

Java中Collection和Collections的区别？

- `java.util.Collection` 是一个**集合接口（集合类的一个顶级接口）**。它提供了对集合对象进行基本操作的通用接口方法。`Collection`接口在Java类库中有很多具体的实现。`Collection`接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有`List`与`Set`。
- `Collections`则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作：

1) 排序(Sort)

2) 混排 (Shuffling)：混排算法所做的正好与 `sort` 相反：它打乱在一个 `List` 中可能有的任何排列的踪迹。也就是说，基于随机源的输入重排该 `List`，这样的排列具有相同的可能性（假设随机源是公正的）。这个算法在实现一个碰运气的游戏中是非常有用的。例如，它可被用来混排代表一副牌的 `Card` 对象的一个 `List`。另外，在生成测试案例时，它也是十分有用的。

3) 反转(Reverse)：使用`Reverse`方法可以根据元素的自然顺序 对指定列表按降序进行排序。

`Collections.reverse(list)`

4) 替换所有的元素(Fill)使用指定元素替换指定列表中的所有元素。

`Collections.fill(li,"aaa");`

5) 拷贝(Copy)：用两个参数，一个目标 `List` 和一个源 `List`，将源的元素拷贝到目标，并覆盖它的内容。目标 `List` 至少与源一样长。如果它更长，则在目标 `List` 中的剩余元素不受影响。

`Collections.copy(list,li)`：前面一个参数是目标列表，后一个是源列表。

6) 返回Collections中最小元素(min): 根据指定比较器产生的顺序, 返回给定 collection 的最小元素。collection 中的所有元素都必须是通过指定比较器可相互比较的。

Collections.min(list)

7) 返回Collections中最小元素(max): 根据指定比较器产生的顺序, 返回给定 collection 的最大元素。collection 中的所有元素都必须是通过指定比较器可相互比较的。

Collections.max(list)

8) lastIndexOfSubList: 返回指定源列表中最后一次出现指定目标列表的起始位置

int count = Collections.lastIndexOfSubList(list,li);

9) IndexOfSubList: 返回指定源列表中第一次出现指定目标列表的起始位置

int count = Collections.indexOfSubList(list,li);

10) Rotate: 根据指定的距离循环移动指定列表中的元素

Collections.rotate(list,-1);

如果是负数, 则正向移动, 正数则方向移动

List面试题:

List与Set区别?

1、List简介

实际上有两种List: 一种是基本的ArrayList,其优点在于随机访问元素, 另一种是LinkedList,它并不是为快速随机访问设计的, 而是快速的插入或删除。

ArrayList: 由数组实现的List。允许对元素进行快速随机访问, 但是向List中间插入与移除元素的速度很慢。

LinkedList: 对顺序访问进行了优化, 向List中间插入与删除的开销并不大。随机访问则相对较慢。

还具有下列方法: addFirst(), addLast(), getFirst(), getLast(), removeFirst() 和 removeLast(), 这些方法 (没有任何接口或基类中定义过)使得LinkedList可以当作堆栈、队列和双向队列使用。

2、Set简介

Set具有与Collection完全一样的接口, 因此没有任何额外的功能。实际上Set就是Collection,只是行为不同。这是继承与多态思想的典型应用: 表现不同的行为。Set不保存重复的元素(至于如何判断元素相同则较为复杂)

Set: 存入Set的每个元素都必须是**唯一**的, 因为Set不保存重复元素。加入Set的元素必须定义equals()方法以确保对象的唯一性。Set与Collection有完全一样的接口。Set接口不保证维护元素的次序。

HashSet: 为快速查找设计的Set。存入HashSet的对象必须定义hashCode()。

TreeSet: 保存次序的Set, 底层为树结构。使用它可以从Set中提取有序的序列。

3、list与Set区别

(1) List,Set都是继承自Collection接口

(2) List特点: 元素有放入顺序, 元素可重复, Set特点: 元素无放入顺序, 元素不可重复, 重复元素会覆盖掉, (元素虽然无放入顺序, 但是元素在set中的位置是有该元素的HashCode决定的, 其位置其实是固定的, 加入Set 的Object必须定义equals()方法, 另外list支持for循环, 也就是通过下标来遍历, 也可以用迭代器, 但是set只能用迭代, 因为他无序, 无法用下标来取得想要的值。)

(3) Set和List对比:

Set: 检索元素效率低下, 删除和插入效率高, 插入和删除不会引起元素位置改变。

List: 和数组类似, List可以动态增长, 查找元素效率高, 插入删除元素效率低, 因为会引起其他元素位置改变。

List有哪些种类?

List接口的实现类有ArrayList与LinkedList, Vector

ArrayList：底层由数组结构实现，数组在内存中的存储顺序是连续的，对集合中的元素可以进行快速访问，更适合用来随机查询数据。

LinkedList：底层由双向链表结构实现，通过节点来存储下一个元素的位置，对集合中的元素可以方便的增加与删除，更适合用于大量修改。

Vector：Vector与ArrayList的区别就是Vector是线程安全的集合，在需要线程安全而且对效率要求比较低的情况下，使用Vector。

ArrayList和LinkedList的区别？

1. ArrayList是实现了**基于动态数组**的数据结构，而LinkedList是**基于链表**的数据结构；
2. 对于**随机访问get和set**，ArrayList**要优于**LinkedList，因为LinkedList要移动指针；
3. 对于添加和删除操作add和remove，一般大家都会说LinkedList要比ArrayList快，因为ArrayList要移动数据。但是实际情况并非这样，对于添加或删除，LinkedList和ArrayList**并不能明确说明谁快谁慢**，下面会详细分析：

因为查找需要删除或添加数据的位置时，ArrayList时间复杂度为 $O(1)$ ，而LinkedList是通过二分查找来找的，时间复杂度为 $O(\log 2n)$ 。但LinkedList的增删是快于ArrayList（因为ArrayList需要移动index下标后面的数据），这就导致两个的增删时间快慢没有那么绝对，由以下规则决定：

- 当数据量较小时，测试程序中，大约小于30的时候，两者效率差不多，没有显著区别；
- 当插入的数据量大时，大约在容量的1/10之前，LinkedList会优于ArrayList。
- 当数据量较大时，大约在容量的1/10处开始，LinkedList的效率就开始没有ArrayList效率高了，特别到一半以及后半的位置插入时，LinkedList效率明显要低于ArrayList，而且数据量越大，越明显。

LinkedList怎么实现栈、队列以及双端队列？

- 实现栈：

```
public class Test {
    public static void main(String[] args) {
        Deque<String> stack = new LinkedList<>();
        stack.push("a");
        stack.push("b");
        stack.push("c");
        while (stack.peek() != null) {
            System.out.println("查看元素,并不删除元素:"+stack.peek());
            System.out.println(stack.pop());
        }
    }
}
```

<http://blog.csdn.net/>

(1)

"C:\Program Files\Java\jdk1.7.0_79\bin\java" ...

