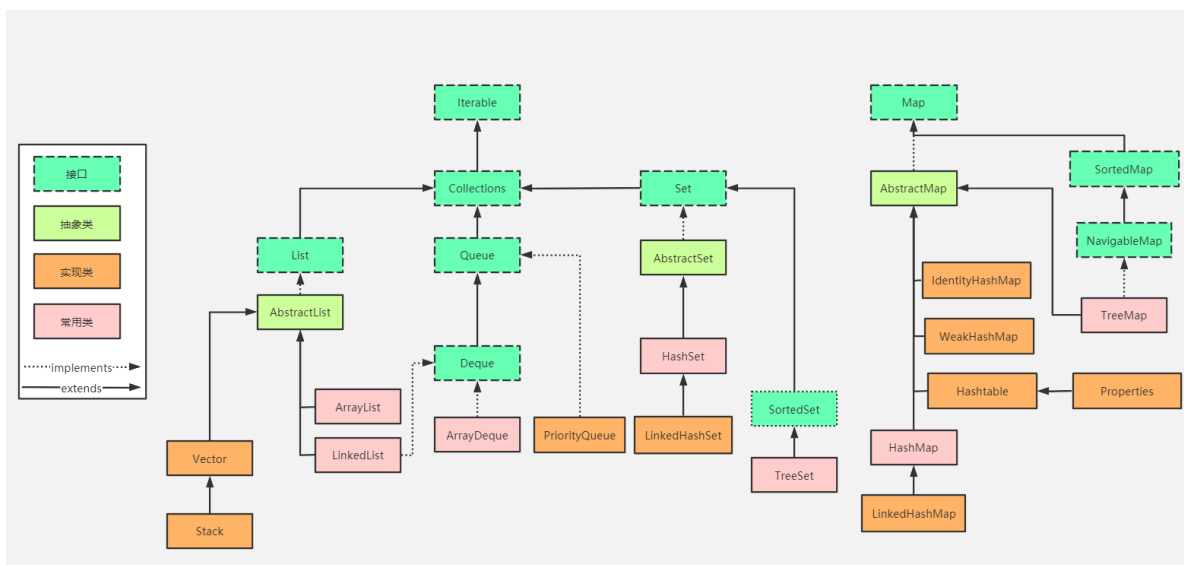


第12章 Java集合框架



#一、集合大纲

#1、集合的继承结构



其实有了我们的超级数组的实战之后，我们学习集合将会很容易，java的集合框架就是给我们提供了一套更加方便的存储数据的类而已。

集合的目的是方便的存储和操作数据，其实说到底无非就是 **增删改查**。

#2、常用接口介绍

List（列表）线性表：

- 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。

Set（表）也是线性表

- 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。

Map（映射）

- Map（映射）用于保存具有映射关系的数据，Map里保存着两组数据：key和value，它们都可以是任何引用类型的数据，但key不能重复。所以通过指定的key就可以取出对应的value。
- List，Set都是继承自Collection接口，Map则不是

#二、集合的增删改查

思考，其实我们能从几个接口源码中看明白集合到底是怎么进行增删改查的。

```
1 public interface List<E> extends Collection<E> {
2
3     int size();
4
5     boolean isEmpty();
6
7     boolean contains(Object o);
8
9     Iterator<E> iterator();
10
11     <T> T[] toArray(T[] a);
12
13     boolean add(E e);
14
15     boolean remove(Object o);
16
17     boolean containsAll(Collection<?> c);
18
19     boolean addAll(Collection<? extends E> c);
20
21     boolean addAll(int index, Collection<? extends E> c);
22
23     boolean removeAll(Collection<?> c);
24
25     void clear();
26
27     E get(int index);
28
29     E set(int index, E element);
30
31     void add(int index, E element);
32
33     E remove(int index);
34
35 }
```

```
1 // 你会发现set天然没有修改的方法
2 public interface Set<E> extends Collection<E> {
3
4     int size();
5
6     boolean isEmpty();
7
8     boolean contains(Object o);
9
10    Iterator<E> iterator();
11
12    Object[] toArray();
13
14    <T> T[] toArray(T[] a);
15
16    boolean add(E e);
17
18    boolean remove(Object o);
19
20    boolean containsAll(Collection<?> c);
```

```

21
22     boolean addAll(Collection<? extends E> c);
23
24     boolean removeAll(Collection<?> c);
25
26     void clear();
27 }

```

```

1  public interface Map<K,V> {
2
3      int size();
4
5      boolean isEmpty();
6
7      boolean containsKey(Object key);
8
9      boolean containsValue(Object value);
10
11     V get(Object key);
12
13     V put(K key, V value);
14
15     V remove(Object key);
16
17     void putAll(Map<? extends K, ? extends V> m);
18
19     void clear();
20
21     Set<K> keySet();
22
23     Collection<V> values();
24
25     Set<Map.Entry<K, V>> entrySet();
26 }

```

#三、源码分析

本节知识比较难，大家量力而行，能学会多少是多少，特别是hashmap。本节的内容可以现在学习，也可以以后学习。

第一次看源码，说说注意的问题：

- 一定要跟着我的节奏看。
- 一定要专注，需要上下文的结合阅读。
- 不要太扣细节，把源代码的整体思路阅读下来就行了。
- 有时间多读几次，慢慢脱离我的视频。

#1、Arraylist

(1) 成员变量

```
1 // 默认的空数组
2 private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
3
4 // 实际存数据的数组
5 transient Object[] elementData;
6
7 // 默认容量
8 private static final int DEFAULT_CAPACITY = 10;
```

(2) 构造器

```
1 // 默认使用空数组
2 public ArrayList() {
3     this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
4 }
5
6 public ArrayList(int initialCapacity) {
7     if (initialCapacity > 0) {
8         this.elementData = new Object[initialCapacity];
9     } else if (initialCapacity == 0) {
10        this.elementData = EMPTY_ELEMENTDATA;
11    } else {
12        throw new IllegalArgumentException("Illegal Capacity: "+
13                                         initialCapacity);
14    }
15 }
16
17 // ArrayList还可以直接传入一个集合
18 public ArrayList(Collection<? extends E> c) {
19     elementData = c.toArray();
20     if ((size = elementData.length) != 0) {
21         // 集合中有数据就拷贝数据
22         if (elementData.getClass() != Object[].class)
23             elementData = Arrays.copyOf(elementData, size, Object[].class);
24     } else {
25         // replace with empty array.
26         this.elementData = EMPTY_ELEMENTDATA;
27     }
28 }
```

(3) add方法

```
1 public boolean add(E e) {
2     // 确保能不能放进去
3     ensureCapacityInternal(size + 1);
4     elementData[size++] = e;
5     return true;
6 }
```

```
1 private void ensureCapacityInternal(int minCapacity) {
2     ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
3 }
4
5 // 根据数组长度和传入的容量值计算容量
6 private static int calculateCapacity(Object[] elementData, int minCapacity) {
```

```

7      // 初始化时 就是空啊，他会选择10当他的容量值。
8      if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
9          return Math.max(DEFAULT_CAPACITY, minCapacity);
10     }
11     return minCapacity;
12 }
13
14 private void ensureExplicitCapacity(int minCapacity) {
15     // 记录了集合被修改的次数
16     modCount++;
17     if (minCapacity - elementData.length > 0)
18         grow(minCapacity);
19 }

```

```

1      // 数组扩容的地方
2      private void grow(int minCapacity) {
3          // 获取旧的容量
4          int oldCapacity = elementData.length;
5          // 很明显扩容了1.5倍
6          int newCapacity = oldCapacity + (oldCapacity >> 1);
7          if (newCapacity - minCapacity < 0)
8              newCapacity = minCapacity;
9          if (newCapacity - MAX_ARRAY_SIZE > 0)
10             newCapacity = hugeCapacity(minCapacity);
11         // 扩容后拷贝数据
12         elementData = Arrays.copyOf(elementData, newCapacity);
13     }

```

(4) 查找和删除

```

1      public E get(int index) {
2          rangeCheck(index);
3          return elementData(index);
4      }
5
6      E elementData(int index) {
7          return (E) elementData[index];
8      }

```

```

1      public E remove(int index) {
2          rangeCheck(index);
3
4          modCount++;
5          E oldValue = elementData(index);
6          int numMoved = size - index - 1;
7          if (numMoved > 0)
8              // System.arraycopy
9              // Object src : 原数组
10             // int srcPos : 从元数据的起始位置开始
11             // Object dest : 目标数组
12             // int destPos : 目标数组的开始起始位置
13             // int length : 要copy的数组的长度
14             System.arraycopy(elementData, index+1, elementData, index,
15                             numMoved);
16         elementData[--size] = null;
17         return oldValue;
18     }

```

源码里能看到的信息：

- 1、arraylist是基于数组实现的。
- 2、默认容量是10，每次扩容是1.5倍的扩容（ $\text{oldCapacity} + (\text{oldCapacity} \gg 1)$ ）。

#2、linkedlist

(1) 成员变量

```
1    transient int size = 0;
2
3    // 保存头结点
4    transient Node<E> first;
5
6    // 保存尾节点
7    transient Node<E> last;
8
9    // 节点的定义
10   private static class Node<E> {
11       E item;
12       Node<E> next;
13       Node<E> prev;
14
15       Node(Node<E> prev, E element, Node<E> next) {
16           this.item = element;
17           this.next = next;
18           this.prev = prev;
19       }
20   }
```

很明显，这里能够看出linkedlist是基于双向链表实现的。

(2) 构造器

```
1    /**
2     * Constructs an empty list.
3     */
4    public LinkedList() {
5    }
6
7    /**
8     * Constructs a list containing the elements of the specified
9     * collection, in the order they are returned by the collection's
10     * iterator.
11     *
12     * @param c the collection whose elements are to be placed into this list
13     * @throws NullPointerException if the specified collection is null
14     */
15   public LinkedList(Collection<? extends E> c) {
16       this();
17       addAll(c);
18   }
```

(3) 添加的方法

```
1  // 头上添加
2  private void linkFirst(E e) {
3      final Node<E> f = first;
4      final Node<E> newNode = new Node<>(null, e, f);
5      first = newNode;
6      if (f == null)
7          last = newNode;
8      else
9          f.prev = newNode;
10     size++;
11     modCount++;
12 }
13
14 // 尾巴上添加
15 void linkLast(E e) {
16     final Node<E> l = last;
17     final Node<E> newNode = new Node<>(l, e, null);
18     last = newNode;
19     if (l == null)
20         first = newNode;
21     else
22         l.next = newNode;
23     size++;
24     modCount++;
25 }
26
27 // 在某个元素之前添加
28 void linkBefore(E e, Node<E> succ) {
29     // assert succ != null;
30     final Node<E> pred = succ.prev;
31     final Node<E> newNode = new Node<>(pred, e, succ);
32     succ.prev = newNode;
33     if (pred == null)
34         first = newNode;
35     else
36         pred.next = newNode;
37     size++;
38     modCount++;
39 }
40
41 // 断开头部
42 private E unlinkFirst(Node<E> f) {
43     // assert f == first && f != null;
44     final E element = f.item;
45     final Node<E> next = f.next;
46     f.item = null;
47     f.next = null; // help GC
48     first = next;
49     if (next == null)
50         last = null;
51     else
52         next.prev = null;
53     size--;
54     modCount++;
55     return element;
}
```

```

56     }
57
58     // 断开尾巴
59     private E unlinkLast(Node<E> l) {
60         // assert l == last && l != null;
61         final E element = l.item;
62         final Node<E> prev = l.prev;
63         l.item = null;
64         l.prev = null; // help GC
65         last = prev;
66         if (prev == null)
67             first = null;
68         else
69             prev.next = null;
70         size--;
71         modCount++;
72         return element;
73     }
74
75     // 断开某一个节点
76     E unlink(Node<E> x) {
77         // assert x != null;
78         final E element = x.item;
79         final Node<E> next = x.next;
80         final Node<E> prev = x.prev;
81
82         if (prev == null) {
83             first = next;
84         } else {
85             prev.next = next;
86             x.prev = null;
87         }
88
89         if (next == null) {
90             last = prev;
91         } else {
92             next.prev = prev;
93             x.next = null;
94         }
95
96         x.item = null;
97         size--;
98         modCount++;
99         return element;
100     }
101
102     // 这两个方法中list接口中没有，是LinkedList类中特有的。
103     public void addFirst(E e) {
104         linkFirst(e);
105     }
106     public void addLast(E e) {
107         linkLast(e);
108     }
109
110     // 默认的添加是给尾巴添加
111     public boolean add(E e) {
112         linkLast(e);
113         return true;

