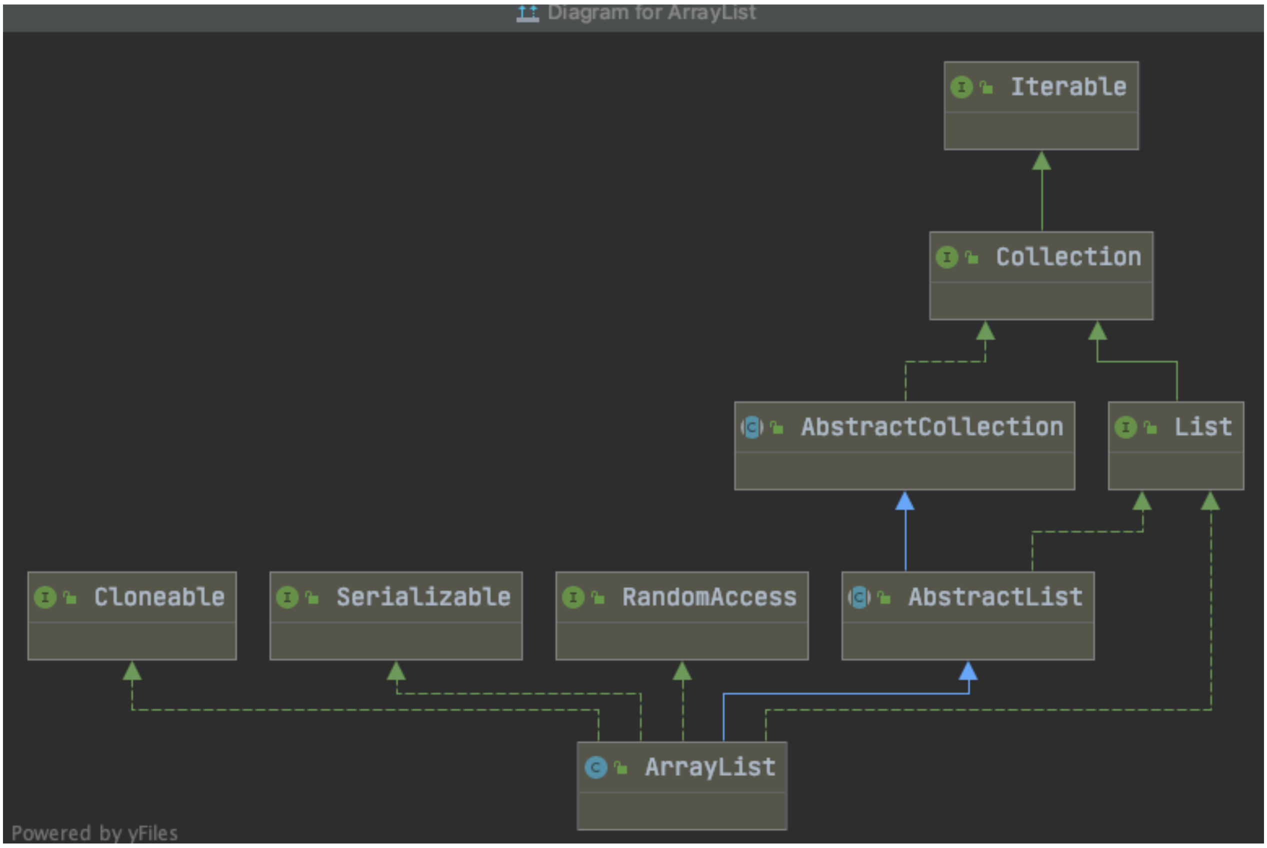
Java集合源码学习

ArrayList

继承关系图



- 实现了`Cloneable`(空接口)代表可以调用object.clone方法
- 实现`Serializable`(空接口)接口 代表可以进行序列化和反序列化
- 实现了`RandomAccess`(空接口) 代表实现了高速随机访问，如果标注了这个接口，则推荐使用for循环遍历，没有则更推荐迭代器
- 继承自`AbstractList`抽象类,继承一些list的通用方法
- 实现List接口

默认容量

```
/**
 * Default initial capacity.
 */
private static final int DEFAULT_CAPACITY = 10;
```

底层数组共享吗?

```java
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable{

    .....

    /**
     * Shared empty array instance used for empty instances.
     */
    private static final Object[] EMPTY_ELEMENTDATA = {};

    /**
     * Shared empty array instance used for default sized empty instances. We
     * distinguish this from EMPTY_ELEMENTDATA to know how much to inflate
when
     * first element is added.
     */
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    ....

    public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }

    /**
     * Constructs an empty list with an initial capacity of ten.
     */
    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }
}
```