查看元素,并不删除元素:c

push：表示入栈，在头部添加元素，栈的空间可能是有限的，如果栈满了，push会抛出异常IllegalStateException。

pop：表示出栈，返回头部元素，并且从栈中删除，如果栈为空，会抛出异常NoSuchElementException。

peek：查看栈头部元素，不修改栈，如果栈为空，返回null。

- 实现队列：

利用队列的接口的实现

```
public class Test {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.offer("1");
        queue.offer("2");
        queue.offer("3");
        while (queue.peek() != null) {
            System.out.println(queue.poll());
        }
    }
}
```

http://blog.csdn.net/

st (1)

"C:\Program Files\Java\jdk1.7.0_79\bin\java" ...

1

2

3

- 实现双端队列：

```
Deque<Object> list = new LinkedList<>();
```

栈和队列只是双端队列的特殊情况，它们的方法都可以使用双端队列的方法替代，使用不同的名称和方法，概念上更为清晰。看看Deque接口里面的方法

Deque

- addFirst(E): void
- addLast(E): void
- offerFirst(E): boolean 双端队列的操作，可以用来实现队列和栈操作。每种增删查都对应2个方法，分别
- offerLast(E): boolean 用在集合元素为空或者满了情况使用。
- removeFirst(): E
- removeLast(): E
- pollFirst(): E
- pollLast(): E
- getFirst(): E
- getLast(): E
- peekFirst(): E
- peekLast(): E
- removeFirstOccurrence(Object): boolean
- removeLastOccurrence(Object): boolean
- add(E): boolean 1 Queue
- offer(E): boolean 1 Queue 队列操作，每个操作对应2种，满或者空特殊情况来使用。
- remove(): E 1 Queue
- poll(): E 1 Queue
- element(): E 1 Queue
- peek(): E 1 Queue
- push(E): void 栈对应操作
- pop(): E
- remove(Object): boolean 1 Collection
- contains(Object): boolean 1 Collection
- size(): int 1 Collection
- iterator(): Iterator<E> 1 Collection
- descendingIterator(): Iterator<E>

493 * @param e the element to push

494 * @throws IllegalStateException if the deque is full

495 * @throws ClassCastException if the element prevents it from being added

496 * @throws NullPointerException if the deque does not permit null elements

497 * @throws IllegalArgumentException if the element prevents it from being added

498 * @return true if the element was added to the front of the deque

499 * @return false if the deque is full

500 * @throws IllegalArgumentException if the element prevents it from being added

501 * @return true if the element was added to the back of the deque

502 * @return false if the deque is full

503 * @return the element at the front of the deque

504 * @throws NoSuchElementException if the deque is empty

505 * @return the element at the back of the deque

506 * @throws NoSuchElementException if the deque is empty

507 * @return the element at the front of the deque

508 * @throws NoSuchElementException if the deque is empty

509 * @return the element at the back of the deque

510 * @throws NoSuchElementException if the deque is empty

511 * @return the element at the front of the deque

512 * @throws NoSuchElementException if the deque is empty

513 * @return the element at the back of the deque

514 * @throws NoSuchElementException if the deque is empty

LinkedList是单向还是双向链表？

双向链表；

LinkedList是循环链表吗？

从JDK1.7开始，LinkedList 由双向循环链表改为双向链表

首先，简单介绍一下LinkedList：

LinkedList是List接口的双向链表实现。由于是链表结构，所以长度没有限制；而且添加/删除元素的时候，只需要改变指针的指向（把链表断开，插入/删除元素，再把链表连起来）即可，非常方便，而ArrayList却需要重整数组（add/remove中间元素）。所以LinkedList适合用于添加/删除操作频繁的情况。

在JDK 1.7之前（此处使用JDK1.6来举例），LinkedList是通过headerEntry实现的一个循环链表的。先初始化一个空的Entry，用来做header，然后首尾相连，形成一个循环链表：

在LinkedList中提供了两个基本属性size、header。

```
1 private transient Entry<E> header = new Entry<E>(null, null, null);
2 private transient int size = 0;
```

其中size表示的LinkedList的大小，header表示链表的表头，Entry为节点对象。

```
1 1 private static class Entry<E> {
2 2     E element;           //元素节点
3 3     Entry<E> next;       //下一个元素
4 4     Entry<E> previous;   //上一个元素
5 5
6 6     Entry(E element, Entry<E> next, Entry<E> previous) {
7 7         this.element = element;
8 8         this.next = next;
9 9         this.previous = previous;
10 10    }
11 11 }
```

每次添加/删除元素都是默认在链尾操作。对应此处，就是在header前面操作，因为遍历是next方向的，所以在header前面操作，就相当于在链表尾操作。

如下面的插入操作addBefore以及图示，如果插入obj_3，只需要修改header.previous和obj_2.next指向obj_3即可。

```
1 1 private Entry<E> addBefore(E e, Entry<E> entry) {
2 2     //利用Entry构造函数构建一个新节点 newEntry,
3 3     Entry<E> newEntry = new Entry<E>(e, entry, entry.previous);
4 4     //修改newEntry的前后节点的引用，确保其链表的引用关系是正确的
5 5     newEntry.previous.next = newEntry;
6 6     newEntry.next.previous = newEntry;
7 7     //容量+1
8 8     size++;
9 9     //修改次数+1
10 10    modCount++;
11 11    return newEntry;
12 12 }
```

在addBefore方法中无非就是做了这件事：构建一个新节点newEntry，然后修改其前后的引用。

在JDK 1.7，1.6的headerEntry循环链表被替换成了first和last组成的非循环链表。

```
1 transient int size = 0;
2
3 /**
4  * Pointer to first node.
5  * Invariant: (first == null && last == null) ||
6  *             (first.prev == null && first.item != null)
7  */
8 transient Node<E> first;
9
```

```

10     /**
11      * Pointer to last node.
12      * Invariant: (first == null && last == null) ||
13      *             (last.next == null && last.item != null)
14      */
15     transient Node<E> last;

```

在初始化的时候，不用去new一个Entry。

```

1     /**
2      * Constructs an empty list.
3      */
4     public LinkedList() {
5     }

```

在插入/删除的时候，也是默认在链尾操作。把插入的obj当成newLast，挂在oldLast的后面。另外还要先判断first是否为空，如果为空则first = obj。

如下面的插入方法linkLast，在尾部操作，只需要把obj_3.next指向obj_4即可。

```

1     void linkLast(E e) {
2         final Node<E> l = last;
3         final Node<E> newNode = new Node<>(l, e, null);
4         last = newNode;
5         if (l == null)
6             first = newNode;
7         else
8             l.next = newNode;
9         size++;
10        modCount++;
11    }

```

其中：

```

1     private static class Node<E> {
2         E item;
3         Node<E> next;
4         Node<E> prev;
5
6         Node(Node<E> prev, E element, Node<E> next) {
7             this.item = element;
8             this.next = next;
9             this.prev = prev;
10        }
11    }

```

单向和双向链表的区别？

双向链表是链表的一种，它和单向链表的区别是双向链表比单向链表每个节点多一个头指针，这个指针指向前一个节点，也就是说，每个节点包含包含头指针、存储元素、尾指针，因此从这个节点可以同时访问到它前面和后面的节点。我们可以想一下，**如果是单向链表，要访问一个节点前面的节点，是不是要从头结点开始遍历，直到找个这个节点前面的节点为止，而双向链表直接就可以访问前一个节点，查询和操作数据就会更加方便。**世界上没有完美的东西，有利就有弊，方便了数据操作的同时，牺牲的是所占的空间，因为每个节点要多出一个头指针，必然会多占用一定的内存空间，这也是空间换时间的一种方式。