```


(4) 查找和删除

```

1  public E get(int index) {
2      checkElementIndex(index);
3      return node(index).item;
4  }
5  // 找到第几个node
6  Node<E> node(int index) {
7      // 看人家的检索，小于一半就从头检索，否则从尾巴检索
8      if (index < (size >> 1)) {
9          Node<E> x = first;
10         for (int i = 0; i < index; i++)
11             x = x.next;
12         return x;
13     } else {
14         Node<E> x = last;
15         for (int i = size - 1; i > index; i--)
16             x = x.prev;
17         return x;
18     }
19 }
20
21 // 获取头结点
22 public E getFirst() {
23     final Node<E> f = first;
24     if (f == null)
25         throw new NoSuchElementException();
26     return f.item;
27 }
28
29 // 获取尾节点
30 public E getLast() {
31     final Node<E> l = last;
32     if (l == null)
33         throw new NoSuchElementException();
34     return l.item;
35 }
36
37 // 删除默认删除头
38 public E remove() {
39     return removeFirst();
40 }
41 // 根据index删除
42 public E remove(int index) {
43     checkElementIndex(index);
44     return unlink(node(index));
45 }
46
47 // 这两个方法中list接口中没有，是LinkedList类中特有的。
48 public E removeFirst() {
49     final Node<E> f = first;
50     if (f == null)
51         throw new NoSuchElementException();
52     return unlinkFirst(f);
53 }

```

```

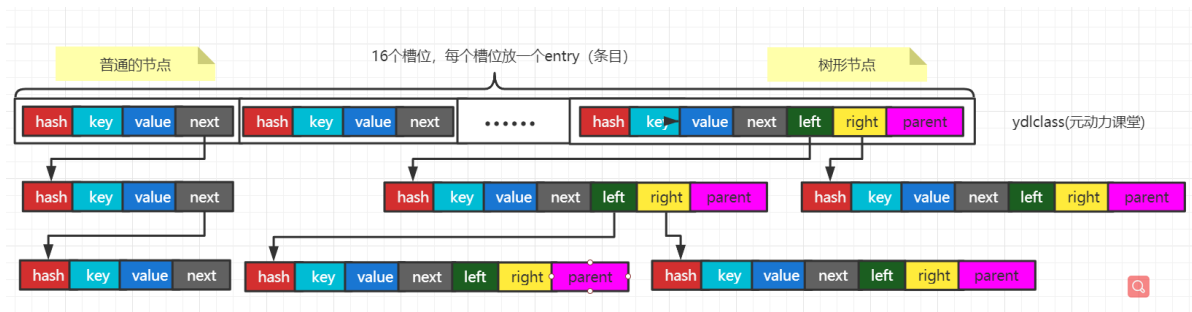
54
55 public E removeLast() {
56     final Node<E> l = last;
57     if (l == null)
58         throw new NoSuchElementException();
59     return unlinkLast(l);
60 }

```

#3、hashmap

(1) 初步了解

hashmap的实现是比较复杂的。



在读map源码之前，我们先看一张图，了解hashmap的存储结构：

简而言之是这样的（不太对，但是有个大概的了解）：

第一步：hashmap构造时（其实不是构造的时候）会创建一个长度为16数组，名字叫table，也叫hash表；

第二步：hashmap在插入数据的时候，首先根据key计算hashcode，然后根据hashcode选择一个槽位。

假设hashmap使用取余的方式计算。（事实上，hashmap不是）

1 我们都知道hashcode会返回一个int值，使用int值除以16取余就能得到一个0~15的数字，就能去定一个具体的槽位。

第三步：确定了具体的槽位之后，我们会封装一个node（节点），里边保存了hash，key，value等数据存入这个槽中。

第四步：当存入新的数据的时候，使用新的hash计算的槽位发现已经有了数据，这个现象叫做hash碰撞，会以链表的形式存储。

第五步：当链表的个数到达了8个，链表开始树化，变成一个红黑树。

通过这五个步骤，大家先有一个基本的了解，更多的细节我们下来看源码。

(2) 成员变量的分析

```

1 // 默认容量
2 static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
3
4 // 最大容量
5 static final int MAXIMUM_CAPACITY = 1 << 30;
6
7 // 默认的加载因子
8 static final float DEFAULT_LOAD_FACTOR = 0.75f;
9
10 // 默认的一个树化的一个阈值（THRESHOLD 阈值）

```

```

11 static final int TREEIFY_THRESHOLD = 8;
12
13 // 非树化的一个阈值
14 static final int UNTREEIFY_THRESHOLD = 6;
15
16 // 树化的最小容量，能看到一些信息，树化除了链表长度，对容量也有要求
17 static final int MIN_TREEIFY_CAPACITY = 64;
18
19 // 存储数据的hash表，就是一个数组
20 transient Node<K,V>[] table;
21
22 // 真实的负载因子
23 final float loadFactor;

```

(3) 构造

```

1 // 只是将默认的负载因子传递给了loadFactor
2 public HashMap() {
3     this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
4 }
5
6 // 有传入的初始化容量
7 public HashMap(int initialCapacity) {
8     this(initialCapacity, DEFAULT_LOAD_FACTOR);
9 }
10
11 // 有传入的初始化容量和负载因子
12 public HashMap(int initialCapacity, float loadFactor) {
13     if (initialCapacity < 0)
14         throw new IllegalArgumentException("Illegal initial capacity: " +
15                                         initialCapacity);
16     if (initialCapacity > MAXIMUM_CAPACITY)
17         initialCapacity = MAXIMUM_CAPACITY;
18     if (loadFactor <= 0 || Float.isNaN(loadFactor))
19         throw new IllegalArgumentException("Illegal load factor: " +
20                                         loadFactor);
21     // 计算新的负载因子和容量
22     this.loadFactor = loadFactor;
23     this.threshold = tableSizeFor(initialCapacity);
24 }
25
26
27
28 /**
29  * 返回一个值，大于等于传入的数字的一个2的次幂的数字，你传入15返回16，传入7返回8、
30  * 保证了容量是2的次幂。为了后来计算hash槽做准备
31  */
32 static final int tableSizeFor(int cap) {
33     int n = cap - 1;
34     n |= n >>> 1;
35     n |= n >>> 2;
36     n |= n >>> 4;
37     n |= n >>> 8;
38     n |= n >>> 16;
39     return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
40 }
41

```

```

42
43 static final int tableSizeFor(int cap) {
44     // 00010000 11101001 10001001 10000101 -- > 00010000 11101001 10001001
10000100 283,740,549
45     // 看完了
46     int n = cap - 1;
47     // 00010000 11101001 10001001 10000101 n
48     // 00001000 01110100 11000100 11000010 右移1位, 保障2位是1
49     // 00011000 11101101 11001101 11000111 n
50     n |= n >>> 1;
51     // 00011000 11101101 11001101 11000111 n
52     // 00000110 00111011 01110011 01110001 右移2位, 保障4位是1
53     // 00011110 11111111 11111111 11110111 n
54     n |= n >>> 2;
55     // 00011110 11111111 11111111 11110111 n
56     // 00000001 11101111 11111111 11111111 右移4位, 保障8位是1
57     // 00011111 11111111 11111111 11111111 n
58     n |= n >>> 4;
59     // 00011111 11111111 11111111 11111111 n
60     // 00000000 00011111 11111111 11111111 右移8位, 保障16位是1
61     // 00011111 11111111 11111111 11111111 n
62     n |= n >>> 8;
63     // 00011111 11111111 11111111 11111111 n
64     // 00000000 00000000 00011111 11111111 右移8位, 保障32位是1
65     // 00011111 11111111 11111111 11111111 n
66     n |= n >>> 16;
67     return n + 1;
68 }

```

在构造的整个过程当中, 并没有初始化hash表table。

(4) put方法

这个方法是核心, 也是我们所需要研究的。很多的问题都是在这个方法当中。

```

1 public V put(K key, V value) {
2     return putVal(hash(key), key, value, false, true);
3 }
4
5 // 第一个关键点: key == null, 说明我们的hashmap支持key为null
6 // 第二个关键点: (h = key.hashCode()) ^ (h >>> 16), 这一点学完, 学完putVal方法再看
7 // h          1010 0010 0001 1001 0010 1100 1010 1001
8 // h >>> 16    0000 0000 0000 0000 1010 0010 0001 1001 ( 0010 1100 1010 1001)
9 // 异或运算    1010 0010 0001 1001 1000 1110 1011 0000
10 // 目的: 让高16位和低16位同时参与计算, 将来计算hash槽时更加均匀
11 static final int hash(Object key) {
12     int h;
13     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
14 }

```

```

1 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
2               boolean evict) {
3     // tab很明显就是hash表
4     Node<K,V>[] tab;
5     // 就是个引用 (指针),
6     Node<K,V> p;
7     // 先不要管,n代表hash表的长度 (tab)
8     int n, i;

```

```

9      // (tab = table) == null 将hash表赋值给tab, 并且判断是不是null
10     // 或者长度等于0, 我就要扩容, 构造没有初始化
11     if ((tab = table) == null || (n = tab.length) == 0){
12         // 那就扩容, 还兼任初始化的责任 (16)
13         n = (tab = resize()).length;
14     }
15     // p == null, 只不过有个给p赋值的过程
16     // p = tab[i = (n - 1) & hash]
17     // 其实 i是计算的槽位, 你的数据往哪个格子里放
18     // (n - 1) & hash 这是真实的计算过程, n确定是一个2的n次幂(100...), hash是一个int值
19     // (n-1)      0000 0000 0000 0000 0000 0000 0000 1111
20     // (hash)     0010 0010 0010 0010 0000 0110 0000 1011
21     // (result)   0000 0000 0000 0000 0000 0000 0000 1011
22     // 与运算之后的结果就是0~15, 正好计算了一个槽位
23     // 第一个思考的问题: 为什么容量必须是2的次幂? 0...01...1
24     // 第二个思考的问题: 为什么使用位移运算而不适用余运算? 效率
25     // 找到槽位, 并且槽位没有数据, 就直接newnode放进去
26     if ((p = tab[i = (n - 1) & hash]) == null){
27         // 创建了一个node
28         tab[i] = newNode(hash, key, value, null);
29     } else {
30         // 只要进入else, 说明这个槽位有数据了, 就要搞链表了
31         //
32         Node<K,V> e;
33         // 键, 泛型, 当前插入数据的键
34         K k;
35         // 根据p = tab[i = (n - 1) & hash], 知道p是放在槽位上的node
36         // p.hash == hash 说明发生了hash碰撞
37         // (k = p.key) == key || (key != null && key.equals(k)) 判断的是key重复了
38         if (p.hash == hash &&
39             ((k = p.key) == key || (key != null && key.equals(k)))){
40             // 覆盖
41             e = p;
42
43             // 判断是不是树形节点
44         } else if (p instanceof TreeNode){
45             e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
46
47             // 否则就是链表的方式
48         } else {
49             for (int binCount = 0; ; ++binCount) {
50                 if ((e = p.next) == null) {
51                     // 这不就是链表吗? 很明显这是尾插
52                     p.next = newNode(hash, key, value, null);
53                     if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
54                         // 树化
55                         treeifyBin(tab, hash);
56                     break;
57                 }
58                 // 判断链表中有没有key一样的, 覆盖
59                 if (e.hash == hash &&
60                     ((k = e.key) == key || (key != null && key.equals(k))))
61                     break;
62                 p = e;
63             }
64         }
65         if (e != null) { // existing mapping for key
66             V oldValue = e.value;

```

```

67         if (!onlyIfAbsent || oldValue == null)
68             e.value = value;
69         afterNodeAccess(e);
70         return oldValue;
71     }
72 }
73 ++modCount;
74 if (++size > threshold)
75     resize();
76 afterNodeInsertion(evict);
77 return null;
78 }