- 如果初始化initialCapacity为0 的list 则底层共用一个空数组实例 `EMPTY_ELEMENTDATA`
- 如果使用无参构造方法也使用了同一个空数组实例 `DEFAULTCAPACITY_EMPTY_ELEMENTDATA`

```java
    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        if ((size = elementData.length) != 0) {
            // c.toArray might (incorrectly) not return Object[] (see 6260652)
            if (elementData.getClass() != Object[].class)
                elementData = Arrays.copyOf(elementData, size,
Object[].class);
        } else {
            // replace with empty array.
            this.elementData = EMPTY_ELEMENTDATA;
        }
    }
```

ArrayList可以从同一个集合的底层array进行构造。这似乎和go的slice一样，两个list共用一个底层数组，但是继续往下看则会发现。

toArray 会生成新数组，所以除了上面总结的两种共用底层数组的情况，Arraylist不会共用一个底层数组。

```java
/**
     * Returns an array containing all of the elements in this list
     * in proper sequence (from first to last element).
     *
     * <p>The returned array will be "safe" in that no references to it are
     * maintained by this list.  (In other words, this method must allocate
     * a new array).  The caller is thus free to modify the returned array.
     *
     * <p>This method acts as bridge between array-based and collection-based
     * APIs.
     *
     * @return an array containing all of the elements in this list in
     *         proper sequence
     */
 public Object[] toArray() {
        return Arrays.copyOf(elementData, size);
    }
```

trimToSize()

> trimToSize() 方法用于将List中的容量调整为数组中的元素个数。

```
    /**
     * Trims the capacity of this <tt>ArrayList</tt> instance to be the
     * list's current size.  An application can use this operation to minimize
     * the storage of an <tt>ArrayList</tt> instance.
     */
    public void trimToSize() {
        modCount++;
        if (size < elementData.length) {
            elementData = (size == 0)
              ? EMPTY_ELEMENTDATA  // 如果size为空 则将底层数组设为共享的空数组
              : Arrays.copyOf(elementData, size); //如果不为空 则根据size copy一个
新数组
        }
    }
```

扩容

```
public void ensureCapacity(int minCapacity) {
        int minExpand = (elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
            // 如果没有使用默认空数组 也就是new Arraylist()产生的实例。则期望所有的> 0
的扩容量
            ? 0
            // 如果使用了默认空数组 则期望的扩容量要 > 默认的10
            : DEFAULT_CAPACITY;

        if (minCapacity > minExpand) {
            ensureExplicitCapacity(minCapacity);
        }
    }
```

```
private void ensureExplicitCapacity(int minCapacity) {
    modCount++; //操作次数加一

    // overflow-conscious code
    if (minCapacity - elementData.length > 0) //如果扩容量大于当前的数组容量就进行扩
容
        grow(minCapacity);
}
```

扩容核心代码

```
private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = elementData.length; //旧的容量
```

```
        int newCapacity = oldCapacity + (oldCapacity >> 1); //新容量为旧容量的1.5倍
        if (newCapacity - minCapacity < 0) //如果新容量小于传进来的最小容量
            newCapacity = minCapacity; //将最小容量赋值给新容量
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity); //如果新容量比规定的最大ArraySize还要大，则需要进行限定
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }

    private static int hugeCapacity(int minCapacity) { //限定的逻辑
        if (minCapacity < 0) // overflow
            throw new OutOfMemoryError();
        return (minCapacity > MAX_ARRAY_SIZE) ?
            Integer.MAX_VALUE :
            MAX_ARRAY_SIZE;
    }
```

add(E e)

```
public boolean add(E e) {
        ensureCapacityInternal(size + 1);  //如果需要扩容则扩容，并且增加操作次数
        elementData[size++] = e;
        return true;
    }
```

add(int index, E element)

```
public void add(int index, E element) {
        rangeCheckForAdd(index); //检查索引越界

        ensureCapacityInternal(size + 1);  //如果需要扩容则扩容，并且增加操作次数
        System.arraycopy(elementData, index, elementData, index + 1,
                         size - index); //使用system.arraycopy 进行数组复制并且在重新赋值给elementData
        elementData[index] = element;
        size++;
    }
```

remove(int index)

```
    public E remove(int index) {
        rangeCheck(index); //检查索引越界

        modCount++; //增加操作次数
        E oldValue = elementData(index); //先保留一份备份
```