ArrayList扩容机制？

扩容是再添加元素时才会出现的情况,有的情况是不指定初始容量第一次添加元素时,直接看add()方法

```
1  /**
2   * Appends the specified element to the end of this list.
3   *
4   * @param e element to be appended to this list
5   * @return <tt>true</tt> (as specified by {@link Collection#add})
6   */
7  public boolean add(E e) {
8      //先将集合的大小加一,代表有一个元素要加进来,开口有没有它的容身之处
9      ensureCapacityInternal(size + 1); // Increments modCount!!
10     //将新元素添加到集合中
11     elementData[size++] = e;
12     return true;
13 }
14
```

跳转到ensureCapacityInternal方法中进行验证

```
1  private void ensureCapacityInternal(int minCapacity) {
2      //DEFAULTCAPACITY_EMPTY_ELEMENTDATA初始化的值,也就是空
3      if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
4          //如果是为空的话,默认的DEFAULT_CAPACITY=10传入的minCapacity哪个大取哪个
5          minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
6      }
7      ensureExplicitCapacity(minCapacity);
8  }
```

继续调用ensureExplicitCapacity方法,传入判断之后的值,第一次add的话这个就是默认的10

```
1  private void ensureExplicitCapacity(int minCapacity) {
2      //对集合操作的次数
3      modCount++;
4
5      // overflow-conscious code
6      //传入的参数减去数组的长度是否大于0,大于0的话就代表要进行扩容了
7      if (minCapacity - elementData.length > 0)
8          grow(minCapacity);
9  }
```

判断传入的参数(第一次为10)减去数组的长度是否大于0,大于0的话调用grow扩容方法,数组的长度是elementData.length也可以说是容量,集合的大小是size,两个值是不同的

```
1  /**
2   * Increases the capacity to ensure that it can hold at least the
3   * number of elements specified by the minimum capacity argument.
4   *
5   * @param minCapacity the desired minimum capacity
6   */
7  private void grow(int minCapacity) {
8      // overflow-conscious code
9      //旧的容量为当前数组的长度
10     int oldCapacity = elementData.length;
11     //新的容量为旧容量1.5倍,>>1代表右移一位,也就是÷2
12     int newCapacity = oldCapacity + (oldCapacity >> 1);
13     //新容量-旧容量是否小于0,一般是不指定容量,第一次add时才会进
14     if (newCapacity - minCapacity < 0)
15         //新容量等于传入的参数
```



```

16         newCapacity = minCapacity;
17         //如果新的容量超过了集合的阈值
18         if (newCapacity - MAX_ARRAY_SIZE > 0)
19             //调用hugeCapacity方法进行进一步的计算
20             newCapacity = hugeCapacity(minCapacity);
21         // minCapacity is usually close to size, so this is a win:
22
23         //底层的数组进行copy后长度变为新的容量
24         elementData = Arrays.copyOf(elementData, newCapacity);
25     }

```

当新容量大于集合的阈值时,调用hugeCapacity方法

```

1     private static int hugeCapacity(int minCapacity) {
2         //为负数的话抛出异常,一般没这个可能
3         if (minCapacity < 0) // overflow
4             throw new OutOfMemoryError();
5         //三元表达式:新容量大于集合容量阈值时,新的容量为Integer的最大阈值,否则为集合的阈值
6         return (minCapacity > MAX_ARRAY_SIZE) ?
7             Integer.MAX_VALUE :
8             MAX_ARRAY_SIZE;
9     }

```

MAX_ARRAY_SIZE的值其实为Integer.MAX_VALUE-8,为什么要减8呢?因为数组也是一个对象,对象需要一定的内存存储对象头信息,对象头信息最大占用内存不可超过8字节。

Vector是什么?

Vector类实现了一个动态数组。和 ArrayList 很相似,但是两者是不同的:

- Vector 是同步访问的。
- Vector 包含了许多传统的方法,这些方法不属于集合框架。

CopyOnWriteArrayList和Vector 的区别?

Vector 介绍:

Vector 内部是使用 synchronized 来保证线程安全的,并且锁的粒度比较大,都是方法级别的锁,在并发量高的时候,很容易发生竞争,并发效率相对较低。在这一点上,Vector 和 Hashtable 很类似。

并且,前面这几种 List 在迭代期间不允许编辑,如果在迭代期间进行添加或删除元素等操作,则会抛出 ConcurrentModificationException 异常,这样的特点也在很多情况下给使用者带来了麻烦。

CopyOnWriteArrayList介绍:

JDK1.5 开始,Java 并发包里提供了使用 CopyOnWrite 机制实现的并发容器,CopyOnWriteArrayList 作为主要的并发 List, CopyOnWrite 的并发集合还包括 CopyOnWriteArraySet, 其底层正是利用 CopyOnWriteArrayList 实现的。

CopyOnWriteArrayList的适用场景:

适合读多写少的场景。

CopyOnWrite思想:

当容器需要修改时,不直接修改当前容器,而是先将当前容器进行 Copy,复制出一个新的容器,然后修改新的容器,完成修改之后,再将原容器的引用指向新的容器。这样就完成了整个修改过程,读取也不会因为修改受影响。

CopyOnWrite读不需要加锁，CopyOnWriteArrayList迭代期间允许修改集合内容，ArrayList 在迭代期间如果修改集合的内容，会抛出 ConcurrentModificationException 异常。

如下代码，CopyOnWriteArrayList能正常输出，将CopyOnWriteArrayList改为ArrayList 运行list.add(4);会报错

```
1 public class CopyOnWriteArrayListDemo {
2     public static void main(String[] args) {
3         CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>(new
4         Integer[]{1, 2, 3});
5         System.out.println(list); //[1, 2, 3]
6         //Get iterator 1
7         Iterator<Integer> itr1 = list.iterator();
8
9         //Add one element and verify list is updated
10        list.add(4);
11        System.out.println(list); //[1, 2, 3, 4]
12
13        //Get iterator 2
14        Iterator<Integer> itr2 = list.iterator();
15        System.out.println("====Verify Iterator 1 content====");
16        itr1.forEachRemaining(System.out::println); //1,2,3
17        System.out.println("====Verify Iterator 2 content====");
18        itr2.forEachRemaining(System.out::println); //1,2,3,4
19    }
20 }
```

CopyOnWriteArrayList缺点：

- 内存占用问题

因为 CopyOnWrite 的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，这一点会占用额外的内存空间。

- 在元素较多或者复杂的情况下，复制的开销很大

复制过程不仅会占用双倍内存，还需要消耗 CPU 等资源，会降低整体性能。

- 数据一致性问题

由于 CopyOnWrite 容器的修改是先修改副本，所以这次修改对于其他线程来说，并不是实时能看到的，只有在修改完之后才能体现出来。如果你希望写入的数据马上能被其他线程看到，CopyOnWrite 容器是不适用的。

