今天粗读了一遍Collection接口下常用的类源码，整理一些笔记方便日后查看，如有错误还请各位大大批评指正。注：文章内容基于jdk1.8

List

部分方法

- iterator()：以适当的顺序在列表中的元素上返回迭代器。
- toArray()：返回一个数组，该数组包含列表中的所有元素，并按正确的顺序(从第一个元素到最后一个元素)。 返回的数组将是“安全的”，因为这个列表不维护对它的引用。(换句话说，这个方法必须分配一个新的数组，即使这个列表是由一个数组支持的)。因此，调用者可以自由地修改返回的数组。 此方法充当基于数组和基于集合的api之间的桥梁。
- toArray(T[] a)：返回一个包含列表中所有元素的数组(从第一个元素到最后一个元素);返回数组的运行时类型为指定数组的类型。如果列表与指定数组匹配，则返回该列表。否则，将为新数组分配指定数组的运行时类型和该列表的大小。 如果列表在指定的数组中有多余的空间(例如。，则数组中紧接在列表末尾的元素被设置为null。(只有当调用者知道列表不包含任何空元素时，这在确定列表的长度时才有用。) 与toArray()方法一样，该方法充当基于数组和基于集合的api之间的桥梁。此外，该方法允许对输出数组的运行时类型进行精确控制，并且在某些情况下可以用于节省分配成本。 假设x是一个已知只包含字符串的列表。下面的代码可以用来转储列表到一个新分配的字符串数组: y = x。toArray(新的字符串[0]); 注意，toArray(新对象[0])在函数上与toArray()相同。
- retainAll(Collection<?> c)：仅保留此列表中包含在指定集合中的元素(可选操作)。换句话说，从该列表中删除指定集合中不包含的所有元素。
- listIterator()：在列表元素上返回一个列表迭代器(按正确的顺序)；List接口提供了一个称为ListIterator的特殊迭代器，除了iterator接口提供的常规操作外，它还允许元素插入和替换以及双向访问。提供了一个方法来获取从列表中指定位置开始的列表迭代器。

AbstractList

hashcode与String类似

```
    public int hashCode() {
        int hashCode = 1;
        for (E e : this)
            hashCode = 31*hashCode + (e==null ? 0 : e.hashCode());
        return hashCode;
    }
```

ArrayList ext AbstractList imp List, RandomAccess, Cloneable,Serializable

特点

- 数据结构：Object数组
- 初始容量：10
- 扩容规则：原始容量的1.5倍，grow方法，newCapacity = oldCapacity + (oldCapacity >> 1);
- 最大容量：Integer.MAX_VALUE - 8，原因是一些虚拟机在数组中保留一些头字。尝试分配更大的数组可能会导致OutOfMemoryError:请求的数组大小超过VM限制，如果再次扩容会设置为Integer.MAX_VALUE即2^{31} -1
- 元素允许为空
- 实现RandomAccess接口
- 采用了Fail-Fast机制，每次添加删除都会自增modCount，并发的修改时，迭代器很快就会完全失败，报异常ConcurrentModificationException
- 非线程安全，可是使用Collections.synchronizedList(List l)或oncurrent并发包下的CopyOnWriteArrayList
- 内部操作数据基于System.arraycopy()方法

基本操作

- 增

```
  public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
  }
  private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
  }

  private static int calculateCapacity(Object[] elementData, int
minCapacity) {
      if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
          return Math.max(DEFAULT_CAPACITY, minCapacity);
      }
      return minCapacity;
  }
  private void ensureExplicitCapacity(int minCapacity) {
      modCount++;

      // overflow-conscious code
      if (minCapacity - elementData.length > 0)
          grow(minCapacity);
  }
  private void grow(int minCapacity) {
      // overflow-conscious code
      int oldCapacity = elementData.length;
      int newCapacity = oldCapacity + (oldCapacity >> 1);
      if (newCapacity - minCapacity < 0)
          newCapacity = minCapacity;
      if (newCapacity - MAX_ARRAY_SIZE > 0)
```

```
            newCapacity = hugeCapacity(minCapacity);
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

- 删

