

# Collection

- 实现了 `Iterable` 接口, 就使用 `forEach` (1.7) 语法 (语法糖);
- 执行 `clear()` 方法, 只是将数组元素设置为 `null`, 分配的内存还是存在的;

## ArrayList

`ArrayList`: 是一个数组队列, 类似动态数组, 但, 不是线程安全的, 多线程中可以选择 `Vector` 或者 `CopyOnWriteArrayList`; `Fail-fast` 机制, 不能充分保证fail-fast一定起作用, 所以fail-fast主要用于bug检测 (*the fail-fast behavior of iterators should be used only to detect bugs*)

- 在 `jdk8` 中, `new ArrayList()` 创建的list长度为 `0`, 即 `{}`;
- 在新建 `ArrayList` 对象的时候, 如果不指定初始大小, 默认大小是 `10`; (实际上是先建立一个空数组 `{}`, 在调用添加 (`add()`) 方法的时候会检查所需数组大小 `minCapacity = size + 1`, 然后取 `Math.max(10, minCapacity)` 作为初始化长度), 但是使用了 `ArrayList(initialCapacity)` 和 `ArrayList(collection)` 构造方法的就另当别论了;
- `ArrayList` 每次长度变化过程是: `minCapacity = size + 添加元素的个数` → 然后 `Math.max(size + (size >> 1), minCapacity)`;
- `&&` 的优先级大于 `||`;
- Java是通过变量 `modCount` 来识别迭代过程中list异常修改的, 然后抛出异常 `ConcurrentModificationException`;
- Arrays的 合并排序 (`mergeSort`) 方法:

```
private static final int INSERTIONSORT_THRESHOLD = 7;

// Insertion sort on smallest arrays
if (length < INSERTIONSORT_THRESHOLD) {
    // 插入排序
    for (int i=low; i<high; i++)
        for (int j=i; j>low &&
            ((Comparable) dest[j-1]).compareTo(dest[j])>0; j--)
            swap(dest, j, j-1);
    return;
}
```

- `Arrays.sort()` 采用了一种名为 `TimSort` 的排序算法, 就是 归并排序 (即 合并排序) 的优化版本 (可以查看 《排序算法》 文档中 `timsort` 模块的介绍):
  - 用户可以通过设置系统属性 `LegacyMergeSort.userRequested` 来决定使用传统数组合并排序方法, 还是使用新数组排序方法 `TimSort`;
- `ArrayList` 和 `LindedList` 性能对比(list的size分别为: 10000,100000, 1000000, 10000000), 内容结果为Array/Linked形式:

循环方式	1,0000(ms/毫秒)	10,0000(ms/毫秒)	100,0000(ms/毫秒)	1000,0000(ms/毫秒)
for-each	1/0	3/1	14/1	152/2
iterator	0/0	1/0	12/0	114/2
for-size	0/0	1/1	13/73	128/7972

总结: 因为 `LinkedList` 内部数据结构是链表, 所以使用 `forEach` 或 `iterator` 这种顺序消费的循环结构不用频繁寻找元素, 性能比较高; 而 `ArrayList` 内部的数据结构是数组, 对 `for-size` 这种通过索引查找元素的方式性能比较高;

- `ArrayList` 的 `removeAll(collection)` 实现方式很高效, 包装了 `batchRemove(Collection<?> c, boolean complement)` 方法;

## CopyOnWriteArrayList

- `CopyOnWriteArrayList` 是 `ArrayList` 的一个线程安全（通过 `ReentrantLock` 实现）的变体，其中所有可变操作（`add`、`set` 等等）都是通过对底层数组进行一次新的复制来实现的；
- `CopyOnWriteArrayList` 适合使用在读操作远远大于写操作的场景里，比如缓存。发生修改时候做copy，新老版本分离，保证读的高性能，适用于以读为主的情况；

## Vector

- 是通过 `synchronized` 实现线程安全；
- `Vector` 的默认大小也是 `10` ；
- `Vector` 的Capacity默认增长率为 `100%`，而 `ArrayList` 的Capacity增长率为 `50%`；
- 如果配置了 `capacityIncrement` 变量，则每次增加量为 `capacityIncrement`；否则，直接增加到 `oldCapacity` 的2倍

```
// minCapacity - 若要添加元素, Vector的最小容量
private int newCapacity(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                    capacityIncrement : oldCapacity);
    if (newCapacity - minCapacity <= 0) {
        if (minCapacity < 0) // overflow
            throw new OutOfMemoryError();
        return minCapacity;
    }
    return (newCapacity - MAX_ARRAY_SIZE <= 0)
        ? newCapacity
        : hugeCapacity(minCapacity);
}
```

