# 第09天【集合\_第二天】

## 主要内容

1. HashMap、TreeMap、LinkedHashMap
2. HashMap和TreeMap源码解析
3. HashSet和TreeSet源码解析
4. Iterator和ListIterator
5. Collections
6. 旧的集合类
7. 最新并发集合类
8. 集合常用概念辨析
9. 泛型
10. JDK8新特征

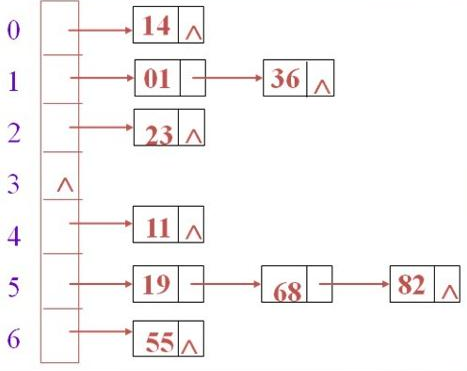
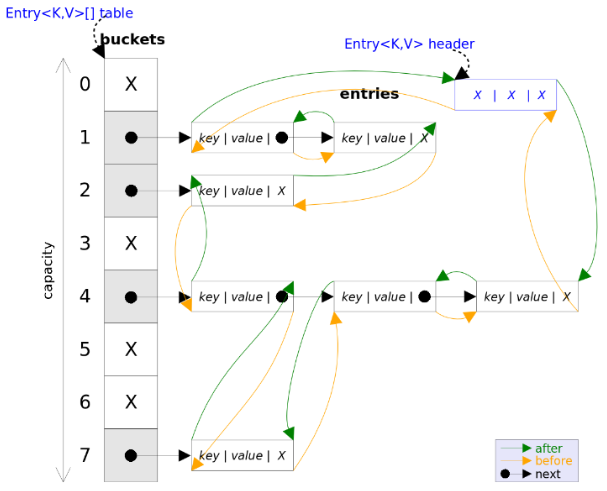
## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| HashMap | 掌握 |
| TreeMap | 掌握 |
| LinkedHashMap | 理解 |
| HashMap源码 | 掌握 |
| TreeMap源码 | 掌握 |
| HashSet源码 | 理解 |
| TreeSet源码 | 理解 |
| Iterator | 掌握 |
| ListIterator | 了解 |
| Collections | 了解 |
| 旧的集合类 | 了解 |
| 最新并发集合类 | 了解 |
| 泛型类 | 了解 |
| 泛型接口 | 了解 |
| 泛型方法 | 了解 |
| Lambda表达式 | 了解 |
| 函数式接口 | 了解 |
| Stream流式编程 | 了解 |
| 方法引用 | 了解 |

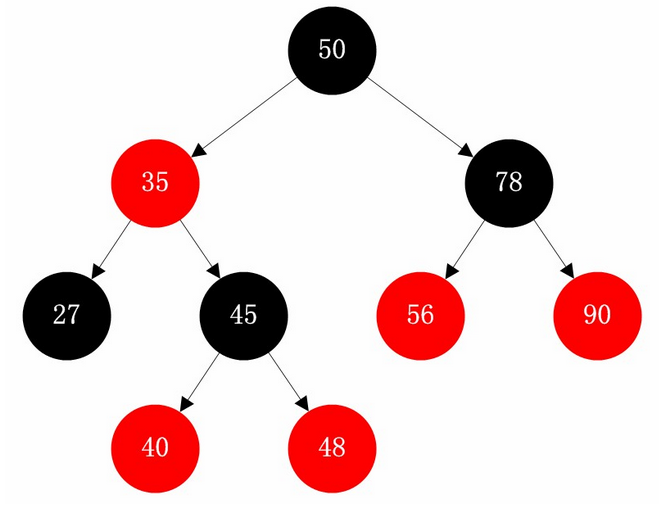
## 一、Map集合的使用

### 1.1 Map集合类型

* Map
  + 特点：存储的键值对映射关系，根据key（唯一）可以找到value
* HashMap
  + 采用Hashtable哈希表存储结构（神奇的结构）
  + 优点：添加速度快 查询速度快 删除速度快
  + 缺点：key无序
* LinkedHashMap
  + 采用哈希表存储结构，同时使用链表维护次序
  + key有序（添加顺序）

* TreeMap
  + 采用二叉树（红黑树）的存储结构
  + 优点：key有序 查询速度比List快（按照内容查询）
  + 缺点：查询速度没有HashMap快



### 1.2 使用Map存储国家简称-国家名称映射

#### 【示例1】使用各种Map存储国家简称-国家名称映射

|  |
| --- |
| **public class** TestMap1 {  **public static void** main(String[] args) {  //创建一个Map集合对象  //Map <String,String>map = new HashMap<String,String>();  //Map <String,String>map =  //new LinkedHashMap<String,String>();  Map<String, String> map = **new** TreeMap<String, String>();  //向Map集合中添加元素（key-value）  map.put(**"cn"**, **"China"**);  map.put(**"jp"**, **"Japan"**);  map.put(**"us"**, **"the United States"**);  map.put(**"us"**, **"America"**);   map.put(**"uk"**, **"England"**);  map.put(**"en"**, **"England"**);  //从Map中根据key获取value  System.**out**.println(map.size()); // System.out.println(map); // System.out.println(map.keySet());//Set 得到所有的key // System.out.println(map.values());//Collection 得到所有的value // System.out.println(map.get("cn")); // System.out.println(map.get("it"));   //Map的遍历  //思路1：先得到所有的key（Set），然后根据key找到value  Set<String> keySet = map.keySet();  **for** (String key : keySet) {  System.**out**.println(key + **"---->"** + map.get(key));  }  //思路2：先得到所有的key-value组成的Set，然后输出每个key-value  Set<Map.Entry<String, String>> entrySet = map.entrySet();  Iterator<Map.Entry<String, String>> it = entrySet.iterator();  **while** (it.hasNext()) {  //取出一个Entry  Map.Entry<String, String> entry = it.next();  //输出一个Entry  System.**out**.println(entry.getKey() + **"->"**+ entry.getValue());  }  } } |

### 1.3 使用各种Map存储学号-学生映射

#### 【示例2】使用各种Map存储学号-学生映射

|  |
| --- |
| **public class** TestMap2 {  **public static void** main(String[] args) {  //创建一个Map对象用户存储key-value  Map<Integer,Student> map = **new** HashMap<Integer,Student>();  //Map <Integer,Student>map = new TreeMap<Integer,Student>();  //使用Map对象存储多个key-value  Student stu1 = **new** Student(1, **"zhangsan"**, 23, 98.2);  Student stu2 = **new** Student(2, **"zhangsan"**, 23, 98);  Student stu3 = **new** Student(3, **"wangwu"**, 22, 98.5);  Student stu4 = **new** Student(1, **"zhangsan"**, 23, 98.2);    map.put(stu1.getSno(), stu1);  map.put(stu2.getSno(), stu2);  map.put(stu3.getSno(), stu3);  map.put(stu4.getSno(), stu4);   //Map的其他方法  //map.remove(1);  //map.clear();  //map.replace(1, new Student(1, "zhaoliu", 23, 100));  map.containsKey(1);  map.containsValue(stu4);  map.isEmpty();   //从Map对象中根据学号找到对应的学生  Student stu = map.get(1);//key 学号 // List list = new ArrayList();; // list.get(1);//索引   **if**(stu == **null**){  System.**out**.println(**"该学生不存在"**);  }**else**{  System.**out**.println(stu);  }   System.**out**.println(map.size());//  System.**out**.println(map.toString());   //遍历  Set<Entry<Integer,Student>> entrySet = map.entrySet();  **for**(Entry<Integer,Student> entry:entrySet){  Student student = entry.getValue();  System.**out**.println(student);  }   } } |

|  |  |
| --- | --- |
| **方法摘要** | |
| void | **[clear](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "clear())**()            从此映射中移除所有映射关系（可选操作）。 |
| boolean | **[containsKey](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "containsKey(java.lang.Object))**([Object](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html" \o "java.lang 中的类) key)            如果此映射包含指定键的映射关系，则返回 true。 |
| boolean | **[containsValue](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "containsValue(java.lang.Object))**([Object](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html" \o "java.lang 中的类) value)            如果此映射将一个或多个键映射到指定值，则返回 true。 |
| [Set](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Set.html" \o "java.util 中的接口)<[Map.Entry](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.Entry.html" \o "java.util 中的接口)<[K](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数),[V](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数)>> | **[entrySet](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "entrySet())**()            返回此映射中包含的映射关系的 [Set](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Set.html" \o "java.util 中的接口) 视图。 |
| boolean | **[equals](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "equals(java.lang.Object))**([Object](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html" \o "java.lang 中的类) o)            比较指定的对象与此映射是否相等。 |
| [V](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数) | **[get](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "get(java.lang.Object))**([Object](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html" \o "java.lang 中的类) key)            返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null。 |
| int | **[hashCode](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "hashCode())**()            返回此映射的哈希码值。 |
| boolean | **[isEmpty](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "isEmpty())**()            如果此映射未包含键-值映射关系，则返回 true。 |
| [Set](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Set.html" \o "java.util 中的接口)<[K](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数)> | **[keySet](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "keySet())**()            返回此映射中包含的键的 [Set](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Set.html" \o "java.util 中的接口) 视图。 |
| [V](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数) | **[put](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "put(K, V))**([K](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数) key, [V](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数) value)            将指定的值与此映射中的指定键关联（可选操作）。 |
| void | **[putAll](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "putAll(java.util.Map))**([Map](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "java.util 中的接口)<? extends [K](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数),? extends [V](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数)> m)            从指定映射中将所有映射关系复制到此映射中（可选操作）。 |
| [V](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数) | **[remove](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "remove(java.lang.Object))**([Object](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html" \o "java.lang 中的类) key)            如果存在一个键的映射关系，则将其从此映射中移除（可选操作）。 |
| int | **[size](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "size())**()            返回此映射中的键-值映射关系数。 |
| [Collection](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html" \o "java.util 中的接口)<[V](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \o "Map 中的类型参数)> | **[values](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Map.html" \l "values())**()            返回此映射中包含的值的 [Collection](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html" \o "java.util 中的接口) 视图。 |