```

(5) 扩容的方法

```

1  final Node<K,V>[] resize() {
2      Node<K,V>[] oldTab = table;
3      // 旧的容量
4      int oldCap = (oldTab == null) ? 0 : oldTab.length;
5      // 旧的阈值
6      int oldThr = threshold;
7      int newCap, newThr = 0;
8      if (oldCap > 0) {
9          // 容量大于最大值就取最大值
10         if (oldCap >= MAXIMUM_CAPACITY) {
11             threshold = Integer.MAX_VALUE;
12             return oldTab;
13         }
14         // 这里体现了扩容的大小
15         // newCap = oldCap << 1 相当于2倍
16         } else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
17             oldCap >= DEFAULT_INITIAL_CAPACITY)
18             // 阈值月扩容二倍
19             newThr = oldThr << 1; // double threshold
20     // 旧的阈值大于零
21 } else if (oldThr > 0){ // initial capacity was placed in threshold
22     // 旧的阈值 = 新的容量
23     newCap = oldThr;
24
25     // 否则就是初始化, 因为 == 0
26 } else { // zero initial threshold signifies using defaults
27     // 否则新的容量就是默认的容量
28     newCap = DEFAULT_INITIAL_CAPACITY;
29     // 新的阈值就是 容量*负载因子
30     newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
31 }
32
33 // 计算新的阈值, 要么是相乘, 要么Integer最大值
34 if (newThr == 0) {
35     float ft = (float)newCap * loadFactor;
36     newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
37         (int)ft : Integer.MAX_VALUE);
38 }
39
40 // 将计算好的阈值赋值给 threshold
41 threshold = newThr;
42 @SuppressWarnings({"rawtypes","unchecked"})

```

```

43
44 // 根据新的容量创建了新的hash表
45 Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
46 table = newTab;
47 // 以下是重新拷贝的过程
48 if (oldTab != null) {
49     for (int j = 0; j < oldCap; ++j) {
50         Node<K,V> e;
51         if ((e = oldTab[j]) != null) {
52             oldTab[j] = null;
53             if (e.next == null)
54                 newTab[e.hash & (newCap - 1)] = e;
55             else if (e instanceof TreeNode)
56                 ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
57             else { // preserve order
58                 Node<K,V> loHead = null, loTail = null;
59                 Node<K,V> hiHead = null, hiTail = null;
60                 Node<K,V> next;
61                 do {
62                     next = e.next;
63                     if ((e.hash & oldCap) == 0) {
64                         if (loTail == null)
65                             loHead = e;
66                         else
67                             loTail.next = e;
68                         loTail = e;
69                     }
70                     else {
71                         if (hiTail == null)
72                             hiHead = e;
73                         else
74                             hiTail.next = e;
75                         hiTail = e;
76                     }
77                 } while ((e = next) != null);
78                 if (loTail != null) {
79                     loTail.next = null;
80                     newTab[j] = loHead;
81                 }
82                 if (hiTail != null) {
83                     hiTail.next = null;
84                     newTab[j + oldCap] = hiHead;
85                 }
86             }
87         }
88     }
89 }
90 return newTab;
91 }

```

(6) 树化的部分代码

```

1 /**
2  * Replaces all linked nodes in bin at index for given hash unless
3  * table is too small, in which case resizes instead.
4  */
5 final void treeifyBin(Node<K,V>[] tab, int hash) {

```

```

6     int n, index; Node<K,V> e;
7     if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
8         // 优先扩容
9         resize();
10    else if ((e = tab[index = (n - 1) & hash]) != null) {
11        TreeNode<K,V> hd = null, tl = null;
12        do {
13            TreeNode<K,V> p = replacementTreeNode(e, null);
14            if (tl == null)
15                hd = p;
16            else {
17                p.prev = tl;
18                tl.next = p;
19            }
20            tl = p;
21        } while ((e = e.next) != null);
22        if ((tab[index] = hd) != null)
23            hd.treeify(tab);
24    }
25 }

```

为什么选择树化的长度是8，泊松分布

```

1    * 0:    0.60653066
2    * 1:    0.30326533
3    * 2:    0.07581633
4    * 3:    0.01263606
5    * 4:    0.00157952
6    * 5:    0.00015795
7    * 6:    0.00001316
8    * 7:    0.00000094
9    * 8:    0.00000006
10   * more: less than 1 in ten million

```

普通节点

```

1    static class Node<K,V> implements Map.Entry<K,V> {
2        final int hash;
3        final K key;
4        V value;
5        Node<K,V> next;
6
7        Node(int hash, K key, V value, Node<K,V> next) {
8            this.hash = hash;
9            this.key = key;
10           this.value = value;
11           this.next = next;
12        }
13    }

```

树形节点

```

1    static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
2        TreeNode<K,V> parent; // red-black tree links
3        TreeNode<K,V> left;
4        TreeNode<K,V> right;
5        TreeNode<K,V> prev; // needed to unlink next upon deletion
6        boolean red;

```



```

7     }
8
9     static class Entry<K,V> extends HashMap.Node<K,V> {
10         Entry<K,V> before, after;
11         Entry(int hash, K key, V value, Node<K,V> next) {
12             super(hash, key, value, next);
13         }
14     }

```

思考题：

- 1、hashmap的key的hash怎么计算？
- 2、hashmap的hash表什么情况下会扩容？
- 3、hashmap中什么时候会树化？
- 4、为什么选择0.75为负载因子，8为树化阈值？

#4、hashset

只要理解了hashmap，hashset不攻自破。

- (1) 首先我们看到hashset内部维护了一个hashmap，其实说明了hashset的实现是基于hashmap的。

```

1     private transient HashMap<E, Object> map;

```

- (2) 我们看到hashset的构造器其实只是new了一个hashmap();

```

1     public HashSet() {
2         map = new HashMap<>();
3     }

```

- (3) 我们以添加为例

```

1     private static final Object PRESENT = new Object();
2     public boolean add(E e) {
3         return map.put(e, PRESENT)!=null;
4     }

```

#四、集合的遍历

#1、普通for循环

能够使用普通for循环的前提是必须可以通过下标获取数据，List天然满足这个特性。

```

1  public class ListTest {
2      public static void main(String[] args) {
3          public List<String> names;
4          names = new ArrayList<>();
5          names.add("lucy");
6          names.add("tom");
7          names.add("jerry");
8          for (int i = 0; i < names.size(); i++) {
9              System.out.println(names.get(i));
10         }
11     }
12 }

```

同理：

我们将 `names = new ArrayList<>();` 改为 `names = new LinkedList<>();` 也是可以的。

思考问题：

hashmap和hashset怎么进行遍历？它们没有下标啊。

这里就必须使用迭代器了。

#2、迭代器

(1) 迭代器介绍

迭代器其实是一种思想。

先看一下迭代器这个接口：

```

1  public interface Iterator<E> {
2      // 是不是有下一个
3      boolean hasNext();
4      // 拿到下一个
5      E next();
6      // 你可以继承重写这个方法，否则将抛出异常
7      default void remove() {
8          throw new UnsupportedOperationException("remove");
9      }
10 }

```

例如：



小丽拿了一篮子苹果，你想把小丽的苹果分给大家吃。

- 我：小丽，篮子里还有吗？
- 小丽：有呢。hasNext()
- 我：给我。
- 小丽：好呢。next()
- 小丽：哎，这个坏了，我扔了吧！remove()

其实小丽就是我们所说的迭代器。

(2) 迭代器的使用

还是上边的例子：

```
1  @Test
2  public void testIterator(){
3      Iterator<String> iterator = names.iterator();
4      // 每次都判断一下是不是有下一个，有的话，继续遍历
5      while (iterator.hasNext()){
6          // 获取下一个
7          String name = iterator.next();
8          System.out.println(name);
9      }
10 }
```

当然换成LinkedList也是可行的。

看看hashSet，居然也行

```
1  /**
2   * @author itnanls
3   * @date 2021/7/16
4   */
5  public class SetTest {
6
7      public Set<String> names;
8
9      @Before
10     public void add() {
11         names = new HashSet<>();
12         names.add("lucy");
13         names.add("tom");
14         names.add("jerry");
15     }
16
17     @Test
18     public void testIterator(){
19         Iterator<String> iterator = names.iterator();
20         // 每次都判断一下是不是有下一个，有的话，继续遍历
21         while (iterator.hasNext()){
22             // 获取下一个
23             String name = iterator.next();
24             System.out.println(name);
25         }
26     }
27 }
```

再看看hashmap，也是可以的

```
1 public class MapTest {
2
3     public Map<String,String> user;
4
5     @Before
6     public void add() {
7         user = new HashMap<>();
8         user.put("username","ydlclass");
9         user.put("password","ydl666888");
10    }
11
12    @Test
13    public void testIterator(){
14        // 拿到一个存有所有entry的set集合。
15        // entry就是一个个的节点node
16        Set<Map.Entry<String, String>> entries = user.entrySet();
17
18        Iterator<Map.Entry<String, String>> iterator = entries.iterator();
19        while (iterator.hasNext()){
20            Map.Entry<String, String> next = iterator.next();
21            System.out.println(next.getKey());
22            System.out.println(next.getValue());
23        }
24    }
25 }
```

也可以先获取一个key的set即可，再用迭代器进行遍历。

这种方式相当于遍历了两次，效率低。

```
1 @Test
2 public void testIterator2(){
3     // 获取一个含有所有key的set集合，去迭代
4     Set<String> keys = user.keySet();
5     Iterator<String> iterator = keys.iterator();
6
7     while (iterator.hasNext()){
8         String key = iterator.next();
9         System.out.println(key);
10        System.out.println(user.get(key));
11    }
12 }
```

千万别以为迭代器牛逼的不行，其实迭代器只是个接口，每个对象都要有对应的实现。

简单的看一下arraylist的实现

```
1 private class Itr implements Iterator<E> {
2
3     // 只要有标没到最后一个就行
4     public boolean hasNext() {
5         return cursor != size();
6     }
7
8     public E next() {
9         checkForComodification();
```

```

10         try {
11             // 大概率就是使用游标控制下一个的位置
12             int i = cursor;
13             // 其实就是返回了下一个
14             E next = get(i);
15             lastRet = i;
16             cursor = i + 1;
17             return next;
18         } catch (IndexOutOfBoundsException e) {
19             checkForComodification();
20             throw new NoSuchElementException();
21         }
22     }
23
24     public void remove() {
25         if (lastRet < 0)
26             throw new IllegalStateException();
27         checkForComodification();
28
29         try {
30             // 直接把当前的删除就行了
31             AbstractList.this.remove(lastRet);
32             if (lastRet < cursor)
33                 cursor--;
34             lastRet = -1;
35             expectedModCount = modCount;
36         } catch (IndexOutOfBoundsException e) {
37             throw new ConcurrentModificationException();
38         }
39     }
40
41     final void checkForComodification() {
42         if (modCount != expectedModCount)
43             throw new ConcurrentModificationException();
44     }
45 }

```

#3、增强for循环

Java提供了一种 **语法糖**（用起来甜甜的，很简单）去帮助我们遍历，叫增强for循环：

List、Set都可以使用这种方式进行遍历：

```

1  @Test
2  public void testEnhancedFor(){
3      for (String name : names){
4          System.out.println(name);
5      }
6  }

```

Map使用这样的写法：

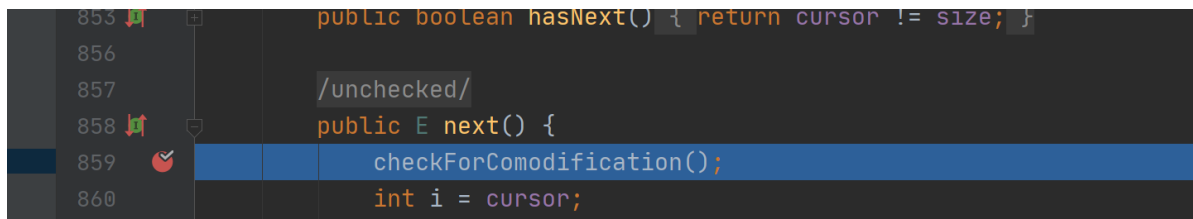
```

1  @Test
2  public void testEnhancedFor(){
3      for (Map.Entry<String,String> entry : user.entrySet()){
4          System.out.println(entry.getKey());
5          System.out.println(entry.getValue());
6      }
7  }

