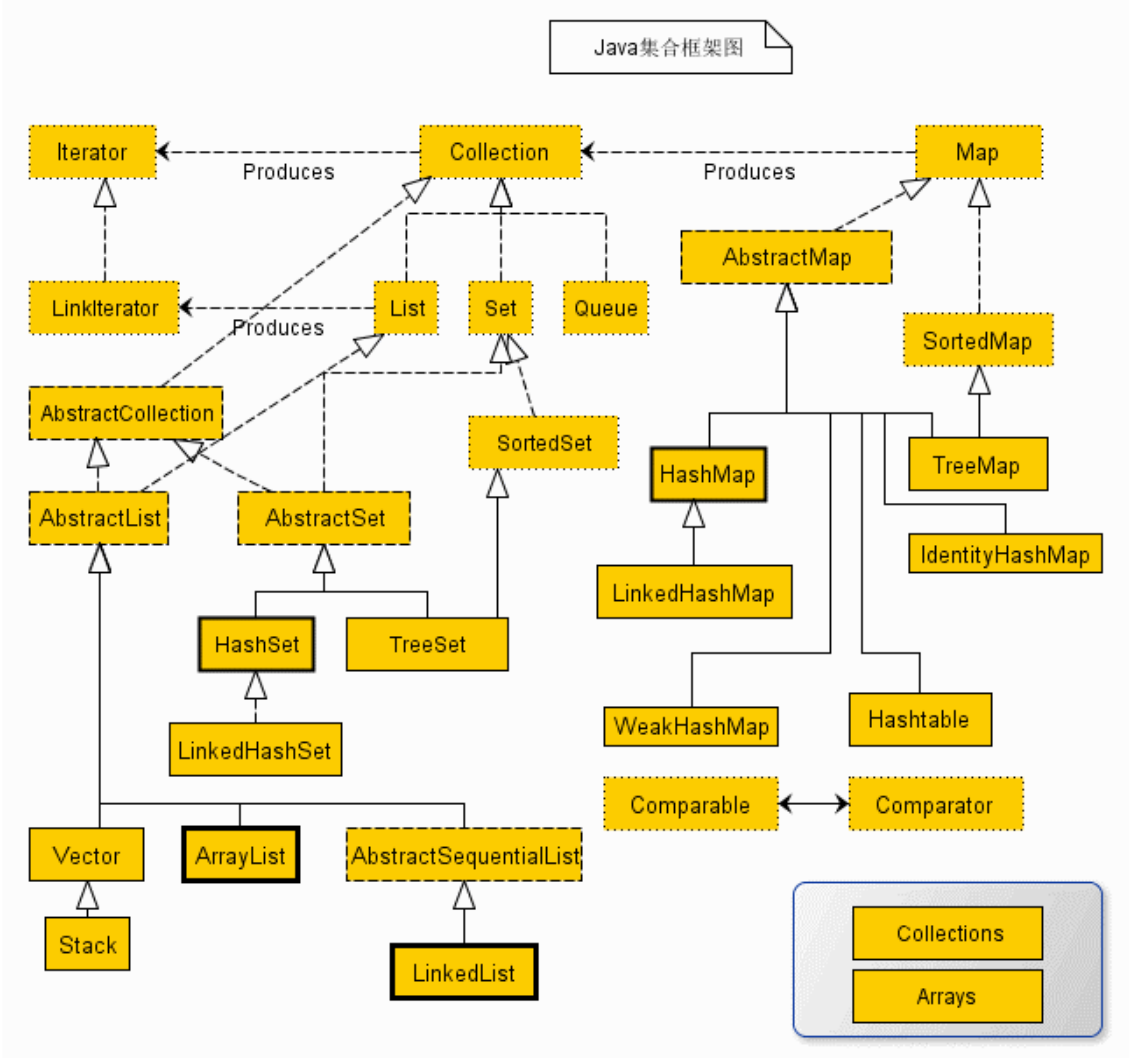
java集合

java集合基本框架

- Collection和Map是最基本的两个接口，该系列集合主要用来盛装其他对象。
- Iterator不同于Map和Iterator，它是用于迭代访问集合中的元素。
- 虚线是接口，长虚线是抽象类，细实线是继承了父类，而粗实线是同时继承了父类和实现了接口。



Iterator

在jdk1.8中，对接口Interator定义了四个方法

- boolean hasNext():判断被迭代的集合是否还有元素没有被遍历，还有元素则返回true
- Object Next()：返回集合里的下一个元素
- void remove()：删除集合里上一次next方法返回的元素,会对collection中的元素进行删除，但是不能进行赋值等操作。
- void forEachRemaining(Consumer<? super E> action)：java8才有，用Lambda表达式来遍历集合。

```
package collection;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

public class test {
    public static void main(String[] args) {
        Collection ex = new HashSet();
        ex.add("111");
        ex.add("222");
        ex.add("333");

        Iterator exIterator = ex.iterator();
        while(exIterator.hasNext()){
            String exSolo = (String) exIterator.next();
            System.out.println(exSolo);
            if(exSolo=="111"){
                exIterator.remove();
            }
        }
        System.out.println(ex);
        Iterator exIterator1= ex.iterator();
        exIterator1.forEachRemaining(obj ->System.out.println(obj));
    }
}
```

AbStractColleciotn

它是Colleciton类的一个实现类，同时集中也是一个抽象类，除开实现一些基本的collection公有方法()以外。

- public boolean isEmpty() { ..... }
- public boolean contains(){ ..... }
- pubcli boolean add(E e){ ..... }
- ....