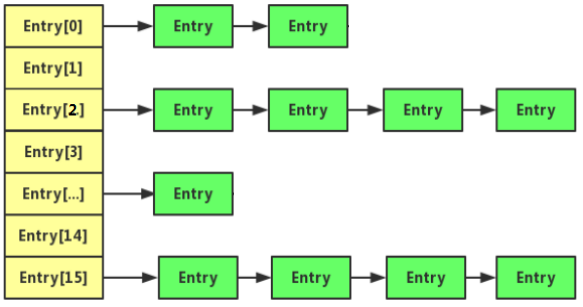
### 本节作业

1. 使用各种Map存储并操作国家简称-国家名称的映射信息
2. 使用各种Map存储并操作学号-学生的映射信息

## 二、Map和Set集合源码

### 2.1理解 HashMap的源码

* JDK1.7及其之前，HashMap底层是一个table数组+链表实现的哈希表存储结构



* 链表的每个节点就是一个Entry，其中包括：键key、值value、键的哈希码hash、执行下一个节点的引用next四部分

|  |
| --- |
| **static class** Entry<K, V> **implements** Map.Entry<K, V> {  **final** K **key**; *//key* V **value**;*//value* Entry<K, V> **next**; *//指向下一个节点的指针* **int hash**;*//哈希码* } |

* JDK1.7中HashMap的主要成员变量及其含义

|  |
| --- |
| **public class** HashMap<K, V> **implements** Map<K, V> {  *//哈希表主数组的默认长度*  **static final int *DEFAULT\_INITIAL\_CAPACITY*** = 16;  *//默认的装填因子* **static final float *DEFAULT\_LOAD\_FACTOR*** = 0.75f;  *//主数组的引用！！！！* **transient** Entry<K, V>[] **table**;**int threshold**;*//界限值 阈值* **final float loadFactor**;*//装填因子* **public** HashMap() {  **this**(***DEFAULT\_INITIAL\_CAPACITY***, ***DEFAULT\_LOAD\_FACTOR***);  }  **public** HashMap(**int** initialCapacity, **float** loadFactor) {  **this**.**loadFactor** = loadFactor;*//0.75* **threshold** = (**int**) Math.min(capacity \* loadFactor,  MAXIMUM\_CAPACITY + 1);*//16\*0.75=12* **table** = **new** Entry[capacity];  ....  } } |

* 调用put方法添加键值对。哈希表三步添加数据原理的具体实现；是计算key的哈希码，和value无关。特别注意：
  1. 第一步计算哈希码时，不仅调用了key的hashCode()，还进行了更复杂处理，目的是尽量保证不同的key尽量得到不同的哈希码
  2. 第二步根据哈希码计算存储位置时，使用了位运算提高效率。同时也要求主数组长度必须是2的幂）
  3. 第三步添加Entry时添加到链表的第一个位置，而不是链表末尾
  4. 第三步添加Entry是发现了相同的key已经存在，就使用新的value替代旧的value，并且返回旧的value

|  |
| --- |
| **public class** HashMap {  **public** V put(K key, V value) {  //如果key是null，特殊处理  **if** (key == **null**) **return** putForNullKey(value);  //1.计算key的哈希码hash   **int** hash = hash(key);  //2.将哈希码代入函数，计算出存储位置 y= x%16；  **int** i = indexFor(hash, table.length);  //如果已经存在链表，判断是否存在该key，需要用到equals()  **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {  Object k;  //如找到了,使用新value覆盖旧的value，返回旧value  **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {   V oldValue = e.value;// the United States  e.value = value;//America  e.recordAccess(**this**);  **return** oldValue;  }  }  //添加一个结点  addEntry(hash, key, value, i);  **return null**;  }  **final int** hash(Object k) {  **int** h = 0;  h ^= k.hashCode();  h ^= (h >>> 20) ^ (h >>> 12);  **return** h ^ (h >>> 7) ^ (h >>> 4); } **static int** indexFor(**int** h, **int** length) {  *//作用就相当于y = x%16,采用了位运算，效率更高*  **return** h & (length-1);  } } |

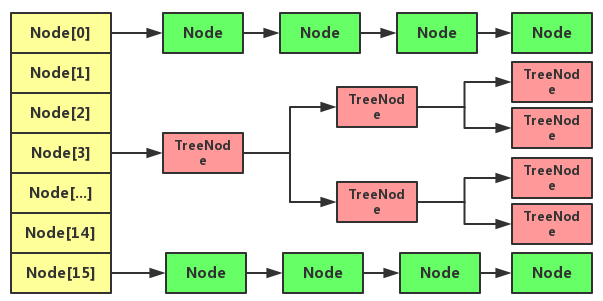
* 调用get方法根据key获取value。
  + 哈希表三步查询数据原理的具体实现
  + 其实是根据key找Entry，再从Entry中获取value即可

|  |
| --- |
| **public** V get(Object key) {  //根据key找到Entry（Entry中有key和value）  Entry<K,V> entry = getEntry(key);  //如果entry== null,返回null，否则返回value  **return null** == entry ? **null** : entry.getValue(); } |

* 添加元素时如达到了阈值，需扩容，每次扩容为原来主数组容量的2倍

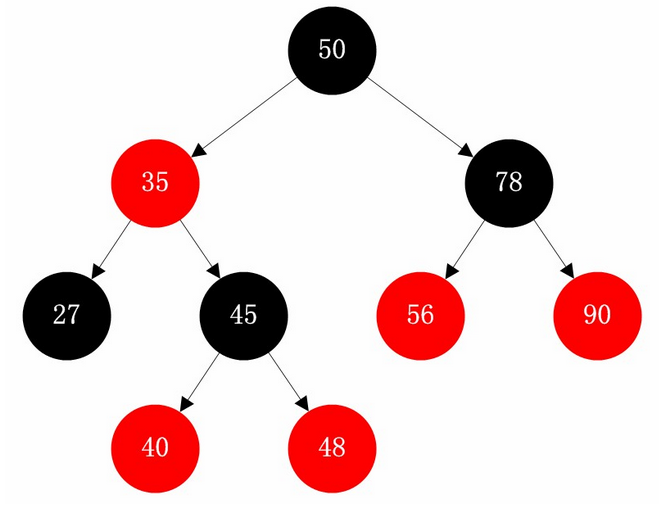
|  |
| --- |
| **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {  *//如果达到了门槛值，就扩容，容量为原来容量的2位 16---32* **if** ((size >= threshold) && (**null** != table[bucketIndex])) {  resize(2 \* table.length);  hash = (**null** != key) ? hash(key) : 0;  bucketIndex = *indexFor*(hash, table.length);  }  *//添加节点* createEntry(hash, key, value, bucketIndex); } |

* 在JDK1.8中有一些变化，当链表的存储数据个数大于等于8的时候，不再采用链表存储，而采用红黑树存储结构。这么做主要是查询的时间复杂度上，链表为O(n)，而红黑树一直是O(logn)。如果冲突多，并且超过8长度小于6 会自动转成链表结构，采用红黑树来提高效率



### 2.2理解 TreeMap的源码

* 基本特征：二叉树、二叉查找树、二叉平衡树、红黑树



* 每个节点的结构



|  |
| --- |
| **static final class** Entry<K,V> **implements** Map.Entry<K,V> {  K **key**;  V **value**;  Entry<K,V> **left**;  Entry<K,V> **right**;  Entry<K,V> **parent**;  **boolean color** = ***BLACK***;  } |

* TreeMap主要的成员变量及其含义

|  |
| --- |
| **public class** TreeMap<K, V> **implements** NavigableMap<K, V> {  **private final** Comparator<? **super** K> **comparator**;*//外部比较器* **private transient** Entry<K, V> **root** = **null**; *//红黑树根节点的引用* **private transient int size** = 0;*//红黑树中节点的个数* **public** TreeMap() {  **comparator** = **null**;*//没有指定外部比较器* }  **public** TreeMap(Comparator<? **super** K> comparator) {  **this**.**comparator** = comparator;*//指定外部比较器* } } |

* 添加原理
  + 从根节点开始比较
  + 添加过程就是构造二叉平衡树的过程，会自动平衡
  + 平衡离不开比较：外部比较器优先，然后是内部比较器。如果两个比较器都没有，就抛出异常

|  |
| --- |
| **public** V put(K key, V value) {  Entry<K,V> t = **root**;  //如果是添加第一个节点，就这么处理  **if** (t == **null**) {  //即使是添加第一个节点，也要使用比较器  compare(key, key); // type (and possibly null) check  //创建根节点  **root** = **new** Entry<>(key, value, **null**);  //此时只有一个节点  **size** = 1;  **return null**;  }  //如果是添加非第一个节点，就这么处理  **int** cmp;  Entry<K,V> parent;   Comparator<? **super** K> cpr = **comparator**;  //如果外部比较器存在，就使用外部比较器  **if** (cpr != **null**) {  **do** {  parent = t;  cmp = cpr.compare(key, t.key);  **if** (cmp < 0)  t = t.left;//在左子树中查找  **else if** (cmp > 0)   t = t.right; //在右子树查找  **else** //找到了对应的key，使用新的value覆盖旧的value   **return** t.setValue(value);  } **while** (t != **null**);  }  **else** {  //如果外部比较器没有，就使用内部比较器  ....  }  //找到了要添加的位置，创建一个新的节点，加入到树中  Entry<K,V> e = **new** Entry<>(key, value, parent);  **if** (cmp < 0)   parent.left = e;  **else** parent.right = e;   **size**++;  **return null**; } |