CopyOnWriteArrayList源码分析：

- CopyOnWriteArrayList添加元素方法add ()：

```
1 public boolean add(E e) {
2     // 加锁
3     final ReentrantLock lock = this.lock;
4     lock.lock();
5     try {
6
7         // 得到原数组的长度和元素
8         Object[] elements = getArray();
9         int len = elements.length;
10
11        // 复制出一个新数组
12        Object[] newElements = Arrays.copyOf(elements, len + 1);
13
14        // 添加时，将新元素添加到新数组中
```

```

15         newElements[len] = e;
16
17         // 将volatile Object[] array 的指向替换成新数组
18         setArray(newElements);
19         return true;
20     } finally {
21         lock.unlock();
22     }
23 }

```

添加过程加锁，添加的过程和之前CopyOnWrite思想是一致的。

▪ 获取元素get ()：

```

1 public E get(int index) {
2     return get(getArray(), index);
3 }
4 final Object[] getArray() {
5     return array;
6 }
7 private E get(Object[] a, int index) {
8     return (E) a[index];
9 }

```

可以看出，get 相关的操作没有加锁，保证了读取操作的高速。

▪ 迭代器COWIterator 类

这个迭代器有两个重要的属性，分别是 Object[] snapshot 和 int cursor。其中 snapshot 代表数组的快照，也就是创建迭代器那个时刻的数组情况，而 cursor 则是迭代器的游标。迭代器的构造方法如下：

```

1 private COWIterator(Object[] elements, int initialCursor) {
2     cursor = initialCursor;
3     snapshot = elements;
4 }

```

可以看出，迭代器在被构建的时候，会把当时的 elements 赋值给 snapshot，而之后的迭代器所有的操作都基于 snapshot 数组进行的，比如：

```

1 public E next() {
2     if (! hasNext())
3         throw new NoSuchElementException();
4     return (E) snapshot[cursor++];
5 }

```

在 next 方法中可以看到，返回的内容是 snapshot 对象，所以，后续就算原数组被修改，这个 snapshot 既不会感知到，也不会受影响，执行迭代操作不需要加锁，也不会因此抛出异常。迭代器返回的结果，和创建迭代器的时候的内容一致。

Set面试题：

HashSet：无序

LinkedHashSet：按照插入顺序

TreeSet：从小到大排序

HashSet：

HashSet是Set接口的典型实现，大多数时候使用Set集合时就是使用这个实现类。HashSet按Hash算法来存储集合中的元素，因此具有很好的存取和查找性能。底层数据结构是**哈希表**。

HashSet具有以下特点：

- 不能保证元素的排列顺序，顺序可能与添加顺序不同，顺序也可能发生变化；
- HashSet不是同步的；
- 集合元素值可以是null,但只能放入一个null(再有null放入不会生效也不会报错)

HashSet内部存储机制：

当向HashSet集合中存入一个元素时，HashSet会调用该对象的hashCode方法来得到该对象的hashCode值，然后根据该hashCode值决定该对象在HashSet中的存储位置。如果有两个元素通过equals方法比较true，但它们的hashCode方法返回的值不相等，HashSet将会把它们存储在不同位置，依然可以添加成功。

也就是说。HashSet集合判断两个元素的标准是两个对象通过equals方法比较相等，并且两个对象的hashCode方法返回值也相等。

靠元素重写hashCode方法和equals方法来判断两个元素是否相等，如果相等则**覆盖**原来的元素，依此来确保元素的唯一性（（HashSet、HashMap、Hashtable默认的负载极限是0.75）

LinkedHashSet：

LinkedHashSet同样是根据**元素的hashCode值**来决定元素的存储位置，只是底层是使用链表存储的，当遍历该集合时候，LinkedHashSet将会以元素的**添加顺序**访问集合的元素。LinkedHashSet在迭代访问Set中的全部元素时，性能比HashSet好，但是插入时性能稍微逊色于HashSet。

TreeSet：

TreeMap 的实现就是红黑树数据结构，**底层以 TreeMap 保存集合元素**。

TreeSet是SortedSet接口的唯一实现类，TreeSet可以确保集合元素处于排序状态。TreeSet支持两种排序方式，自然排序 和 定制排序，其中自然排序为默认的排序方式。向TreeSet中加入的应该是同一个类的对象。

TreeSet判断两个对象不相等的方式是两个对象通过equals方法返回false，或者通过CompareTo方法比较没有返回0

自然排序：

自然排序使用要排序元素的CompareTo（Object obj）方法来比较元素之间大小关系，然后将元素按照升序排列。Java提供了一个Comparable接口，该接口里定义了一个compareTo(Object obj)方法，该方法返回一个整数值，实现了该接口的对象就可以比较大小。obj1.compareTo(obj2)方法如果返回0，则说明被比较的两个对象相等，如果返回一个正数，则表明obj1大于obj2，如果是 负数，则表明obj1小于obj2。如果我们将两个对象的equals方法总是返回true，则这两个对象的compareTo方法返回应该返回0

定制排序：

自然排序是根据集合元素的大小，以升序排列，如果要定制排序，应该使用Comparator接口，实现 int compare(T o1,T o2)方法

Set怎么保证唯一性？

HashSet和LinkedHashSet是通过重写hashCode方法和equals方法来判断两个元素是否相等，如果相等则**覆盖**原来的元素；

TreeSet通过CompareTo比较，小的存左边，大的存右边，**相等就不存**。

Map面试题：

HashMap 和 Hashtable 有什么区别？

1. HashMap是线程不安全的，HashTable是线程安全的；

2. HashMap中允许键和值为null，HashTable不允许；
3. HashMap的默认容器是16，为2倍扩容，HashTable默认是11，为2倍+1扩容；

ConcurrentHashMap 和 Hashtable 的区别？

ConcurrentHashMap 与 Hashtable都是线程安全的。

1. 出现版本不同：

Hashtable 在JDK1.0 就存在了，JDK1.2 版本中实现了 Map 接口，成为了集合框架的一员。ConcurrentHashMap 则是在 JDK1.5 中才出现的，也正是因为它们出现的年代不同，而后出现的往往是对前面出现的类的优化，所以它们在实现方式以及性能上，也存在着较大的不同。

2. 实现线程安全的方式不同：

- Hashtable 实现并发安全的原理是通过 synchronized 关键字实现线程安全：

举例：

```
1 public synchronized void clear() {
2     Entry<?,?> tab[] = table;
3     modCount++;
4     for (int index = tab.length; --index >= 0; )
5         tab[index] = null;
6     count = 0;
7 }
```

可以看出这个 clear() 方法是被 synchronized 关键字所修饰的，同理其他的方法例如 put、get、size 等，也同样是 synchronized 关键字修饰的。之所以 Hashtable 是线程安全的，是因为几乎每个方法都被 synchronized 关键字所修饰了，这也就保证了线程安全。

- ConcurrentHashMap实现线程安全的原理是利用了 CAS + synchronized + Node 节点的方式，这和 Hashtable 的完全利用 synchronized 的方式有很大的不同。

3. 性能不同：

多线程情况下，Hashtable 的性能很差，，因为每一次修改都需要锁住整个对象，而其他线程在此期间是不能操作的。不仅如此，还会带来额外的上下文切换等开销，**所以此时它的吞吐量甚至还不如单线程的情况。**

