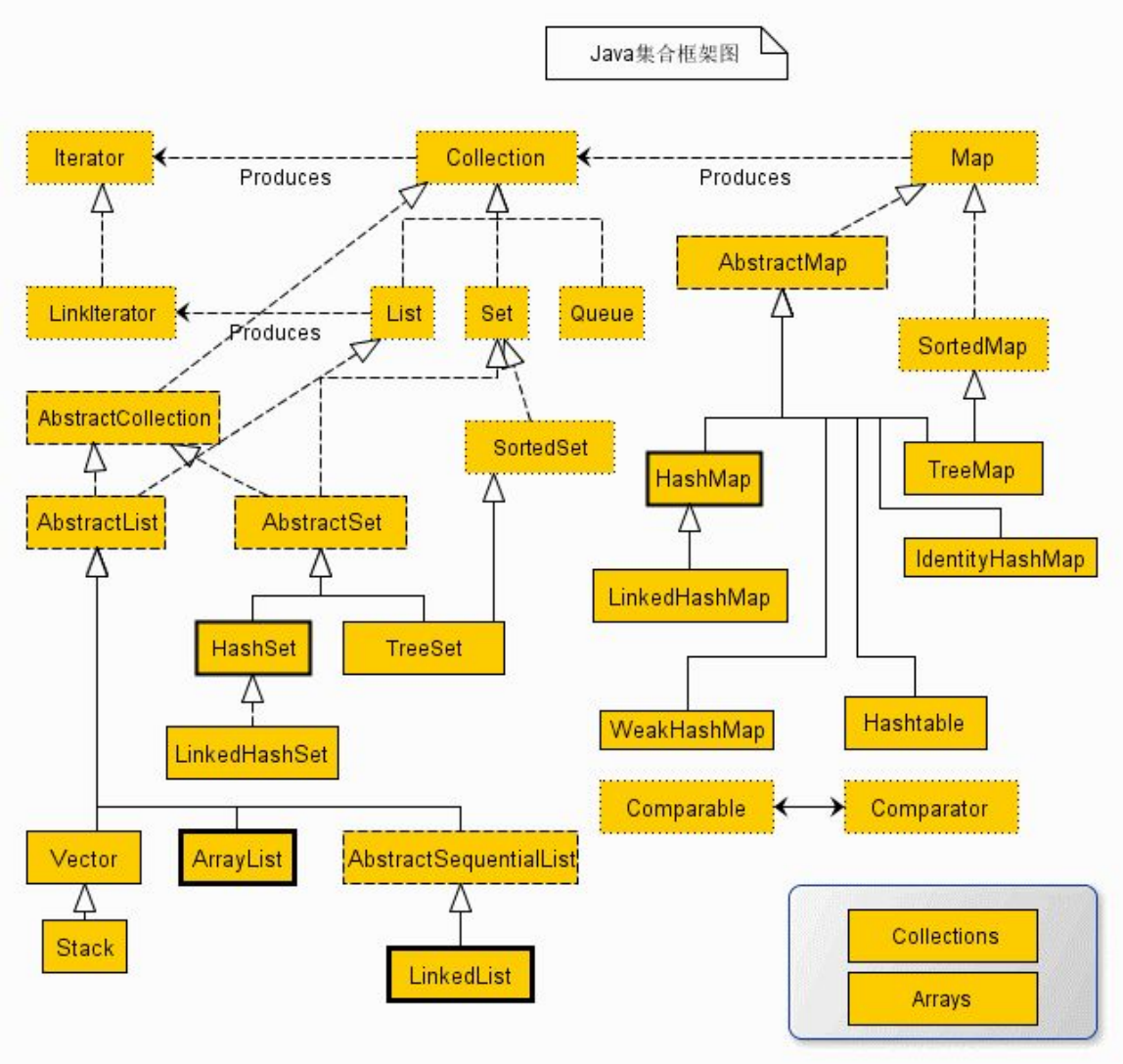


ArrayList：

List特点：元素有放入顺序，元素可重复 。

存储结构：底层采用数组来实现的。

源码：

```
1 public class ArrayList<E> extends AbstractList<E>
```

2、Cloneable

支持拷贝：实现Cloneable接口，重写clone方法、方法内容默认调用父类的clone方法。

2.1、浅拷贝

基础类型的变量拷贝之后是独立的，不会随着源变量变动而变

String类型拷贝之后也是独立的

引用类型拷贝的是引用地址，拷贝前后的变量引用同一个堆中的对象

public Object clone() throws CloneNotSupportedException { Study s = (Study) super.clone(); return s; }