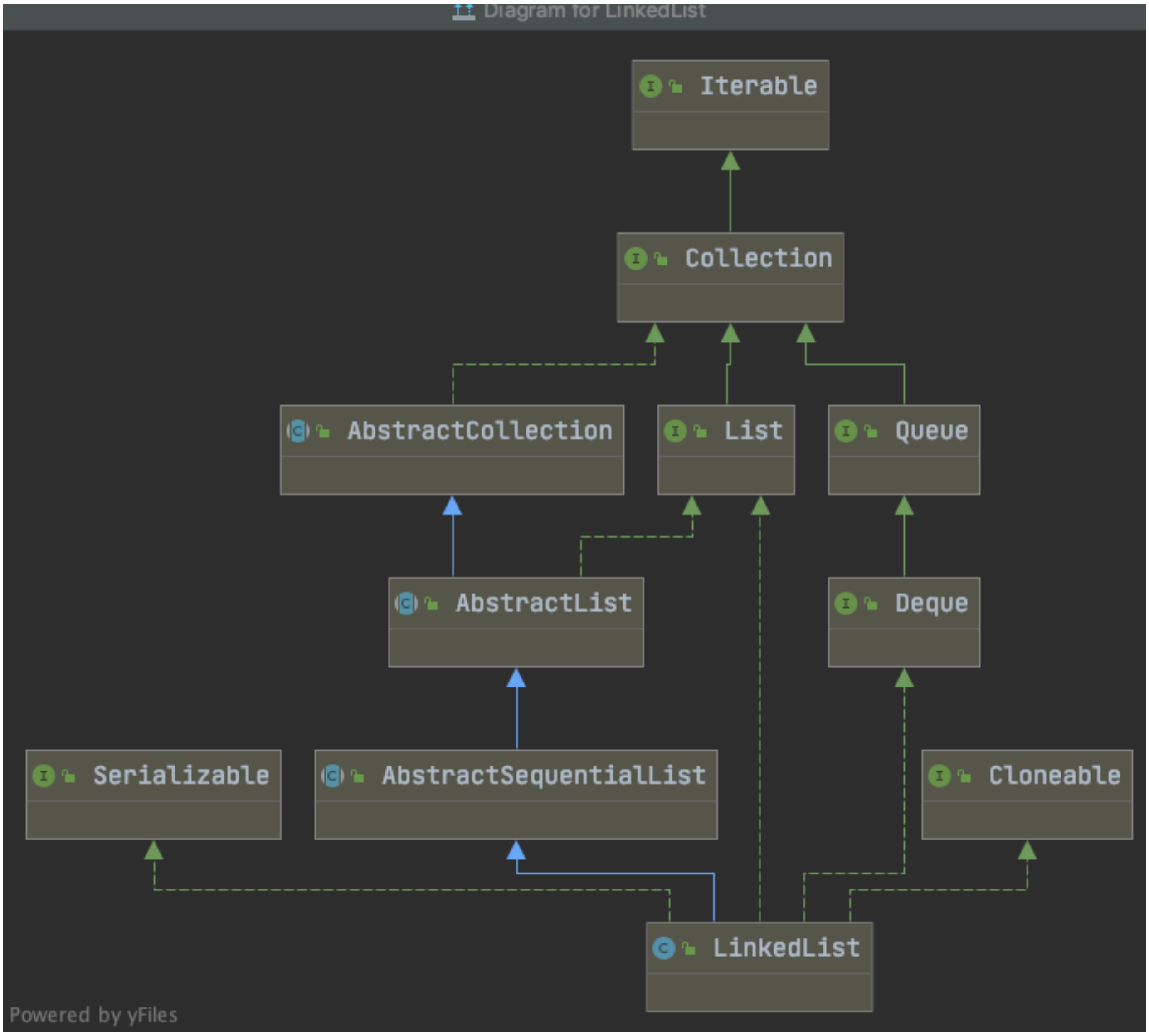
```
        int numMoved = size - index - 1; //获取index 的倒序index
        if (numMoved > 0) //如果>0 则需要将数组index+1到最后的数据  覆盖到 index 到
size-1
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--size] = null; // 如果numMoved == 0 相当于直接remove最后一个

        return oldValue; //返回备份
    }
```

addAll()

```
public boolean addAll(int index, Collection<? extends E> c) {
        rangeCheckForAdd(index);

        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacityInternal(size + numNew);

        int numMoved = size - index;
        if (numMoved > 0) //现将index之后的数据往后挪numMoved位置，给插入的数据集腾地
方
            System.arraycopy(elementData, index, elementData, index + numNew,
                             numMoved);

        System.arraycopy(a, 0, elementData, index, numNew);
        size += numNew;
        return numNew != 0;
    }
```

LinkedList

继承关系图



- 实现了`Cloneable`(空接口)代表可以调用`object.clone`方法
- 实现`Serializable`(空接口)接口 代表可以进行序列化和反序列化
- 实现了`Deque`接口 说明`LinkedList` 还是一个`双端队列`并且包含了`队列`和`栈`的所有功能
- 继承了`AbstractSequentialList`只支持迭代器按顺序 访问，不支持`RandomAccess`，所以遍历`AbstractSequentialList` 的子类，使用 for 循环 get() 的效率要 <= 迭代器遍历：