而在 ConcurrentHashMap 中，就算上锁也仅仅会对一部分上锁而不是全部都上锁，所以多线程中的吞吐量通常都会大于单线程的情况，也就是说，在并发效率上，ConcurrentHashMap 比 Hashtable 提高了很多。

4. 迭代时修改的不同：

Hashtable在迭代时修改会抛出ConcurrentModificationException 异常，ConcurrentHashMap 在迭代时是可以修改的。

总结：Hashtable 已经不再推荐使用。

HashMap为什么不完全采用数组？

数组的好处是抄可以根据下标快速的找到对应的元素。而链表的好处是只用知道插入位置的前后，不需要一个一个的位置。这样就提高了插入的速度或者删除的速度。就样二者的优势结合一下就提高了查找的速度 也提高了增删的速度。

还有一个原因是数组长度是固定的，不好扩展长度。

HashMap扩容：

什么时候触发扩容？

一般情况下，**当元素数量超过阈值时**便会触发扩容。每次扩容的容量都是之前容量的2倍。

HashMap的容量是有上限的，必须小于 $1 \ll 30$ ，即1073741824。如果容量超出了这个数，则不再增长，且阈值会被设置为Integer.MAX_VALUE ($2^{31} - 1$ ，即永远不会超出阈值了)。

JDK7中的扩容机制

JDK7的扩容机制相对简单，有以下特性：

- 空参数的构造函数：以默认容量、默认负载因子、默认阈值初始化数组。内部数组是**空数组**。
- 有参构造函数：根据参数确定容量、负载因子、阈值等。
- 第一次put时会初始化数组，其容量变为**不小于指定容量的2的幂数**。然后根据负载因子确定阈值。
- 如果不是第一次扩容，则新容量 = 旧容量 * 2，新阈值 = 新容量 * 负载因子。

JDK8的扩容机制

JDK8的扩容做了许多调整。

HashMap的容量变化通常存在以下几种情况：

1. 空参数的构造函数：实例化的HashMap默认内部数组是null，即没有实例化。第一次调用put方法时，则会开始第一次初始化扩容，长度为16。
2. 有参构造函数：用于指定容量。会根据指定的正整数找到**不小于指定容量的2的幂数**，将这个数设置赋值给**阈值**（threshold）。第一次调用put方法时，会将阈值赋值给容量，然后让 阈值 = 容量 * 负载因子。（因此并不是我们手动指定了容量就一定不会触发扩容，超过阈值后一样会扩容！！）
3. 如果不是第一次扩容，则容量变为原来的2倍，阈值也变为原来的2倍。（*容量和阈值都变为原来的2倍时，负载因子还是不变*）

此外还有几个细节需要注意：

- 首次put时，先会触发扩容（算是初始化），然后存入数据，然后判断是否需要扩容；
- 不是首次put，则不再初始化，直接存入数据，然后判断是否需要扩容；

扩容时容量的计算方法说完了，下面说一说元素的迁移。

JDK7的元素迁移

JDK7中，HashMap的内部数据保存的都是链表。因此逻辑相对简单：在准备好新的数组后，map会遍历数组的每个“桶”，然后遍历桶中的每个Entity，重新计算其hash值（也有可能不计算），找到新数组中的对应位置，以**头插法**插入新的链表。

这里有几个注意点：

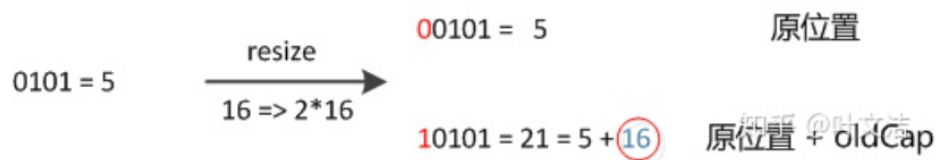
- 是否要重新计算hash值的条件这里不深入讨论，读者可自行查阅源码。
- 因为是头插法，因此新旧链表的元素位置会发生转置现象。
- 元素迁移的过程中在多线程情境下有可能会触发死循环（无限进行链表反转）。

JDK8的元素迁移

JDK8则因为巧妙的设计，性能有了大大的提升：由于数组的容量是以2的幂次方扩容的，那么一个Entity在扩容时，新的位置要么在**原位置**，要么在**原长度+原位置**的位置。原因如下图：

n - 1	0000 0000 0000 0000 0000 0000 0000 1111	→	1111 1111 1111 1111 0000 1111 0001 1111
hash1	1111 1111 1111 1111 0000 1111 0000 0101		1111 1111 1111 1111 0000 1111 0000 0101
hash2	1111 1111 1111 1111 0000 1111 0001 0101		1111 1111 1111 1111 0000 1111 0001 0101

数组长度变为原来的2倍，表现在二进制上就是**多了一个高位参与数组下标确定**。此时，一个元素通过hash转换坐标的方法计算后，恰好出现一个现象：最高位是0则坐标不变，最高位是1则坐标变为“10000+原坐标”，即“原长度+原坐标”。如下图：



因此，在扩容时，不需要重新计算元素的hash了，只需要判断最高位是1还是0就好了。

JDK8的HashMap还有以下细节：

- JDK8在迁移元素时是正序的，不会出现链表转置的发生。
- 如果某个桶内的元素超过8个，则会将链表转化成红黑树，加快数据查询效率。

版本		JDK 1.8	JDK 1.7
扩容流程		<p>前置条件</p> <div> <div>插入键值对时，发现容量不足</div> </div> <p>开始</p> <p>异常情况判断</p> <ul style="list-style-type: none"> 是否需 初始化 若当前容量 > 最大值 则不扩容 <p>根据新容量 (2倍) 新建数组 (new table)</p> <p>保存旧的数组 (old table)</p> <p>将旧数组上的数据 (键值对) 转移到新数组中 (old table -> new table)</p> <div> <div>遍历旧数组的每个数据 (old table)</div> <div> 重新计算每个数据在新数组中的存储位置 <ul style="list-style-type: none"> 原位置 Or 原位置 + 旧容量) 含需插入的数据 </div> <div> 将旧数组上的每个数据 逐个转移到新数组中 <ul style="list-style-type: none"> 尾插法，即插入到链表尾部 含需插入的数据 </div> </div> <p>新数组table引用到HashMap的table属性上</p> <p>重新设置扩容阈值 (threshold)</p> <p>扩容结束 (此时哈希表table = 扩容后 (2倍) & 转移了旧数据的新table)</p>	<p>前置条件</p> <div> <div>插入键值对时，发现容量不足</div> </div> <p>开始</p> <p>保存旧的数组 (old table)</p> <p>根据新容量 (2倍) 新建数组 (new table)</p> <p>将旧数组上的数据 (键值对) 转移到新数组中 (old table -> new table)</p> <div> <div>遍历旧数组的每个数据 (old table)</div> <div> 重新计算每个数据在新数组中的存储位置 <ul style="list-style-type: none"> 同put () 过程: h & (length-1) 不含需插入的数据 </div> <div> 将旧数组上的每个数据 逐个转移到新数组中 <ul style="list-style-type: none"> 头插法，即插入到链表头部 不含需插入的数据 </div> </div> <p>新数组table引用到HashMap的table属性上</p> <p>重新设置扩容阈值 (threshold)</p> <p>扩容结束 (此时哈希表table = 扩容后 (2倍) & 转移了旧数据的新table)</p> <p>后置条件</p> <div> <div>重新计算 准备插入数据对应的 hash值 (计算方式同 put函数过程)</div> <div>重新计算 准备插入数据对应的 存储数组位置 (计算方式同 put函数过程)</div> <div>将数据插入到对应的存储数组位置</div> </div>
区别	扩容后存储位置的计算方式	按照扩容后的规律计算 (即 扩容后的位置 = 原位置 or 原位置 + 旧容量)	全部按照原来方法进行计算 (即 hashCode () ->> 扰动处理 ->> (h & length-1))
	转移数据方式	尾插法 (直接插入到链表尾部 / 红黑树，不会出现逆序 & 环形链表死循环问题)	头插法 (先将原位置的数据移到后1位，再插入数据到该位置；会出现逆序 & 环形链表死循环问题)
	需插入数据的插入时机 & 位置重计算时机	扩容前插入、转移数据时统一计算	扩容后插入、单独计算 (即 转移数据时无统一计算) https://blog.csdn.net/qq_36520235