```
    public boolean remove(Object o) {
        if (o == null) {
            for (int index = 0; index < size; index++)
                if (elementData[index] == null) {
                    fastRemove(index);
                    return true;
                }
        } else {
            for (int index = 0; index < size; index++)
                if (o.equals(elementData[index])) {
                    fastRemove(index);
                    return true;
                }
        }
        return false;
    }
    //先将数据中要删除元素后的所有元素copy到删除元素的下标处，然后设置最后一个元素为空。
    private void fastRemove(int index) {
        modCount++;
        int numMoved = size - index - 1;

        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--size] = null; // clear to let GC do its work
    }
```

- 改

```
    public E set(int index, E element) {
        rangeCheck(index);

        E oldValue = elementData(index);
        elementData[index] = element;
        return oldValue;
    }
```

- 查

```
public E get(int index) {
    rangeCheck(index);

    return elementData(index);
}
```

如何复制某个ArrayList到另一个ArrayList中去?

- 使用clone()方法，比如ArrayList newArray = oldArray.clone(); // 浅拷贝
- 使用ArrayList构造方法，比如：ArrayList myObject = new ArrayList(myTempObject); //浅拷贝
- 使用Collection的copy方法。 //浅拷贝

ArrayList 中 elementData 为什么使用 transient 修饰?

由于 ArrayList 是基于动态数组实现的，所以并不是所有的空间都被使用。因此使用了 transient 修饰，可以防止被自动序列化。

因此 ArrayList 自定义了序列化与反序列化，具体可以看 writeObject 和 readObject 两个方法。

需要注意的一点是，当对象中自定义了 writeObject 和 readObject 方法时，JVM 会调用这两个自定义方法来实现序列化与反序列化。

ArrayList 的插入删除一定慢么?

取决于你删除的元素离数组末端有多远，ArrayList拿来作为堆栈来用还是挺合适的，push和pop操作完全不涉及数据移动操作。

ArrayList 中的 elementData 为什么是 Object 而不是泛型 E ?

Java 中泛型运用的目的就是实现对象的重用，泛型T和Object类其实在编写时没有太大区别,只是JVM中没有T这个概念，T只是存在于编写时,进入虚拟机运行时,虚拟机会对泛型标志进行擦除，也就是替换T会限定类型替换（根据运行时类型）,如果没有限定就会用Object替换。同时Object可以new Object()，就是说可以实例化，而T则不能实例化。在反射方面来说，从运行时,返回一个T的实例时,不需要经过强制转换,然后Object则需要经过转换才能得到。

ArrayList的遍历和LinkedList遍历性能比较如何?

论遍历ArrayList要比LinkedList快得多，ArrayList遍历最大的优势在于内存的连续性，CPU的内部缓存结构会缓存连续的内存片段，可以大幅降低读取内存的性能开销。

ArrayList 做队列合适么？ 队列一般是FIFO的，如果用ArrayList做队列，就需要在数组尾部追加数据，数组头部删除数组，反过来也可以。但是无论如何总会有一个操作会涉及到数组的数据搬迁，这个是比较耗费性能的。 这个回答是错误的！ ArrayList固然不适合做队列，但是数组是非常合适的。比如ArrayBlockingQueue内部实现就是一个环形队列，它是一个定长队列，内部是用一个定长数组来实现的。另外著名的Disruptor开源Library也是用环形数组来实现的超高性能队列，具体原理不做解释，比较复杂。简单点说就是使用两个偏移量来标记数组的读位置和写位置，如果超过长度就折回到数组开头，前提是它们是定长数组。

参考文章： Java基础之一个 ArrayList 就能让你面试到哭！ 坑人无数的Java面试题之ArrayList

LinkedList ext AbstractSequentialList imp List,Deque,cloneable

特点

- 数据结构： LinkedList.Node双向链表
- 元素允许为空
- 实现RandomAccess接口

特殊方法

- peek 检索但不删除此列表的头(第一个元素)，first为空返回null
  - peekFirst
  - peekLast
- element 检索但不删除此列表的头(第一个元素)，first为空抛出NoSuchElementException
- poll 删除并返回第一个元素，first为空时返回null
  - pollFirst
  - pollLast
- remove 删除并返回第一个元素，first为空时抛出 NoSuchElementException
- offer 将指定的元素添加为该列表的尾部
  - offerFirst 将指定的元素添加为该列表的头部
  - offerLast
- push 将一个元素压入此列表所表示的堆栈。换句话说，将元素插入到这个列表的前面。 这个方法相当于addFirst
- pop 从这个列表所表示的堆栈中弹出一个元素。换句话说，删除并返回该列表的第一个元素。 这个方法等同于removeFirst()。 first为空时抛出 NoSuchElementException

基本操作

- 增