* 查询原理基本同添加

|  |
| --- |
| **public** V get(Object key) {  *//根据key（cn）找Entry（cn--China）* Entry<K,V> p = getEntry(key);  *//如果Entry存在，返回value：China* **return** (p==**null** ? **null** : p.value); }  **final** Entry<K, V> getEntry(Object key) {  *//如果外部比较器存在，就使用外部比较器* **if** (**comparator** != **null**)  **return** getEntryUsingComparator(key);  **if** (key == **null**)  **throw new** NullPointerException();  @SuppressWarnings(**"unchecked"**)  *//如果外部比较器不存在，就使用内部比较器* Comparable<? **super** K> k = (Comparable<? **super** K>) key;  Entry<K, V> p = **root**;  **while** (p != **null**) {  **int** cmp = k.compareTo(p.key);  **if** (cmp < 0)  p = p.left;  **else if** (cmp > 0)  p = p.right;  **else** *//如果找到了，就返回Entry* **return** p;  }  *//如果没有找到，就返回null* **return null**; } |

### 2.3理解 HashSet的源码

* HashSet的底层使用的是HashMap，所以底层结构也是哈希表
* HashSet的元素到HashMap中做key，value统一是同一个Object()

|  |
| --- |
| **public class** HashSet<E> **implements** Set<E> {  **private transient** HashMap<E, Object> **map**;  **private static final** Object ***PRESENT*** = **new** Object();  **public** HashSet() {  **map** = **new** HashMap<>();  }  **public boolean** add(E e) {  **return map**.put(e, **new** Object()) == **null**;  **return map**.put(e, ***PRESENT***) == **null**;  }  **public int** size() {  **return map**.size();  }  **public** Iterator<E> iterator() {  **return map**.keySet().iterator();  } } |

### 2.4理解 TreeSet的源码

* TreeSet的底层使用的是TreeMap，所以底层结构也是红黑树
* TreeSet的元素e是作为TreeMap的key存在的，value统一为同一个 Object()

|  |
| --- |
| **public class** TreeSet<E> **implements** NavigableSet<E> {  *//底层的TreeMap的引用*  **private transient** NavigableMap<E, Object> **m**;**private static final** Object ***PRESENT*** = **new** Object();  **public** TreeSet() {  *//创建TreeSet对象就是创建一个TreeMap对象*  **this**(**new** TreeMap<E, Object>());}  TreeSet(NavigableMap<E, Object> m) {  **this**.**m** = m;  }**public boolean** add(E e) {  **return m**.put(e, ***PRESENT***) == **null**;  }  **public int** size() {  **return m**.size();  } } |

### 本节作业

1. 理解HashMap源码。
2. 理解TreeMap底层源码
3. 理解HashSet和TreeSet的源码

## 三、集合的其他内容

### 3.1 Iterator

* Iterator专门为遍历集合而生，集合并没有提供专门的遍历的方法
* Iterator实际上迭代器设计模式的实现
* Iterator的常用方法
  + boolean hasNext(): 判断是否存在另一个可访问的元素
  + Object next(): 返回要访问的下一个元素
  + void remove(): 删除上次访问返回的对象。
* 哪些集合可以使用Iterator遍历
  + 层次1：Collection、List、Set可以、Map不可以
  + 层次2：提供iterator()方法的就可以将元素交给Iterator;
  + 层次3：实现Iterable接口的集合类都可以使用迭代器遍历
* for-each循环和Iterator的联系
  + for-each循环(遍历集合)时，底层使用的是Iterator
  + 凡是可以使用for-each循环(遍历的集合)，肯定也可以使用Iterator进行遍历
* for-each循环和Iterator的区别
  + for-each还能遍历数组，Iterator只能遍历集合
  + 使用for-each遍历集合时不能删除元素，会抛出异常ConcurrentModificationException使用Iterator遍历合时能删除元素
* Iterator是一个接口，它的实现类在哪里？

在相应的集合实现类中 ,比如在ArrayList中存在一个内部了Itr implements Iterator

* 为什么Iterator不设计成一个类，而是一个接口

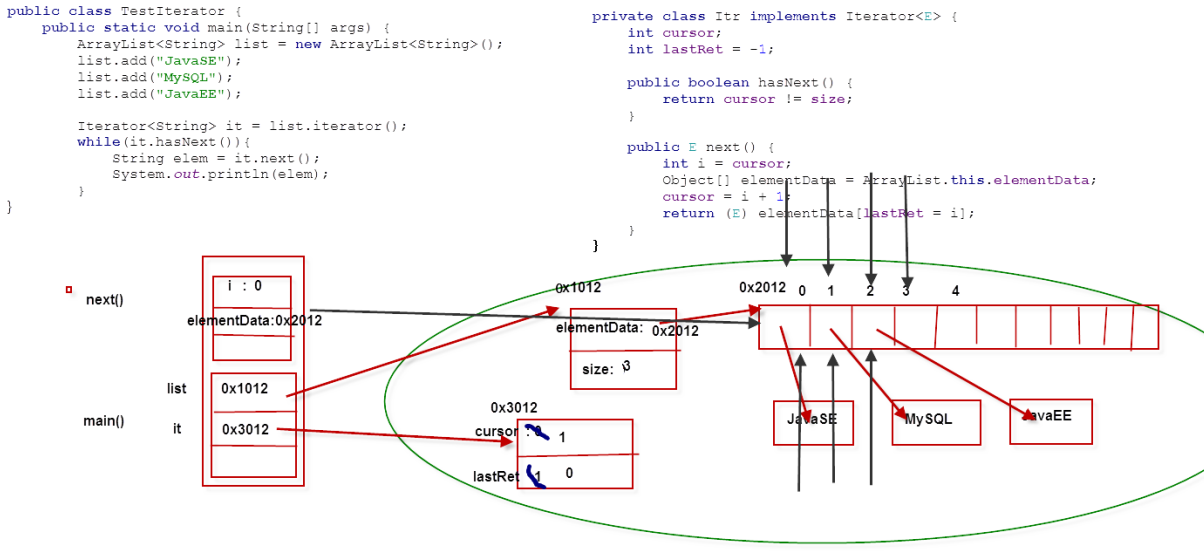
不同的集合类，底层结构不同，迭代的方式不同，所以提供一个接口，让相应的实现类来实现

#### 【示例3】使用Iterator迭代集合

|  |
| --- |
| **public class** TestIterator {  **public static void** main(String[] args) {   //创建一个集合对象  ArrayList<Integer> list = **new** ArrayList<Integer>();   //向集合中添加分数  list.add(78);  list.add(80);  list.add(89);  System.**out**.println(list);  Iterator<Integer> it = list.iterator();  **while**(it.hasNext()){  **int** elem = it.next();  **if**(elem == 78){  //list.remove(new Integer(78));  it.remove();  }  //System.out.println(elem);  }  System.**out**.println(list);  } } |

* Iterator到底是怎么工作的；不同集合的遍历有Iterator的不同的实现类完成，以Iterator遍历ArrayList为例进行说明

|  |
| --- |
| **public** Iterator<E> iterator() {  **return new** Itr(); } **private class** Itr **implements** Iterator<E> {  **int cursor**; **int lastRet** = -1;  **public boolean** hasNext() {  **return cursor** != **size**;  }  **public** E next() {  **int** i = **cursor**;  Object[] elementData = ArrayList.**this**.**elementData**;  **cursor** = i + 1;  **return** (E) elementData[**lastRet** = i];  }  } |



### 3.2 ListIterator

* ListIterator和Iterator的关系
  + public interface ListIterator<E> extends Iterator<E>
  + 都可以遍历List
* ListIterator和Iterator的区别
  + 使用范围不同
    - Iterator可以应用于更多的集合，Set、List和这些集合的子类型。
    - ListIterator只能用于List及其子类型。
  + 遍历顺序不同
    - Iterator只能顺序向后遍历; ListIterator还可以逆序向前遍历
  + Iterator可以在遍历的过程中remove();ListIterator可以在遍历的过程中remove()、add()、set()
  + ListIterator可以定位当前的索引位置，nextIndex()和previousIndex()可以实现。Iterator没有此功能。

#### 【示例4】使用ListIterator迭代List

|  |
| --- |
| **public class** TestListIterator {  **public static void** main(String[] args) {  *//创建一个集合对象* List<Integer> list = **new** ArrayList<Integer>();  *//向集合中添加分数* list.add(78);  list.add(80);  list.add(89);   ListIterator<Integer> lit = list.listIterator();   **while**(lit.hasNext()){  lit.next();  }   **while**(lit.hasPrevious()){  **int** elem = lit.previous();  System.***out***.println(elem +**" "**+lit.nextIndex()  +**" "**+lit.previousIndex());  }  } } |

### 3.3 Collections工具类

* 关于集合操作的工具类，好比Arrays，Math
* 唯一的构造方法private，不允许在类的外部创建对象
* 提供了大量的static方法，可以通过类名直接调用