Deque方法详解

`Deque`既可以用作后进先出的栈，也可以用作先进先出的队列。

先看一下`Deque`接口中的方法

```
public interface Deque<E> extends Queue<E> {
    // *** Deque methods ***
    void addFirst(E e);  //在链表前面插入一个数据
    void addLast(E e);   //在链表后面一个数据
    boolean offerFirst(E e); //调用了addFirst 只不过最后返回true
    boolean offerLast(E e);  //调用了addLast 只不过最后返回true
```

```
    E removeFirst(); //删除并返回链表第一个数据 若数据为空则抛出NoSuchElementException
    E removeLast();  //删除并返回链表最后一个数据 若数据为空则抛出NoSuchElementException
    E pollFirst();   //删除并返回链表第一个数据 若数据为null则返回null
    E pollLast();    //删除并返回链表最后一个数据 若数据为null则返回null
    E getFirst();    //返回链表第一个数据 若数据为空则抛出NoSuchElementException
    E getLast();     //返回链表最后一个数据 若数据为空则抛出NoSuchElementException
    E peekFirst();   //返回链表第一个数据 若数据为空则返回null
    E peekLast();    //返回链表最后一个数据 若数据为空则返回null
    boolean removeFirstOccurrence(Object o);  //删除链表中指定元素的第一次出现（从头部遍历列表时尾）。如果不包含该元素，则不变。
    boolean removeLastOccurrence(Object o);  //删除链表中指定元素的最后一次出现（从头部遍历列表时尾）。如果不包含该元素，则不变。

    // *** Queue methods ***
    boolean add(E e); //在链表后面追加一个数据
    boolean offer(E e); //在链表后面追加一个数据
    E remove();   //删除链表第一个数据 若数据为空则抛出NoSuchElementException
    E poll();     //删除链表第一个数据 若数据为空则返回null
    E element();  //调用getFirst  返回链表第一个数据 若数据为空则抛出NoSuchElementException
    E peek();     //返回链表第一个数据 若数据为空则返回null

    // *** Stack methods ***
    void push(E e); //在链表前面插入一个数据
    E pop(); //返回链表第一个数据 若数据为空则返回null

    // *** Collection methods ***
    boolean remove(Object o);
    boolean contains(Object o);
    public int size();
    Iterator<E> iterator();
    Iterator<E> descendingIterator();
}
```

LinkedList 里面的核心数据结构 双向链表

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable{
  /**
     * Pointer to first node.
     * Invariant: (first == null && last == null) ||
     *            (first.prev == null && first.item != null)
     */
    transient Node<E> first; //头节点
```

```java
    /**
     * Pointer to last node.
     * Invariant: (first == null && last == null) ||
     *            (last.next == null && last.item != null)
     */
    transient Node<E> last;  //尾节点

    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
}
```

link相关方法 添加新的节点到链表

```java
/**
     * Links e as first element.
     */
    private void linkFirst(E e) { //将一个数据链接到链表的头部
        final Node<E> f = first;
        final Node<E> newNode = new Node<>(null, e, f);
        first = newNode;
        if (f == null) //如果链表头为空 则将头尾都设置成newNode
            last = newNode;
        else //否则将f的前驱节点设备newNode
            f.prev = newNode;
        size++;  //链表节点数+1
        modCount++; //操作次数+1
    }

    /**
     * Links e as last element.
     */
    void linkLast(E e) { //和上面的逻辑一样
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null)
            first = newNode;
```

```java
        else
            l.next = newNode;
        size++;
        modCount++;
    }

    /**
     * Inserts element e before non-null Node succ.
     */
    void linkBefore(E e, Node<E> succ) {
        // assert succ != null;
        final Node<E> pred = succ.prev; //获取succ的前驱节点
        final Node<E> newNode = new Node<>(pred, e, succ); //构造newNode 前驱节
点为pred 后驱节点为succ
        succ.prev = newNode; //将succ的前驱节点设置为newNode
        if (pred == null)
            first = newNode; //如果succ没有前驱节点，则将newNode设置为头节点
        else
            pred.next = newNode; //将pred的后驱节点设置为newNode
        size++;
        modCount++;
    }
```

unlink相关方法 链表里删除节点

```java
/**
     * Unlinks non-null first node f.
     */
    private E unlinkFirst(Node<E> f) {
        // assert f == first && f != null;
        final E element = f.item;
        final Node<E> next = f.next;
        f.item = null;
        f.next = null; // help GC
        first = next;
        if (next == null)
            last = null;
        else
            next.prev = null;
        size--;
        modCount++;
        return element;
    }

    /**
     * Unlinks non-null last node l.
     */
    private E unlinkLast(Node<E> l) {
```

```java
        // assert l == last && l != null;
        final E element = l.item;
        final Node<E> prev = l.prev;
        l.item = null;
        l.prev = null; // help GC
        last = prev;
        if (prev == null)
            first = null;
        else
            prev.next = null;
        size--;
        modCount++;
        return element;
    }

    /**
     * Unlinks non-null node x.
     */
    E unlink(Node<E> x) { //删除x节点 并将x的前驱和后驱节点相连
        // assert x != null;
        final E element = x.item;
        final Node<E> next = x.next;
        final Node<E> prev = x.prev;

        if (prev == null) { //如果没有前驱节点 说明这个节点是头节点
            first = next; //将此节点的后驱节点赋值给头节点
        } else {
            prev.next = next;  //将前驱节点的next节点 连接到 后驱节点
            x.prev = null; //clear
        }

        if (next == null) {
            last = prev; ///将此节点的前驱节点赋值给尾节点
        } else {
            next.prev = prev; //将后驱节点的前驱节点 连接到前驱节点
            x.next = null; //clear
        }