他作为一个抽象类，也有两个抽象方法方法是需要子类去实现，

- public abstract Iterator<E> iterator();
- public abstract int size();

Set

HashSet

实现和继承状态

- extends AbstractSet<E>：其中AbstractSet是extends AbstractCollection<E> implements Set<E>
- implements Set<E> ：实现set接口
- implements Cloneable ：Cloneable是标记型的接口，它们内部都没有方法和属性，实现Cloneable来表示该对象能被克隆，能使用Object.clone()方法。如果没有实现 Cloneable的类对象调用clone()就会抛出CloneNotSupportedException。（这个之后再去琢磨下Cloneable的作用）
- implements java.io.Serializable: 实现了可序列化

源码解析

```
package java.util;

import java.io.InvalidObjectException;
import sun.misc.SharedSecrets;

/**

 * 这个类是继承了Set接口，被哈希表支持（hashMap），它不保证set集合中元素的顺序，也不能保证，这个顺序会一直保持不变，同时它允许空值存在
 *
 *这个类提供了基本的add,remove,contains,size方法。假设哈希函数在buckets中散列分布，遍历这个set集合时间和实例大小加上buckets的数量成正相关，初始容量不能太高
 *
 *注意到这个实现类不是同步的，如果多线程同时访问这个hashset，b并至少有一个要修改的话，它必须要外部同步，这个通过在某个对象上来同步封装该集合实现
 *
 * 即在用iterator遍历set时，如果原set的数据被修改，就会发生ConcurrentModificationException错误。只能iterator的remove方法。这个称为fail-fast机制
 *
 */

public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    static final long serialVersionUID = -5024744406713321676L;

    //底层使用HashMap来保存HashSet中所有元素
    private transient HashMap<E,Object> map;
```

```
    // 定义一个虚拟的Object对象作为HashMap的value，将此对象定义为static final。
    private static final Object PRESENT = new Object();

    /**
     * 默认的无参构造器，会初始化一个初始容量为16，和加载因子为0.75的hashmap
     */
    public HashSet() {
        map = new HashMap<>();
    }

    /**
     * 构造一个包含指定collection中的元素的新set
     * 实际底层是按默认的加载因子和初始容量来进行map的初始化
     * 通过addAll的方法存放在次set中的collection中
     */
    public HashSet(Collection<? extends E> c) {
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
    }

    /**
     * 以指定的initialCapacity和loadFactor构造一个空的HashSet，实际底层是以相应的参数构造一个空的HashMap
     *
     * @param      initialCapacity   初始容量
     * @param      loadFactor        加载因子
     * @throws     IllegalArgumentException 当为负数时报错
     */
    public HashSet(int initialCapacity, float loadFactor) {
        map = new HashMap<>(initialCapacity, loadFactor);
    }

    /**
     * 以指定的initialCapacity构造一个空的HashSet，实际底层是以相应参数和0.75的加载因子初始化的空hashmap。
     *
     * @param      initialCapacity   the initial capacity of the hash table
     * @throws     IllegalArgumentException if the initial capacity is less
     *             than zero
     */
    public HashSet(int initialCapacity) {
        map = new HashMap<>(initialCapacity);
    }

    /**
     * 以指定的initialCapacity和loadFactor构造一个新的空链接哈希集合。
     * 此构造函数为包访问权限，不对外公开，实际只是是对LinkedHashSet的支持。
     *
     * 实际底层会以指定的参数构造一个空LinkedHashMap实例来实现。
     *
     * @param      initialCapacity   the initial capacity of the hash map
     * @param      loadFactor        the load factor of the hash map
     * @param      dummy             标记
     * @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);
    }

    /**
     * 返回对set中元素的迭代，返回顺序不确定
     *
     * @return an Iterator over the elements in this set
     * @see ConcurrentModificationException
     */
    public Iterator<E> iterator() {
        return map.keySet().iterator();
    }

    /**
     * 返回此set中的元素的数量（set的容量）。
     *
     * 底层实际调用HashMap的size()方法返回Entry的数量，就得到该Set中元素的个数。
     */
    public int size() {
        return map.size();
    }

    /**
     * 如果此set不包含任何元素，则返回true。
     *
     * 底层实际调用HashMap的isEmpty()判断该HashSet是否为空。
     * @return 如果此set不包含任何元素，则返回true。
     */
    public boolean isEmpty() {
        return map.isEmpty();
    }

    /**
     * 如果此set包含指定元素，则返回true。
     * 更确切地讲，当且仅当此set包含一个满足(o==null ? e==null : o.equals(e))
     * 的e元素时，返回true。
     *
     * 底层实际调用HashMap的containsKey判断是否包含指定key。
     * @param o 在此set中的存在已得到测试的元素。
     * @return 如果此set包含指定元素，则返回true。
     */
    public boolean contains(Object o) {
        return map.containsKey(o);
    }

    /**
     * 如果此set中尚未包含指定元素，则添加指定元素。
     * 更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2))
     * 的元素e2，则向此set 添加指定的元素e。
     * 如果此set已包含该元素，则该调用不更改set并返回false。
     *
     * 底层实际将将该元素作为key放入HashMap。
     * 由于HashMap的put()方法添加key-value对时，当新放入HashMap的Entry中key
     * 与集合中原有Entry的key相同（hashCode()返回值相等，通过equals比较也返回true），
     * 新添加的Entry的value会将覆盖原来Entry的value，但key不会有任何改变，
```

```
     * 因此如果向HashSet中添加一个已经存在的元素时，新添加的集合元素将不会被放入HashMap中，
     * 原来的元素也不会有任何改变，这也就满足了Set中元素不重复的特性。
     * @param e 将添加到此set中的元素。
     * @return 如果此set尚未包含指定元素，则返回true
     */
    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }

    /**
     * 如果指定元素存在于此set中，则将其移除。
     * 更确切地讲，如果此set包含一个满足(o==null ? e==null : o.equals(e))的元素e，
     * 则将其移除。如果此set已包含该元素，则返回true
     * （或者：如果此set因调用而发生更改，则返回true）。（一旦调用返回，则此set不再包含该元
素）。
     *
     * 底层实际调用HashMap的remove方法删除指定Entry。