- 类似 `ArrayList`，可以添加任意数量的 `Null`；

## HashMap

- 默认初始大小是 `16`，如果添加了初始化大小 `initialCapacity`（使用带参数`initialCapacity`的构造方法），临界值为 `threshold` 为不小于`initialCapacity`的2的最小幂值。装载因子 `loadFactor` 默认值为 `0.75F`；  
`threshold = tableCapacity * loadFactor` 等式会在调用 `put` 方法的时候保证；
- 无论 `initialCapacity` 设置为多少，其最终的初始容量会是不小于 `initialCapacity` 的2的最小幂值；
- `HashMap` 的 `hash` 桶的最大数量是 `1<<30 = 1G` 的容量

```
static final int MAXIMUM_CAPACITY = 1 << 30;
```

- `HashMap` 的容量：  
此处说的`HashMap`的容量其实就是`hashmap`的`hash`桶的数量
- 1. 无参构造器  
如果使用了 `HashMap()` 无参构造器，那么 `hashMap` 的初始容量是0，在第一次添加元素的时候，容量被设置为默认容量 `DEFAULT_INITIAL_CAPACITY (16)`

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

`putAll(Map map)` 方法是将参数map中元素拆开，一个一个地通过调用 `put(key, value)` 方法添加到当前HashMap中；

2. 有参构造器  
如果使用 `HashMap(int capacity)` 方法：
  - `capacity=0`, 结果: `hashMap`的实际容量为1;

- capacity>0, 结果: hashMap的实际容量为 不小于capacity的2的最小次幂:

capacity=0时, hashMap的实际容量其实也满足这个规律, 但实际算法并不一样:

如何得到 不小于capacity的2的最小次幂?

1. 先计算 capacity 的二进制形式前面有几个零, 用 n 表示, 如: 1的进制前面有31个零;
  2. `-1 >> n`, 用tCap表示;
  3. `tCap + 1` 就是 不小于capacity的2的最小次幂了;
- 向 hashMap 中添加元素的时候, 如果某个hash桶中的node个数小于某个 阈值 ( `final TREEIFY_THRESHOLD`, 值是 8 ) 并且map的Capacity大于等于64 ( `final MIN_TREEIFY_CAPACITY` ) 时, 桶中的元素会使用链表的形式存储具体元素是Node类型; 如果大于这个阈值 ( `TREEIFY_THRESHOLD` ), 桶中元素会被重构成 tree结构, 具体对象类型是 `TreeNode<K, V>`。插入 `treenode` 的方法同 `treenmap` 一样, 可参照 `treenmap` ;
  - `HashMap` 有几个回调函数, 可通过继承 `hashmap` 重写方法, 实现客户逻辑

```
// Callbacks to allow LinkedHashMap post-actions
void afterNodeAccess(Node<K,V> p) { }
void afterNodeInsertion(boolean evict) { }
void afterNodeRemoval(Node<K,V> p) { }
```

- `HashMap` 中有一个计算余数的高效方式: ( 博客中-为什么HashMap容量一定要为2的幂呢 )

```
// 在HashMap类的final Node<K,V> getNode(int hash, Object key)方法中
(n - 1) & hash;
```

- 可以使用 `Map m = Collections.synchronizedMap(new HashMap(...));` 得到线程安全的map, 但是这个实现中的所有方法均使用了 `synchronized` 关键字, 此关键字对性能不友好;
- `fail-fast` 一旦创建了迭代器, 除了迭代器自身的删除 ( `remove` ) 方法, 其他任何元素删除方法都会抛出 `ConcurrentModificationException` 异常; ( 集合类通则 )
- 更多参考
  - [Java 8 HashMap键与Comparable接口](#)
  - [HashMap的加载因子为何是0.75?](#)

## TreeMap

- `TreeMap` 中的键是有序的;
- `TreeMap` 主要使用到的数据结构是 红黑树, 红黑树有三个主要性质: ①根节点必须是黑色; ②红色节点不能连续; ③每条路径上的黑色节点数必须相等, 上面的这些性质是为了保持树的平衡性的, 数据的大小顺序还是通过左小右大 (子节点) 的方式保持的;
- `TreeMap` 的key 不允许为 null;
- 插入节点步骤: 通过比较把节点插入到相应位置 (此时tree会失衡) → 再调用 `fixAfterInsertion()` 方法, 回归平衡。