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

ArrayDeque

> `ArrayDeque`是`Deque`接口的一个实现，使用了可变数组，所以没有容量上的限制。

> 同时，`ArrayDeque`是线程不安全的，在没有外部同步的情况下，不能再多线程环境下使用。
>
> `ArrayDeque`是`Deque`的实现类，可以作为栈来使用，效率高于`Stack`；
>
> 也可以作为队列来使用，效率高于`LinkedList`。

```
public class ArrayDeque<E> extends AbstractCollection<E>
                           implements Deque<E>, Cloneable, Serializable
{
    /**存储双端队列的元素的数组。
      *双端队列的容量是该数组的长度,始终是2的幂。
      *数组永远不能成为已满，除了在addX方法中短暂存在的地方
      *装满后立即调整大小（请参阅doubleCapacity），从而避免头和尾缠绕在一起以使它们相等
      *我们还保证所有不包含双端队列元素的数组单元始终为空。
      */
    transient Object[] elements;

    /**
     * The index of the element at the head of the deque (which is the
     * element that would be removed by remove() or pop()); or an
     * arbitrary number equal to tail if the deque is empty.
     */
    transient int head; //头节点index

    /**
     * The index at which the next element would be added to the tail
     * of the deque (via addLast(E), add(E), or push(E)).
     */
    transient int tail; //尾节点index

    /**
     * The minimum capacity that we'll use for a newly created deque.
     * Must be a power of 2.
     */
    private static final int MIN_INITIAL_CAPACITY = 8;
}
```

add操作

上面例子使用的add方法，其实内部使用了addLast方法，addLast也就添加数据到双向队列尾端：

```
public void addLast(E e) {
    if (e == null)
        throw new NullPointerException();
    elements[tail] = e; // 根据尾索引，添加到尾端
    if ( (tail = (tail + 1) & (elements.length - 1)) == head) // 尾索引+1，如果尾索引和头索引重复了，说明数组满了，进行扩容
        doubleCapacity();
}
```

addFirst方法跟addLast方法相反，添加数据到双向队列头端：

```
public void addFirst(E e) {
    if (e == null)
        throw new NullPointerException();
    elements[head = (head - 1) & (elements.length - 1)] = e; // 根据头索引，添加到头端，头索引-1
    if (head == tail) // 如果头索引和尾索引重复了，说明数组满了，进行扩容
        doubleCapacity();
}
```

remove操作

remove操作分别removeFirst和removeLast，removeLast代码如下：

```
public E removeLast() {
    E x = pollLast(); // 调用pollLast方法
    if (x == null)
        throw new NoSuchElementException();
    return x;
}

public E pollLast() {
    int t = (tail - 1) & (elements.length - 1); // 尾索引 -1
    @SuppressWarnings("unchecked")
    E result = (E) elements[t]; // 根据尾索引，得到尾元素
    if (result == null)
        return null;
    elements[t] = null; // 尾元素置空
    tail = t;
    return result;
}
```

removeFirst方法原理一样，remove头元素。 头索引 +1

扩容

ArrayDeque的扩容会把数组容量扩大2倍，同时还会重置头索引和尾索引，头索引置为0，尾索引置为原容量的值。

比如容量为8，扩容为16，头索引变成0，尾索引变成8。

扩容代码如下：

```
private void doubleCapacity() {
    assert head == tail;
    int p = head;
    int n = elements.length;
    int r = n - p;
    int newCapacity = n << 1;
    if (newCapacity < 0)
        throw new IllegalStateException("Sorry, deque too big");
    Object[] a = new Object[newCapacity];
    System.arraycopy(elements, p, a, 0, r);
    System.arraycopy(elements, 0, a, r, p);
    elements = a;
    head = 0;  // 头索引重置
    tail = n;  // 尾索引重置
}
```

LinkedBlockingDeque

使用`双向链表`和`ReentrantLock`实现的双向的阻塞队列

```
  */
    transient Node<E> first;  //头节点

    /**
     * Pointer to last node.
     * Invariant: (first == null && last == null) ||
     *            (last.next == null && last.item != null)
     */
    transient Node<E> last;  //尾节点

    /** Number of items in the deque */
    private transient int count; //队列里的元素数量
    /** Maximum number of items in the deque */

    private final int capacity; //队列容量

    /** Main lock guarding all access */
    final ReentrantLock lock = new ReentrantLock(); //🔒

    /** Condition for waiting takes */
```

```
    private final Condition notEmpty = lock.newCondition(); //通过Condition来阻塞当队列为空时来take元素的线程

    /** Condition for waiting puts */
    private final Condition notFull = lock.newCondition(); //通过Condition来阻塞当队列满了时来put元素的线程
```