```

增强for循环其实也是使用了迭代器。我们可以在ArrayList中的迭代器中打一个断点，debug运行一下即可。

增强for循环只是一种语法糖，用起来甜甜的简单而已。



#4、迭代中删除元素

有同一个题目：我想把下边的集合中的lucy全部删除？

```

1  public void add() {
2      List<String> names = new ArrayList<>();
3      names.add("tom");
4      names.add("lucy");
5      names.add("lucy");
6      names.add("lucy");
7      names.add("jerry");
8  }

```

(1) for循环中删除

```

1  public void testDelByFor(){
2      for (int i = 0; i < names.size(); i++) {
3          if("lucy".equals(names.get(i))){
4              names.remove(i);
5          }
6      }
7      System.out.println(names);
8  }
9
10  结果:
11  [tom, lucy, jerry]

```

我们发现并没有删除干净，中间的lucy好像被遗忘了。

当第一次遍历发现lucy时就删除了。
删除了lucy之后，数组会进行调整，
后边的数据会向前移动。

tom	lucy	lucy	lucy	jerry
-----	------	------	------	-------



所以，以第一个删除的lucy相邻的lucy
就逃过了一劫。

tom	•	lucy	•	lucy	•	jerry
-----	---	------	---	------	---	-------



ydlclass

合适的解决方式有两种：

第一种：回调指针

```
1  for (int i = 0; i < names.size(); i++) {
2      if("lucy".equals(names.get(i))) {
3          names.remove(i);
4          // 回调指针:
5          i--;
6      }
7  }
8  System.out.println(names);
9
10
11  结果:
12  [tom, jerry]
```

第二种：逆序遍历

```
1  for (int i = names.size()-1; i > 0; i--) {
2      if("lucy".equals(names.get(i))) {
3          names.remove(i);
4      }
5  }
6  System.out.println(names);
7
8  结果:
9  [tom, jerry]
```

但是最好的删除方法是使用迭代器。

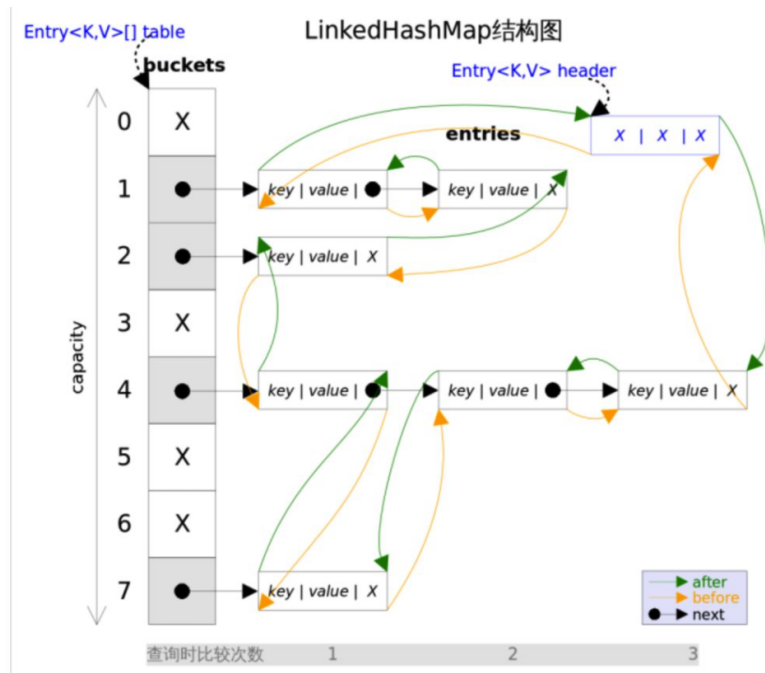
(2) 使用迭代器删除元素

```
1  public static void main(String[] args){
2      Iterator<String> iterator = names.iterator();
3      while (iterator.hasNext()){
4          // 记住next(), 只能调用一次, 因为每次调用都会选择下一个
5          String name = iterator.next();
6          if("lucy".equals(name)){
7              iterator.remove();
8          }
9      }
10     System.out.println(names);
11 }
```

#五、其他的集合类

#1、Linkedhashmap

Linkedhashmap在原来的基础上维护了一个双向链表，用来维护，插入的顺序。



```
1 public class LinkedHashMapTest {
2
3     public static void main(String[] args){
4         Map<String,String> map = new LinkedHashMap<>(16);
5         map.put("m", "abc");
6         map.put("a", "abc");
7         map.put("g", "bcd");
8         map.put("s", "cde");
9         map.put("z", "def");
10
11         Iterator<Map.Entry<String, String>> iterator = map.entrySet().iterator();
12         while (iterator.hasNext()){
13             Map.Entry<String, String> next = iterator.next();
14             System.out.println(next.getKey());
15             System.out.println(next.getValue());
16         }
17     }
18 }
19
20 结果：结果是有序的
21 m
22 abc
23 a
24 abc
25 g
26 bcd
27 s
28 cde
29 z
30 def
31
```



```

32 Map<String,String> map = new HashMap<>();
33 如果换成hashmap的结果是：很明显无序
34 a
35 abc
36 s
37 cde
38 g
39 bcd
40 z
41 def
42 m
43 abc

```

```

1 public LinkedHashMap() {
2     super();
3     accessOrder = false;
4 }
5
6 static class Entry<K,V> extends HashMap.Node<K,V> {
7     Entry<K,V> before, after;
8     Entry(int hash, K key, V value, Node<K,V> next) {
9         super(hash, key, value, next);
10    }
11 }
12
13 transient LinkedHashMap.Entry<K,V> head;
14
15 /**
16     * The tail (youngest) of the doubly linked list.
17     */
18 transient LinkedHashMap.Entry<K,V> tail;

```

如果accessOrder为true的话，则会把访问过的元素放在链表后面，放置顺序是访问的顺序 如果accessOrder为false的话，则按插入顺序来遍历

在Linkedhashmap中有几个顺序，一个是插入顺序，一个是访问顺序。

我们还可以使用linkedhashmap实现LRU算法的缓存，所谓LRU:Least Recently Used,最近最少使用,即当缓存了,会优先淘汰那些最近不常访问的数据.即冷数据优先淘汰.

```

1 public class LRU<K,V> extends LinkedHashMap<K,V> {
2
3     private int max_capacity;
4
5     public LRU(int initialCapacity,int max_capacity) {
6         super(initialCapacity, 0.75F, true);
7         this.max_capacity = max_capacity;
8     }
9
10    public LRU() {
11        super(16, 0.75F, true);
12        max_capacity = 8;
13    }
14
15    @Override
16    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
17        return size() > max_capacity;
18    }

```

```
19
20 }
```

#2、TreeMap

TreeMap底层实现是红黑树，所以天然支持排序。

```
1  public TreeMap() {
2      comparator = null;
3  }
4
5  public TreeMap(Comparator<? super K> comparator) {
6      this.comparator = comparator;
7  }
8
9
10
11 final int compare(Object k1, Object k2) {
12     return comparator==null ? ((Comparable<? super K>)k1).compareTo((K)k2)
13         : comparator.compare((K)k1, (K)k2);
14 }
15
16     会吧key1强转为Comparable
```

我们尝试把一个没有实现Comparable的类传入TreeMap中，发现会抛出异常。

```
1  Map<Dog,String> map = new TreeMap<>();
2
3  for (int i = 0; i < 100; i++) {
4      map.put(new Dog(), "a");
5  }
6
7
8  Exception in thread "main" java.lang.ClassCastException: bb.Dog cannot be cast to
java.lang.Comparable
9      at java.util.TreeMap.compare(TreeMap.java:1294)
10     at java.util.TreeMap.put(TreeMap.java:538)
11     at bb.Animal.main(Animal.java:14)
```

已经很明显了，这就是我们之前学习的策略设计模式啊，我们可以自定义比较器，实现key的有序性。

```
1  public static void main(String[] args) {
2      Map<Integer,String> map = new TreeMap<>(new Comparator<Integer>() {
3          @Override
4          public int compare(Integer o1, Integer o2) {
5              return o1 - o2;
6          }
7      });
8
9      for (int i = 0; i < 100; i++) {
10         map.put(i, "a");
11     }
12     System.out.println(map);
13
14 }
15
16 {0=a, 1=a, 2=a, 3=a, 4=a, 5=a, 6=a, 7=a, 8=a, 9=a, 10=a, 11=a, 12=a, 13=a, 14=a,
17 15=a,
```

我们修改一个比较器，立马就发生了变化。

```
1    return o2 - o1;
2    {99=a, 98=a, 97=a, 96=a, 95=a, 94=a, 93=a, 92=a, 91=a, 90=a, 89=a
```

我们当然可以让Dog类实现Comparable接口来使Dog作为key传入Map中。

#3、Collections

Collections是一个工具类，它给我们提供了一些常用的好用的操作集合的方法。

```
1    ArrayList<Integer> list = new ArrayList<>();
2    list.add(12);
3    list.add(4);
4    list.add(3);
5    list.add(5);
6    //将集合按照默认的规则排序,按照数字从小到大的顺序排序
7    Collections.sort(list);
8    System.out.println("list = " + list);
9    System.out.println("=====");
10   //将集合中的元素反转
11   Collections.reverse(list);
12   System.out.println("list = " + list);
13   //addAll方法可以往集合中添加元素，也可往集合中添加一个集合
14   Collections.addAll(list,9,20,56);
15   //打乱集合中的元素
16   Collections.shuffle(list);
17   System.out.println("list = " + list);
18
19   //Arrays.asList方法可以返回一个长度内容固定的List集合
20   List<String> list2 = Arrays.asList("tom", "kobe", "jordan",
21   "tracy", "westbook", "yaoming", "ace", "stephen");
22   //按照字符串首字符的升序排列
23   Collections.sort(list2);
```

小问题：

Arrays.asList(...)返回的是ArrayList吗？

#六、线程安全问题

#1、并发修改异常

使用增强for循环中删除元素会抛异常

```
1    public void testDelByEnhancedFor(){
2        for (String name : names){
3            if("lucy".equals(name)){
4                names.remove(name);
5            }
6        }
7    }
8
9
10   // 并发修改异常
```

```

11 java.util.ConcurrentModificationException
12     at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
13     at java.util.ArrayList$Itr.next(ArrayList.java:859)
14     at com.ydlclass.ListTest.testDelByEnhancedFor(ListTest.java:51)
15     ....不仅仅是上边的情况，下边的情况也会出现：

```

```

1 public void addDelByEnhancedFor(){
2     for (String name : names){
3         if("lucy".equals(name)){
4             names.add(name);
5         }
6     }
7 }

```

```

1 public void testDelByIterator(){
2     Iterator<String> iterator = names.iterator();
3     while (iterator.hasNext()){
4         // 记住next(),只能调用一次,因为每次调用都会选择下一个
5         String name = iterator.next();
6         if("lucy".equals(name)){
7             names.add("hello");
8         }
9     }
10    System.out.println(names);
11 }

```

产生的原因：

我们可以把普通的方法和迭代器的方法看成两个人，一个小丽，一个小红。

你用小丽迭代的时候，用小红的方法删除，或者用小红的方法迭代，用小丽的方法删除就会出错。

迭代器是依赖于集合而存在的，在判断成功后，集合的中新添加了元素，而迭代器却不知道，所以就报错了，这个错叫并发修改异常。

如何解决呢？

- 迭代器迭代元素，迭代器修改元素。
- 集合遍历元素，集合修改元素(普通for)。

#2、数据错误的问题

我们学了并发编程，知道当多个线程同时操作共享资源时会有线程安全问题。

```

1 public static void main(String[] args) throws InterruptedException {
2     final ArrayList<Integer> list = new ArrayList<>();
3     CountDownLatch countDownLatch = new CountDownLatch(200);
4     for (int i = 0; i < 200; i++) {
5         new Thread(()->{
6             try {
7                 Thread.sleep(10);
8             } catch (InterruptedException e) {
9                 e.printStackTrace();
10            }
11            list.add(1);
12            countDownLatch.countDown();
13        }).start();
14    }
15    countDownLatch.await();

