集合的快速失败机制 “fail-fast” ？出现这个原因是什么？有何解决办法？

- Java集合的快速失败机制 “fail-fast”
  - java集合的一种错误检测机制，当多个线程对集合进行结构上的改变的操作时，有可能会产生 fail-fast 机制。
    - 例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集

- 出现这个原因是什么？
  - 迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变modCount的值。每当迭代器使用hashNext()/next()遍历下一个元素之前，都会检测modCount变量是否为expectedmodCount值，是的话就返回遍历；否则抛出异常，终止遍历。
- 有何解决办法：博客
  - 1.在遍历过程中，所有涉及到改变modCount值得地方全部加上synchronized。
  - 2.使用CopyOnWriteArrayList来替换ArrayList
- 看源码如下所示
  - modCount 用来记录 ArrayList 结构发生变化的次数。结构发生变化是指添加或者删除至少一个元素的所有操作，或者是调整内部数组的大小，仅仅只是设置元素的值不算结构发生变化。
  - 在进行序列化或者迭代等操作时，需要比较操作前后 modCount 是否改变，如果改变了需要抛出 ConcurrentModificationException。

```
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    // Write out element count, and any hidden stuff
    int expectedModCount = modCount;
    s.defaultWriteObject();

    // Write out size as capacity for behavioural compatibility with clone()
    s.writeInt(size);

    // Write out all elements in the proper order.
    for (int i=0; i<size; i++) {
        s.writeObject(elementData[i]);
    }

    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}
```

3.0.0.9 LinkedList集合有何特点？LinkedList相比ArrayList效率如何？用什么进行论证？

- LinkedList集合有何特点
  - LinkedList 同时实现了 List 接口和 Deque 接口，所以既可以将 LinkedList 当做一个有序容器，也可以将之看作一个队列（Queue），同时又可以看作一个栈（Stack）。虽然 LinkedList 和 ArrayList 一样都实现了 List 接口，但其底层是通过**双向链表**来实现的，所以

- 简单总结一下
  - 实现了List接口和Deque接口，是双端链表
  - 支持高效的插入和删除操作
  - 不是线程安全的
- 如果想使LinkedList变成线程安全的
  - 可以调用静态类

```
List list =Collections.synchronizedList(new LinkedList(...));
```

- LinkedList相比ArrayList效率如何
  - LinkedList 相比 ArrayList 添加和移除元素的效率会高些，但随机访问元素的效率要比 ArrayList 低，这里我也来做个测试，看下两者之间的差距
  - 分别向 ArrayList 和 LinkedList 存入同等数据量的数据，然后各自移除 100 个元素以及遍历 10000 个元素，观察两者所用的时间。博客

```
public static void main(String[] args) {

        List<String> stringArrayList = new ArrayList<>();
        for (int i = 0; i < 300000; i++) {
            stringArrayList.add("leavesC " + i);
        }
        //开始时间
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            stringArrayList.remove(100 + i);
        }
        //结束时间
        long endTime = System.currentTimeMillis();
        System.out.println("移除 ArrayList 中的100个元素，所用时间：" + (endTime - startTime) + "

        //开始时间
        startTime = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            stringArrayList.get(i);
        }
        //结束时间
        endTime = System.currentTimeMillis();
        System.out.println("遍历 ArrayList 中的10000个元素，所用时间：" + (endTime - startTi

        List<String> stringLinkedList = new LinkedList<>();
        for (int i = 0; i < 300000; i++) {
```

```
        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            stringLinkedList.remove(100 + i);
        }
        //结束时间
        endTime = System.currentTimeMillis();
        System.out.println("移除 LinkedList 中的100个元素，所用时间：" + (endTime - startTime) +
        //开始时间
        startTime = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            stringLinkedList.get(i);
        }
        //结束时间
        endTime = System.currentTimeMillis();
        System.out.println("遍历 LinkedList 中的10000个元素，所用时间：" + (endTime - startTime)
    }
```

- 可以看出来，两者之间的差距还是非常大的，因此，在使用集合时需要根据实际情况来判断到底哪一种数据结构才更加适合。

3.0.1.0 HashSet和TreeSet的区别？是如何保证唯一值的，底层怎么做到的？

- HashSet
  - 不能保证元素的排列顺序；使用Hash算法来存储集合中的元素，有良好的存取和查找性能；通过equal()判断两个元素是否相等，并两个元素的hashCode()返回值也相等
- TreeSet
  - 是SortedSet接口的实现类，根据元素实际值的大小进行排序；采用红黑树的数据结构来存储集合元素；支持两种排序方法：自然排序（默认情况）和定制排序。前者通过实现Comparable接口中的compareTo()比较两个元素之间大小关系，然后按升序排列；后者通过实现Comparator接口中的compare()比较两个元素之间大小关系，实现定制排列