核心代码

```
private boolean linkFirst(Node<E> node) {
  // assert lock.isHeldByCurrentThread();
  if (count >= capacity) //如果队列已满
    return false;
  Node<E> f = first;
  node.next = f;
  first = node;
  if (last == null)
    last = node;
  else
    f.prev = node;
  ++count;
  notEmpty.signal(); //加入了节点，唤醒某个等待take的线程
  return true;
}

public E takeLast() throws InterruptedException {
  final ReentrantLock lock = this.lock;
  lock.lock();
  try {
    E x;
    while ( (x = unlinkLast()) == null) //如果队列为空，则进入等待
      notEmpty.await();
    return x;
  } finally {
    lock.unlock();
  }
}

private E unlinkLast() {
  // assert lock.isHeldByCurrentThread();
  Node<E> l = last;
  if (l == null) //如果队列为空，则返回null
    return null;
  Node<E> p = l.prev;
  E item = l.item;
  l.item = null;
  l.prev = l; // help GC
```

```java
    last = p;
    if (p == null)
      first = null;
    else
      p.next = null;
    --count;
    notFull.signal(); //如果正常删掉节点 则唤醒某个等待put的线程
    return item;
  }

  public void putFirst(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    Node<E> node = new Node<E>(e);
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
      while (!linkFirst(node))
        notFull.await(); //如果队列已满 则让出锁，进入等待
    } finally {
      lock.unlock();
    }
  }
```