```

```
16     System.out.println(list.size());
17 }
```

第一次，我们居然发现arraylist也会有空指针，盲猜大概是，获取大小的时候没问题，插入的时候有人捷足先登，你就插不进去了。

```
1   Exception in thread "Thread-193" java.lang.ArrayIndexOutOfBoundsException: 163
2       at java.util.ArrayList.add(ArrayList.java:463)
3       at aaa.Test.lambda$main$0(Test.java:21)
4       at java.lang.Thread.run(Thread.java:748)
```

第二次

```
1   195
```

那怎么解决线程安全的问题啊。

加锁，其实JDK开始也是这样想的，于是有这两个类。

#3、加锁解决

HashTable和Vector，这是两个很古老的类

HashTable

```
1   public class Hashtable<K,V>
2       extends Dictionary<K,V>
3       implements Map<K,V>, Cloneable, java.io.Serializable {
4
5       public synchronized V get(Object key) {
6           Entry<?,?> tab[] = table;
7           int hash = key.hashCode();
8           int index = (hash & 0x7FFFFFFF) % tab.length;
9           for (Entry<?,?> e = tab[index] ; e != null ; e = e.next) {
10              if ((e.hash == hash) && e.key.equals(key)) {
11                  return (V)e.value;
12              }
13          }
14          return null;
15      }
16 }
```

Vector

```
1   public class Vector<E>
2       extends AbstractList<E>
3       implements List<E>, RandomAccess, Cloneable, java.io.Serializable
4
5
6   public synchronized int size() {
7       return elementCount;
8   }
9
10
11  public synchronized boolean add(E e) {
12      modCount++;
13      ensureCapacityHelper(elementCount + 1);
14      elementData[elementCount++] = e;
15      return true;
16 }
```

```

16     }
17
18
19
20     public synchronized boolean removeElement(Object obj) {
21         modCount++;
22         int i = indexOf(obj);
23         if (i >= 0) {
24             removeElementAt(i);
25             return true;
26         }
27         return false;
28     }

```

这两个类，其实很久没更新了，但是还是有面试会问，其实这两个类都有历史渊源，最开始就是在ArrayList和HashMap的基础上增加了Synchronized，但是后来ArrayList和HashMap一直在改进，这两个就成了历史了，反而现在问它们的区别其实意义不大了。

HashMap和HashTable区别

1. HashMap允许将 null 作为一个 entry 的 key 或者 value，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，而 HashMap 是 Java1.2 引进的 Map interface 的一个实现。
4. Hashtable 的方法是 Synchronized 修饰的，而 HashMap 不是，这也是是否能保证线程安全的重要保障。
5. Hashtable 和 HashMap 采用的 hash/rehash 算法都不一样。
6. 获取数组下标的算法不同，

ArrayList和Vector的区别

1. Vector是多线程安全的，线程安全就是说多线程访问同一代码，不会产生不确定的结果。而ArrayList不是，这个可以从源码中看出，Vector类中的方法很多，有synchronized进行修饰，这样就导致了Vector在效率上无法与ArrayList相比；
2. 两个都是采用线性连续空间存储元素，但是当空间不足的时候，两个类的扩容方式是不同的。
3. Vector是一种老的动态数组，是线程同步的，效率很低，一般不赞成使用。

#4、目前常用的线程安全集合

(1) CopyOnWriteList

目前我们有更好的解决方案：

CopyOnWriteList的核心就是写入的时候加锁，保证线程安全，读取的时候不加锁。不是一股脑，给所有的方法加锁。

```

1     public boolean add(E e) {
2         final ReentrantLock lock = this.lock;
3         lock.lock();
4         try {
5             // 复制一个新的数组，在新的独立空间进行添加操作
6             Object[] elements = getArray();
7             int len = elements.length;
8             Object[] newElements = Arrays.copyOf(elements, len + 1);
9             newElements[len] = e;

```

```

10         // 修改引用
11         setArray(newElements);
12         return true;
13     } finally {
14         lock.unlock();
15     }
16 }
17
18
19 final void setArray(Object[] a) {
20     array = a;
21 }

```

(2) ConcurrentHashMap

1.8中的ConcurrentHashMap和HashMap的代码基本一样，只不过在有些操作上使用了cas，有些地方加了锁。

```

1 public class ConcurrentHashMap<K,V> extends AbstractMap<K,V>
2     implements ConcurrentMap<K,V>, Serializable {

```

构造器：

```

1 public ConcurrentHashMap(int initialCapacity) {
2     if (initialCapacity < 0)
3         throw new IllegalArgumentException();
4     int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
5         MAXIMUM_CAPACITY :
6         tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
7     this.sizeCtl = cap;
8 }

```

我们简单看看putVal算法

```

1 final V putVal(K key, V value, boolean onlyIfAbsent) {
2     if (key == null || value == null) throw new NullPointerException();
3     // 计算hash
4     int hash = spread(key.hashCode());
5     int binCount = 0;
6     for (Node<K,V>[] tab = table;;) {
7         Node<K,V> f; int n, i, fh;
8         // 如果没有hash表，就创建一个
9         if (tab == null || (n = tab.length) == 0)
10             tab = initTable();
11         // 给f赋值就是hash表中的元素
12         else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
13             // 这里也是线程安全的
14             // 如果没有就使用cas的方式添加
15             if (casTabAt(tab, i, null,
16                 new Node<K,V>(hash, key, value, null)))
17                 break; // no lock when adding to empty bin
18         }
19         else if ((fh = f.hash) == MOVED)
20             tab = helpTransfer(tab, f);
21         else {
22             V oldVal = null;
23             // 看这是关键，这加了锁，f是什么啊？
24             // f是头节点

```

```

25         synchronized (f) {
26             if (tabAt(tab, i) == f) {
27                 if (fh >= 0) {
28                     binCount = 1;
29                     for (Node<K,V> e = f;; ++binCount) {
30                         K ek;
31                         if (e.hash == hash &&
32                             ((ek = e.key) == key ||
33                              (ek != null && key.equals(ek)))) {
34                             oldVal = e.val;
35                             if (!onlyIfAbsent)
36                                 e.val = value;
37                             break;
38                         }
39                         Node<K,V> pred = e;
40                         if ((e = e.next) == null) {
41                             pred.next = new Node<K,V>(hash, key,
42                                                         value, null);
43                             break;
44                         }
45                     }
46                 }
47                 else if (f instanceof TreeBin) {
48                     Node<K,V> p;
49                     binCount = 2;
50                     if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
51                                                             value)) != null) {
52                         oldVal = p.val;
53                         if (!onlyIfAbsent)
54                             p.val = value;
55                     }
56                 }
57             }
58         }
59         if (binCount != 0) {
60             if (binCount >= TREEIFY_THRESHOLD)
61                 treeifyBin(tab, i);
62             if (oldVal != null)
63                 return oldVal;
64             break;
65         }
66     }
67 }
68 addCount(1L, binCount);
69 return null;
70 }

```

其实，面试很喜欢问1.7和1.8的区别

主要是1.7的分段锁是一个很经典的案例，造成这个的原因还有一个更重要的就是JDK1.7使用的是头插，而1.8改成尾插

我们简单的看一下1.7的put方法实现：

```

1     public V put(K key, V value) {
2         if (key == null)
3             return putForNullKey(value);
4         int hash = hash(key);

```



```

5     int i = indexFor(hash, table.length);
6     // 找到相同的key, 覆盖
7     for (Entry<K,V> e = table[i]; e != null; e = e.next) {
8         Object k;
9         if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
10             V oldValue = e.value;
11             e.value = value;
12             e.recordAccess(this);
13             return oldValue;
14         }
15     }
16
17     modCount++;
18     // 否则就是新增
19     addEntry(hash, key, value, i);
20     return null;
21 }
22
23
24 void addEntry(int hash, K key, V value, int bucketIndex) {
25     // 判断是否需要扩容
26     if ((size >= threshold) && (null != table[bucketIndex])) {
27         resize(2 * table.length);
28         hash = (null != key) ? hash(key) : 0;
29         bucketIndex = indexFor(hash, table.length);
30     }
31     // 创建
32     createEntry(hash, key, value, bucketIndex);
33 }
34
35 void createEntry(int hash, K key, V value, int bucketIndex) {
36     Entry<K,V> e = table[bucketIndex];
37     // 头插啊
38     table[bucketIndex] = new Entry<>(hash, key, value, e);
39     size++;
40 }
41
42 // 1.7中居然直接就是Entry不是node
43 Entry(int h, K k, V v, Entry<K,V> n) {
44     value = v;
45     next = n;
46     key = k;
47     hash = h;
48 }
49

```

JDK8以前是头插法，JDK8后是尾插法，那为什么要从尾插法改成头插法？

1. 因为头插法会造成循环链表
2. JDK7用头插是考虑到了一个所谓的热点数据的点(新插入的数据可能会更早用到)，但这其实是个伪命题,因为JDK7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置(就是因为头插)，所以最后的结果 还是打乱了插入的顺序，所以总的来看支撑JDK7使用头插的这点原因也不足以支撑下去了，所以就干脆换成尾插一举多得。

1.7的加锁实现

```

1  /**
2   * Mask value for indexing into segments. The upper bits of a

```

```

3  * key's hash code are used to choose the segment.
4  */
5  final int segmentMask;
6
7  /**
8  * Shift value for indexing within segments.
9  */
10 final int segmentShift;
11
12 /**
13 * The segments, each of which is a specialized hash table.
14 */
15 final Segment<K,V>[] segments;
16
17 transient Set<K> keySet;
18 transient Set<Map.Entry<K,V>> entrySet;
19 transient Collection<V> values;

```

```

1  static final class Segment<K,V> extends ReentrantLock implements Serializable {
2
3      private static final long serialVersionUID = 2249069246763182397L;
4
5      static final int MAX_SCAN_RETRIES =
6          Runtime.getRuntime().availableProcessors() > 1 ? 64 : 1;
7
8      // 我们陡然发现每一个分段里边保存了一个数组，这不就是数组套数组吗？
9      transient volatile HashEntry<K,V>[] table;
10
11      transient int count;
12
13      final float loadFactor;

```

```

1  @SuppressWarnings("unchecked")
2  public ConcurrentHashMap(int initialCapacity,
3                          float loadFactor, int concurrencyLevel) {
4      if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
5          throw new IllegalArgumentException();
6      if (concurrencyLevel > MAX_SEGMENTS)
7          concurrencyLevel = MAX_SEGMENTS;
8      // Find power-of-two sizes best matching arguments
9      int sshift = 0;
10     int ssize = 1;
11     while (ssize < concurrencyLevel) {
12         ++sshift;
13         ssize <= 1;
14     }
15     this.segmentShift = 32 - sshift;
16     this.segmentMask = ssize - 1;
17     if (initialCapacity > MAXIMUM_CAPACITY)
18         initialCapacity = MAXIMUM_CAPACITY;
19     int c = initialCapacity / ssize;
20     if (c * ssize < initialCapacity)
21         ++c;
22     int cap = MIN_SEGMENT_TABLE_CAPACITY;
23     while (cap < c)
24         cap <= 1;
25     // create segments and segments[0]

```

```

26     Segment<K,V> s0 =
27         new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
28             (HashEntry<K,V>[])new HashEntry[cap]);
29     Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
30     UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]
31     this.segments = ss;
32 }

```

```

1     public V put(K key, V value) {
2         Segment<K,V> s;
3         if (value == null)
4             throw new NullPointerException();
5         int hash = hash(key);
6         int j = (hash >>> segmentShift) & segmentMask;
7         if ((s = (Segment<K,V>)UNSAFE.getObject           // nonvolatile; recheck
8             (segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment
9             s = ensureSegment(j);
10        return s.put(key, hash, value, false);
11    }