#### 【示例5】使用Collections简化集合操作

|  |
| --- |
| **public class** TestCollections {  **public static void** main(String[] args) {  //添加元素  List<Integer> list = **new** ArrayList();  Collections.addAll(list, 10, 50, 30, 90, 85, 100);//6  System.**out**.println(list);  //排序  Collections.sort(list);//默认按照内部比较器  System.**out**.println(list);  //查找元素(元素必须有序)  **int** index = Collections.binarySearch(list, 500);//不存在返回负数  System.**out**.println(index);  //获取最大值和最小值  **int** max = Collections.max(list);  **int** min = Collections.min(list);  System.**out**.println(max + **" "** + min);  //填充集合  Collections.fill(list, **null**);  System.**out**.println(list);  //复制集合  List list2 = **new** ArrayList();  Collections.addAll(list2, 10, 20, 30, 50);  System.**out**.println(list2);  Collections.copy(list, list2);//dest.size >= src.size 目标列表的长度至少必须等于源列表。  System.**out**.println(list);  //同步集合  //StringBuffer 线程安全效率低 StringBuilder 线程不安全，效率高  //Vector 线程安全 效率低 ArrayList 线程不安全，效率高  //难道是要性能不要安全吗，肯定不是。  //在没有线程安全要求的情况下可以使用ArrayList  //如果遇到了线程安全的情况怎么办  //方法1：程序员手动的将不安全的变成安全的   //方法2：提供最新的线程安全并且性能高的集合类  List list3 = **new** ArrayList();  Collections.addAll(list3, 10, 90, 30, 40, 50, 23);  System.**out**.println(list3);  //将list3转换成线程安全的集合类  list3 = Collections.synchronizedList(list3);  //下面再操作，就线程安全了  } } |

### 3.4旧的集合类

* Vector
  + 实现原理和ArrayList相同，功能相同，都是长度可变的数组结构，很多情况下可以互用
  + 两者的主要区别如下
    - Vector是早期JDK接口，ArrayList是替代Vector的新接口
    - Vector线程安全，效率低下；ArrayList重速度轻安全，线程非安全
    - 长度需增长时，Vector默认增长一倍，ArrayList增长50%
* Hashtable类
  + 实现原理和HashMap相同，功能相同，底层都是哈希表结构，查询速度快，很多情况下可互用
  + 两者的主要区别如下
    - Hashtable是早期JDK提供，HashMap是新版JDK提供
    - Hashtable继承Dictionary类，HashMap实现Map接口
    - Hashtable线程安全，HashMap线程非安全
    - Hashtable不允许null值，HashMap允许null值

#### 【示例6】使用Vector类进行集合操作

|  |
| --- |
| **public class** TestVector {  **public static void** main(String[] args) {  *//泛型是1.5开始的，重新改写了Vector，ArrayList* Vector<Integer> v = **new** Vector<Integer>();   v.addElement(123);  v.addElement(456);  v.addElement(345);  v.addElement(100);   Enumeration<Integer> en = v.elements();  **while**(en.hasMoreElements()){  Integer elem = en.nextElement();  System.***out***.println(elem);  }  } } |

### 3.5新一代并发集合类

#### 3.5.1 集合类的发展历程

* 早期集合类Vector、Hashtable：线程安全的。是怎么保证线程安排的，使用synchronized修饰方法
* 为了提高性能，使用ArrayList、HashMap替换，线程不安全，但是性能好。使用ArrayList、HashMap，需要线程安全怎么办呢？

使用Collections.synchronizedList(list)、Collections.synchronizedMap(m)解决，底层使用synchronized代码块锁。

虽然也是锁住了所有的代码，但是锁在方法里边，并所在方法外边性能可以理解为稍有提高吧。毕竟进方法本身就要分配资源的

* 在大量并发情况下如何提高集合的效率和安全呢？

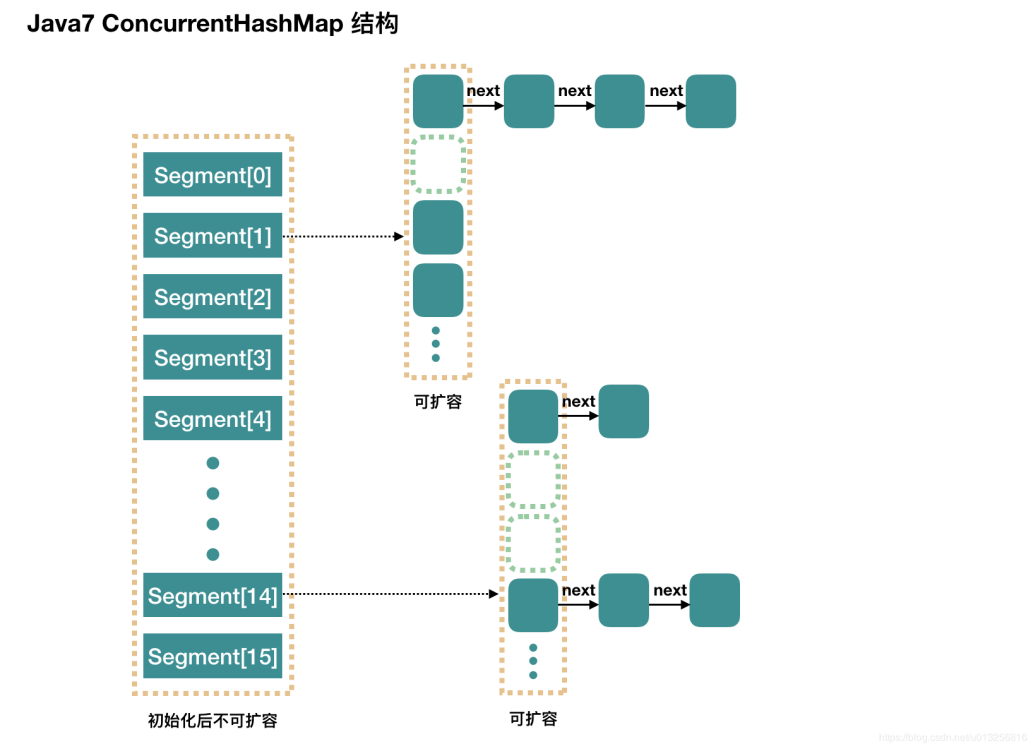
提供了新的线程同步集合类，位于java.util.concurrent包下，使用Lock锁或者volatile+CAS的无锁化。

* + ConcurrentHashMap
  + CopyOnWriteArrayList
  + CopyOnWriteArraySet：

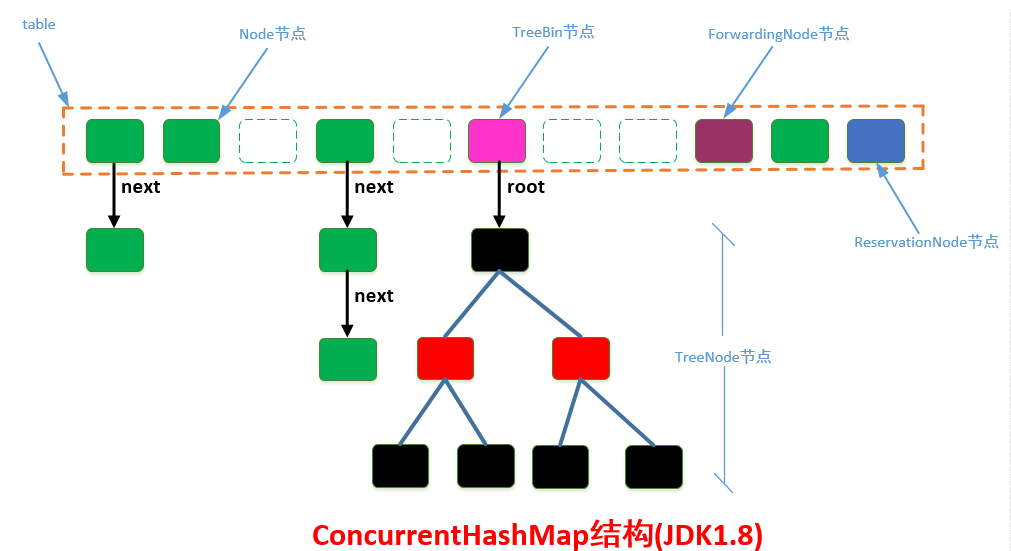
#### 3.5.2 新一代并发集合类

* **ConcurrentHashMap: 分段（segment）锁定+Lock锁**

HashMap的线程安全班，性能比Hashtable、Collections.*synchronizedMap*(m);都有提高。使用的不是synchronized代码块锁，也不是synchronzied方法锁。并且使用了锁分离技术，**使用多个锁来控制对hash表的不同部分（段segment）进行的修改**，采用ReentrantLock锁来实现。如果多个修改操作发生在不同的段上，他们就可以并发进行，从而提高了效率。JDK1.7和JDK1.8的关于ConcurrentHashMap的实现差异较大，以上理论属于JDK1.7。



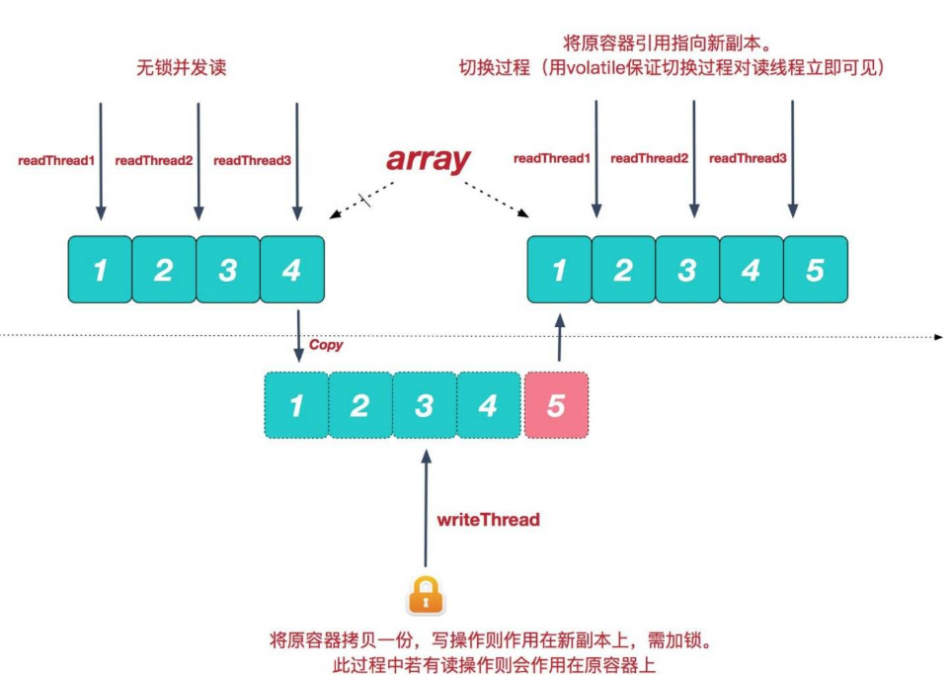
Concurr**entHashMap在JDK8中进行了巨大改动。**它摒弃了Segment（锁段）的概念，而是启用了一种全新的方式实现,利用 volatile + CAS 实现无锁化操作。 它底层由"数组"+链表+红黑树的方式思想(JDK8中HashMap的实现)， 为了做到并发，又增加了很多辅助的类，例如TreeBin，Traverser等对象内部类。