3.0.1.3 HashMap有哪些特点，简单说一下？HashMap内部的结构是怎样的？简单说一下什么是桶，作用是什么？

- HashMap有哪些特点，简单说一下？
  - 几个关键的信息：
    - 基于Map接口实现、允许null键/值、是非同步(这点很重要，多线程注意)、不保证有(比如插入的顺序)、也不保证序不随时间变化。
  - 如何理解允许null键/值？

        - HashMap 不保证元素顺序，根据需要该容器可能会对元素重新哈希，元素的顺序也会被重新打散，因此在不同时间段迭代同一个 HashMap 的顺序可能会不同。
    - 如何理解非同步？
        - HashMap 非线程安全，即任一时刻有多个线程同时写 HashMap 的话可能会导致数据的不一致
- HashMap内部的结构是怎样的
    - HashMap 内部的结构，它可以看作是数组（Node[] table）和链表结合组成的复合结构，数组被分为一个个桶（bucket），通过哈希值决定了键值对在这个数组的寻址；哈希值相同的键值对，则以链表形式存储，你可以参考下面的示意图。
    - 这里需要注意的是，如果链表大小超过阈值（TREEIFY_THRESHOLD, 8，图中的链表就会被改造为树形结构。
- 结构图如下所示
    -

3.0.1.4 当有键值对插入时，HashMap会发生什么？对于查找一个key时，HashMap会发生什么？

- 当有键值对插入时，HashMap会发生什么？
    - 首先，键的哈希值被计算出来，然后这个值会赋给 Entry 类中对应的 hashCode 变量。
    - 然后，使用这个哈希值找到它将要被存入的数组中“桶”的索引。
    - 如果该位置的“桶”中已经有一个元素，那么新的元素会被插入到“桶”的头部，next 指向上一个元素——本质上使“桶”形成链表。
- 对于查找一个key时，HashMap会发生什么？
    - 键的哈希值先被计算出来

mArray[2index+1]。博客

- 这里的时间复杂度从 O(1) 上升到 O(logN)，但是内存效率提升了。当我们在 100 左右的数据量范围内尝试时，没有耗时的问题，察觉不到时间上的差异，但我们应用的内存效率获得了提高。

3.0.1.5 HashMap和Hashtable的区别？HashMap在put、get元素的过程？体现了什么数据结构？

- HashMap
  - 基于AbstractMap类，实现了Map、Cloneable（能被克隆）、Serializable（支持序列化）接口； 非线程安全；允许存在一个为null的key和任意个为null的value；采用链表散列的数据结构，即数组和链表的结合；初始容量为16，填充因子默认为0.75，扩容时是当前容量翻倍，即2capacity
- Hashtable
  - 基于Map接口和Dictionary类；线程安全，开销比HashMap大，如果多线程访问一个Map对象，使用Hashtable更好；不允许使用null作为key和value；底层基于哈希表结构；初始容量为11，填充因子默认为0.75，扩容时是容量翻倍+1，即2capacity+1
  - HashTable里使用的是synchronized关键字，这其实是对对象加锁，锁住的都是对象整体，当Hashtable的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。
- HashMap在put、get元素的过程
  - 向Hashmap中put元素时，首先判断key是否为空，为空则直接调用putForNullKey()，不为空则计算key的hash值得到该元素在数组中的下标值；如果数组在该位置处没有元素，就直接保存；如果有，还要比较是否存在相同的key，存在的话就覆盖原来key的value，否则将该元素保存在链头，先保存的在链尾。
  - 从Hashmap中get元素时，计算key的hash值找到在数组中的对应的下标值，返回该key对应的value即可，如果有冲突就遍历该位置链表寻找key相同的元素并返回对应的value
- 体现了什么数据结构
  - HashMap采用链表散列的数据结构，即数组和链表的结合，在Java8后又结合了红黑树，当链表元素超过8个将链表转换为红黑树
  - 技术博客大总结

3.0.1.6 如何保证HashMap线程安全？底层怎么实现的？HashMap是有序的吗？如何实现有序？

- 使用ConcurrentHashMap可保证线程安全
  - ConcurrentHashMap是线程安全的HashMap，它采取锁分段技术，将数据分成一段一段存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段