```

```

1     final V put(K key, int hash, V value, boolean onlyIfAbsent) {
2         // 先尝试加锁，加不上再疯狂加锁，反正能加上锁，他继承了ReentrantLock
3         HashEntry<K,V> node = tryLock() ? null :
4             scanAndLockForPut(key, hash, value);
5         V oldValue;
6         try {
7             HashEntry<K,V>[] tab = table;
8             int index = (tab.length - 1) & hash;
9             HashEntry<K,V> first = entryAt(tab, index);
10            for (HashEntry<K,V> e = first;;) {
11                if (e != null) {
12                    K k;
13                    if ((k = e.key) == key ||
14                        (e.hash == hash && key.equals(k))) {
15                        oldValue = e.value;
16                        if (!onlyIfAbsent) {
17                            e.value = value;
18                            ++modCount;
19                        }
20                        break;
21                    }
22                    e = e.next;
23                }
24                else {
25                    if (node != null)
26                        node.setNext(first);
27                    else
28                        node = new HashEntry<K,V>(hash, key, value, first);
29                    int c = count + 1;
30                    if (c > threshold && tab.length < MAXIMUM_CAPACITY)
31                        rehash(node);
32                    else
33                        setEntryAt(tab, index, node);
34                    ++modCount;
35                    count = c;
36                    oldValue = null;
37                    break;

```

```

38         }
39     }
40     } finally {
41         unlock();
42     }
43     return oldValue;
44 }

```

```

1  private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {
2      HashEntry<K,V> first = entryForHash(this, hash);
3      HashEntry<K,V> e = first;
4      HashEntry<K,V> node = null;
5      int retries = -1; // negative while locating node
6      // 不定的重新抢锁，抢锁的过程当中完成很多初始化的工作
7
8      while (!tryLock()) {
9          HashEntry<K,V> f; // to recheck first below
10         // 第一次再次抢锁时顺便初始化了entry
11         if (retries < 0) {
12             if (e == null) {
13                 if (node == null) // speculatively create node
14                     node = new HashEntry<K,V>(hash, key, value, null);
15                 retries = 0;
16             }
17             // 发现重复的key就不用初始化entry了
18             else if (key.equals(e.key))
19                 retries = 0;
20             else
21                 e = e.next;
22         }
23         // 如果超过最大的抢锁的次数直接调用lock
24         else if (++retries > MAX_SCAN_RETRIES) {
25             lock();
26             break;
27         }
28         else if ((retries & 1) == 0 &&
29                 (f = entryForHash(this, hash)) != first) {
30             e = first = f; // re-traverse if entry changed
31             retries = -1;
32         }
33     }
34     return node;
35 }

```

#5、guava提供的不可变集合

#七、JUnit单元测试

#1、JUnit 入门

JUnit 是一个 Java 编程语言的单元测试框架。JUnit 在测试驱动的开发方面有很重要的发展，是起源于 JUnit 的一个统称为 xUnit 的单元测试框架之一。

JUnit的好处：

1. 可以书写一系列的测试方法，对项目所有的接口或者方法进行单元测试。
2. 启动后，自动化测试，并判断执行结果,不需要人为的干预。
3. 只需要查看最后结果，就知道整个项目的方法接口是否通畅。
4. 每个单元测试用例相对独立，由JUnit 启动，自动调用。不需要添加额外的调用语句。
5. 添加，删除，屏蔽测试方法，不影响其他的测试方法。开源框架都对JUnit 有相应的支持。

JUnit其实就是一个jar包，idea中可以通过自动修复功能直接添加。但是为了演示清楚，我们还是安装规范引入jar包完成。

使用JUnit我们需要引入下边两个jar包即可：

1. hamcrest-core-1.1.jar
2. junit-4.12.jar

网站有提供。

加入JUnit后，我们可以创建测试类，测试方法，每一个测试方法都可以独立运行：



```
3  */
9  public class JunitTest {
10     @Test
11     public void print(){
12         System.out.println("hello junit!");
13     }
14 }
```

#2、JUnit 断言

JUnit所有的断言都包含在 Assert 类中。

这个类提供了很多有用的断言方法来编写测试用例。只有失败的断言才会被记录。Assert 类中的一些有用的方法列式如下：

1. `void assertEquals(boolean expected, boolean actual)` :检查两个变量或者等式是否平衡
2. `void assertTrue(boolean expected, boolean actual)` :检查条件为真
3. `void assertFalse(boolean condition)` :检查条件为假
4. `void assertNotNull(Object object)` :检查对象不为空
5. `void assertNull(Object object)` :检查对象为空
6. `void assertSame(boolean condition)` :assertSame() 方法检查两个相关对象是否指向同一个对象
7. `void assertNotSame(boolean condition)` :assertNotSame() 方法检查两个相关对象是否不指向同一个对象

8. `void assertEquals(expectedArray, resultArray)` :assertEquals() 方法检查两个数组是否相等

断言不成功会抛出异常，会有红色的进度条，断言能够帮助我们很好的预判结果，即使程序正常运行但是结果不正确，也会以失败结束。



```
11  @Test
12  public void print(){
13      System.out.println("hello junit!");
14      Assert.assertFalse(condition: true);
15  }
16  }
```

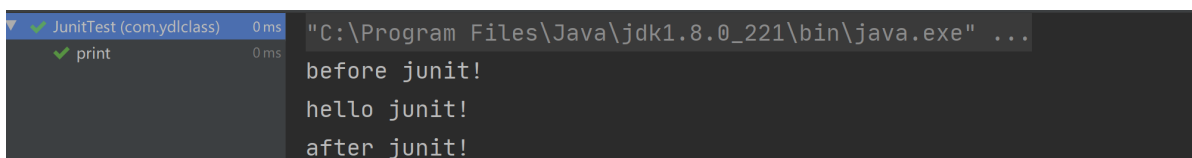
Tests failed: 1 of 1 test - 4 ms

java.lang.AssertionError <4 internal calls>
at com.ydlclass.JunitTest.print(JunitTest.java:14) <22 internal calls>

#3、JUnit 注解

1. `@Test` :这个注释说明依附在 JUnit 的 public void 方法可以作为一个测试案例。
2. `@Before` :有些测试在运行前需要创造几个相似的对象。在 public void 方法加该注释是因为该方法需要在 test 方法前运行。
3. `@After` :如果你将外部资源在 Before 方法中分配，那么你需要在测试运行后释放他们。在 public void 方法加该注释是因为该方法需要在 test 方法后运行。

```
1  public class JunitTest {
2
3      @Before
4      public void before(){
5          System.out.println("before junit!");
6      }
7
8      @Test
9      public void print(){
10         System.out.println("hello junit!");
11     }
12
13     @After
14     public void after(){
15         System.out.println("after junit!");
16     }
17 }
```



```
JUnitTest (com.ydlclass) 0 ms
print 0 ms
before junit!
hello junit!
after junit!
```

#4、命名规范

单元测试类的命名规范为：被测试类的类名+Test。 .

单元测试类中测试方法的命名规范为：test+被测试方法的方法名+AAA，其中AAA为对同一个方法的不同单元测试用例的自定义名称。 .

#六、性能对比

#1、Hashtable和ConcurrentHashMap

我们尝试开辟50个线程，每个线程向集合中put100000个元素，测试两个类所需要的时间。

```
1  @Test
2  public void hashtableTest() throws InterruptedException {
3      final Map<Integer,Integer> map = new Hashtable<>(500000);
4      final CountDownLatch countDownLatch = new CountDownLatch(50);
5      System.out.println("-----开始测试Hashtable-----");
6      long start = System.currentTimeMillis();
7      for (int i = 0; i < 50; i++) {
8          final int j = i;
9          new Thread()->{
10              for (int k = 0; k < 100000; k++) {
11                  map.put(j*k,1);
12              }
13              countDownLatch.countDown();
14          }).start();
15      }
16      countDownLatch.await();
17      long end = System.currentTimeMillis();
18      System.out.println("hashtable:(end-start) = " + (end - start));
19
20      // -----开始测试ConcurrentHashMap-----
21      System.out.println("-----开始测试ConcurrentHashMap-----");
22
23      final Map map2 = new ConcurrentHashMap<>(500000);
24      final CountDownLatch countDownLatch2 = new CountDownLatch(50);
25      start = System.currentTimeMillis();
26      for (int i = 0; i < 50; i++) {
27          final int j = i;
28          new Thread()->{
29              for (int k = 0; k < 100000; k++) {
30                  map2.put(j*k,1);
31              }
32              countDownLatch2.countDown();
33          }).start();
34      }
35      countDownLatch2.await();
36      end = System.currentTimeMillis();
37      System.out.println("ConcurrentHashMap:(end-start) = " + (end - start));
38  }
```

得到的结果：性能真的差距很大

```
1  -----开始测试Hashtable-----
2  hashtable:(end-start) = 777
3  -----开始测试ConcurrentHashMap-----
4  ConcurrentHashMap:(end-start) = 2
```

、arraylist和linkedlist

(1) 顺序添加

```
1  @Test
2  public void testArrayListAdd(){
3      List<Integer> list = new ArrayList<>();
4      Long start = System.currentTimeMillis();
5      for (int i = 0; i < 10000000; i++) {
6          list.add((int)(Math.random()*100));
7      }
8      Long end = System.currentTimeMillis();
9      System.out.printf("用时%d毫秒。",end-start);
10 }
11 结果:
12      用时243毫秒。
13
14  @Test
15  public void testLinkedListAdd(){
16      List<Integer> list = new LinkedList<>();
17      Long start = System.currentTimeMillis();
18      for (int i = 0; i < 10000000; i++) {
19          list.add((int)(Math.random()*100));
20      }
21      Long end = System.currentTimeMillis();
22      System.out.printf("用时%d毫秒。",end-start);
23  }
24 结果:
25      用时2524毫秒。
```

(2) 使用for循环迭代获取

```
1  @Test
2  public void testArrayListFor(){
3      List<Integer> list = new ArrayList<>();
4      for (int i = 0; i < 10000000; i++) {
5          list.add((int)(Math.random()*100));
6      }
7      System.out.println("开始-----");
8      Long start = System.currentTimeMillis();
9      for (int i = 0; i < list.size(); i++) {
10         list.get(i);
11     }
12     Long end = System.currentTimeMillis();
13     System.out.printf("用时%d毫秒。",end-start);
14 }
15 结果:
16     用时2毫秒。
17
18  @Test
19  public void testLinkedListFor(){
20      List<Integer> list = new LinkedList<>();
21      for (int i = 0; i < 10000000; i++) {
22          list.add((int)(Math.random()*100));
23      }
24      System.out.println("开始-----");
25      Long start = System.currentTimeMillis();
26      for (int i = 0; i < list.size(); i++) {
27          list.get(i);
```



```

28     }
29     Long end = System.currentTimeMillis();
30     System.out.printf("用时%d毫秒。",end-start);
31 }
32 结果:
33     无法计算时间。

```

(3) 使用迭代器迭代获取

```

1  @Test
2  public void testArrayListIterator(){
3      List<Integer> list = new ArrayList<>();
4      for (int i = 0; i < 10000000; i++) {
5          list.add((int)(Math.random()*100));
6      }
7      System.out.println("开始-----");
8      Long start = System.currentTimeMillis();
9      Iterator<Integer> iterator = list.iterator();
10     while (iterator.hasNext()){
11         iterator.next();
12     }
13     Long end = System.currentTimeMillis();
14     System.out.printf("用时%d毫秒。",end-start);
15 }
16 结果:
17     开始-----
18     用时4毫秒。
19
20 @Test
21 public void testLinkedListIterator(){
22     List<Integer> list = new LinkedList<>();
23     for (int i = 0; i < 10000000; i++) {
24         list.add((int)(Math.random()*100));
25     }
26     System.out.println("开始-----");
27     Long start = System.currentTimeMillis();
28     Iterator<Integer> iterator = list.iterator();
29     while (iterator.hasNext()){
30         iterator.next();
31     }
32     Long end = System.currentTimeMillis();
33     System.out.printf("用时%d毫秒。",end-start);
34 }
35
36 结果:
37     开始-----
38     用时42毫秒。

```

(4) 头插

```

1  @Test
2  public void testArrayListAddHeader(){
3      List<Integer> list = new ArrayList<>();
4      Long start = System.currentTimeMillis();
5      for (int i = 0; i < 10000000; i++) {
6          list.add(0,(int)(Math.random()*100));
7      }