* **CopyOnWriteArrayList ：CopyOnWrite+Lock锁**

对于set()、add()、remove()等方法使用ReentrantLock的lock和unlock来加锁和解锁。读操作不需要加锁（之前集合安全类，即使读操作也要加锁，保证数据的实时一致）。

|  |
| --- |
| **CopyOnWrite原理：写时复制**。  通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，  添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。  对于读操作远远多于写操作的应用非常适合，特别在并发情况下，可以提供高性能的并发读取。  CopyOnWrite容器只能保证数据的最终一致性，不能保证数据实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。 |



|  |
| --- |
| **public boolean** add(E e) {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  Object[] elements = getArray();  **int** len = elements.**length**;  // 复制出新数组  Object[] newElements = Arrays.*copyOf*(elements, len + 1);  // 把新元素添加到新数组里  newElements[len] = e;  // 把原数组引用指向新数组  setArray(newElements);  **return true**;  } **finally** {  lock.unlock();  } }  **final void** setArray(Object[] a) {  **array** = a; }  **public** E get(**int** index) {  **return** get(getArray(), index); } |

注意：读的时候不需要加锁，如果读的时候有多个线程正在向ArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的ArrayList

* **CopyOnWriteArraySet：CopyOnWrite+Lock锁**

它是线程安全的无序的集合，可以将它理解成线程安全的HashSet。 有意思的是，CopyOnWriteArraySet和HashSet虽然都继承于共同的父类AbstractSet； 但是，HashSet是通过"散列表(HashMap)"实现的，而CopyOnWriteArraySet则是通过"动态数组(CopyOnWriteArrayList)"实现的，并不是散列表。

CopyOnWriteArraySet在CopyOnWriteArrayList 的基础上使用了Java的装饰模式，所以底层是相同的。而CopyOnWriteArrayList本质是个动态数组队列，所以CopyOnWriteArraySet相当于通过通过动态数组实现的"集合"！

CopyOnWriteArrayList中允许有重复的元素；但CopyOnWriteArraySet是一个集合，所以它不能有重复集合。因此，CopyOnWriteArrayList额外提供了addIfAbsent()和addAllAbsent()这两个添加元素的API，通过这些API来添加元素时，只有当元素不存在时才执行添加操作！

### 3.6集合常用概念辨析

**集合和数组的比较**

数组不是面向对象的，存在明显的缺陷，集合完全弥补了数组的一些缺点，比数组更灵活更实用，可大大提高软件的开发效率而且不同的集合框架类可适用于不同场合。具体如下：

1 : 数组容量固定且无法动态改变，集合类容量动态改变。

2：数组能存放基本数据类型和引用数据类型的数据，而集合类中只能放引用数据类型的数据。

3：数组无法判断其中实际存有多少元素，length只告诉了array容量；集合可以判断实际存有多少元素，而对总的容量不关心

4：集合有多种数据结构（顺序表、链表、哈希表、树等）、多种特征（是否有序，是否唯一）、不同适用场合（查询快，便于删除、有序），不像数组仅采用顺序表方式

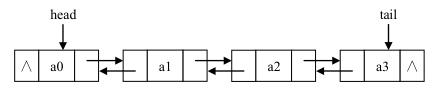
5：集合以类的形式存在，具有封装、继承、多态等类的特性，通过简单的方法和属性调用即可实现各种复杂操作，大大提高软件的开发效率。

**ArrayList和LinkedList 的联系和区别**

* 联系：
  + 都实现了List接口
  + 有序 不唯一（可重复）
* ArrayList
  + 特点：在内存中分配连续的空间，实现了长度可变的数组
  + 优点：遍历元素和随机访问元素的效率比较高
  + 缺点：添加和删除需大量移动元素效率低，按照内容查询效率低

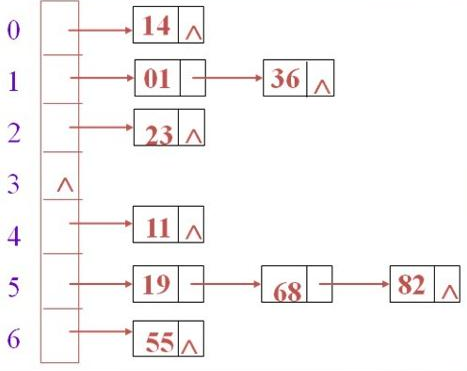
Snap4

* LinkedList
  + 特点：采用链表存储方式，底层是双向链表。
  + 缺点：遍历和随机访问元素效率低下
  + 优点：插入、删除元素效率比较高（但是前提也是必须先低效率查询才可。如果插入删除发生在头尾可以减少查询次数）



**哈希表的原理（ HashMap的底层原理）**

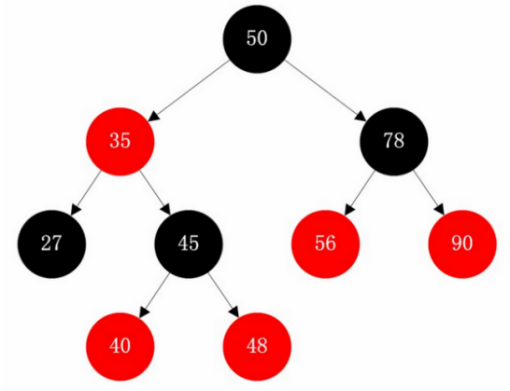
* 哈希表的特征
  + 快：查询快、添加快
* 哈希表的结构
  + 最常用、最容易理解的结构（JDK1.7）:数组+链表
  + JDK1.8:数组+链表/红黑树（链表长度>=8）



* 哈希表添加原理
  + 计算哈希码 （hashCode()）；
  + 计算存储位置（存储位置就是数组的索引）；
  + 存入指定位置（要处理冲突，可能重复。需借助equals()进行比较）
* 哈希表查询原理
  + 同哈希表添加原理
* 其他
  + hashCode()和equals()的作用；
  + 如何减少冲突；
  + 如何生产不同数据类型的哈希码

**TreeMap的底层原理（红黑树的底层原理）**

* 基本特征
  + 二叉树、二叉查找树、二叉平衡树、红黑树



* 每个节点的结构



* 添加原理
  + 从根节点开始比较
  + 添加过程就是构造二叉平衡树的过程，会自动平衡
  + 平衡离不开比较：外部比较器优先，然后是内部比较器，否则出错
* 查询原理基本同添加

**Collection和Collections的区别**

* Collection是Java提供的集合接口，存储一组不唯一，无序的对象。它有两个子接口List和Set。
* Java中还有一个Collections类，专门用来操作集合类 ，它提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

**Vector和ArrayList的联系和区别**

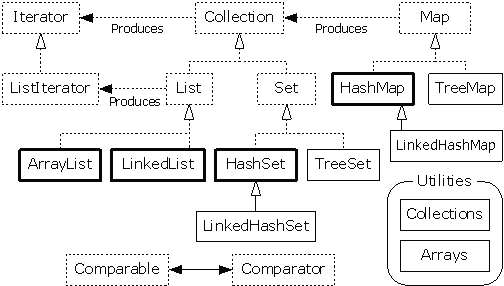
实现原理和ArrayList相同，功能相同，都是长度可变的数组结构，很多情况下可互用

* 两者的主要区别如下
  + Vector是早期JDK接口，ArrayList是替代Vector的新接口
  + Vector线程安全效率低下；ArrayList重速度轻安全，线程非安全
  + 长度需增长时，Vector默认增长一倍，ArrayList增长50%

**HashMap和Hashtable的联系和区别**

* 实现原理相同，功能相同，底层都是哈希表，查询速度快，在很多情况下可互用
* 两者的主要区别如下
  + Hashtable是早期JDK提供的接口，HashMap是新版JDK提供的接口
  + Hashtable继承Dictionary类，HashMap实现Map接口
  + Hashtable线程安全，HashMap线程非安全
  + Hashtable不允许null值，HashMap允许null值