基本类型、引用类型。

浅拷贝 基本类型（int类型）是独立的（改了值会生效），引用类型不独立的（同一份数据，string也是同样的）

2.2、深拷贝

变量的所有引用类型变量（除了String）都需要实现Cloneable（数组可以直接调用clone方法），clone方法中，引用类型需要各

自调用clone，重新赋值

```
public Object clone() throws CloneNotSupportedException {
    Study s = (Study) super.clone();
    s.setScore(this.score.clone());
    return s;
}
```

java的传参，基本类型和引用类型传参

java在方法传递参数时，是将变量复制一份，然后传入方法体去执行。复制的是栈中的内容

所以基本类型是复制的变量名和值，值变了不影响源变量

引用类型复制的是变量名和值（引用地址），对象变了，会影响源变量（引用地址是一样的）

String：是不可变对象，重新赋值时，会在常量表新生成字符串（如果已有，直接取他的引用地址），将新字符串的引用地址赋值给栈中的新变量，因此源变量不会受影响

3、Serializable

序列化：将对象状态转换为可保持或传输的格式的过程。与序列化相对的是反序列化，它将流转换为对象。这两个过程结合起来，可以轻松地存储和传输数据，在Java中的这个Serializable接口其实是给jvm看的，通知jvm，我不对这个类做序列化了，你(jvm)帮我序列化就好了。如果我们没有自己声明一个serialVersionUID变量,接口会默认生成一个serialVersionUID，默认的serialVersinUID对于class的细节非常敏感，反序列化时可能会导致InvalidClassException这个异常（每次序列化都会重新计算该值）

存盘 网络

序列化 返序列化

序列化 对象》二级制

二进制 》对象 反序列化 校验 这个serivresionid

4、AbstractList

继承了AbstractList ，说明它是一个列表，拥有相应的增，删，查，改等功能。

5、List

为什么继承了 AbstractList 还需要 实现List 接口?

1、在StackOverFlow 中：传送门 得票最高的答案的回答者说他问了当初写这段代码的 Josh Bloch，得知这就是一个写法错误。 I’ ve asked Josh Bloch, and he informs me that it was a mistake. He used to think, long ago, that there was some value in it, but he since “saw the light”. Clearly JDK maintainers haven’ t considered this to be worth backing out later.

基本属性：

```
private static final long serialVersionUID = 8683452581122892189L;//序列化版本号（类文件签
private static final int DEFAULT_CAPACITY = 10;//如果实例化时未指定容量，则在初次添加元素时会
//static修饰，所有的未指定容量的实例(也未添加元素)共享此数组，两个空的数组有什么区别呢？ 就是第一
private static final Object[] EMPTY_ELEMENTDATA = {};
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
transient Object[] elementData; // arrayList真正存放元素的地方，长度大于等于size
```

构造器：

```
//无参构造器，构造一个容量大小为 10 的空的 list 集合，但构造函数只是给 elementData 赋值了一个空
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
//当使用无参构造函数时是把 DEFAULTCAPACITY_EMPTY_ELEMENTDATA 赋值给 elementData。 当 initial
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);
    }
}
//将 Collection 转化为数组，数组长度赋值给 size。 如果 size 不为零，则判断 elementData 的 cla
public ArrayList(Collection<? extends E> c) {
    Object[] a = c.toArray();
    if ((size = a.length) != 0) {
        if (c.getClass() == ArrayList.class) {
            elementData = a;
```

```
        } else {
            elementData = Arrays.copyOf(a, size, Object[].class);
        }
    } else {
        // 指向空数组
        elementData = EMPTY_ELEMENTDATA;
    }
```

添加元素--默认尾部添加

效率比较高

指定下标添加元素

```
public void add(int index, E element) {
    rangeCheckForAdd(index);//下标越界检查
    ensureCapacityInternal(size + 1);  //同上  判断扩容,记录操作数
    //依次复制插入位置及后面的数组元素，到后面一格，不是移动，因此复制完后，添加的下标位置和下一
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;//再将元素赋值给该下标
    size++;
```