```

```

8     Long end = System.currentTimeMillis();
9     System.out.printf("用时%d毫秒。",end-start);
10 }
11 结果:
12     无法算出, 太慢
13
14 @Test
15 public void testLinkedListAddHeader(){
16     List<Integer> list = new LinkedList<>();
17     Long start = System.currentTimeMillis();
18     for (int i = 0; i < 10000000; i++) {
19         list.add(0,(int)(Math.random()*100));
20     }
21     Long end = System.currentTimeMillis();
22     System.out.printf("用时%d毫秒。",end-start);
23 }
24 结果:
25     用时2487毫秒。

```

(5) 随机删除

```

1  @Test
2  public void testLinkedListDel(){
3      List<Integer> list = new LinkedList<>();
4      for (int i = 0; i < 10000000; i++) {
5          list.add(0,(int)(Math.random()*100));
6      }
7      Long start = System.currentTimeMillis();
8      // 不用管为啥, 这就是排序, 复制过来用就行, 写个冒泡也行
9      Iterator<Integer> iterator = list.iterator();
10     while (iterator.hasNext()){
11         if(iterator.next()>5000000){
12             iterator.remove();
13         }
14     }
15     Long end = System.currentTimeMillis();
16     System.out.printf("用时%d毫秒。",end-start);
17 }
18 结果:
19     用时45毫秒。
20
21 @Test
22 public void testArrayListDel(){
23     List<Integer> list = new ArrayList<>();
24     for (int i = 0; i < 10000000; i++) {
25         list.add(0,(int)(Math.random()*100));
26     }
27     Long start = System.currentTimeMillis();
28     // 不用管为啥, 这就是排序, 复制过来用就行, 写个冒泡也行
29     Iterator<Integer> iterator = list.iterator();
30     while (iterator.hasNext()){
31         if(iterator.next()>5000000){
32             iterator.remove();
33         }
34     }
35     Long end = System.currentTimeMillis();
36     System.out.printf("用时%d毫秒。",end-start);

```

```
37     }
38     结果:
39         太慢, 时间没出来
```

(6) 自带的排序方法

排序比较耗费资源, 所以我们把量级调整到了十万。

```
1  @Test
2  public void testArrayListSort(){
3      List<Integer> list = new ArrayList<>();
4      for (int i = 0; i < 100000; i++) {
5          list.add(0, (int)(Math.random()*100));
6      }
7      Long start = System.currentTimeMillis();
8      // 不用管为啥, 这就是排序, 复制过来用就行, 写个冒泡也行
9      list.sort(Comparator.comparingInt(num -> num));
10     Long end = System.currentTimeMillis();
11     System.out.printf("用时%d毫秒。", end-start);
12 }
13 结果:
14     用时49毫秒。
15
16 @Test
17 public void testLinkedListSort(){
18     List<Integer> list = new LinkedList<>();
19     for (int i = 0; i < 100000; i++) {
20         list.add(0, (int)(Math.random()*100));
21     }
22     Long start = System.currentTimeMillis();
23     // 不用管为啥, 这就是排序, 复制过来用就行, 写个冒泡也行
24     list.sort(Comparator.comparingInt(num -> num));
25     Long end = System.currentTimeMillis();
26     System.out.printf("用时%d毫秒。", end-start);
27 }
28
29 结果:
30     用时53毫秒。
```

(7) 思考

其实我们学习时, 总是去背诵概念:

数组查询快, 插入慢。链表插入慢, 查询快。

- 但是经过测试, 尾插反而是数组快, 而尾插的使用场景极多。
- 测试了各种迭代, 遍历方法, ArrayList基本都是比LinkedList要快。
- 随机插入, 链表会快很多, 确实有一些特殊的场景LinkedList更合适, 比如以后我们学的过滤器链。
- 随机删除, 链表的效率也是无比优于数组, 如果我们存在需要过滤删除大量随机元素的场景也能使用linkedlist。
- 我们工作中的使用还是以ArrayList为主, 因为它的使用场景最多。

#七、Java8特性

#1、接口默认方法

在JDK8之前，接口不能定义任何实现，这意味着之前所有的JAVA版本中，接口制定的方法是抽象的，不包含方法体。从JDK8开始，添加了一种新功能-默认方法。默认方法允许接口方法定义默认实现，而所有子类都将拥有该方法及实现。

默认方法的主要优势是提供一种拓展接口的方法，而不破坏现有代码。假如我们有一个已经投入使用接口，需要拓展一个新的方法，在JDK8以前，如果为一个使用的接口增加一个新方法，则我们必须在所有实现类中添加该方法的实现，否则编译会出现异常。如果实现类数量少并且我们有权限修改，可能会工作量相对较少。如果实现类比较多或者我们没有权限修改实现类源代码，这样可能就比较麻烦。而默认方法则解决了这个问题，它提供了一个实现，当没有显示提供其他实现时就采用这个实现。这样新添加的方法将不会破坏现有代码。

#2、函数式接口

函数式接口在Java中是指：有且仅有一个抽象方法的接口

函数式接口，即适用于函数式编程场景的接口。而Java中的函数式编程体现就是Lambda，所以函数式接口就是可以适用于Lambda使用的接口。只有确保接口中有且仅有一个抽象方法，Java中的Lambda才能顺利地进行推导。

接下来给大家介绍几个常用的函数式接口，在我们接下来要学习的Lambda表达式中大量使用。

消费者，消费数据

```
1  @FunctionalInterface
2  public interface Consumer<T> {
3      void accept(T t);
4  }
```

供应商，给我们产生数据

```
1  @FunctionalInterface
2  public interface Supplier<T> {
3      T get();
4  }
```

断言，判断传入的t是不是满足条件

```
1  @FunctionalInterface
2  public interface Predicate<T> {
3
4      boolean test(T t);
5  }
```

函数，就是将一个数据转化成另一个数据

```
1  @FunctionalInterface
2  public interface Function<T, R> {
3      R apply(T t);
4  }
```

我们在思考上边的代码的时候，不要胡思乱想，它们就是一组接口，和我们的普通接口一样，每个接口代表一种能力，需要子类去实现，因为它们是函数式接口，所以匿名内部类都可以写成箭头函数的形式。

#3、Optional

(1) 简介

```
1      Optional类是Java8为了解决null值判断问题，借鉴google guava类库的Optional类而引入的一个同名Optional类，使用Optional类可以避免显式的null值判断（null的防御性检查），避免null导致的NPE（NullPointerException）。
```

(2) Optional对象的创建

Optional类提供了三个静态方法empty()、of(T value)、ofNullable(T value)来创建Optional对象，示例如下：

```
1      // 1、创建一个包装对象值为空的Optional对象
2      Optional<String> optStr = Optional.empty();
3      // 2、创建包装对象值非空的Optional对象
4      Optional<String> optStr1 = Optional.of("optional");
5      // 3、创建包装对象值允许为空的Optional对象
6      Optional<String> optStr2 = Optional.ofNullable(null);
```

(3) Optional 类典型接口的使用

下面以一些典型场景为例，列出Optional API常用接口的用法，并附上相应代码。

get()方法

简单看下get()方法的源码：

```
1      public T get() {
2          if (value == null) {
3              throw new NoSuchElementException("No value present");
4          }
5          return value;
6      }
```

可以看到，get()方法主要用于返回包装对象的实际值，但是如果包装对象值为null，会抛出NoSuchElementException异常。

isPresent()方法

isPresent()方法的源码：

```
1      public boolean isPresent() {
2          return value != null;
3      }
```

可以看到，isPresent()方法用于判断包装对象的值是否非空。下面我们来看一段糟糕的代码：

```

1  public static String getGender(Student student){
2      Optional<Student> stuOpt = Optional.ofNullable(student);
3      if(stuOpt.isPresent())
4      {
5          return stuOpt.get().getGender();
6      }
7
8      return "Unkown";
9  }

```

这段代码实现的是第一章(简介)中的逻辑，但是这种用法不但没有减少null的防御性检查，而且增加了Optional包装的过程，违背了Optional设计的初衷，因此开发中要避免这种糟糕的使用

ifPresent()方法

ifPresent()方法的源码：

```

1  public void ifPresent(Consumer<? super T> consumer) {
2      if (value != null)
3          consumer.accept(value);
4  }

```

ifPresent()方法接受一个Consumer对象（消费函数），如果包装对象的值非空，运行Consumer对象的accept()方法。示例如下：

```

1  public static void printName(Student student){
2      Optional.ofNullable(student).ifPresent(u -> System.out.println("The student
   name is : " + u.getName()));
3  }

```

上述示例用于打印学生姓名，由于ifPresent()方法内部做了null值检查，调用前无需担心NPE问题。

orElse()方法

orElse()方法的源码：

```

1  public T orElse(T other) {
2      return value != null ? value : other;
3  }

```

orElse()方法功能比较简单，即如果包装对象值非空，返回包装对象值，否则返回入参other的值（默认值）。

```

1  public static String getGender(Student student){
2      return Optional.ofNullable(student).map(u -> u.getGender()).orElse("Unkown");
3  }

```

orElseGet()方法

orElseGet()方法的源码：

```

1  public T orElseGet(Supplier<? extends T> other) {
2      return value != null ? value : other.get();
3  }

```

orElseGet()方法与orElse()方法类似，区别在于orElseGet()方法的入参为一个Supplier对象，用Supplier对象的get()方法的返回值作为默认值。如：

```
1 public static String getGender(Student student)
2 {
3     return Optional.ofNullable(student).map(u -> u.getGender()).orElseGet(() -
4     > "Unkown");
5 }
```

orElseThrow()方法

orElseThrow()方法的源码：

```
1 public <X extends Throwable> T orElseThrow(Supplier<? extends X>
2     exceptionSupplier) throws X {
3     if (value != null) {
4         return value;
5     } else {
6         throw exceptionSupplier.get();
7     }
8 }
```

orElseThrow()方法其实与orElseGet()方法非常相似了，入参都是Supplier对象，只不过orElseThrow()的Supplier对象必须返回一个Throwable异常，并在orElseThrow()中将异常抛出：

```
1 public static String getGender1(Student student){
2     return Optional.ofNullable(student).map(u -> u.getGender()).orElseThrow(() ->
3     new RuntimeException("Unkown"));
4 }
```

orElseThrow()方法适用于包装对象值为空时需要抛出特定异常的场景。

#八、Stream编程

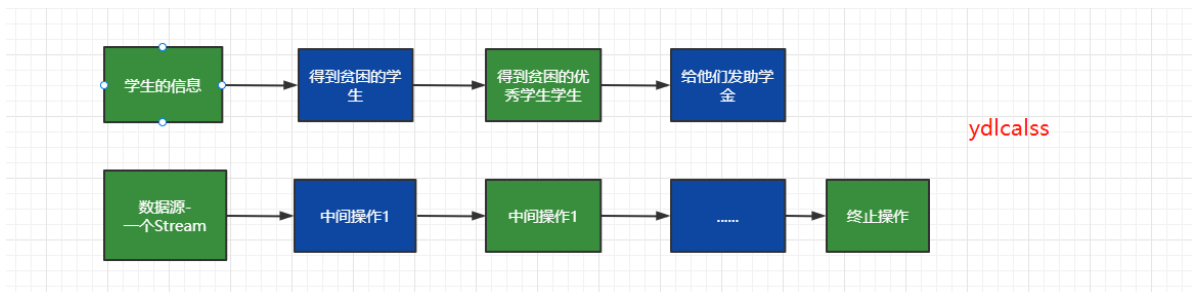
Java8中的Stream是对容器对象功能的增强，它专注于对容器对象进行各种非常便利、高效的聚合操作（aggregate operation），或者大批量数据操作（bulk data operation）。Stream API借助于同样新出现的Lambda表达式，极大的提高编程效率和程序可读性。同时，它提供串行和并行两种模式进行汇聚操作，并发模式能够充分利用多核处理器的优势。通常，编写并行代码很难而且容易出错，但使用Stream API无需编写一行多线程的代码，就可以很方便地写出高性能的并发程序。

我觉得我们可以将流看做流水线，这个流水线是处理数据的流水线，一个产品经过流水线会有一道道的工序就如同对数据的中间操作，比如过滤我不需要的，给数据排序能，最后的终止操作就是产品从流水线下来，我们就可以统一打包放入仓库了。

当我们使用一个流的时候，通常包括三个基本步骤：获取一个数据源（source）→ 数据转换 → 执行操作获取想要的结果。**每次转换原有Stream对象不改变，返回一个新的Stream对象（可以有多次转换），这就允许对其操作可以像链条一样排列，变成一个管道，如下图所示：**