将数组元素作为锁，对每一行数据进行加锁，可减少并发冲突的概率
        - 数据结构由“数组＋单向链表”变为“数组＋单向链表＋红黑树”，使得查询的时间复杂度可以降低到O(logN)，改进一定的性能。
    - 通俗一点解释：ConcurrentHashMap引入了分割(Segment)，可以理解为把一个大的Map拆分成N个小的HashTable，在put方法中，会根据hash(paramK.hashCode())来决定具体存放进哪个Segment，如果查看Segment的put操作，我们会发现内部使用的同步机制是基于lock操作的，这样就可以对Map的一部分（Segment）进行上锁，这样影响的只是将要放入同一个Segment的元素的put操作，保证同步的时候，锁住的不是整个Map（HashTable就是这么做的），相对于HashTable提高了多线程环境下的性能，因此HashTable已经被淘汰了。技术博客大总结
- 使用LinkedHashMap可实现有序
    - HashMap是无序的，而LinkedHashMap是有序的HashMap，默认为插入顺序，还可以是访问顺序，基本原理是其内部通过Entry维护了一个双向链表，负责维护Map的迭代顺序

3.0.1.7 HashMap存储两个对象的hashcode相同会发生什么？如果两个键的hashcode相同，你如何获取值对象？

- HashMap存储两个对象的hashcode相同会发生什么？
    - 错误回答：因为hashcode相同，所以两个对象是相等的，HashMap将会抛出异常，或者不会存储它们。
    - 正确回答：两个对象就算hashcode相同，但是它们可能并不相等。如果不明白，可以先看看我的这篇博客：Hash和HashCode深入理解。回答“因为hashcode相同，所以它们的bucket位置相同，‘碰撞’会发生。因为HashMap使用链表存储对象，这个Entry(包含有键值对的Map.Entry对象)会存储在链表中。
- HashMap1.7和1.8的区别
    - 在JDK1.6，JDK1.7中，HashMap采用数组+链表实现，即使用链表处理冲突，同一hash值的链表都存储在一个链表里。但是当位于一个链表中的元素较多，即hash值相等的元素较多时，通过key值依次查找的效率较低。
    - JDK1.8中，HashMap采用位数组+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树，这样大大减少了查找时间。
- 如果两个键的hashcode相同，你如何获取值对象？
    - 当调用get()方法，HashMap会使用键对象的hashcode找到bucket位置，然后获取值对象。当然如果有两个值对象储存在同一个bucket，将会遍历链表直到找到值对象。
    - 在没有值对象去比较，如何确定确定找到值对象的？因为HashMap在链表中存储的是键值对，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，最终找到找的值对象。
    - 技术博客大总结

    - hashCode()方法返回的是int整数类型，其范围为-(2^31)~(2^31-1)，约有40亿个映射空间，而HashMap的容量范围是在16（初始化默认值）~2 ^ 30，HashMap通常情况下是取不到最大值的，并且设备上也难以提供这么多的存储空间，从而导致通过hashCode()计算出的哈希值可能不在数组大小范围内，进而无法匹配存储位置；
- HashMap是使用了哪些方法来有效解决哈希冲突的
    - 1.使用链地址法（使用散列表）来链接拥有相同hash值的数据；
    - 2.使用2次扰动函数（hash函数）来降低哈希冲突的概率，使得数据分布更平均；
    - 3.引入红黑树进一步降低遍历的时间复杂度，使得遍历更快；
- 如何解决匹配存储位置问题
    - HashMap自己实现了自己的hash()方法，通过两次扰动使得它自己的哈希值高低位自行进行异或运算，降低哈希碰撞概率也使得数据分布更平均；
    - 在保证数组长度为2的幂次方的时候，使用hash()运算之后的值与运算（&）（数组长度 - 1）来获取数组下标的方式进行存储，这样一来是比取余操作更加有效率，二来也是因为只有当数组长度为2的幂次方时，h&(length-1)才等价于h%length，三来解决了“哈希值与数组大小范围不匹配”的问题；
- 为什么数组长度要保证为2的幂次方呢？
    - 只有当数组长度为2的幂次方时，h&(length-1)才等价于h%length，即实现了key的定位，2的幂次方也可以减少冲突次数，提高HashMap的查询效率；
    - 技术博客大总结
    - 如果 length 为 2 的次幂 则 length-1 转化为二进制必定是 11111……的形式，在于 h 的二进制与操作效率会非常的快，而且空间不浪费；如果 length 不是 2 的次幂，比如 length 为 15，则 length - 1 为 14，对应的二进制为 1110，在于 h 与操作，最后一位都为 0 ，而 0001，0011，0101，1001，1011，0111，1101 这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！这样就会造成空间的浪费。