     * @param o 如果存在于此set中则需要将其移除的对象。
     * @return 如果set包含指定元素，则返回true。
     */
    public boolean remove(Object o) {
        return map.remove(o)==PRESENT;
    }

    /**
     * 从此set中移除所有元素。此调用返回后，该set将为空。
     * 底层实际调用HashMap的clear方法清空Entry中所有元素。
     */
    public void clear() {
        map.clear();
    }

    /**
     * Returns a shallow copy of this <tt>HashSet</tt> instance: the elements
     * themselves are not cloned.
     *
     * @return a shallow copy of this set
     */
    @SuppressWarnings("unchecked")
    public Object clone() {
        try {
            HashSet<E> newSet = (HashSet<E>) super.clone();
            newSet.map = (HashMap<E, Object>) map.clone();
            return newSet;
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e);
        }
    }

    /**
     * Save the state of this <tt>HashSet</tt> instance to a stream (that is,
     * serialize it).
     *
     * @serialData The capacity of the backing <tt>HashMap</tt> instance
     *             (int), and its load factor (float) are emitted, followed by
     *             the size of the set (the number of elements it contains)
     *             (int), followed by all of its elements (each an Object) in
```

```
     *             no particular order.
     */
    private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException {
        // Write out any hidden serialization magic
        s.defaultWriteObject();

        // Write out HashMap capacity and load factor
        s.writeInt(map.capacity());
        s.writeFloat(map.loadFactor());

        // Write out size
        s.writeInt(map.size());

        // Write out all elements in the proper order.
        for (E e : map.keySet())
            s.writeObject(e);
    }

    /**
     * Reconstitute the <tt>HashSet</tt> instance from a stream (that is,
     * deserialize it).
     */
    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        // Read in any hidden serialization magic
        s.defaultReadObject();

        // Read capacity and verify non-negative.
        int capacity = s.readInt();
        if (capacity < 0) {
            throw new InvalidObjectException("Illegal capacity: " +
                                             capacity);
        }

        // Read load factor and verify positive and non NaN.
        float loadFactor = s.readFloat();
        if (loadFactor <= 0 || Float.isNaN(loadFactor)) {
            throw new InvalidObjectException("Illegal load factor: " +
                                             loadFactor);
        }

        // Read size and verify non-negative.
        int size = s.readInt();
        if (size < 0) {
            throw new InvalidObjectException("Illegal size: " +
                                             size);
        }
        // Set the capacity according to the size and load factor ensuring that
        // the HashMap is at least 25% full but clamping to maximum capacity.
        capacity = (int) Math.min(size * Math.min(1 / loadFactor, 4.0f),
                HashMap.MAXIMUM_CAPACITY);

        // Constructing the backing map will lazily create an array when the first
element is
        // added, so check it before construction. Call HashMap.tableSizeFor to compute
the
        // actual allocation size. Check Map.Entry[].class since it's the nearest public
```

```
type to
        // what is actually created.

        SharedSecrets.getJavaOISAccess()
                     .checkArray(s, Map.Entry[].class, HashMap.tableSizeFor(capacity));

        // Create backing HashMap
        map = (((HashSet<?>)this) instanceof LinkedHashSet ?
               new LinkedHashMap<E,Object>(capacity, loadFactor) :
               new HashMap<E,Object>(capacity, loadFactor));

        // Read in all elements in the proper order.
        for (int i=0; i<size; i++) {
            @SuppressWarnings("unchecked")
                E e = (E) s.readObject();
            map.put(e, PRESENT);
        }
    }

    /**
     * Creates a <em><a href="Spliterator.html#binding">late-binding</a></em>
     * and <em>fail-fast</em> {@link Spliterator} over the elements in this
     * set.
     *
     * <p>The {@code Spliterator} reports {@link Spliterator#SIZED} and
     * {@link Spliterator#DISTINCT}.  Overriding implementations should document
     * the reporting of additional characteristic values.
     *
     * @return a {@code Spliterator} over the elements in this set
     * @since 1.8
     */
    public Spliterator<E> spliterator() {
        return new HashMap.KeySpliterator<E,Object>(map, 0, -1, 0, 0);
    }
}
```