LinkedHashMap:

LinkedHashMap是HashMap的子类，它是一个将所有Entry节点链入一个双向链表的HashMap。LinkedHashMap的元素存取过程基本与HashMap基本类似，只是在细节实现上稍有不同。当然，这是由LinkedHashMap本身的特性所决定的，因为它额外维护了一个双向链表用于保持迭代顺序（即：插入顺序和访问顺序）；此外，LinkedHashMap可以很好的支持LRU算法（这个后面了解：https://blog.csdn.net/justloveyou_/article/details/71713781）。

LinkedHashMap的存入源码：

- LinkedHashMap 没有对 put(key,value) 方法进行任何直接的修改，完全继承了HashMap的put(Key,Value)方法
- LinkedHashMap 没有对 put(key,value) 方法进行任何直接的修改，完全继承了HashMap的put(Key,Value)方法，下面我们对比地看一下LinkedHashMap和HashMap的addEntry方法的具体实现：

```
1      /**
2       * This override alters behavior of superclass put method. It causes newly
3       * allocated entry to get inserted at the end of the linked list and
4       * removes the eldest entry if appropriate.
5       *
6       * LinkedHashMap中的addEntry方法
7       */
8      void addEntry(int hash, K key, V value, int bucketIndex) {
9
10         //创建新的Entry，并插入到LinkedHashMap中
11         createEntry(hash, key, value, bucketIndex); // 重写了HashMap中的
createEntry方法
12
13         //双向链表的第一个有效节点（header后的那个节点）为最近最少使用的节点，这是用来
支持LRU算法的
14         Entry<K,V> eldest = header.after;
15         //如果有必要，则删除掉该近期最少使用的节点，
16         //这要看对removeEldestEntry的覆写，由于默认为false，因此默认是不做任何处理
的。
17         if (removeEldestEntry(eldest)) {
18             removeEntryForKey(eldest.key);
19         } else {
20             //扩容到原来的2倍
21             if (size >= threshold)
22                 resize(2 * table.length);
23         }
24     }
25
26     -----我是分割线-----
27
28     /**
29      * Adds a new entry with the specified key, value and hash code to
30      * the specified bucket. It is the responsibility of this
31      * method to resize the table if appropriate.
32      *
33      * Subclass overrides this to alter the behavior of put method.
34      *
35      * HashMap中的addEntry方法
36      */
37     void addEntry(int hash, K key, V value, int bucketIndex) {
38         //获取bucketIndex处的Entry
39         Entry<K,V> e = table[bucketIndex];
40
41         //将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry
```

```

42         table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
43
44         //若HashMap中元素的个数超过极限了，则容量扩大两倍
45         if (size++ >= threshold)
46             resize(2 * table.length);
47     }

```

以上源码我们可以知道，在LinkedHashMap中向哈希表中插入新Entry的同时，还会通过Entry的addBefore方法将其链入到双向链表中。其中，addBefore方法本质上是一个双向链表的插入操作，其源码如下：

```

1 //在双向链表中，将当前的Entry插入到existingEntry(header)的前面
2 private void addBefore(Entry<K,V> existingEntry) {
3     after = existingEntry;
4     before = existingEntry.before;
5     before.after = this;
6     after.before = this;
7 }

```

到此为止，我们分析了在LinkedHashMap中put一条键值对的完整过程。总的来说，相比HashMap而言，LinkedHashMap在向哈希表添加一个键值对的同时，也会将其链入到它所维护的双向链表中，以便设定迭代顺序。

TreeSet和TreeMap的关系

与HashSet完全类似，TreeSet里面绝大部分方法都是直接调用TreeMap方法来实现的。

相同点：

1. TreeMap和TreeSet都是有序的集合，也就是说他们存储的值都是排好序的。
2. TreeMap和TreeSet都是非同步集合，因此他们不能在多线程之间共享，不过可以使用方法Collections.synchronizedMap()来实现同步。
3. 运行速度都要比Hash集合慢，他们内部对元素的操作时间复杂度为 $O(\log N)$ ，而HashMap/HashSet则为 $O(1)$ 。

不同点：

1. 最主要的区别就是TreeSet和TreeMap分别实现Set和Map接口
2. TreeSet只存储一个对象，而TreeMap存储两个对象Key和Value（仅仅key对象有序）
3. TreeSet中不能有重复对象，而TreeMap中可以重复

TreeMap 组成：

TreeMap的实现使用了红黑树数据结构，也就是一棵自平衡的排序二叉树，这样就可以保证快速检索指定节点。对于TreeMap而言，它采用一种被称为“红黑树”的排序二叉树来保存Map中每个Entry——每个Entry都被当成“红黑树”的一个节点对待

为什么开发中使用HashMap和ArrayList效率要比其他集合高？

- 1、ArrayList底层数据结构是数组，数组的内存分配空间是连续的，所以通过偏移量来查询某个元素的值效率是最高的。
- 2、Hashmap底层数据结构是数组+链表+红黑树，内存分布非连续的，而链表和红黑树在插入和删除操作上效率相对于ArrayList和AVL树是较高的，当然在查询的效率上也是很高。
- 3、ArrayList和Hashmap非线程安全，减少了使用锁的消耗

4、其他集合，例如 Hashset 基于 hashmap, LinkedHashMap 继承 Hashset, TreeSet 继承 Abstractmap，由此可见，其他集合类是在 hashmap 的基础上进行新增一些其他功能，也是因为 hashmap 效率之高从而应用之广泛。

HashMap特点：

是否可以空	key允许有一个null，value你随意
是否有序（插入时候的顺序和读取时是否一致）	hash码具有随机性，所以无序
是否允许重复	key肯定不行，value你随意
是否线程安全	线程不安全（jdk8以下会出现链表成环）
特性	时间复杂度为常数级

手写HashMap

■ 新增：

1. 首先拿到key，获取key的hashCode码：

```
1 static final int hash(Object key) { //jdk1.8获取hashCode（为了保证散列性才这么计算）
2     int h;
3     return key == null ? 0 : (h = key.hashCode()) ^ h >>> 16;
4 }
```