**各种集合类的特点**

****

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **存储结构** | **顺序** | **唯一** | **查询** | **添加/删除** |
| **ArrayList** | 顺序表 | 有序（添加） | 不唯一 | 索引查询效率最高 | 效率低 |
| **LinkedList** | 双向链表 | 有序（添加） | 不唯一 | 效率低 | 效率高 |
| **HashSet** | 哈希表 | 无序 | 唯一 | 效率最高 | 效率最高 |
| **HashMap** | 哈希表 | Key无序 | key唯一 | 效率最高 | 效率最高 |
| **LinkedHashSet** | 哈希表+链表 | 有序(添加) | 唯一 | 效率最高 | 效率最高 |
| **LinkedHashMap** | 哈希表+链表 | Key有序（添加） | key唯一 | 效率最高 | 效率最高 |
| **TreeSet** | 红黑树 | 有序（自然） | 唯一 | 效率中等 | 效率中等 |
| **TreeMap** | 红黑树 | 有序（自然） | key唯一 | 效率中等 | 效率中等 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Collection** | | **Map** | |
| **无序 不唯一** | Collection | | Map.values() |  |
| **有序 不唯一** | ArrayList | LinkedList |  |  |
| **无序 唯一** | HashSet |  | HashMap keySet |  |
| **有序 唯一** | LinkedHashSet | TreeSet | LinkedHashMap keySet | TreeMap keySet |

### 本节作业

1. for-each循环和Iterator的联系和区别
2. Iterator迭代ArrayList的源码分析
3. Iterator和ListIterator的联系和区别
4. 使用Collections类简化集合操作
5. 主要的新一代并发集合类及其特点
6. 案例1：添加和查看商品
7. 使用集合模拟数据库
8. 添加商品
9. 查看商品
10. 案例2：完成购物车的添加和查看操作
11. 购物车条目CartItem
12. 购物车
13. 完成添加操作
14. 完成查看操作

## 四、泛型

### 4.1为什么需要泛型

没有采用泛型之前

1.不安全：添加元素是无检查 宽进

2.繁琐：获取元素时需要强制类型转换 严出

采用泛型之后

1.安全 严进

2.简单 宽出

### 4.2什么是泛型generic

JDK1.5引入。泛型，即"参数化类型"。

方法定义和调用：定义方法时有形参，然后调用此方法时传递实参。  
**定义方法**：

public int add(int num1,int num2){ //形参

return num1+num2;

}

**调用方法**：

add(10,20) //实参

add(100,200) //实参

**泛型类的定义和创建对象**：  
public class ArrayList<T> implements List<T> {} 类型形参  
List<String> list = new ArrayList<String>(); 类型实参

List<Date> list = new ArrayList<Date>(); 类型实参

### 4.3泛型内容

1).泛型接口

#### 【示例7】泛型接口

|  |
| --- |
| **public interface** Comparable<T> {  **public int** compareTo(T obj); }  **public interface** List<E> {  **public void** add(E obj);  **public** E get(**int** index); } |

2).泛型类

#### 【示例8】泛型类

|  |
| --- |
| **public class** ArrayList<E> **implements** List<E> {  **private** E **elem**;  @Override  **public void** add(E elem) {  **this**.**elem** = elem;  }  @Override  **public** E get(**int** index) {  **return null**;  }  **public static** <E> **void** method1(E e){  }  **public static void** main(String[] args) {  ArrayList.*method1*(**null**);  ArrayList <Student> list1 = **new** ArrayList<Student>();  ArrayList <String> list2 = **new** ArrayList<String>();  ArrayList <Integer>list3 = **new** ArrayList<Integer>();  } }  **public class** Student **implements** Comparable<Student>{  **private int sno**;  **private** String **name**;  **public int** compareTo(Student obj) {  **return** 0;  } } |

3).泛型方法

泛型类中方法，即使使用了泛型，也不是泛型方法；泛型方法将泛型定义在方法上。

静态方法无法访问类上定义的泛型；如果静态方法操作的引用数据类型不确定的时候，必须要将泛型定义在方法上。

#### 【示例9】泛型方法

|  |
| --- |
| **public class** Collections {  **public** <T> **void** addAll( T t){  }  //泛型方法  **public static** <T> **void** method1(T t){  }  } |

4).上限和下限通配符

* <? extends E> 上限通配符，用来限制类型的上限
* <? super E> 下限通配符，用来限制类型的下限
* 注意了，此处’？’是类型实参，而不是类型形参 。可以解决当具体类型不确定的时候，这个通配符就是 ?

#### 【示例10】上限和下限通配符

|  |
| --- |
| **public class** Collections {  **public** <T> **void** addAll( T t){ }  **public static** <T> **void** method1(T t){ }  **public static** <E>**void** method2(List<E> list){ }  **public static** <E>**void** method3(List<Teacher> list){ }  **public static** <E>**void** method4(List<? **super** Teacher> list){ //下限  }  **public static** <E>**void** method5(List<? **extends** Teacher> list){ //上限  }  **public static** <E> **void** method6(List<?> list){ //下限  }  **public static void** main(String[] args) {  *method2*(**new** ArrayList<String>());  *method2*(**new** ArrayList<Person>());  *method2*(**new** ArrayList<Teacher>());  *method2*(**new** ArrayList<MiddleTeacher>());  //method3(new ArrayList<String>());  //method3(new ArrayList<Person>());  *method3*(**new** ArrayList<Teacher>());  //method3(new ArrayList<MiddleTeacher>());   *method4*(**new** ArrayList<Object>());  *method4*(**new** ArrayList<Person>());  *method4*(**new** ArrayList<Teacher>());  //method4(new ArrayList<MiddleTeacher>());   //method5(new ArrayList<Person>());  *method5*(**new** ArrayList<Teacher>());  *method5*(**new** ArrayList<MiddleTeacher>());   *method6*(**new** ArrayList<String>());  *method6*(**new** ArrayList<Person>());  *method6*(**new** ArrayList<Teacher>());  *method6*(**new** ArrayList<MiddleTeacher>());  } } **class** Person{ } **class** Teacher **extends** Person{ } **class** MiddleTeacher **extends** Teacher{} |

### 4.4.泛型注意事项

* 在于Java中的泛型这一概念提出的目的，导致其只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦出。也就是说，成功编译过后的class文件中是不包含泛型信息的。泛型信息不会全部进入到运行时阶段。
* 泛型类，是在实例化类的时候指明泛型的具体类型；泛型方法，是在调用方法的时候指明泛型的具体类型

### 本节作业

1. 泛型的好处
2. 练习泛型类、泛型接口、泛型方法的定义

## 五、JDK8新特征

### 5.1 JDK版本变化

|  |  |  |
| --- | --- | --- |
| **JDK版本** | **名称** | **发布时间** |
| 1.0 | Oak(橡树) | 1996/1/23 |
| 1.1 |  | 1997/2/19 |
| 1.2 | Playground（运动场） | 1998/12/4 |
| 1.3 | Kestrel（美洲红隼） | 2000/5/8 |
| 1.4.0 | Merlin（灰背隼） | 2002/2/13 |
| Java SE 5.0 / 1.5 | Tiger（老虎） | 2004/9/30 |
| Java SE 6.0 / 1.6 | Mustang（野马） | 2006/4/1 |
| Java SE 7.0 / 1.7 | Dolphin（海豚） | 2011/7/28 |
| Java SE 8.0 / 1.8 | Spider（蜘蛛） | 2014/3/18 |
| Java SE 9.0 |  | 2017/9/21 |
| Java SE 10.0 |  | 2018/3/21 |
| Java SE 11.0 |  | 2018/9/25 |
| Java SE 12.0 |  | 2019/3/19 |
| Java SE 13.0 |  | 2019/9/17 |
| Java SE 14.0 |  | 2020/3/17 |

2009年4月20日晚，甲骨文和Sun宣布，两家公司已达成正式收购协议。根据协议，甲骨文将以每股9.5美元的价格收购Sun，交易总价值约为74亿美元。所以Java7及其后面的版本都是Oracle公司推出的。

从Java9这个版本开始，Java 的计划发布周期是6个月。这意味着Java的更新从传统的以特性驱动的发布周期，转变为以时间驱动的发布模式，并逐步的将Oracle JDK原商业特性进行开源。

理念：小步快跑，快速迭代。后面周期变短，对之前的东西固定下来保存下来抛弃一些，版本趋于稳定，更新内容新特性就少。使用者就像小白鼠，我们相当于测试了，有一些功能优缺点进行测试和反馈，测试完它就改动，他掌握用户需求就可以改。针对企业客户的需求，Oracle将以三年为周期发布长期支持版本(long term support)。

其中Java5，Java8是革命性的版本。Java11也将提供长期的支持。Java9,10都是“功能性的版本”，支持时间只有半年左右。**特别是JDK8是JDK5以来最具革命性的版本**。主要新特征包括Lambda表达式、函数式接口、方法引用和构造器引用、Stream API、新日期类、接口新变化等。其中的新日期类、接口新变化在前面章节已经讲解。

总的来说,JDK8中的Lambda表达式和Stream 是自Java语言添加泛型(Generics)和注解(annotation)以来最大的变化。

### 5.2 Lambda表达式

#### 5.2.1引入Lambda表达式

#### 【示例11】引入Lambda表达式

|  |
| --- |
| **public interface** MyInterface {  **public void** method(); } **class** MyClass **implements** MyInterface{  **public void** method() {  System.***out***.println(**"使用类实现接口"**);  } } **class** Test{  **public static void** main(String[] args) {  //1.定义类实现接口  MyInterface myInferface1 = **new** MyClass();  myInferface1.method();  //2.使用匿名内部类  MyInterface myInferface2 = **new** MyInterface() {  **public void** method() {  System.***out***.println(**"使用匿名内部类实现类"**);  }  };  myInferface2.method();  //3.使用Lambda表达式  MyInterface myInferface3 = ()->System.***out***.println(**"使用Lambda表达式"**);  myInferface3.method();  } } |

**Lambda表达式：**

Lambda表达式基于数学中的λ演算得名，直接对应于其中的lambda抽象，是一个匿名函数，即没有函数名的函数。

**Lambda表达式好处**：

使用 Lambda 表达式可使代码变的更加简洁紧凑。并且Lambda表达式可和Stream API等相结合，使代码更加简洁紧凑。Lambda 表达式经常用来替代部分匿名内部类。