相等判断

因为hashset底层是用hashmap实现的，然后hashmap的k是set的值。在判断是否相同时，主要通过equals和hashcode两个方法来进行判断，两个返回的结果都相同时，我们才会默认两个元素是相同的。

hashset集合里没有索引，当进行添加元素时，实质是通过计算该值的hashcode，然后来计算其存储位置，。优点是可以长度是可变的而不是固定的。

LinkedHashSet

能够对其中的元素进行排序，即其中的元素顺序和插入的顺序始终保持一致。

继承和实现关系

- extends HashSet

- implements Cloneable
- implements java.io.Serializable
- 四个基本方法的super其实是通过LinkedHashMap实现的

源码

```
package java.util;

/**
 *
 *
 */

public class LinkedHashSet<E>
    extends HashSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {

    private static final long serialVersionUID = -2851667679971038690L;

    /**
     * 调用父类方法，点进去是用LiknedHashMap实现的
     */
    public LinkedHashSet(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor, true);
    }

    /**
     * 调用父类方法，点进去是用LiknedHashMap实现的
     */
    public LinkedHashSet(int initialCapacity) {
        super(initialCapacity, .75f, true);
    }

    /**
     * 调用父类方法，点进去是用LiknedHashMap实现的
     */
    public LinkedHashSet() {
        super(16, .75f, true);
    }

    /**
     * 调用父类方法，点进去是用LiknedHashMap实现的
     */
    public LinkedHashSet(Collection<? extends E> c) {
        super(Math.max(2*c.size(), 11), .75f, true);
        addAll(c);
    }

    /**
     * Creates a <em><a href="Spliterator.html#binding">late-binding</a></em>
     * and <em>fail-fast</em> {@code Spliterator} over the elements in this set.
     *
     * <p>The {@code Spliterator} reports {@link Spliterator#SIZED},
     * {@link Spliterator#DISTINCT}, and {@code ORDERED}.  Implementations
     * should document the reporting of additional characteristic values.
```

```
     *
     * @implNote
     * The implementation creates a
     * <em><a href="Spliterator.html#binding">late-binding</a></em> spliterator
     * from the set's {@code Iterator}.  The spliterator inherits the
     * <em>fail-fast</em> properties of the set's iterator.
     * The created {@code Spliterator} additionally reports
     * {@link Spliterator#SUBSIZED}.
     *
     * @return a {@code Spliterator} over the elements in this set
     * @since 1.8
     */
    @Override
    public Spliterator<E> spliterator() {
        return Spliterators.spliterator(this, Spliterator.DISTINCT |
Spliterator.ORDERED);
    }
}
```

TreeSet

继承和父类关系

- extends AbstractSet<E>：所以它是一个Set集合，具有Set的属性和方法。
- implements NavigableSet<E>:NavigableSet也是继承了SortedSet父类的接口。意味着它支持一系列的导航方法。比如查找与指定目标最匹配项。
- implements Cloneable,:意味着它能被克隆。
- implements java.io.Serializable:意味着它支持序列化

源码分析

```
    private transient NavigableMap<E,Object> m;

    //新建一个空白的Object对象作为底层map实现的value
    private static final Object PRESENT = new Object();

    /**
     * TreeSet实质都是利用NavigebleMap去实现自己
     */
    TreeSet(NavigableMap<E,Object> m) {
        this.m = m;
    }

    public TreeSet() {
        this(new TreeMap<E,Object>());
    }

    public TreeSet(Collection<? extends E> c) {
        this();
        addAll(c);
    }
```

排序

- 自然排序

```
    public TreeSet(Comparator<? super E> comparator) {
        this(new TreeMap<>(comparator));
    }
```

TreeSet底层的comparator实质是利用treemap的comparator来实现。

通常默认实现了Comparable接口的类就可以调用obj.compareTo(obj2)方法来进行比较。包括Data，Time,String,Boolean,Character,BigDecimal&BigInteger以及所有数值类型对应的包装类。

所以往TreeSet里添加的元素必须是实现了Comparable接口的元素，否则无法进行大小比较从而排序，同时还必须是同一个类的对象，才可以进行比较，比如String和Date是无法进行比较的，放进去也会报错。
TreeSet是采用红黑树的数据结构进行存储，而不是通过hash算法计算其hash值来决定存储位置。

同样，对某个类重写compareTo方法时，也要与equals方法保持一致，如果compareTo返回0，那equals也要返回true。不然会发生冲突

- 定制排序 在创建TreeSet集合对象时，提供一个Comparator对象与该TreeSet集合关联，由该Comparator对象负责集合元素的排序逻辑。两种思路
  - 某个实现了Comparable接口的类，直接重写compareTo方法，然后在treeset中直接添加该元素
  - 在新建TreeSet时，用Lambda对象来重写comparator对象

Set实现类的性能分析

TreeSet：需要额外的红黑树算法类维持集合顺序，线程不安全

HashSet：查询效率极高，线程不安全

LinkedHashSet：HashSet的子类，底层也是用hash算法，但也使用了链表来维持元素的先后添加顺序。比HashSet速度慢，但因为有链表存在，所以遍历就很快，线程不安全

List

ArrayList

继承和实现关系

- extends AbstractList<E>：继承AbstractList，主要是实现了List接口的一些方法，同时有个抽象方法abstract public E get(int index);，这个是需要子类自己去实现的

- implements List<E>：实现list接口，一些基本方法等
- RandomAccess: RandomAccess 就是一个标记接口，用于标明实现该接口的List支持快速随机访问，主要目的是使算法能够在随机和顺序访问的List中性能更加高效（在Collections二分查找时）。

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
{
    if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
        return Collections.indexedBinarySearch(list, key);
    else
        return Collections.iteratorBinarySearch(list, key);
}
```

- Cloneable, ：克隆
- java.io.Serializable：可序列化

源码解析

jdk1.8源码解析

```
/**
     * 初始化容量大小为10
     */
    private static final int DEFAULT_CAPACITY = 10;

    /**
     * 指定数组容量为0时，返回该空数组
     */
    private static final Object[] EMPTY_ELEMENTDATA = {};

    /**
     * 在调用无参构造器返回的数组实例，其内部数据量是0，当第一次添加元素时，会通过ensureCapacityInternal()变成默认容量为10的数组。
     */
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    /**
     * 存储ArrayList元素的缓冲数组，ArrayList的容量就是这个array缓冲数组的长度，每一个elementData=默认空数组的空arraylist添加第一个元素时都会默认容量为10
     */
    transient Object[] elementData; // non-private to simplify nested class access

    /**
     * The size of the ArrayList (the number of elements it contains).实际的元素值数量
     */
    private int size;

    /**
     * Constructs an empty list with the specified initial capacity.
     * 按给定的容量大小创建空数组
     * @param  initialCapacity  the initial capacity of the list
     * @throws IllegalArgumentException if the specified initial capacity
```