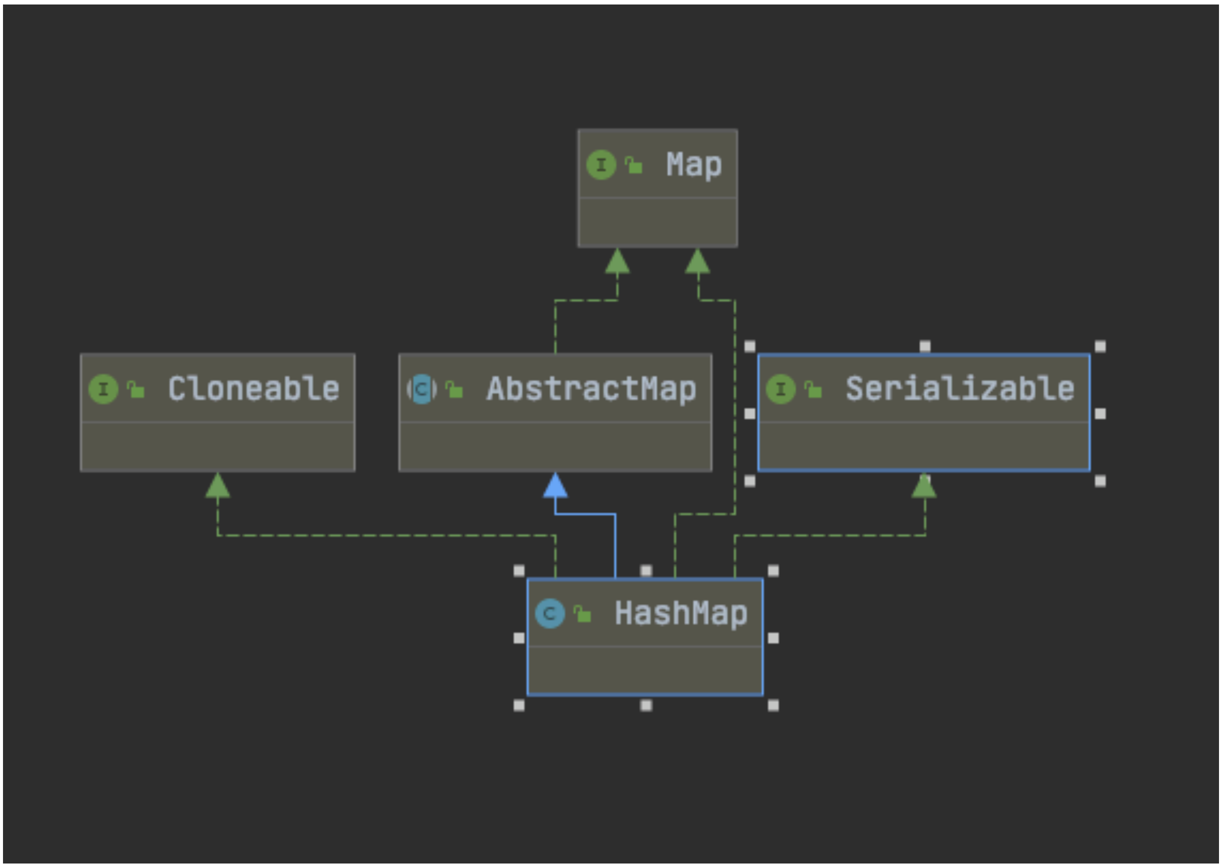
CopyOnWriteArrayList

HashSet

TreeSet

HashMap

继承关系图



默认的配置参数

```
/**
 * 默认的容量 16 (必须是2的幂)
 */
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

/**
 * 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; //最大容量 2的30次方

/**
 * The load factor used when none specified in constructor.
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f; //默认的负载因子0.75

/**
 * The bin count threshold for using a tree rather than list for a
 * bin.  Bins are converted to trees when adding an element to a
 * bin with at least this many nodes. The value must be greater
 * than 2 and should be at least 8 to mesh with assumptions in
 * tree removal about conversion back to plain bins upon
```

```
 * shrinkage.
 */
static final int TREEIFY_THRESHOLD = 8; //链表转化为树的阈值 8

/**
 * The bin count threshold for untreeifying a (split) bin during a
 * resize operation. Should be less than TREEIFY_THRESHOLD, and at
 * most 6 to mesh with shrinkage detection under removal.
 */
static final int UNTREEIFY_THRESHOLD = 6; //取消树化的阈值6

/**
 * The smallest table capacity for which bins may be treeified.
 * (Otherwise the table is resized if too many nodes in a bin.)
 * Should be at least 4 * TREEIFY_THRESHOLD to avoid conflicts
 * between resizing and treeification thresholds.
 */
static final int MIN_TREEIFY_CAPACITY = 64;

   /**
     * The table, initialized on first use, and resized as
     * necessary. When allocated, length is always a power of two.
     * (We also tolerate length zero in some operations to allow
     * bootstrapping mechanics that are currently not needed.)
     */
    transient Node<K,V>[] table; //hash表

    /**
     * Holds cached entrySet(). Note that AbstractMap fields are used
     * for keySet() and values().
     */
    transient Set<Map.Entry<K,V>> entrySet;
```

Node

hashmap中真正的数据存储结构

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;
}


 static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
        TreeNode<K,V> parent;  // red-black tree links
```

```
        TreeNode<K,V> left;
        TreeNode<K,V> right;
        TreeNode<K,V> prev;    // needed to unlink next upon deletion
        boolean red;
 }
```

get(Object key)

```
public V get(Object key) {
        Node<K,V> e;
        return (e = getNode(hash(key), key)) == null ? null : e.value;
    }

    /**
     * Implements Map.get and related methods.
     *
     * @param hash hash for key
     * @param key the key
     * @return the node, or null if none
     */
    final Node<K,V> getNode(int hash, Object key) {
        Node<K,V>[] tab; Node<K,V> first, e; int n; K k;

//如果table == null 或者 table.length==0 或者在hash表里找不到元素 则返回null  hash公式 (n-1)& hash 获取此hash值在表里的索引
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            if (first.hash == hash &&
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;  // 如果 first的hash值等于传进来的hash 并且key相等 则first就是要找的node
            if ((e = first.next) != null) { //如果first 不是要找的值 则寻找它连着的next
                if (first instanceof TreeNode) //如果first 是红黑树节点 则调用getTreeNode方法
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                do { //first 是普通的链表节点 则遍历链表找到key对应的value
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null &&
key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }
```

```java
    /**
     * Calls find for root node.
     */
    final TreeNode<K,V> getTreeNode(int h, Object k) {
      return ((parent != null) ? root() : this).find(h, k, null); //调用find
方法查询树中的节点
    }

    final TreeNode<K,V> find(int h, Object k, Class<?> kc) {
          TreeNode<K,V> p = this;
          do {
              int ph, dir; K pk;
              TreeNode<K,V> pl = p.left, pr = p.right, q;
              if ((ph = p.hash) > h) //p.hash > h 就将 p的左节点赋给p
                  p = pl;
              else if (ph < h) //p.hash < h 就将 p的右节点赋给p
                  p = pr;
              else if ((pk = p.key) == k || (k != null && k.equals(pk)))
                  return p; //如果 p的key与k相等 则返回p
              else if (pl == null)
                  p = pr; //如果没有左节点 就将右节点赋值给p
              else if (pr == null)
                  p = pl; //如果没有右节点 就将左节点赋值给p
              else if ((kc != null ||
                          //判断是否实现comparable接口 如果没实现则返回null 实现了则
返回key的class
                          (kc = comparableClassFor(k)) != null) &&
                          //根据具体类实现的comparable接口的compareTo方法判断k 与当
前pk的大小关系
                         (dir = compareComparables(kc, k, pk)) != 0)
                  p = (dir < 0) ? pl : pr; //根据大小关系 将左右节点赋给p
              else if ((q = pr.find(h, k, kc)) != null) //递归
                  return q;
              else
                  p = pl; //如果从右孩子节点递归查找后仍未找到，那么从左孩子节点进行下
一轮循环
          } while (p != null);
          return null;
      }
```

put(K key,V value)