**Lambda表达式的语法**

(parameters) -> expression或 (parameters) ->{ statements; }

参数：要重写的方法的形参列表

-> ：lambda运算符

表达式/语句体：要重写的方法的方法体

**Lambda表达式的本质：**Lambda 表达式是一种匿名函数（不是匿名内部类），简单地说，它是没有声明的方法，也即没有访问修饰符、返回值声明和名字。它实质属于函数式编程的概念。（不是匿名内部类的，此点在此不做展开）。

#### 5.2.2 使用Lambda表达式

1. **无参数无返回值**

#### 【示例12】无参数无返回值的Lambda表达式

|  |
| --- |
| **public interface** MyInterface {  **public void** method(); } **class** Test1{  **public static void** main(String[] args) {  MyInterface myInterface = ()->{System.***out***.println(**"无参数无返回值,一条语句"**);};  myInterface.method();  //一条语句：{}可以省略  myInterface = ()->System.***out***.println(**"无参数无返回值,一条语句,省略{}"**);  myInterface.method();  //多条语句：{}不可以省略  /\*final\*/ **int** n = 5;  MyInterface myInferface2 = ()->{  System.***out***.println(**"无参数无返回值"**);  //n = 6;  System.***out***.println(n);  };  myInferface2.method();  } } |

注意：

* 如果Lambda表达式的语句体只有一条语句，{}可以省略；如多条语句，{}不可以省略。
* Lambda 表达式只能引用标记了 final 的外层局部变量。lambda 表达式的局部变量可以不用声明为 final，但是必须不可被后面的代码修改（即隐性的具有 final 的语义）。

1. **一个参数无返回值**

#### 【示例13】一个参数无返回值的Lambda表达式

|  |
| --- |
| **public interface** MyInterface2 {  **public void** method(**int** num1); } **class** Test2{  **public static void** main(String[] args) {  MyInterface2 myInterface = (**int** x)->System.***out***.println(x);  myInterface.method(10);  //类型可以省略  myInterface = (x)->System.***out***.println(x);  myInterface.method(20);  //()可以省略  **myInterface = x->System.*out*.println(x);**  myInterface.method(30);  myInterface = x->{  System.***out***.println(**"-----"**);  System.***out***.println(x);  System.***out***.println(**"-----"**);  };  myInterface.method(40);  } } |

注意：

* Lambda表达式的参数类型可以省略，参数名可以是任意名称；
* 如果只有一个参数,参数外的()也可以省略。

1. **多个参数无返回值**

#### 【示例14】多个参数无返回值的Lambda表达式

|  |
| --- |
| **public interface** MyInterface3 {  **public void** method(**int** num,String str); } **class** Test3{  **public static void** main(String[] args) {  MyInterface3 myInterface=(**int** num,String str)-> System.***out***.println(num+**" "**+str);  myInterface.method(20,**"bjsxt"**);  //参数的小括号不能省略  myInterface = (num,str)-> System.***out***.println(num+**" "**+str);  myInterface.method(20,**"bjsxt"**);  myInterface = (num,str)-> {  System.***out***.println(**"----------"**);  System.***out***.println(num+**" "**+str);  System.***out***.println(**"----------"**);  };  myInterface.method(20,**"bjsxt"**);  } } |

注意：

* Lambda表达式如果多个参数，参数外的 （）不可省略。

1. **有参数有返回值**

#### 【示例15】有参数有返回值的Lambda表达式

|  |
| --- |
| **public interface** MyInterface4 {  **public int** method(**int** num); } **class** Test4{  **public static void** main(String[] args) {  MyInterface4 myInterface = (x)-> {**return** x+10; };  **int** result = myInterface.method(10);  System.***out***.println(result);  //只有一条return语句,{}和return可以均省去  **myInterface = (x)-> x+10;**  result = myInterface.method(10);  System.***out***.println(result);  myInterface = (x)-> {  x+=10;  **return** x+10;  };  result = myInterface.method(10);  System.***out***.println(result);  } } |

注意：

* 有返回值的Lambda表达式，如果方法体只有一条语句，可同时省略return和{}。
* 虽然使用 Lambda 表达式可以对某些接口进行简单的实现，但并不是所有的接口都可以使用 Lambda 表达式来实现。**Lambda 规定接口中只能有一个需要被实现的抽象方法，不是规定接口中只能有一个方法，称为函数式接口**。

**总结：**

->左面：

* 参数类型可以省略不写！->类型推断
* 如果只有一个参数，()可以省略不写

->右侧：

* {}将方法体的具体内容包裹起来
* 只有一个方法体执行语句的话，{}可以省略不写
* 如果一句执行语句是return语句，return可以省略不写。

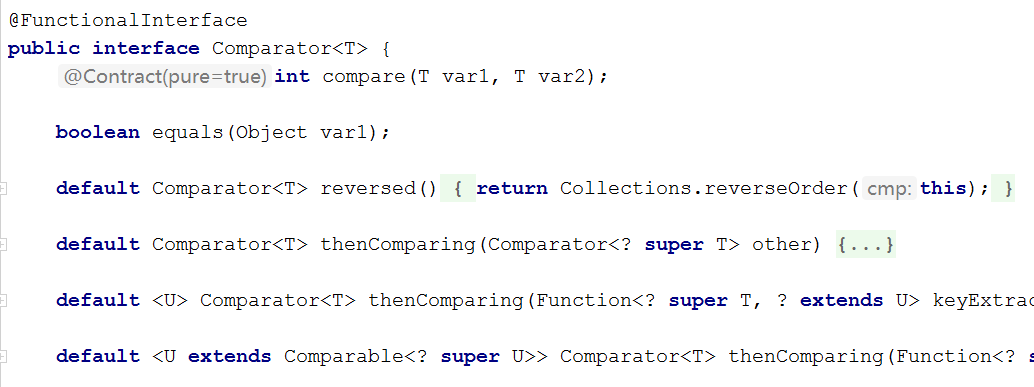
### 5.2函数式接口

#### 5.2.1 认识函数式接口

函数式接口：只能有一个抽象方法，其他的可以有default、static、Object里public方法等。作用：在Java中主要用在Lambda表达式和方法引用（实际上也可认为是Lambda表达式）上。

JDK8专门提供了@FunctionalInterface注解，用来进行编译检查。

已经使用过的函数式接口，比如Comparator等，后面在多线程阶段要学习的函数式接口有Runnable、Callable等。（注意：Comparable并没有被标记为函数式接口）



#### 【示例16】认识函数式接口

|  |
| --- |
| @FunctionalInterface **public interface** FuncInterface {  //只有一个抽象方法  **public void** method1();  //default方法不计  **default void** method2(){  }  //static方法不计  **static void** method3(){  }  //从Object过来的public方法不计  **public boolean** equals(Object obj); } |

#### 5.2.2内置函数式接口。

JDK 也提供了大量的内置函数式接口，使得 Lambda 表达式的运用更加方便、高效。这些内置的函数式接口已经可以解决我们开发过程中绝大部分的问题，只有一小部分比较特殊得情况需要我们自己去定义函数式接口。在这里特别介绍四个函数式接口。

* Consumer<T>：消费型接口（void accept(T t)）。有参数，无返回值
* Supplier<T>：供给型接口（T get（））。只有返回值，没有入参
* Function<T, R>：函数型接口（R apply（T t））。一个输入参数，一个输出参数，两种类型不可不同、可以一致
* Predicate<T>：断言型接口（boolean test（T t））。输入一个参数，输出一个boolean类型得返回值

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函数式接口 | 方法名 | 输入参数 | 输出参数 | 作用 |
| 消费型接口  Consumer | void accept(T t) | T | void | 对类型为T的对象进行操作 |
| 供给型接口  Supplier | T get（） | void | T | 返回类型为T的对象 |
| 函数型接口  Function | R apply（T t） | T | R | 对类型为T的对象进行操作，返回类型为R的对象 |
| 断言型接口  Predicate | boolean test（T t） | T | boolean | 对类型为T的对象进行操作，返回布尔类型结果 |

1. **消费型接口Consumer<T>**

#### 【示例17】消费型接口

|  |
| --- |
| **public class** TestFunctional1 {  **public static void** main(String[] args) {  List<Integer > list = **new** ArrayList<>();  Collections.*addAll*(list,34,56,89,65,87);  //使用匿名内部类实现  Consumer consumer = **new** Consumer<Integer>() {  @Override  **public void** accept(Integer elem) {  System.***out***.println(elem);  }  };  list.forEach(consumer);  //使用Lambda表达式  //list.forEach((elem)->{System.out.println(elem);});  list.forEach((elem)->System.***out***.println(elem));  } } |

1. **断言型接口Predicate<T>**

#### 【示例18】断言型接口1

|  |
| --- |
| **public class** TestFunctional2 {  **public static void** main(String[] args) {  List<Integer > list = **new** ArrayList<>();  Collections.*addAll*(list,34,56,89,65,87);  //使用匿名内部类实现  System.***out***.println(list);  Predicate predicate = **new** Predicate<Integer>(){  @Override  **public boolean** test(Integer i) {  **if**(i<60){  **return true**;  }  **return false**;  }  };  list.removeIf(predicate);  System.***out***.println(list);  //使用Lambda表达式实现  list.removeIf((i)->{**if**(i>80) {  **return true**;  }  **return false**;  });  System.***out***.println(list);  } } |

#### 【示例19】断言型接口2

|  |
| --- |
| **public class** TestFunctional3 {  **public static void** main(String[] args) {  List<String > list = **new** ArrayList<>();  Collections.*addAll*(list,**"Java"**,**"MySQL"**,**"HTML"**,**"JSP"**,**"SSM"**);  System.***out***.println(list);   **int** length=4;  Predicate <String>predicate =  (elem)->{**if**(elem.length()>=length ) **return true**; **return false**;};  List<String> list2 = *retailIf*(predicate,list);  System.***out***.println(list2);  }  **public static** List retailIf(Predicate predicate,List<String> list){  List<String> list2 = **new** ArrayList<>();  **for** (String elem :list){  **if**(predicate.test(elem)){  list2.add(elem);  }  }  **return** list2;  } } |