## LinkedHashMap

- `LinkedHashMap` 既集成了 `hashmap` 的基本特性, 也实现了一个双向链表;
- `LinkedHashMap` 默认顺序是 插入顺序 ( `accessOrder=false` ), 当设置 `accessOrder=true` 时, 则按照访问排序, 被get过的元素会被放在link的最后; 参考: 博客;

## HashSet

- `HashSet` 的内部结构其实是一个 `hashMap<E, Object>`, 是通过map的key实现去重复的;

```
private static final Object PRESENT = new Object();

public boolean add(E e) {
    // 内部map数据结构的value值都是PRESENT
```

```
        return map.put(e, PRESENT)==null;
    }
```

- `HashSet` 默认使用的是 `HashMap`，`HashSet(int initialCapacity, float loadFactor, boolean dummy)`：这个构造函数，内部使用的是 `LinkedHashMap`；参数 `dummy` 为无效参数，没有实际意义：

```
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
    map = new LinkedHashMap<>(initialCapacity, loadFactor);
}
```

## TreeSet

- `TreeSet` 的大部分方法基本都是在 `TreeMap` 的基础上实现的；

## Enumeration

自行搜索吧

## Queue

- `Add / remove / element` 方法是在 `offer / poll / peek` 方法的基础上实现的

```
public boolean add(E e) {
    if (offer(e))
        return true;
    else
        throw new IllegalStateException("Queue full");
}

public E remove() {
    E x = poll();
    if (x != null)
        return x;
    else
        throw new NoSuchElementException();
}
```

- `queue` 中不能插入 `null` 对象，因为 `offer / poll / peek` 都用到`null`对象来判断队列是否结束，所以`queue`的实现类中也都做有相应的非空判断；

## Deque

- **Deque**: Double Ended Queue
- `LinkedList` 就是`Deque`的一个实现
- `LinkedBlockingDeque` 是一个链表阻塞双向队列

## PriorityQueue

- `PriorityQueue` 不允许插入没有实现排序接口（`comparable`）的对象；
- `PriorityQueue` 的默认初始大小是 `11`；
- `PriorityQueue` 中 `siftDown()/siftUp()` 方法是建立堆的过程
- `PriorityQueue` 的构造方法 `PriorityQueue(collection)` 没有充分地检查`collection`中是否包含`null`，某些情况下可以构建`PriorityQueue`对象，但是执行方法的时候却包 `NullPointerException` 异常。比如：`toString()` 方法
  - 但是好像新版jdk（如jdk11）完善了这个bug；
- `PriorityQueue`的扩容函数

```
private void grow(int minCapacity) {
    int oldCapacity = queue.length;
```

```
// Double size if small; else grow by 50%
int newCapacity = oldCapacity + ((oldCapacity < 64) ?
    (oldCapacity + 2) :
    (oldCapacity >> 1));

// overflow-conscious code
if (newCapacity - MAX_ARRAY_SIZE > 0)
    newCapacity = hugeCapacity(minCapacity);
queue = Arrays.copyOf(queue, newCapacity);
}
```

## EnumSet

- `EnumSet` 设计牛逼，但是不知道什么场景能够使用

```
// RegularEnumSet类
public boolean add(E e) {
    typeCheck(e);

    long oldElements = elements;
    elements |= (1L << ((Enum<?>)e).ordinal());
    return elements != oldElements;
}
```

- 优秀源码：猜猜 `unseen & -unseen` 的计算结果：

提示：数字的位与运算时，如果有负数，则负数先转换成负数的补码，再参与运算；

```
public E next() {
    if (unseen == 0)
        throw new NoSuchElementException();
    lastReturned = unseen & -unseen;
    unseen -= lastReturned;
    return (E) universe[Long.numberOfTrailingZeros(lastReturned)];
}
```

其实 `unseen & -unseen` 的结果是：只保留 `unseen` 从低位开始第一个非0位，其他所有位清零；  
如：6L（二进制-1010），`6L & -6L` 结果为2（二进制-10）；

- 类似的方案应用：`Redis` 的 `bitmap` 类型统计用户在线状态；

## EnumMap

- `EnumMap` 使用的是数组进行数据保存，随机读取效率比较高；
- `EnumMap` 的key不能是null，不然抛出异常，`value`可以为null；
- `EnumMap` 的key必须是枚举类型；
- `EnumMap` 是保证顺序的，输出是按照键（`枚举`）顺序；