

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:

```
// INSERTIONSORT_THRESHOLD
if (length < INSERTIONSORT_THRESHOLD) {
    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, dest[j-1]);
    return;
}
```



长度小于7时, 直接进行
插入排序

- `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);
- 在没有传入比较器的情况下, 还有一种排序分支

```
public static void sort(Object[] a) {
    if (LegacyMergeSort.userRequested)    传统合并排序（1.7之前）
        legacyMergeSort(a);
    else
        ComparableTimSort.sort(a, lo: 0, a.length, work: null, workBase: 0, workLen: 0);
}
```

我注意到1.7下的sort有一个分支判断，当LegacyMergeSort.userRequested为true的情况下，采用legacyMergeSort，否则采用ComparableTimSort。LegacyMergeSort.userRequested的字面意思大概就是“用户请求传统归并排序”的意思，这个分支调用的是与jdk1.5相同的方法来实现功能。

我没有花时间去研究jdk1.7为什么要这么做，以及在另外一个分支判断中ComparableTimSort的具体实现是如何的，我只需要尽量解决当前发现的问题，网上搜索过一下后，发现在执行Collections.sort进行排序前可以调用：

Java代码  

```
1. System.setProperty("java.util.Arrays.useLegacyMergeSort", "true");
```

来实现设置LegacyMergeSort.userRequested的赋值，测试过以后确实有效，这样能够在jdk1.7下强制使用老的排序方式达到预期功能。

其他的问题：

1、如果客户生产环境中的某些软件需要升级需要变更（升级）到高版本的jdk，还是可能会遇到不少难以预期的问题，例如像我遇到的排序结果本末倒置，所以升级真是挺危险的一件事。

2、在jdk1.7的legacyMergeSort方法注释中，注意到了：“To be removed in a future release.”，这么说在1.7以后的版本中，老的mergeSort方法可能会被ComparableTimSort替代，这样也就不能通过改变userLegacyMergeSort的值来影响排序的实现了，这可能确实会带来一些问题，如果需要进行自定义逻辑的排序，那么以后也需要留意。

- ArrayList 和 LindedList 性能对比

1	compare loop performance of ArrayList				
2	-----				
3	list size	10,000	100,000	1,000,000	10,000,000
4	-----				
5	for each	1 ms	3 ms	14 ms	152 ms
6	-----				
7	for iterator	0 ms	1 ms	12 ms	114 ms
8	-----				
9	for list.size()	1 ms	1 ms	13 ms	128 ms
10	-----				
11	for size = list.size()	0 ms	0 ms	6 ms	62 ms
12	-----				
13	for j--	0 ms	1 ms	6 ms	63 ms
14	-----				
15					
16	compare loop performance of LinkedList				
17	-----				
18	list size	100	1,000	10,000	100,000
19	-----				
20	for each	0 ms	1 ms	1 ms	2 ms
21	-----				
22	for iterator	0 ms	0 ms	0 ms	2 ms
23	-----				
24	for list.size()	0 ms	1 ms	73 ms	7972 ms
25	-----				
26	for size = list.size()	0 ms	0 ms	67 ms	8216 ms
27	-----				
28	for j--	0 ms	1 ms	67 ms	8277 ms
29	-----				

退出全屏模式 (F11)

linkedlist的迭代器
性能更加出色

ArrayList的随机读取
速度更加好

优

优

优

- 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

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                    capacityIncrement : oldCapacity);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

- 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 $1 \ll 30 = 1G$ 的容量

```
/**
 * The maximum capacity, used if a higher value is implicitly specified
 * by either of the constructors with arguments.
 * MUST be a power of two <= 1<<30.
```



```
*/
```

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

```
@Test
```

```
public void testTheMid() {
```

```
    int x = 32;
```

```
    int n = x - 1;
```

```
    n |= n >> 1;
```

```
    n |= n >> 2;
```

```
    n |= n >> 4;
```

```
    n |= n >> 8;
```

```
    n |= n >> 16;
```

```
    System.out.println(n + 1);
```

```
}
```

hashmap中扩容时使用到的：
计算不小于x的最小的2的幂值

- 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 **treemap** 一样，可参照 **treemap**；
- **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](#)）

```

/**
 * Returns index for hash code h.
 */
static int indexFor(int h, int length) {
    return h & (length-1);
}

```

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；

```

public boolean add(E e) {
    return map.put(e, PRESENT) == null;
}

```

set.add(e)
PRESENT = new Object()

- **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；

```

* @param      initialCapacity    the initial capacity of the hash map
* @param      loadFactor          the load factor of the hash map
* @param      dummy                ignored (distinguishes this
*                                  constructor from other int, float constructor.)
* @throws      IllegalArgumentException if the initial capacity is less
*                                  than zero, or if the load factor is nonpositive
*/

```

```

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

Iterator在功能上可以完全替代**Enumeration**。后者目前还保留在Java标准库里纯粹是为了兼容老API（例如Hashtable、Vector、Stack等老的collection类型）。

Iterator相比**Enumeration**有以下区别：

- 前者的方法名比后者简明扼要；
- 前者添加了一个可选的**remove()**方法（“可选”意味着一个实现**Iterator**接口的类可以选择不实现**remove()**方法）；
- 前者是“fail-fast”的——如果它在遍历过程中，底下的容器发生了结构变化（例如**add**或者**remove**了元素），则它会抛出**ConcurrentModificationException**；后者没有这种检查机制；
- 前者可以配合**Iterable<E>**接口用于Java 5的**for-each**循环中。

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

- PriorityQueue的扩容函数

```
int newCapacity = oldCapacity + ((oldCapacity < 64) ?
                                (oldCapacity + 2) :
                                (oldCapacity >> 1));
// overflow-conscious code
```

EnumSet

- EnumSet 设计牛逼，但是不知道什么场景能够使用

```
public boolean add(E e) {
    typeCheck(e);

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

每一位代表一个枚举类型，共64位

- 超级6的一个算法：关键是 `unseen & -unseen` 的计算结果：提示：使用utf8转码

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：这个例子中的 `unseen & -unseen` 计算，得到的结果刚好就是末尾0的长度，计算末尾0位的位数的方法还有 `Long.numberOfTrailingZeros(long i)`

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

RegularEnumSet内部迭代器

- 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;