Stream有几个特性：

1. Stream不存储数据，而是按照特定的规则对数据进行计算，一般会输出结果。
2. Stream不会改变数据源，通常情况下会产生一个新的集合或一个值。
3. Stream具有延迟执行特性，只有调用终端操作时，中间操作才会执行。



#1、Stream流的创建

(1) Stream可以通过集合数组创建。

1、通过 `java.util.Collection.stream()` 方法用集合创建流，我们发现

```
1 default Stream<E> stream() {
2     return StreamSupport.stream(spliterator(), false);
3 }
```

```
1 List<String> list = Arrays.asList("a", "b", "c");
2 // 创建一个顺序流
3 Stream<String> stream = list.stream();
4 // 创建一个并行流
5 Stream<String> parallelStream = list.parallelStream();
```

(2) 使用 `java.util.Arrays.stream(T[] array)` 方法用数组创建流

```
1 int[] array={1,3,5,6,8};
2 IntStream stream = Arrays.stream(array);
```

(3) 使用Stream的静态方法: `of()`、`iterate()`、`generate()`

```
1 Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6);
2
3 Stream<Integer> stream2 = Stream.iterate(0, (x) -> x + 3).limit(4);
4 stream2.forEach(System.out::println);
5
6 Stream<Double> stream3 = Stream.generate(Math::random).limit(3);
7 stream3.forEach(System.out::println);
```

#2、Stream的终止操作

为了方便我们后续的使用，我们先初始化一部分数据：

```
1 public class Person {
2     private String name; // 姓名
3     private int salary; // 薪资
4     private int age; // 年龄
5     private String sex; // 性别
6     private String area; // 地区
7
8     public Person() {
9     }
10
11     public Person(String name, int salary, int age, String sex, String area) {
12         this.name = name;
13         this.salary = salary;
14         this.age = age;
```



```

15         this.sex = sex;
16         this.area = area;
17     }
18 }

```

初始化数据，我们设计一个简单的集合和一个复杂的集合。

```

1  public class LambdaTest {
2
3      List<Person> personList = new ArrayList<Person>();
4      List<Integer> simpleList = Arrays.asList(15, 22, 9, 11, 33, 52, 14);
5
6      @Before
7      public void initData(){
8          personList.add(new Person("张三", 3000, 23, "男", "太原"));
9          personList.add(new Person("李四", 7000, 34, "男", "西安"));
10         personList.add(new Person("王五", 5200, 22, "女", "太原"));
11         personList.add(new Person("小黑", 1500, 33, "女", "上海"));
12         personList.add(new Person("狗子", 8000, 44, "女", "北京"));
13         personList.add(new Person("铁蛋", 6200, 36, "女", "南京"));
14     }
15 }

```

(1) 遍历/匹配 (foreach/find/match)

将数据流消费掉

```

1  @Test
2  public void foreachTest(){
3      // 打印集合的元素
4      simpleList.stream().forEach(System.out::println);
5      // 其实可以简化操作的
6      simpleList.forEach(System.out::println);
7  }
8
9
10 @Test
11 public void findTest(){
12     // 找到第一个
13     Optional<Integer> first = simpleList.stream().findFirst();
14     // 随便找一个, 可以看到findAny()操作, 返回的元素是不确定的,
15     // 对于同一个列表多次调用findAny()有可能会返回不同的值。
16     // 使用findAny()是为了更高效的性能。如果是数据较少, 串行地情况下, 一般会返回第一个结果,
17     // 如果是并行的情况, 那就不能确保是第一个。
18     Optional<Integer> any = simpleList.parallelStream().findAny();
19     System.out.println("first = " + first.get());
20     System.out.println("any = " + any.get());
21 }
22
23 @Test
24 public void matchTest(){
25     // 判断有没有任意一个人年龄大于35岁
26     boolean flag = personList.stream().anyMatch(item -> item.getAge() > 35);
27     System.out.println("flag = " + flag);
28
29     // 判断是不是所有人年龄都大于35岁
30     flag = personList.stream().allMatch(item -> item.getAge() > 35);
31     System.out.println("flag = " + flag);

```

(2) 归集(toList/toSet/toMap)

因为流不存储数据，那么在流中的数据完成处理后，需要将流中的数据重新归集到新的集合里。

`toList`、`toSet` 和 `toMap` 比较常用。

下面用一个案例演示 `toList`、`toSet` 和 `toMap`：

```
1  @Test
2  public void collectTest(){
3      // 判断有没有任意一个人年龄大于35岁
4      List<Integer> collect = simpleList.stream().collect(Collectors.toList());
5      System.out.println(collect);
6      Set<Integer> collectSet = simpleList.stream().collect(Collectors.toSet());
7      System.out.println(collectSet);
8      Map<Integer,Integer> collectMap =
9      simpleList.stream().collect(Collectors.toMap(item->item,item->item+1));
10     System.out.println(collectMap);
11 }
```

(3) 统计(count/averaging/sum/max/min)

```
1  @Test
2  public void countTest(){
3      // 判断有没有任意一个人年龄大于35岁
4      long count = new Random().ints().limit(50).count();
5      System.out.println("count = " + count);
6      OptionalDouble average = new Random().ints().limit(50).average();
7      average.ifPresent(System.out::println);
8      int sum = new Random().ints().limit(50).sum();
9      System.out.println(sum);
10 }
```

案例：获取员工工资最高的人

```
1  Optional<Person> max = personList.stream().max((p1, p2) -> p1.getSalary() -
2  p2.getSalary());
3  max.ifPresent(item -> System.out.println(item.getSalary()));
4  里边的比较器可以改为：Comparator.comparingInt(Person::getSalary)
```

(4) 归约(reduce)

归约，也称缩减，顾名思义，是把一个流缩减成一个值，能实现对集合求和、求乘积和求最值操作。

案例：求 `Integer` 集合的元素之乘积。

```
1  @Test
2  public void reduceTest(){
3      Integer result = simpleList.stream().reduce(1,(n1, n2) -> n1*n2);
4      System.out.println(result);
5  }
```

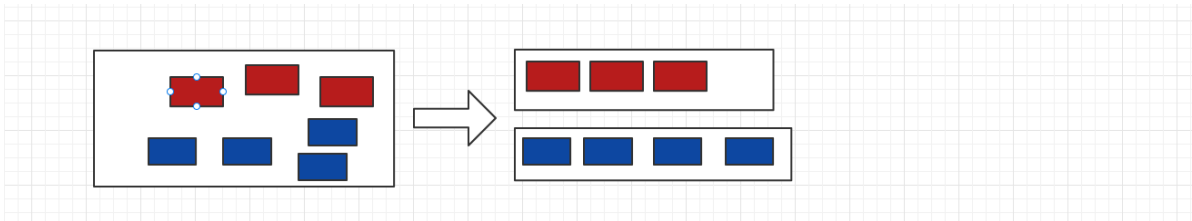
(5) 接合(joining)

`joining` 可以将Stream中的元素用特定的连接符（没有的话，则直接连接）连接成一个字符串。

```
1  @Test
2  public void joiningTest(){
3      List<String> list = Arrays.asList("A", "B", "C");
4      String string = list.stream().collect(Collectors.joining("-"));
5      System.out.println("拼接后的字符串: " + string);
6  }
7  }
```

(6) 分组(partitioningBy/groupingBy)

- 分区：将 `stream` 按条件分为两个 `Map`，比如员工按薪资是否高于8000分为两部分。
- 分组：将集合分为多个Map，比如员工按性别分组。



案例：将员工按薪资是否高于8000分为两部分；将员工按性别和地区分组

```
1  @Test
2  public void groupingByTest(){
3      // 将员工按薪资是否高于8000分组
4      Map<Boolean, List<Person>> part =
5      personList.stream().collect(Collectors.partitioningBy(x -> x.getSalary() >
6      8000));
7      // 将员工按性别分组
8      Map<String, List<Person>> group =
9      personList.stream().collect(Collectors.groupingBy(Person::getSex));
10     // 将员工先按性别分组，再按地区分组
11     Map<String, Map<String, List<Person>>> group2 =
12     personList.stream().collect(Collectors.groupingBy(Person::getSex,
13     Collectors.groupingBy(Person::getArea)));
14     System.out.println("员工按薪资是否大于8000分组情况: " + part);
15     System.out.println("员工按性别分组情况: " + group);
16     System.out.println("员工按性别、地区: " + group2);
17 }
```

#3、Stream中间操作

(1) 筛选 (filter)

该操作符需要传入一个function函数

筛选出 `simpleList` 集合中大于17的元素，并打印出来

```
1  simpleList.stream().filter(item -> item > 17).forEach(System.out::println);
```

筛选员工中工资高于8000的人，并形成新的集合。

```

1 List<Person> collect = personList.stream().filter(item -> item.getSalary() >
  8000).collect(Collectors.toList());
2 System.out.println("collect = " + collect);

```

(2) 映射(map/flatMap)

映射，可以将一个流的元素按照一定的映射规则映射到另一个流中。分为 `map` 和 `flatMap`：

- `map`：接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
- `flatMap`：接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。

案例：将员工的薪资全部增加1000。

```

1 personList.stream().map(item -> {
2     item.setSalary(item.getSalary()+1000);
3     return item;
4 }).forEach(System.out::println);

```

将simpleList转化为字符串list

```

1 List<String> collect = simpleList.stream().map(num -> Integer.toString(num))
2     .collect(Collectors.toList());

```

(3) 排序(sorted)

sorted，中间操作。有两种排序：

- `sorted()`：自然排序，流中元素需实现Comparable接口
- `sorted(Comparator com)`：Comparator排序器自定义排序

案例：将员工按工资由高到低（工资一样则按年龄由大到小）排序

```

1 @Test
2 public void sortTest(){
3     // 按工资升序排序（自然排序）
4     List<String> newList =
5     personList.stream().sorted(Comparator.comparing(Person::getSalary)).map(Person::g
6     etName)
7     .collect(Collectors.toList());
8     // 按工资倒序排序
9     List<String> newList2 =
10    personList.stream().sorted(Comparator.comparing(Person::getSalary).reversed())
11    .map(Person::getName).collect(Collectors.toList());
12    // 先按工资再按年龄升序排序
13    List<String> newList3 = personList.stream()
14    .sorted(Comparator.comparing(Person::getSalary).thenComparing(Person::getAge)).ma
15    p(Person::getName)
16    .collect(Collectors.toList());
17    // 先按工资再按年龄自定义排序（降序）
18    List<String> newList4 = personList.stream().sorted((p1, p2) -> {
19        if (p1.getSalary() == p2.getSalary()) {
20            return p2.getAge() - p1.getAge();
21        } else {
22            return p2.getSalary() - p1.getSalary();
23        }
24    })
25    .map(Person::getName).collect(Collectors.toList());
26 }

```

```

20     }).map(Person::getName).collect(Collectors.toList());
21
22     System.out.println("按工资升序排序: " + newList);
23     System.out.println("按工资降序排序: " + newList2);
24     System.out.println("先按工资再按年龄升序排序: " + newList3);
25     System.out.println("先按工资再按年龄自定义降序排序: " + newList4);
26 }

```

(4) peek操作

peek的调试作用

```

1  @Test
2  public void peekTest(){
3      // 在stream中间进行调试, 因为stream不支持debug
4      List<Person> collect = personList.stream().filter(p -> p.getSalary() > 5000)
5          .peek(System.out::println).collect(Collectors.toList());
6      // 修改元素的信息, 给每个员工涨工资一千
7      personList.stream().peek(p -> p.setSalary(p.getSalary() + 1000))
8          .forEach(System.out::println);
9  }

```

(5) 其他操作

流也可以进行合并、去重、限制、跳过等操作。

```

1  @Test
2  public void otherTest(){
3      // distinct去掉重复数据
4      // skip跳过几个数据
5      // limit限制使用几个数据
6      simpleList.stream().distinct().skip(2).limit(3).forEach(System.out::println);
7  }
8
9  // 11,11,22,22,11,23,43,55,78
10 // 去重 11, 22,23,43,55,78
11 // 跳过两个 23, 43,55,78
12 // 使用3个 23,43,55

```