```
public boolean add(E e) {
    linkLast(e);
    return true;
}
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
```

```
        size++;
        modCount++;
    }
```

- 删

```
    public boolean remove(Object o) {
        if (o == null) {
            for (Node<E> x = first; x != null; x = x.next) {
                if (x.item == null) {
                    unlink(x);
                    return true;
                }
            }
        } else {
            for (Node<E> x = first; x != null; x = x.next) {
                if (o.equals(x.item)) {
                    unlink(x);
                    return true;
                }
            }
        }
        return false;
    }
    E unlink(Node<E> x) {
        // assert x != null;
        final E element = x.item;
        final Node<E> next = x.next;
        final Node<E> prev = x.prev;

        if (prev == null) {
            first = next;
        } else {
            prev.next = next;
            x.prev = null;
        }

        if (next == null) {
            last = prev;
        } else {
            next.prev = prev;
            x.next = null;
        }

        x.item = null;
        size--;
        modCount++;
        return element;
    }
```

- 改

```
public E set(int index, E element) {
    checkElementIndex(index);//验证下标是否正确
    Node<E> x = node(index);
    E oldVal = x.item;
    x.item = element;
    return oldVal;
}
```

- 查

```
public E get(int index) {
    checkElementIndex(index);//验证下标是否正确
    return node(index).item;
}
Node<E> node(int index) {
    // assert isElementIndex(index);
    //二分法查找
    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

Vector

特点

- 数据结构：Object数组
- 初始容量：10
- 扩容规则：原始容量的2倍，grow方法，newCapacity = oldCapacity + ((capacityIncrement > 0) ?capacityIncrement : oldCapacity)；如果初始化时指定了capacityIncrement就按参数扩容。

- 最大容量：Integer.MAX_VALUE - 8，原因是一些虚拟机在数组中保留一些头字。尝试分配更大的数组可能会导致OutOfMemoryError:请求的数组大小超过VM限制，如果再次扩容会设置为Integer.MAX_VALUE即2^{31} -1
- 元素允许为空
- 实现RandomAccess接口
- setSize(int size);设置这个向量的大小。如果新大小大于当前大小，则在向量的末尾添加新的空项。如果新大小小于当前大小，则丢弃索引newSize和更大的所有组件
- 线程安全
  - 方法内部使用synchronized (Vector.this)
  - public synchronized boolean isEmpty()
- 内部操作数据基于System.arraycopy()方法

基本操作

- 增

```
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}
private void ensureCapacityHelper(int minCapacity) {
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
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);
}
private static int hugeCapacity(int minCapacity) {
    if (minCapacity < 0) // overflow
        throw new OutOfMemoryError();
    return (minCapacity > MAX_ARRAY_SIZE) ?
        Integer.MAX_VALUE :
        MAX_ARRAY_SIZE;
}
```