时间复杂度为O(n)，与移动的元素个数正相关

扩容:

```
private void grow(int minCapacity) {
    int oldCapacity = elementData.length;//获取当前数组长度
    int newCapacity = oldCapacity + (oldCapacity >> 1);//默认将扩容至原来容量的 1.5 倍
    if (newCapacity - minCapacity < 0)//如果1.5倍太小的话，则将我们所需的容量大小赋值给newCa
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)//如果1.5倍太大或者我们需要的容量太大，那就直接拿 n
        newCapacity = hugeCapacity(minCapacity);
    elementData = Arrays.copyOf(elementData, newCapacity);//然后将原数组中的数据复制到大小为
```

迭代器 iterator

```java
public Iterator<E> iterator() {
    return new Itr();
}
private class Itr implements Iterator<E> {
    int cursor;       // 代表下一个要访问的元素下标
    int lastRet = -1; // 代表上一个要访问的元素下标
    int expectedModCount = modCount;//代表对 ArrayList 修改次数的期望值，初始值为 modCount
    //如果下一个元素的下标等于集合的大小 ，就证明到最后了
    public boolean hasNext() {
        return cursor != size;
    }
    @SuppressWarnings("unchecked")
    public E next() {
        checkForComodification();//判断expectedModCount和modCount是否相等,ConcurrentModifi
        int i = cursor;
        if (i >= size)//对 cursor 进行判断，看是否超过集合大小和数组长度
            throw new NoSuchElementException();
        Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new ConcurrentModificationException();
        cursor = i + 1;//自增 1。开始时，cursor = 0，lastRet = -1；每调用一次next方法，curso
        return (E) elementData[lastRet = i];//将cursor赋值给lastRet，并返回下标为 lastRet
    }
    public void remove() {
        if (lastRet < 0)//判断 lastRet 的值是否小于 0
            throw new IllegalStateException();
        checkForComodification();//判断expectedModCount和modCount是否相等,ConcurrentModifi
        try {
            ArrayList.this.remove(lastRet);//直接调用 ArrayList 的 remove 方法删除下标为 la
            cursor = lastRet;//将 lastRet 赋值给 curso
            lastRet = -1;//将 lastRet 重新赋值为 -1，并将 modCount 重新赋值给 expectedModCo
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
    final void checkForComodification() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
```

```
40      }
```

remove 方法的弊端。
1、只能进行remove操作，add、clear 等 Itr 中没有。
2、调用 remove 之前必须先调用 next。因为 remove 开始就对 lastRet 做了校验。而 lastRet 初始化时为 -1。
3、next 之后只可以调用一次 remove。因为 remove 会将 lastRet 重新初始化为 -1

什么是fail-fast?

fail-fast机制是java集合中的一种错误机制。
当使用迭代器迭代时，如果发现集合有修改，则快速失败做出响应，抛出ConcurrentModificationException异常。
这种修改有可能是其它线程的修改，也有可能是当前线程自己的修改导致的，比如迭代的过程中直接调用remove()删除元素等。
另外，并不是java中所有的集合都有fail-fast的机制。比如，像最终一致性的ConcurrentHashMap、CopyOnWriterArrayList等都是没有fast-fail的。
fail-fast是怎么实现的：
ArrayList、HashMap中都有一个属性modcount，每次对集合的修改这个值都会加1，在遍历前记录这个值expert*count中，遍历中检查两者是否一致，如果出现不一致就说明有修改，则抛出ConcurrentModificationException异常。
底层数组存/取元素效率非常的高(get/set)，时间复杂度是O(1)，而查找（比如：indexOf，contain），插入和删除元素效率不太高，时间复杂度为O(n)。
插入/删除元素会触发底层数组频繁拷贝，效率不高，还会造成内存空间的浪费，解决方案：linkedList
查找元素效率不高，解决方案：HashMap（红黑树）