3.0.1.9 为什么HashMap中String、Integer这样的包装类适合作为K？如果要用对象最为key，该如何操作？

- 为什么HashMap中String、Integer这样的包装类适合作为K？
    - String、Integer等包装类的特性能够保证Hash值的不可更改性和计算准确性，能够有效的减少Hash碰撞的几率
        - 都是final类型，即不可变性，保证key的不可更改性，不会存在获取hash值不同的情况
        - 内部已重写了equals()、hashCode()等方法，遵守了HashMap内部的规范（不清楚去上面看看putValue的过程），不容易出现Hash值计算错误的情况；
- 想要让自己的Object作为K应该怎么办呢？
    - 重写hashCode()和equals()方法

撞；

- 重写equals()方法，需要遵守自反性、对称性、传递性、一致性以及对于任何非null的引用值x，x.equals(null)必须返回false的这几个特性，目的是为了保证key在哈希表中的唯一性；

- 总结
  - 采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择。
  - 技术博客大总结
- 如果要用对象最为key，该如何操作？
  - 需要重写hashCode()和equals()方法，实例代码如下所示：

```
public class Key {

    private final String name;
    private final int width;
    private final int heifht;

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        Key key = (Key) o;
        if (width != key.width) {
            return false;
        }
        if (heifht != key.heifht) {
            return false;
        }
        return name != null ? name.equals(key.name) : key.name == null;
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + width;
        result = 31 * result + heifht;
        return result;
```

3.0.2.0 HashMap是如何扩容的？如何理解HashMap的大小超过了负载因子定义的容量？重新调整HashMap大小存在什么问题吗？

- HashMap是为啥要扩容
  - 当链表数组的容量超过初始容量*加载因子（默认0.75）时，再散列将链表数组扩大2倍，把原链表数组的搬移到新的数组中。为什么需要使用加载因子？为什么需要扩容呢？因为如果填充比很大，说明利用的空间很多，如果一直不进行扩容的话，链表就会越来越长，这样查找的效率很低，扩容之后，将原来链表数组的每一个链表分成奇偶两个子链表分别挂在新链表数组的散列位置，这样就减少了每个链表的长度，增加查找效率。
- 如何理解HashMap的大小超过了负载因子(load factor)定义的容量？
  - 默认的负载因子大小为0.75，也就是说，当一个map填满了75%的bucket时候，和其它集合类(如ArrayList等)一样，将会创建原来HashMap大小的两倍的bucket数组，来重新调整map的大小，并将原来的对象放入新的bucket数组中。这个过程叫作rehashing，因为它调用hash方法找到新的bucket位置。
- 重新调整HashMap大小存在什么问题吗？技术博客大总结
  - 当多线程的情况下，可能产生条件竞争。当重新调整HashMap大小的时候，确实存在条件竞争，因为如果两个线程都发现HashMap需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免尾部遍历(tail traversing)。如果条件竞争发生了，那么就死循环了。

3.0.2.1 HashMap是线程安全的吗？多线程条件下put存储数据会发生什么情况？如何理解它并发性？

- HashMap是非线程安全的，那么测试一下，先看下测试代码

```
private HashMap map = new HashMap();
private void test(){
    Thread t1 = new Thread() {
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                map.put(new Integer(i), i);
            }
            LogUtils.d("yc-----执行结束----t1");
        }
    };
    //省略一部分线程代码，和t1一样
    t1.start();
```

```
        t5.start();
        t6.start();
    }
```

- 就是启了6个线程，不断的往一个非线程安全的HashMap中put/get内容，put的内容很简单，key和value都是从0自增的整数（这个put的内容做的并不好，以致于后来干扰了我分析问题的思路）。对HashMap做并发写操作，我原以为只不过会产生脏数据的情况，但反复运行这个程序，会出现线程t1、t2被卡住的情况，多数情况下是一个线程被卡住另一个成功结束，偶尔会6个线程都被卡住。博客
- 多线程条件下put存储数据会发生什么情况
  - CPU利用率过高一般是因为出现了出现了死循环，导致部分线程一直运行，占用cpu时间。问题原因就是HashMap是非线程安全的，多个线程put的时候造成了某个key值Entry key List的死循环，问题就这么产生了。
  - 当另外一个线程get 这个Entry List 死循环的key的时候，这个get也会一直执行。最后结果是越来越多的线程死循环，最后导致卡住。我们一般认为HashMap重复插入某个值的时候，会覆盖之前的值，这个没错。但是对于多线程访问的时候，由于其内部实现机制(在多线程环境且未作同步的情况下，对同一个HashMap做put操作可能导致两个或以上线程同时做rehash动作，就可能导致循环键表出现，一旦出现线程将无法终止，持续占用CPU，导致CPU使用率居高不下)，就可能出现安全问题了。
- 如何解决HashMap多线程并发问题？
  - 多线程下直接使用ConcurrentHashMap，解决了这个问题。