### 5.3方法引用

有时候，Lambda体可能仅调用一个已存在方法，而不做任何其它事，对于这种情况，通过一个方法名字来引用这个已存在的方法会更加清晰。方法引用是一个更加紧凑，易读的 Lambda 表达式，注意方法引用是一个 Lambda 表达式，方法引用操作符是双冒号 "::"。

#### 【示例20】方法引用1

|  |
| --- |
| **public class** TestMethodRef1 {  **public static void** main(String[] args) {  //使用匿名内部类实现  Consumer consumer = **new** Consumer<Integer>() {  @Override  **public void** accept(Integer i) {  System.***out***.println(i);  }  };  consumer.accept(56);  //使用lambda表达式实现  Consumer<Integer> consumer1 = (i)->System.***out***.println(i);  consumer1.accept(56);  //使用方法引用  //println()的参数类型、返回值类型正好和accept方法的参数类型、返回值类型相同  Consumer<Integer> consumer2 = System.***out***::println;  consumer2.accept(56);  } } |

在本示例中，println()的参数类型、返回值类型正好和Consumer接口的accept方法的参数类型、返回值类型相同，此时可以采用方法引用来简化语法。语法为对象名::实例方法名。

#### 【示例21】方法引用2

|  |
| --- |
| **public class** TestMethodRef2 {  **public static void** main(String[] args) {  Student stu = **new** Student(10,**"zhangsan"**,23,100);  //使用匿名内部类实现  Supplier<String> supplier1= **new** Supplier<String>(){  **public** String get() {  **return** stu.getName();  }  };  System.***out***.println(supplier1.get());  //使用Lambda表达式实现  Supplier<String> supplier2 = ()-> stu.getName();  System.***out***.println(supplier2.get());  //使用方法引用实现  Supplier<String> supplier3 = stu::getName;  System.***out***.println(supplier3.get());  } } |

#### 【示例22】方法引用3

|  |
| --- |
| **public class** TestMethodRef3 {  **public static void** main(String[] args) {  //使用匿名内部类实现  Comparator comparator1 = **new** Comparator<Integer>() {  **public int** compare(Integer in1, Integer in2) {  //return in1.intValue()-in2.intValue();  **return** Integer.*compare*(in1,in2);  }  };  System.***out***.println(comparator1.compare(12,34));  //使用Lambda表达式实现  Comparator<Integer> comparator2 = (in1,in2)->{ **return** Integer.*compare*(in1,in2);};  System.***out***.println(comparator2.compare(12,34));  //使用方法引用实现  Comparator<Integer> comparator3 =Integer::*compare*;  System.***out***.println(comparator3.compare(12,34));  } } |

**方法引用有下面几种方式：**

1. 对象引用::实例方法名
2. 类名::静态方法名
3. 类名::实例方法名
4. 类名::new （也称为构造方法引用）
5. 类型[]::new （也称为数组引用）

这里讲解了第一种方式（System.***out***::println、stu::getName）和第二种方式（Integer::*compare*）， 对于另外三种形式，感兴趣的同学可以自己查询资料进行学习。

### 5.4流式编程

Stream作为Java8的一大亮点，它与java.io包里的InputStream和OutputStream是完全不同的概念。它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、高效的聚合操作或者大批量数据操作。

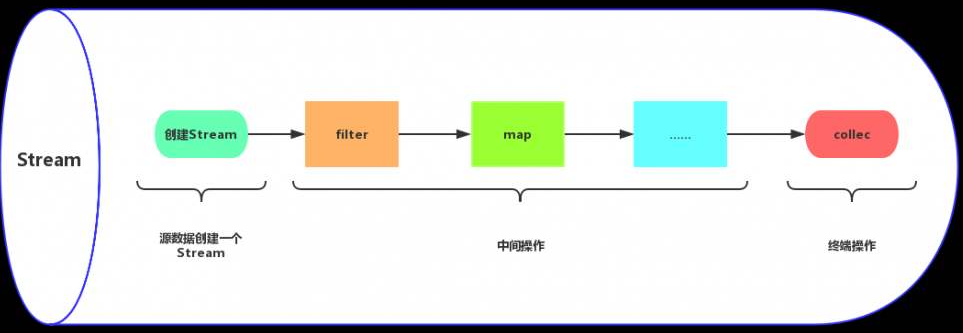
Stream API借助于同样新出现的Lambda表达式，极大的提高编程效率和程序可读性。同时，它提供串行和并行两种模式进行汇聚操作，并发模式能够充分利用多核处理器的优势，使用fork/join并行方式来拆分任务和加速处理过程。所以说，Java8中首次出现的 java.util.stream是一个函数式语言+多核时代综合影响的产物。

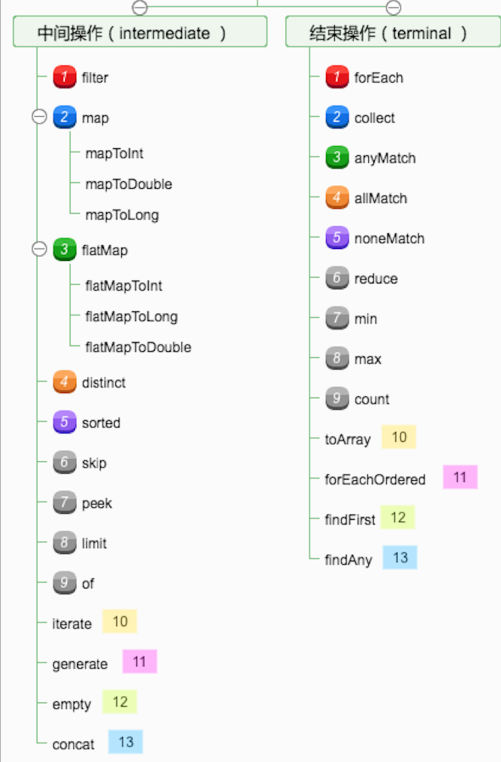
Stream有如下三个操作步骤：

一、创建Stream：从一个数据源，如集合、数组中获取流。

二、中间操作：一个操作的中间链，对数据源的数据进行操作。

三、终止操作：一个终止操作，执行中间操作链，并产生结果。





当数据源中的数据上了流水线后，这个过程对数据进行的所有操作都称为“中间操作”。中间操作仍然会返回一个流对象，因此多个中间操作可以串连起来形成一个流水线。比如map (mapToInt, flatMap 等)、filter、distinct、sorted、peek、limit、skip、parallel、sequential、unordered。

当所有的中间操作完成后，若要将数据从流水线上拿下来，则需要执行终止操作。终止操作将返回一个执行结果，这就是你想要的数据。比如：forEach、forEachOrdered、toArray、reduce、collect、min、max、count、anyMatch、allMatch、noneMatch、findFirst、findAny、iterator。

多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何处理！而在终止操作时一次性全部处理，称作“惰性求值”。

#### 【示例23】流式编程1：中间操作和中止操作

|  |
| --- |
| **public class** TestStream1 {  **public static void** main(String[] args) {  List<Integer > list = **new** ArrayList<>();  Collections.*addAll*(list,34,56,89,65,87,80,87,95,100,34,45);  //创建Stream  Stream<Integer> stream = list.stream();  //进行中间操作  stream = stream.filter((x)->{**if**(x>=60) **return true**; **return false**;})//刷选掉不及格的  .distinct() //去重  .sorted((x1,x2)->{**return** -Integer.*compare*(x1,x2);})//降序排列  .limit(4)//只要前四个  .map((x)->x+5)//每个成绩加5分  .skip(2);//跳过前两个，从第三个开始  //进行终止操作:stream has already been operated upon or closed  //stream.forEach(System.out::println);//遍历  //System.out.println(stream.max((x1,x2)->x1-x2));;  //System.out.println(stream.count());  System.***out***.println(stream.findFirst());   } } |

#### 【示例24】流式编程2：创建Stream

|  |
| --- |
| **public class** TestStream2 {  **public static void** main(String[] args) {  //创建Stream方式1  List<Integer > list = **new** ArrayList<>();  Collections.*addAll*(list,34,56,89,65,87,80,87,95,100,34,45);  Stream stream = list.stream();  stream.forEach(System.***out***::println);  //创建Stream方式2：并行流，底层采用ForkJoin框架，结果并不按照集合原有顺序输出  System.***out***.println(**"----------------"**);  Stream stream2 = list.parallelStream();//  stream2.forEach((x)->System.***out***.println(x+**"---"**+Thread.*currentThread*().getName()));  //创建Stream方式3:of()  System.***out***.println(**"----------------"**);  Stream stream3 = Stream.*of*(34,56,89,65,87,80,87,95,100,34,45);  stream3.forEach(System.***out***::println);  //创建Stream方式4  **int** [] arr = {34,56,89,65,87,80,87,95,100,34,45};  IntStream stream4 = Arrays.*stream*(arr);  stream4.forEach(System.***out***::println);  //创建Stream方式5  Stream stream5 = Stream.*generate*(()->Math.*random*());  // stream5.forEach(System.out::println);  //创建Stream方式6  Stream stream6 = Stream.*iterate*(6,(i)->2+i);  stream6.limit(5).forEach(System.***out***::println);  } } |

### 本节作业

1. Lambda表达式的语法和本质
2. 函数式接口的特点
3. 四大内置函数式接口
4. 方法引用的5种语法
5. 流式编程的操作步骤