```
     *         is negative
     */
    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);
        }
    }

    /**
     * 无参构造函数：
     * 创建一个 空的 ArrayList，此时其内数组缓冲区 elementData = {}, 长度为 0
     * 当元素第一次被加入时，扩容至默认容量 10
     */
    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    /**
     * 创建一个包含collection的ArrayList
     * @param c 要放入 ArrayList 中的集合，其内元素将会全部添加到新建的 ArrayList 实例中
     * @throws NullPointerException 当参数 c 为 null 时抛出异常
     */
    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;
        }
    }
```

特点

- arraylist的判断size>capacity的时候，会通过grow方法来增加数组长度，通常是增加1.5倍，是通过位移的方法实现。初始化是从0-10，然后之后是每次扩大1.5倍
- arraylist是线程不安全的，需要通过Collecionts的工具类来使其变得线程安全。

LinkedList

继承和实现关系

- extends AbstractSequentialList<E>：java类库中只有 LinkedList继承了这个抽象类，正如其名，它提供了对序列的连续访问的抽象：

- implements List<E>,：实现list接口
- Deque<E>:LinkedList的底层是 Deque双向链表，实现了 Deque接口，而 Deque接口继承于 Queue接口，因此，在java中，如果要实现队列，一般都使用 LinkedList来实现。
- Cloneable,：克隆
- java.io.Serializable： 序列化

源码分析

```
//实现Serilizable接口时，将不需要序列化的属性前添加关键字transient，序列化对象的时候，这个属性就不会序列化到指定的目的地中。
      transient int size = 0;
      //指向首节点
      transient Node<E> first;
      //指向最后一个节点
      transient Node<E> last;
      //构建一个空列表
      public LinkedList() {
      }
      //构建一个包含集合c的列表
      public LinkedList(Collection<? extends E> c) {
          this();
          addAll(c);
      }
```

LinkedList底层是一个双向链表去实现，首尾结点可以为null，相比于arraylist它的优势是对数据进行删除或者添加，或者顺序访问时，性能更好。在进行根据index的随机访问时，会先用二分法判断从头开始，还是从尾开始。
但arraylist因为底层是用数组实现，所以它的随机访问效率更高，而且在末尾进行add操作的话效率也更高，但是要在除尾部外进行增加或者删除操作，则要进行列表分段复制再还原的操作，耗能更大。

Vector

继承和实现关系

- extends AbstractList<E>：继承AbstractList抽象类的实现方法，没有重写get方法
- implements List<E>：实现list接口
- RandomAccess, ： 标记接口，用于标明实现该接口的List支持快速随机访问，主要目的是使算法能够在随机和顺序访问的List中性能更加高效
- Cloneable：克隆
- java.io.Serializable：序列化

源码分析

```
protected Object[] elementData;
```

```java
    /**
     * The number of valid components in this {@code Vector} object.
     * Components {@code elementData[0]} through
     * {@code elementData[elementCount-1]} are the actual items.
     *
     * @serial
     */
    protected int elementCount;

    /**
     * The amount by which the capacity of the vector is automatically
     * incremented when its size becomes greater than its capacity.  If
     * the capacity increment is less than or equal to zero, the capacity
     * of the vector is doubled each time it needs to grow.
     *
     * @serial
     */
    protected int capacityIncrement;

    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = -2767605614048989439L;

    /**
     * Constructs an empty vector with the specified initial capacity and
     * capacity increment.
     *
     * @param   initialCapacity     the initial capacity of the vector
     * @param   capacityIncrement   the amount by which the capacity is
     *                              increased when the vector overflows
     * @throws IllegalArgumentException if the specified initial capacity
     *         is negative
     */
    public Vector(int initialCapacity, int capacityIncrement) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        this.elementData = new Object[initialCapacity];
        this.capacityIncrement = capacityIncrement;
    }

    /**
     * Constructs an empty vector with the specified initial capacity and
     * with its capacity increment equal to zero.
     *
     * @param   initialCapacity   the initial capacity of the vector
     * @throws IllegalArgumentException if the specified initial capacity
     *         is negative
     */
    public Vector(int initialCapacity) {
        this(initialCapacity, 0);
    }

    /**
     * Constructs an empty vector so that its internal data array
     * has size {@code 10} and its standard capacity increment is
     * zero.
     */
```