- 删

```
public boolean remove(Object o) {
    return removeElement(o);
}
public synchronized boolean removeElement(Object obj) {
    modCount++;
    int i = indexOf(obj);
    if (i >= 0) {
        removeElementAt(i);
        return true;
    }
    return false;
}
public int indexOf(Object o) {
    return indexOf(o, 0);
}
public synchronized int indexOf(Object o, int index) {
    if (o == null) {
        for (int i = index ; i < elementCount ; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = index ; i < elementCount ; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
public synchronized void removeElementAt(int index) {
    modCount++;
    if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " +
                                                 elementCount);
    }
    else if (index < 0) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    int j = elementCount - index - 1;
    if (j > 0) {
        System.arraycopy(elementData, index + 1, elementData, index, j);
    }
    elementCount--;
    elementData[elementCount] = null; /* to let gc do its work */
}
```

- 改

```
public synchronized E set(int index, E element) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;
}
E elementData(int index) {
    return (E) elementData[index];
}
```

- 查

```
public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    return elementData(index);
}
```

Stack ext Vector

特点与Vector类似，在vector基础上增加如下方法

- push 将一项压入该堆栈的顶部
- pop 删除堆栈顶部的对象，并将该对象作为此函数的值返回。
- peek 查看堆栈顶部的对象，但不将其从堆栈中移除
- empty
- search 返回对象在堆栈上的基于栈顶的位置。

CopyOnWriteArrayList imp List, RandomAccess, Cloneable, java.io.Serializable

特点

- 数据结构：Object数组
- 初始容量：0
- 扩容规则：每次添加元素都会使用Arrays.copyOf(elements, len + 1);复制原始数组长度加1
- 最大容量：无
- 元素允许为空
- 实现RandomAccess接口

- 没有Fail-Fast机制 在遍历时仅保存了一份数据的快照，而且iterator不支持remove、set、add等操作，异常UnsupportedOperationException
- 线程安全，内部持有一把ReenTrantLock锁，数组对象添加volatile保证可见性，每次增删改都会获取锁，复制出一个新数组，完成插入、修改或者移除操作后将新数组赋值给array，释放锁
- 内部操作数据基于System.arraycopy()、Arrays.copyOf方法

基本操作

- 增

```
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock(); //获取锁
    try {
        Object[] elements = getArray();//取得原始数组
        int len = elements.length;
        //复制一个新的数组，长度是原始数组长度+1
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        newElements[len] = e;
        setArray(newElements);//将新数组放到全局变量中
        return true;
    } finally {
        lock.unlock();//释放锁
    }
}
```

- 删

```
public E remove(int index) {
    final ReentrantLock lock = this.lock;
    lock.lock();//获取锁
    try {
        Object[] elements = getArray();//取得原始数组
        int len = elements.length;
        E oldValue = get(elements, index);
        int numMoved = len - index - 1;
        if (numMoved == 0)
            //如果删除最后一位元素，直接复制数组，长度-1
            setArray(Arrays.copyOf(elements, len - 1));
        else {
            //new 新的数组，长度-1
            Object[] newElements = new Object[len - 1];
            //将要删除的index前的copy到newElements
            System.arraycopy(elements, 0, newElements, 0, index);
            //将要删除的index后的copy到newElements
            System.arraycopy(elements, index + 1, newElements, index,
                             numMoved);
            setArray(newElements);//将新数组放到全局变量中
```

```
            }
            return oldValue;
        } finally {
            lock.unlock();//释放锁
        }
    }
```

- 改

```
    public E set(int index, E element) {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            Object[] elements = getArray();
            E oldValue = get(elements, index);

            if (oldValue != element) {
                int len = elements.length;
                //创建一个新的数组
                Object[] newElements = Arrays.copyOf(elements, len);
                newElements[index] = element;
                setArray(newElements);
            } else {
                // Not quite a no-op; ensures volatile write semantics
                setArray(elements);
            }
            return oldValue;
        } finally {
            lock.unlock();
        }
    }
```

- 查

```
public E get(int index) {
    return get(getArray(), index);
}
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

迭代器

当调用iterator()方法获取迭代器会返回一个COWIterator对象，COWIterator对象的snapshot变量保存了当前list的快照，cursor是遍历list数组元素的下标。如果在遍历期间其它线程对该list进行增删改，snapshot还是更改前的数组，遍历的结果也就不回出现更改后的内容。