2. 将hashCode对数组的长度取余，找到数组位置

3. 如果数组为空，直接放入；

如果不为空，如果key存在，则覆盖数据；

如果为树节点，走红黑树插入；

其他情况，走链表尾插法逻辑（需要判断链表长度是否 ≥ 7 ，满足转红黑树，以及判断是否需要扩容）；

是否需要扩容：若entry数量 \geq 数组长度 * 阈值，则需要扩容。

■ 获取：

调用get方法时，我们根据key的hashcode计算它对应的index，然后直接去table中的对应位置查找即可，如果有链表就遍历。

■ 删除：

移除某个节点时，如果该key对应的index处没有形成链表，那么直接置为null。如果存在链表，我们需要将目标节点的前驱节点的next引用指向目标节点的后继节点。由于我们的Entry节点没有previous引用，因此我们要基于目标节点的前驱节点进行操作，即：

```
1 Copycurrent.next = current.next.next;
```

current代表我们要删除的节点的前驱节点。

还有一些简单的size()、isEmpty()等方法都很简单，这里就不再赘述。现在，我们自定义的MyHashMap基本可以使用了。

■ 扩容：

参考上方扩容

Arrays.sort 和 Collections.sort 实现原理和区别

Collection和Collections区别

java.util.Collection 是一个集合接口。它提供了对集合对象进行基本操作的通用接口方法。

java.util.Collections 是针对集合类的一个帮助类，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全等操作。 然后还有混排（Shuffling）、反转（Reverse）、替换所有的元素（fill）、拷贝（copy）、返

回Collections中最小元素（min）、返回Collections中最大元素（max）、返回指定源列表中最后一次出现指定目

标列表的起始位置（lastIndexOfSubList）、返回指定源列表中第一次出现指定目标列表的起始位置

（IndexOfSubList）、根据指定的距离循环移动指定列表中的元素（Rotate）；

事实上Collections.sort方法底层就是调用的array.sort方法，

```
1 public static void sort(Object[] a) {
2     if (LegacyMergeSort.userRequested)
3         legacyMergeSort(a);
4     else
5         ComparableTimSort.sort(a, 0, a.length, null, 0, 0);
6 }
7 //void java.util.ComparableTimSort.sort()
8 static void sort(Object[] a, int lo, int hi, Object[] work, int workBase, int
workLen){
9     assert a != null && lo >= 0 && lo <= hi && hi <= a.length;
10    int nRemaining = hi - lo;
11    if (nRemaining < 2)
12        return; // Arrays of size 0 and 1 are always sorted
13    // If array is small, do a "mini-TimSort" with no merges
14    if (nRemaining < MIN_MERGE) {
15        int initRunLen = countRunAndMakeAscending(a, lo, hi);
16        binarySort(a, lo, hi, lo + initRunLen);
17        return;
18    }
19 }
```

legacyMergeSort (a): 归并排序 ComparableTimSort.sort(): Timsort 排序

Timsort 排序是结合了合并排序（merge sort）和插入排序（insertion sort）而得出的排序算法

Timsort的核心过程

TimSort 算法为了减少对升序部分的回溯和对降序部分的性能倒退，将输入按其升序和降序特点进行了分

区。排序的输入的单位不是一个个单独的数字，而是一个个的块-分区。其中每一个分区叫一个run。针对这

些 run 序列，每次拿一个 run 出来按规则进行合并。每次合并会将两个 run合并成一个 run。合并的结果保

存到栈中。合并直到消耗掉所有的 run，这时将栈上剩余的 run合并到只剩一个 run 为止。这时这个仅剩的

run 便是排好序的结果。

综上所述过程，Timsort算法的过程包括

- (0) 如何数组长度小于某个值，直接用二分插入排序算法
- (1) 找到各个run，并入栈
- (2) 按规则合并run

HashSet是如何保证不重复的

向 HashSet 中 add ()元素时，判断元素是否存在的依据，不仅要比较hash值，同时还要结合 equals 方法比较。

HashSet 中的add ()方法会使用HashMap 的add ()方法。以下是HashSet 部分源码：

```
1 private static final Object PRESENT = new Object();
2 private transient HashMap<E,Object> map;
3 public HashSet() {
4     map = new HashMap<>();
5 }
6 public boolean add(E e) {
7     return map.put(e, PRESENT)!=null;
8 }
```

HashMap 的key 是唯一的，由上面的代码可以看出HashSet 添加进去的值就是作为HashMap 的key。所以不会重复（HashMap 比较key是否相等是先比较hashcode 在比较equals）。

list和数组的区别

1. 数组存储时是连续的，List是不连续的存储结构
2. 数组必须要在初始化时分配固定的大小，List无须指定初始大小
3. Array数组可以包含基本类型和对象类型，ArrayList却只能包含对象类型。但是需要注意的是：Array数组在存放的时候一定是同种类型的元素。ArrayList就不一定了，因为ArrayList可以存储Object。

hashmap为什么不弄成数组

一个是很容易出现扩容；还有一个是删除和增加节点很耗时（因为数组+链表就是为了结合两个的优点）。

map和list使用要注意什么问题

1. 不是所有map都允许key为null

集合类	Key	Value	Super	说明
Hashtable	不允许为 null	不允许为 null	Dictionary	线程安全
ConcurrentHashMap	不允许为 null	不允许为 null	AbstractMap	锁分段技术 (JDK8:CAS)
TreeMap	不允许为 null	允许为 null	AbstractMap	线程不安全
HashMap	允许为 null	允许为 null	AbstractMap	线程不安全

读取配置文件的时候，有可能key为null

2. ArrayList遍历删除失败：

```

1 public static void remove(ArrayList<String> list)
2 {
3     for(int i=0;i<list.size();i++)
4     {
5         Strings=list.get(i);
6         if(s.equals("b")){
7             list.remove(s); //第二个“b”的字符串没有删掉
8         }
9     }
10 }
11 原因: remove方法的底层删除后会移动, 导致遍历不到第二个“b”
12 解决方案: 使用倒序遍历

```

```

1 public static void remove(ArrayList<String> list)
2 {
3     for(Strings:list){
4         if(s.equals("b"))
5         {
6             list.remove(s); //会报出著名的并发修改异常
7             java.util.ConcurrentModificationException。
8         }
9     }
10 }
11 原因: foreach写法是对实际的Iterable、hasNext、next方法的简写, remove方法底层会对
    modCount变量的值加一, 而迭代器父类会对modCount检查, 导致修改异常
12 解决方案: 改为使用Iterator的remove即可。

```

3. 错用 ConcurrentHashMap 导致线程不安全

```

1 ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
2 map.put("key", 1);
3 ExecutorService executorService = Executors.newFixedThreadPool(100);
4 for (int i = 0; i < 1000; i++) {
5     executorService.execute(() -> {
6         int value = map.get("key") + 1; //第一步, 值+1
7         map.put("key", value); //第二步, 重新设值
8     });
9 }
10 TimeUnit.SECONDS.sleep(51);
11 System.out.println("-----" + map.get("key") + "-----");
12 executorService.shutdown();

```

如上面代码, 我们原本期望输出 1001, 但是运行几次, 得到结果都是小于 1001。ConcurrentHashMap 只能保证这两步单的操作是个原子操作, 线程安全。但是并不能保证两个组合逻辑线程安全, 很有可能 A 线程刚通过 get 方法取到值, 还未来得及加 1, 线程发生了切换, B 线程也进来取到同样的值。

4. List转Map的坑

https://blog.csdn.net/qq_38066812/article/details/109443362

5. key为自定义对象需要重写hashCode 与 equals 方法

<https://blog.csdn.net/j3T9Z7H/article/details/106152226>

hashmap为什么线程不安全?