```
    public Vector() {
        this(10);
    }

    /**
     * Constructs a vector containing the elements of the specified
     * collection, in the order they are returned by the collection's
     * iterator.
     *
     * @param c the collection whose elements are to be placed into this
     *       vector
     * @throws NullPointerException if the specified collection is null
     * @since   1.2
     */
    public Vector(Collection<? extends E> c) {
        elementData = c.toArray();
        elementCount = elementData.length;
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, elementCount, Object[].class);
    }
```

特点

除开构造器以外的各种方法，都加上了synchronized来进行修饰，即本身就是线程安全的，其他的初始化和基本方法和arraylist差不多

Stack

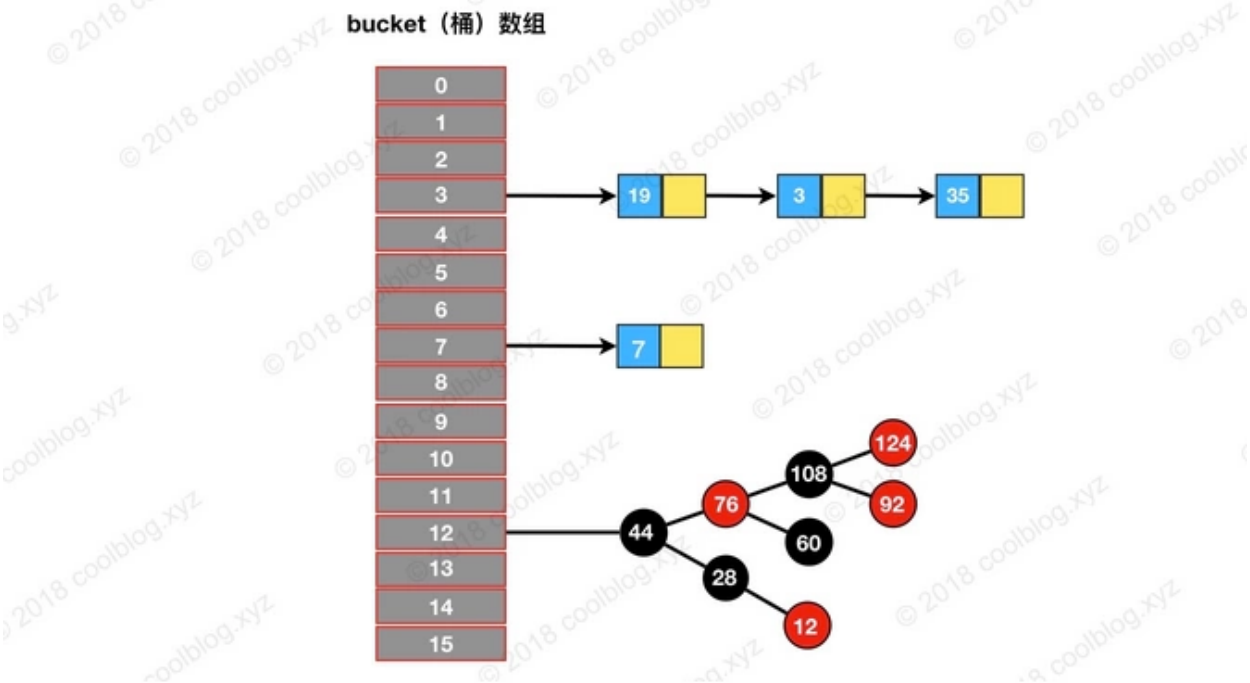
继承Vector，除开空参构造，其他的方法也是都加了synchronized的修饰，来保证同步
（peek,pop,empty,push,search）

Map

values():方法是获取集合中的所有的值----没有键，没有对应关系，
KeySet(): 将Map中所有的键存入到set集合中。因为set具备迭代器。所有可以迭代方式取出所有的键，再根据get方法。获取每一个键对应的值。 keySet():迭代后只能通过get()取key
entrySet()： Set<Map.Entry<K,V>> entrySet() //返回此映射中包含的映射关系的 Set 视图。
Map.Entry表示映射关系。entrySet()：迭代后可以e.getKey()，e.getValue()取key和value。返回的是Entry接口 。

HashMap

HashMap底层实质是基于hash算法实现的。散列算法分为散列再探测和拉链式，hashmap采用的是拉链式。整个的数据结构如下。



对于拉链式的算法，其数据结构是由数组+链表（红黑树）来形成的，在进行增删改查的操作时，需要先定位到元素所在的桶的位置，之后再在链表中遍历，确定元素的位置，没有就添加，相同则覆盖，

继承和实现关系

- extends AbstractMap<K,V>：继承AbstractMap的一些基本方法，
- implements Map<K,V>：实现Map接口
- implements Cloneable：实现克隆方法
- implements Serializable：实现可序列化

源码中的关键变量

源码分析链接

- initialCapacity(hashmap初始容量)：在初始化HashMap时的容量大小，初始值默认是16。1<<4.

+ loadFactor(负载因子)：当负载因子变小时，HashMap 所能容纳的键值对数量变少。所以键值对数量很容易大于阈值，当大于时就会发生扩容，重新将键值对存储新的桶数组里，键的键之间产生的碰撞会下降，链表长度变短。此时，HashMap的增删改查等操作的效率将会变高，这里是典型的拿空间换时间。相反，如果增加负载因子（负载因子可以大于1），HashMap所能容纳的键值对数量变多，空间利用率高，但碰撞率也高。这意味着链表长度变长，效率也随之降低，这种情况是拿时间换空间。至于负载因子怎么调节，这个看使用场景了
+ threshold(当前hashmap所能容纳键值对数量的最大值：阈值):阈值等于capacity*loadfactor。当实际需要存储的数量大于阈值时，就会发生扩容。

```
        /** The default initial capacity - MUST be a power of two. */
        static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; //即默认初始容量是16
```

```
        /** The load factor used when none specified in constructor. */
        static final float DEFAULT_LOAD_FACTOR = 0.75f;

        final float loadFactor;

        /** The next size value at which to resize (capacity * load factor). */
        int threshold;

        //当add一个元素到某个位桶，其链表长度达到8时将链表转换为红黑树
        static final int TREEIFY_THRESHOLD = 8;

        //位桶的链表长度小于6时，解散红黑树
        static final int UNTREEIFY_THRESHOLD = 6;

        //默认的最小的扩容量64，为避免重新扩容冲突，至少为4 * TREEIFY_THRESHOLD=32，即默认初始容量的2倍
        static final int MIN_TREEIFY_CAPACITY = 64;
```

HashMap如何确定元素位置

1.确定bucket中的位置 先通过key.hashCode计算其hash值，返回一个int类型的数值，直接用这个结果去映射键值对，大概有40亿映射空间。h ^ (h >>> 16)：让hash值高位参与计算，增加扰动，优化散列值的分布。

```
 static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }
```

计算出hash值之后，是通过(length - 1) & hash方法来确定在桶数组中的位置，HashMap 中桶数组的大小 length 总是2的幂，此时，(n - 1) & hash 等价于对 length取余。但取余的计算效率没有位运算高，所以(n - 1) & hash也是一个小的优化