```
public Iterator<E> iterator() {
    return new COWIterator<E>(getArray(), 0);
}
static final class COWIterator<E> implements ListIterator<E> {
    /** Snapshot of the array */
    private final Object[] snapshot;
    /** Index of element to be returned by subsequent call to next.  */
    private int cursor;

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

    public boolean hasNext() {
        return cursor < snapshot.length;
    }

    public boolean hasPrevious() {
        return cursor > 0;
    }

    @SuppressWarnings("unchecked")
    public E next() {
        if (! hasNext())
            throw new NoSuchElementException();
        return (E) snapshot[cursor++];
    }

    @SuppressWarnings("unchecked")
    public E previous() {
        if (! hasPrevious())
            throw new NoSuchElementException();
        return (E) snapshot[--cursor];
    }

    public int nextIndex() {
        return cursor;
    }

    public int previousIndex() {
        return cursor-1;
    }

    /**
     * Not supported. Always throws UnsupportedOperationException.
     * @throws UnsupportedOperationException always; {@code remove}
     *         is not supported by this iterator.
```

```
         */
        public void remove() {
            throw new UnsupportedOperationException();
        }

        /**
         * Not supported. Always throws UnsupportedOperationException.
         * @throws UnsupportedOperationException always; {@code set}
         *         is not supported by this iterator.
         */
        public void set(E e) {
            throw new UnsupportedOperationException();
        }

        /**
         * Not supported. Always throws UnsupportedOperationException.
         * @throws UnsupportedOperationException always; {@code add}
         *         is not supported by this iterator.
         */
        public void add(E e) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void forEachRemaining(Consumer<? super E> action) {
            Objects.requireNonNull(action);
            Object[] elements = snapshot;
            final int size = elements.length;
            for (int i = cursor; i < size; i++) {
                @SuppressWarnings("unchecked") E e = (E) elements[i];
                action.accept(e);
            }
            cursor = size;
        }
    }
```

Set

AbstractSet

hashCode是所有元素的hashcode合

```
    public int hashCode() {
        int h = 0;
        Iterator<E> i = iterator();
        while (i.hasNext()) {
            E obj = i.next();
            if (obj != null)
                h += obj.hashCode();
        }
        return h;
    }
```

HashSet ext AbstractSet imp Set, Cloneable, Serializable

特点

- 数据结构：`HashMap<E,Object> map;`每次添加的元素做为`key`，`value`是全局对象`Object PRESENT = new Object();`
- 默认长度与`HashMap`相同，因为用hashmap存数据，所以扩容也与`HahsMap`相同
- 排序规则：无序，由一个哈希表(实际上是一个HashMap实例)支持。它不能保证集合的迭代顺序;特别是，它不能保证顺序随时间保持不变。这个类允许空元素

增删

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

LinkedHashSet ext HashSet imp Set, Cloneable,Serializable

。

特点

- 数据结构：继承`HashSet`，但内部使用`LinkedHashMap`存储数据
- 默认长度无，与`LinkedHashMap`一致
- 排序规则：按插入的顺序

TreeSet ext AbstractSet imp NavigableSet, Cloneable, Serializable

特点

- 数据结构：`TreeMap<E,Object> m;` 每次添加的元素做为key，value是全局对象`Object PRESENT = new Object();`
- 增删与HashSet同理
- 排序规则：默认按照自然顺序，如果初始化时指定了`Comparator`则按照其规则升序

CopyOnWriteArraySet ext AbstractSet imp Serializable

特点

- 数据结构：使用`final CopyOnWriteArrayList<E> al`存放数据
- 添加元素：`al.addIfAbsent(e);//如果不存在时才添加`
- 删除元素：`al.remove(o);`
- iterator：`al.iterator();`
- 排序规则：按照插入顺序排序