- 1.7版本的头插法扩容时容易产生死循环
- 1.8版本会出现数据覆盖:

这是 jdk1.8 中 HashMap 中 put 操作的主函数, 注意第 6 行代码, 如果没有 hash 碰撞则会直接插入元素。

如果线程 A 和线程 B 同时进行 put 操作，刚好这两条不同的数据 hash 值一样，并且该位置数据为 null，所以这线程 A、B 都会进入第 6 行代码中。

假设一种情况，线程 A 进入后还未进行数据插入时挂起，而线程 B 正常执行，从而正常插入数据，

然后线程 A 获取 CPU 时间片，此时线程 A 不用再进行 hash 判断了，问题出现：线程 A 会把线程 B 插入的数据给覆盖，发生线程不安全。

HashSet 是如何保证不重复的？

向 HashSet 中 add ()元素时，判断元素是否存在的依据，不仅要比对hash值，同时还要结合 equals 方法比较。HashSet 中的 add ()方法会使用 HashMap 的 add ()方法。以下是 HashSet 部分源码：

```
1 private static final Object PRESENT = new Object();
2 private transient HashMap<E,Object> map;
3 public HashSet() {
4     map = new HashMap<>();
5 }
6 public boolean add(E e) {
7     return map.put(e, PRESENT)!=null;
8 }
```

HashMap 的 key 是唯一的，由上面的代码可以看出 HashSet 添加进去的值就是作为 HashMap 的key。所以不会重复（HashMap 比较key是否相等是先比较 hashCode 在比较 equals）。

数组在内存中如何分配？

对于 Java 数组的初始化，有以下两种方式，这也是面试中经常考到的经典题目：

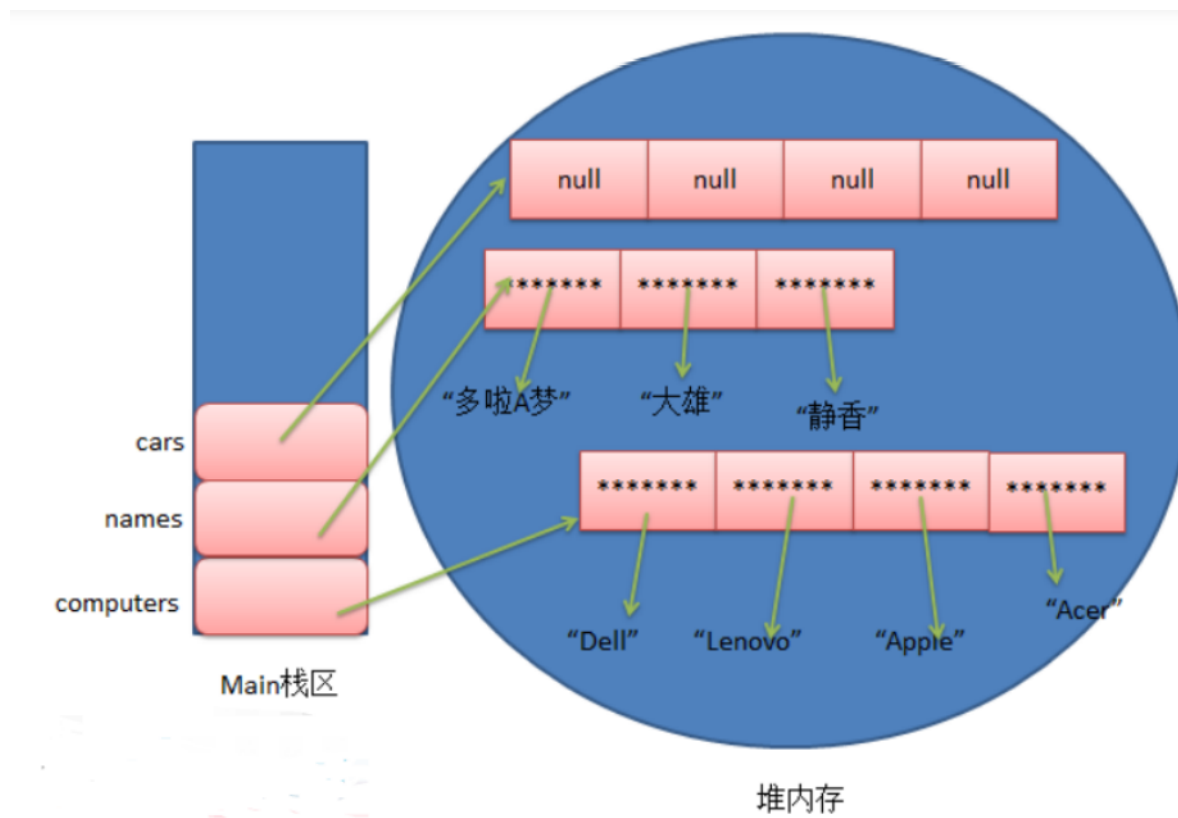
静态初始化：初始化时由程序员显式指定每个数组元素的初始值，由系统决定数组长度，如：

```
1 //只是指定初始值，并没有指定数组的长度，但是系统为自动决定该数组的长度为4
2 String[] computers = {"Dell", "Lenovo", "Apple", "Acer"}; //①
3 //只是指定初始值，并没有指定数组的长度，但是系统为自动决定该数组的长度为3
4 String[] names = new String[]{"多啦A梦", "大雄", "静香"}; //②
```

动态初始化：初始化时由程序员显示的指定数组的长度，由系统为数据每个元素分配初始值，如：

```
1 //只是指定了数组的长度，并没有显示的为数组指定初始值，但是系统会默认给数组数组元素分配初始值为null
2 String[] cars = new String[4];
```

因为 Java 数组变量是引用类型的变量，所以上述几行初始化语句执行后，三个数组在内存中的分配情况如下图所示：



由上图可知，静态初始化方式，程序员虽然没有指定数组长度，但是系统已经自动帮我们给分配了，而动态初始化方式，程序员虽然没有显示的指定初始化值，但是因为 Java 数组是引用类型的变量，所以系统也为每个元素分配 初始化值 null，当然不同类型的初始化值也是不一样的，假设是基本类型int类型，那么为系统分配的初始化值也是对应的默认值0。

红黑树原理？

红黑树是一个二叉搜索树。在每个节点增加了一个存储位记录节点的颜色，可以是 RED，也可以是 BLACK，通过任意一条从根到叶子简单路径上颜色的约束，红黑树保证最长路径不超过最短路径的两倍，加以平衡。

性质如下：

- 每个节点颜色不是黑色就是红色
- 根节点的颜色是黑色的
- 如果一个节点是红色，那么他的两个子节点就是黑色的，没有持续的红节点
- 对于每个节点，从该节点到其后代叶节点的简单路径上，均包含相同数目的黑色节点