```
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;
        //定位键值所在桶的位置
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
```

```
            //当桶的第一个位置hash值不为空且相同，则返回第一个结点
            if (first.hash == hash && // always check first node
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;
            if ((e = first.next) != null) {
                //如果first是TrreNode类型，则用红黑树的查找法
                if (first instanceof TreeNode)
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                do {
                //按链表进行查找
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }
```

2.确定在链表中的位置

- 当确定是哪一个桶之后，先判断这个点是否为null，如果是null，则直接对这个Node进行赋值。
- 若该结点有值，则对该链表(红黑树)进行遍历
  - 若遍历完没有相同值，则在末尾添加新的结点
    - 若当链表长度大于8时，则转换成红黑树然后添加。
  - 若有相同的值，则进行覆盖

HashMap中数组容量扩充的条件

在 HashMap 中，桶数组的长度均是2的幂，阈值大小为桶数组长度与负载因子的乘积。当 HashMap 中的键值对数量超过阈值时，进行扩容。

HashMap 的扩容机制与其他变长集合的套路不太一样，HashMap 按当前桶数组长度的2倍进行扩容，阈值也变为原来的2倍（如果计算过程中，阈值溢出归零，则按阈值公式重新计算）。扩容之后，要重新计算键值对的位置，并把它们移动到合适的位置上去。以上就是 HashMap 的扩容大致过程，接下来我们来看看具体的实现：

解决hash冲突的开放地址法

hash冲突:指的是不同的key，经过hash算法得出的hash值，是相同的，hash(key)，即在bucket的数组中，他们是在同一个bucket的位置，这个时候就会发生冲突。
拉链法：在每个bucket中，我们用链表（jdk1.8后红黑树）来实现在同一个bucket下存储不同的key值。

- 优点
  - 拉链法处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，因此平均查找长度较短；
  - 由于拉链法中各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况
  - 开放定址法为减少冲突，要求装填因子α较小，故当结点规模较大时会浪费很多空间。而拉链法中可取α≥1，且结点较大时，拉链法中增加的指针域可忽略不计，因此节省空间；

    - 在用拉链法构造的散列表中，删除结点的操作易于实现。只要简单地删去链表上相应的结点即可。
- 缺点：指针需要额外的空间，故当结点规模较小时，开放定址法较为节省空间，而若将节省的指针空间用来扩大散列表的规模，可使装填因子变小，这又减少了开放定址法中的冲突，从而提高平均查找速度。

开放地址法：开放地址法有个非常关键的特征，就是所有输入的元素全部存放在哈希表里，也就是说，位桶的实现是不需要任何的链表来实现的，换句话说，也就是这个哈希表的装载因子不会超过1。它的实现是在插入一个元素的时候，先通过哈希函数进行判断，若是发生哈希冲突，就以当前地址为基准，根据再寻址的方法（探查序列），去寻找下一个地址，若发生冲突再去寻找，直至找到一个为空的地址为止。所以这种方法又称为再散列法

- 线性查找法：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。缺点是容易造成联系的位置都被占用，容易产生聚集。
- 二次查找，伪随机数查找法等：都是用一种特定的方法，判定下一次空bucket的位置，若不为空则继续遍历。
- 再哈希法：构造不同的哈希算法，当第一个冲突时，换第二个方法算，直到冲突不再产生，这种方法不易产生聚集，但增加了计算时间。

利用开放地址法，在对结点进行删除时，不像拉链法那么简单，因为删除某节点后，将截断在它之后填人散列表的同义词结点的查找路径。这是因为各种开放地址法中，空地址单元(即开放地址)都是查找失败的条件。因此在 用开放地址法处理冲突的散列表上执行删除操作，只能在被删结点上做删除标记，而不能真正删除结点
装填因子：a=n/m 其中n 为关键字个数，m为表长。
加载因子是表示Hsah表中元素的填满的程度.若:加载因子越大,填满的元素越多,好处是,空间利用率高了,但:冲突的机会加大了.反之,加载因子越小,填满的元素越少,好处是:冲突的机会减小了,但:空间浪费多了.
冲突的机会越大,则查找的成本越高.反之,查找的成本越小.因而,查找时间就越小.

表 9.2　不同处理冲突的平均查找长度

处理冲突的方法	平均查找长度	
	查找成功时	查找不成功时
线性探测法	$S_{nl}\approx\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$U_{nl}\approx\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
二次探测法与双哈希法	$S_{nr}\approx-\frac{1}{\alpha}\ln(1-\alpha)$	$U_{nr}\approx\frac{1}{1-\alpha}$
链地址法	$S_{nc}\approx1+\frac{\alpha}{2}$	$U_{nc}\approx\alpha+e^{-\alpha}$

hashmap产生闭环的原因

在rehash扩容过程中，会将原先bucket里的元素通过transfer的方法转移到新的数组和链当中去，但因为链表头插法的会颠倒原来一个散列桶里面链表的顺序。在并发的时候原来的顺序被另外一个线程a颠倒了，而被挂起线程b恢复后拿扩容前的节点和顺序继续完成第一次循环后，又遵循a线程扩容后的链表顺序重新排列链表中的顺序，最终形成了环。

比如本来是3->7->9.若是单线程操作经过transfer后是9—>7->3。若是两个线程同时发生操作，A线程在确定e是3，next是7后被挂起。B以单线程形式走完变成了9—>7->3->null。A重新开始运行，遵循9—>7->3->null这个进行颠倒，于是就形成了3->7->9.然后形成了闭环。