3.0.2.2 TreeMap集合结构有何特点？使用场景是什么？将"aababcabcdabcde"打印成a(5)b(4)c(3)d(2)e(1)？

- TreeMap集合结构特点
  - 键的数据结构是红黑树,可保证键的排序和唯一性
  - 排序分为自然排序和比较器排序，如果使用的是自然排序,对元素有要求,要求这个元素需要实现 Comparable 接口
  - 线程是不安全的效率比较高

```
public TreeMap(): 自然排序
public TreeMap(Comparator<? super K> comparator):  使用的是比较器排序
```

- 使用场景是什么？
  - 之前已经学习过HashMap和LinkedHashMap了，HashMap不保证数据有序，LinkedHashMap保证数据可以保持插入顺序，而如果我们希望Map可以**保持key的大小**的时候，我们就需要利用TreeMap了。博客

- "aababcabcdabcde" 按照键值对的形式存储到TreeMap集合中
- 分析:博客
  - 1，遍历字符串,获取每一个字符,然后将当前的字符作为键，上map集合中查找对应的值
  - 2，如果返回的值不是null 对值进行+1，在把当前的元素作为键，值是+1以后的结果存储到集合中
  - 3，如果返回的是是null，不存在，就把当前遍历的元素作为键，1 作为值,添加到集合中
- 代码如下

```
public static void main(String[] args) {
    // 定义字符串
    String s = "aababcabcdabcde" ;
    // 创建TreeMap集合对象
    TreeMap<Character , Integer> tm = new TreeMap<Character , Integer>() ;
    // 遍历字符串
    for(int x = 0 ; x < s.length() ; x++) {
        // 获取当前索引出对应的字符
        char ch = s.charAt(x) ;
        // 找值
        Integer value = tm.get(ch) ;
        // 判断
        if(value == null) {
            tm.put(ch, 1) ;
        }else {
            value += 1 ;
            tm.put(ch, value) ;
        }
    }
    // 遍历Map集合按照指定的形式拼接字符串
    StringBuilder sb = new StringBuilder() ;
    Set<Entry<Character,Integer>> entrySet = tm.entrySet() ;
    for(Entry<Character,Integer> en : entrySet) {
        // 获取键
        Character key = en.getKey() ;
        // 获取值
        Integer value = en.getValue() ;
        // a(5)b(4)c(3)d(2)e(1)
        // 拼接
        sb.append(key).append("(").append(value).append(")") ;
    }
    // 把sb转换成String
    String result = sb.toString() ;
    // 输出
    System.out.println(result);
}
```

- 说一下HashSet集合特点？
  - HashSet 实现了 Set 接口，不允许插入重复的元素，允许包含 null 元素，且不保证元素迭代顺序，特别是不保证该顺序恒久不变
  - HashSet 的代码十分简单，去掉注释后的代码不到两百行。HashSet 底层是通过 HashMap 来实现的。
- 案例测试
  - HashSet是根据hashCode来决定存储位置的，是通过HashMap实现的，所以对象必须实现hashCode()方法，存储的数据无序不能重复，可以存储null,但是只能存一个。
- 如何存储null值的？
- HashSet是如何去重操作？
  - 在向 HashSet 添加元素时，HashSet 会将该操作转换为向 HashMap 添加键值对，如果 HashMap 中包含 key 值与待插入元素相等的键值对（hashCode() 方法返回值相等，通过 equals() 方法比较也返回 true），则待添加的键值对的 value 会覆盖原有数据，但 key 不会有所改变，因此如果向 HashSet 添加一个已存在的元素时，元素不会被存入 HashMap 中，从而实现了 HashSet 元素不重复的特征。博客
- **产生10个1-20之间的随机数要求随机数不能重复案例**

```
/**
 * 产生10个1-20之间的随机数,要求不能重复
 * 分析:
 *         1: 创建一个HashSet集合对象 , 作用: 存储产生的随机数
 *         2: 生成随机数 , 把随机数添加到集合中
 *         3: 使用循环,当集合的长度大于等于10退出循环 , 小于10就一直循环
 */
// 创建一个HashSet集合对象 , 作用: 存储产生的随机数
HashSet<Integer> hs = new HashSet<Integer>() ;
while(hs.size() < 10) {
    // 使用Random类
    Random random = new Random() ;
    int num = random.nextInt(20) + 1 ;
    // 把num添加到集合中
    hs.add(num) ;
}
// 遍历
for(Integer i : hs) {
    System.out.println(i);
}
```

其他介绍