```java
public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
```

```
    }

    /**
     * Implements Map.put and related methods.
     *
     * @param hash hash for key
     * @param key the key
     * @param value the value to put
     * @param onlyIfAbsent if true, don't change existing value
     * @param evict if false, the table is in creation mode.
     * @return previous value, or null if none
     */
    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        if ((tab = table) == null || (n = tab.length) == 0) //如果tab.length
!=0 则赋值给n
            n = (tab = resize()).length; //如果table == null 则将resize之后的
tab.length 赋给n
        if ((p = tab[i = (n - 1) & hash]) == null) //如果当前hash值在表里面没有存在
则存到当前位置
            tab[i] = newNode(hash, key, value, null);
        else { //hash表里面的位置被占据了
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p; // key相同
            else if (p instanceof TreeNode) //发生hash冲突 并且目前存储结构是树
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);
            else { //发生hash冲突 并且目前存储结构是链表
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                      //如果hash表中的冲突位置的节点没有next，则将put进来的key,value
链接到此节点后面
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash); //如果当前链表的长度超过了8则转
换成红黑树
                        break;
                    }
                    if (e.hash == hash && //如果在链表中找到了key相同的，则跳出，覆盖
它的value
                        ((k = e.key) == key || (key != null &&
key.equals(k))))
                        break;
                    p = e; //链表循环next
                }
            }
```

```
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value; //替换新值
                afterNodeAccess(e);
                return oldValue; //返回旧值
            }
        }
        ++modCount;
        if (++size > threshold)
            resize(); ////如果当前的 hash表的长度已经超过了当前 hash 需要扩容的长度，
重新扩容,条件是 haspmap 中存放的数据超过了临界值(经过测试),而不是数组中被使用的下标
        afterNodeInsertion(evict);
        return null;
    }
```

resize()

```
    final Node<K,V>[] resize() {
        Node<K,V>[] oldTab = table;
        int oldCap = (oldTab == null) ? 0 : oldTab.length; //获取当前hash表的长度
        int oldThr = threshold; //获取当前的扩容阈值
        int newCap, newThr = 0; //新的大小和扩容阈值
        if (oldCap > 0) { //hashmap已经初始化过了
            if (oldCap >= MAXIMUM_CAPACITY) { //如果旧的hash表长度大于等于最大限定长
度 则不进行扩容
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }
          // 如果数组元素个数在正常范围内，那么新的数组容量为老的数组容量的2倍（左移1位相当
于乘以2）
          //  如果扩容之后的新容量小于最大容量  并且  老的数组容量大于等于默认初始化容量
（16），那么新数组的扩容阈值设置为老阈值的2倍。
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY)
                newThr = oldThr << 1;
        }
        else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;  //如果老数组的扩容阈值大于0，那么设置新数组的容量为该阈值
这一步也就意味着构造该map的时候，指定了初始化容量。
        else {               // 第一次初始化reszie 使用默认参数进
            newCap = DEFAULT_INITIAL_CAPACITY; //默认容量
            newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY); //
默认扩容阈值 默认容量*负载因子0.75
        }
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor; //新长度*负载因子
```

```java
            newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
                      (int)ft : Integer.MAX_VALUE);
        }
        threshold = newThr; //将新阈值 赋给threshold 覆盖老的阈值
        @SuppressWarnings({"rawtypes","unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap]; //根据新容量创建hash
表
        table = newTab; //将新的hash表赋给table
        if (oldTab != null) { //如果原来的hash表里有值 则进行元素的转移
            for (int j = 0; j < oldCap; ++j) { //遍历旧的hash表
                Node<K,V> e;
                if ((e = oldTab[j]) != null) { //如果j处有值则赋值给e
                    oldTab[j] = null; //将旧表j处的数据清空
                    if (e.next == null) //如果e后没有连接节点
                        newTab[e.hash & (newCap - 1)] = e;//直接hash到新表位置
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap); //如
果当前是树存储 则调用split方法
                    else { // preserve order
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
                        Node<K,V> next;
                        do {
                            next = e.next;
                            //拿元素的hash值  和  老数组的长度  做与运算
                            // 数组的长度一定是2的N次方（例如16），如果hash值和该长度
做与运算，结果为0，就说明该hash值小于数组长度（例如hash值为7），
                            // 那么该hash值再和新数组的长度取摸的话mod值也不会放生变
化，所以该元素的在新数组的位置和在老数组的位置是相同的，所以该元素可以放置在低位链表中。
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null) //如果链表没有尾节点
                                    loHead = e; //就将e赋给链表的头节点
                                else
                                    loTail.next = e; //挂到链表尾部
                                loTail = e; //然后将尾节点设为e
                            }
                            else { //说明hash值大于数组长度 所以需要放到高位链表中
                                if (hiTail == null) //同上
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        if (loTail != null) {
                            loTail.next = null;
                            newTab[j] = loHead; // 低位的元素组成的链表还是放置在原
来的位置
```

```
                        }
                        if (hiTail != null) {
                            hiTail.next = null;// 高位的元素组成的链表放置的位置只是
在原有位置上偏移了老数组的长度个位置
                            newTab[j + oldCap] = hiHead;
                        }
                    }
                }
            }
        }
        return newTab;
    }
```

TreeMap

LinkedHashMap

ConcurrentHashMap