```
/**
 * Rehashes the contents of this map into a new array with a
 * larger capacity.  This method is called automatically when the
 * number of keys in this map reaches its threshold.
 *
 * If current capacity is MAXIMUM_CAPACITY, this method does not
 * resize the map, but sets threshold to Integer.MAX_VALUE.
 * This has the effect of preventing future calls.
 *
 * @param newCapacity the new capacity, MUST be a power of two;
 *        must be greater than current capacity unless current
 *        capacity is MAXIMUM_CAPACITY (in which case value
 *        is irrelevant).
 */
void resize(int newCapacity) {
    // 缓存就哈希表数据
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }
    // 用扩容容量创建一个新的哈希表
    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable, initHashSeedAsNeeded(newCapacity));
    table = newTable;
    threshold = (int)Math.min(newCapacity * loadFactor, MAXIMUM_CAPACITY + 1);
}

/**
 * 把所有条目从当前哈希表转移到新哈希表
 */
void transfer(Entry[] newTable, boolean rehash) {
    int newCapacity = newTable.length;
    for (Entry<K,V> e : table) {
        while(null != e) {
            Entry<K,V> next = e.next;
            if (rehash) {
                e.hash = null == e.key ? 0 : hash(e.key);
            }
            int i = indexFor(e.hash, newCapacity);
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```

HashTable

继承和实现关系

- extends Dictionary<K,V>：继承字典？
- implements Map<K,V>,：实现Map接口
- implements Cloneable, java.io.Serializable：实现了克隆和可序列化接口

源码特点

resize的扩容方式是old*2+1.且初始化的bucket长度是16.
除开构造方法以外，其他方法都加上synchronized的方法进行修饰来保证同步，但效率过低。
在确定是哪一个bucket时，采取的是计算hash值后，用模的方法，而不是位运算，效率相对hashmap更低。
key和value的值都不允许是null;

LinkedHashMap

继承和实现关系

- extends HashMap<K,V>
- implements Map<K,V>

源码分析

```
/**
     * 相比于hashmap的结点，Linkedhashmap多了前后两个结点的定位，改成了双向链表
     */
    static class Entry<K,V> extends HashMap.Node<K,V> {
        Entry<K,V> before, after;
        Entry(int hash, K key, V value, Node<K,V> next) {
            super(hash, key, value, next);
        }
    }

    private static final long serialVersionUID = 3801124242820219131L;

    /**
     * 链表头
     */
    transient LinkedHashMap.Entry<K,V> head;

    /**
     * 链表尾t.
     */
    transient LinkedHashMap.Entry<K,V> tail;
```

源码分析

特点

LinkedHashMap相对于HashMap的源码比，是很简单的。因为大树底下好乘凉。它继承了HashMap，仅重写了几个方法，以改变它迭代遍历时的顺序。这也是其与HashMap相比最大的不同。 在每次插入数据，或者访问、修改数据时，会增加节点、或调整链表的节点顺序。以决定迭代时输出的顺序。

- accessOrder ,默认是false，则迭代时输出的顺序是插入节点的顺序。若为true，则输出的顺序是按照访问节点的顺序。为true时，可以在这基础之上构建一个LruCache.
- LinkedHashMap并没有重写任何put方法。但是其重写了构建新节点的newNode()方法.在每次构建新节点时，将新节点链接在内部双向链表的尾部
- accessOrder=true的模式下,在afterNodeAccess()函数中，会将当前被访问到的节点e，移动至内部的双向链表的尾部。值得注意的是，afterNodeAccess()函数中，会修改modCount,因此当你正在accessOrder=true的模式下,迭代LinkedHashMap时，如果同时查询访问数据，也会导致fail-fast，因为迭代的顺序已经改变。
- nextNode() 就是迭代器里的next()方法 。 该方法的实现可以看出，迭代LinkedHashMap，就是从内部维护的双链表的表头开始循环输出。而双链表节点的顺序在LinkedHashMap的增、删、改、查时都会更新。以满足按照插入顺序输出，还是访问顺序输出。它与HashMap比，还有一个小小的优化，重写了containsValue()方法，直接遍历内部链表去比对value值是否相等。 ！

SortedMap接口和TreeMap实现类

treemap的继承和实现关系

- extends AbstractMap<K,V> ：继承map的一些基本方法
- implements NavigableMap<K,V>: NavigableMap implements SortedMap，
- implements Cloneable, java.io.Serializable:实现了克隆和可序列化

源码分析

总结来说，它是底层基于红黑树来实现存储的，每个key-value就是红黑树的一个结点。通过红黑树来维护自己key的顺序。

这个判断是通过equals和compareTo方法

(了解完红黑树再来了解具体的查询，添加等逻辑)

IdentityHashMap

- 简单说IdentityHashMap与常用的HashMap的区别是：前者比较key时是“引用相等”而后者是“对象相等” ，即对于k1和k2，当k1==k2时，IdentityHashMap认为两个key相等，而HashMap只有在k1.equals(k2) == true 时才会认为两个key相等。IdentityHashMap 允许使用null作为key和value. 不保证任何Key-value对的之间的顺序, 更不能保证他们的顺序随时间的推移不会发生变化.

- IdentityHashMap有其特殊用途，比如序列化或者深度复制。或者记录对象代理。

举个例子，jvm中的所有对象都是独一无二的，哪怕两个对象是同一个class的对象，而且两个对象的数据完全相同，对于jvm来说，他们也是完全不同的，如果要用一个map来记录这样jvm中的对象，你就需要用IdentityHashMap，而不能使用其他Map实现。
ConcurrentHashMap ConcurrentHashMap降低了锁的粒度，其中在1.7中，设置了Segment数组，来表示不同数据段，在1.8中，取消了Segment数组，进一步降低了锁的粒度。由于本文是分析1.8的ConcurrentHashMap，所以不对1.7的版本过多的解释。