LinkedList：

存储结构：底层采用链表来实现的。

HashSet（Set）：

特点：

元素无放入顺序，元素不可重复（注意：元素虽然无放入顺序，但是元素在set中的位置是有该元素的HashCode决定的，其位置其实是固定的）

存储结构：

底层采用HashMap来实现

原理讲解：

源码分析：

HashMap（Map）：

特点：

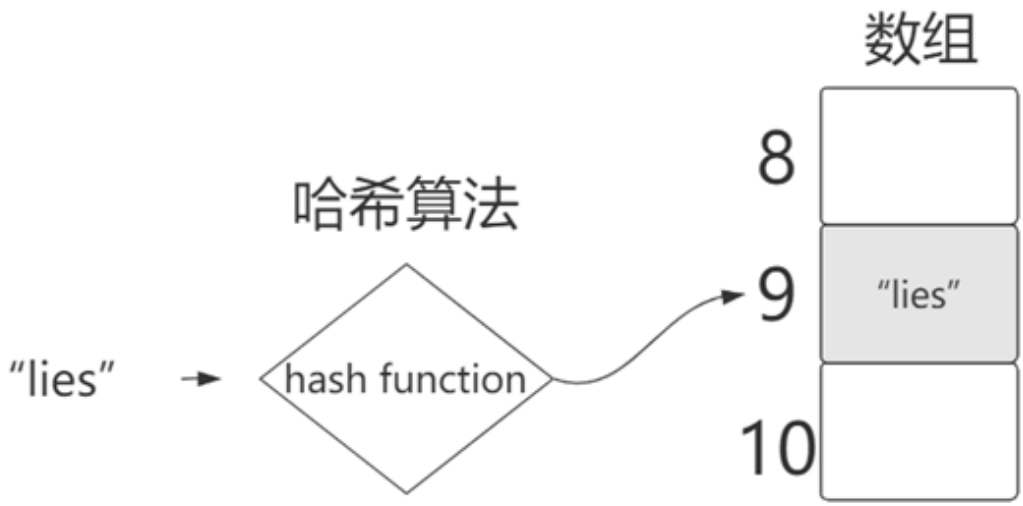
key，value存储，key可以为null，同样的key会被覆盖掉

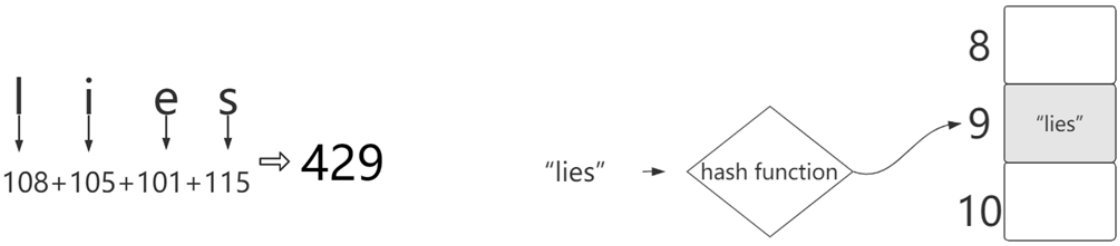
存储结构：

底层采用数组、链表、红黑树来实现的。

原理讲解：

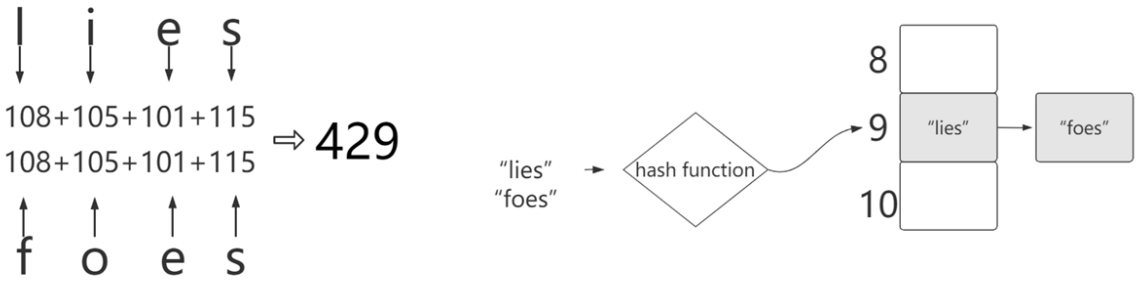
哈希算法（也叫散列），就是把任意长度值(Key)通过散列算法变换成固定长度的key（地址）通过这个地址进行访问的数据结构它通过把关键码值映射到表中一个。位置来访问记录，以加快查找的速度。





Hashcode：通过字符串算出它的ascii码，进行mod（取模），算出哈希表中的下标

哈希冲突

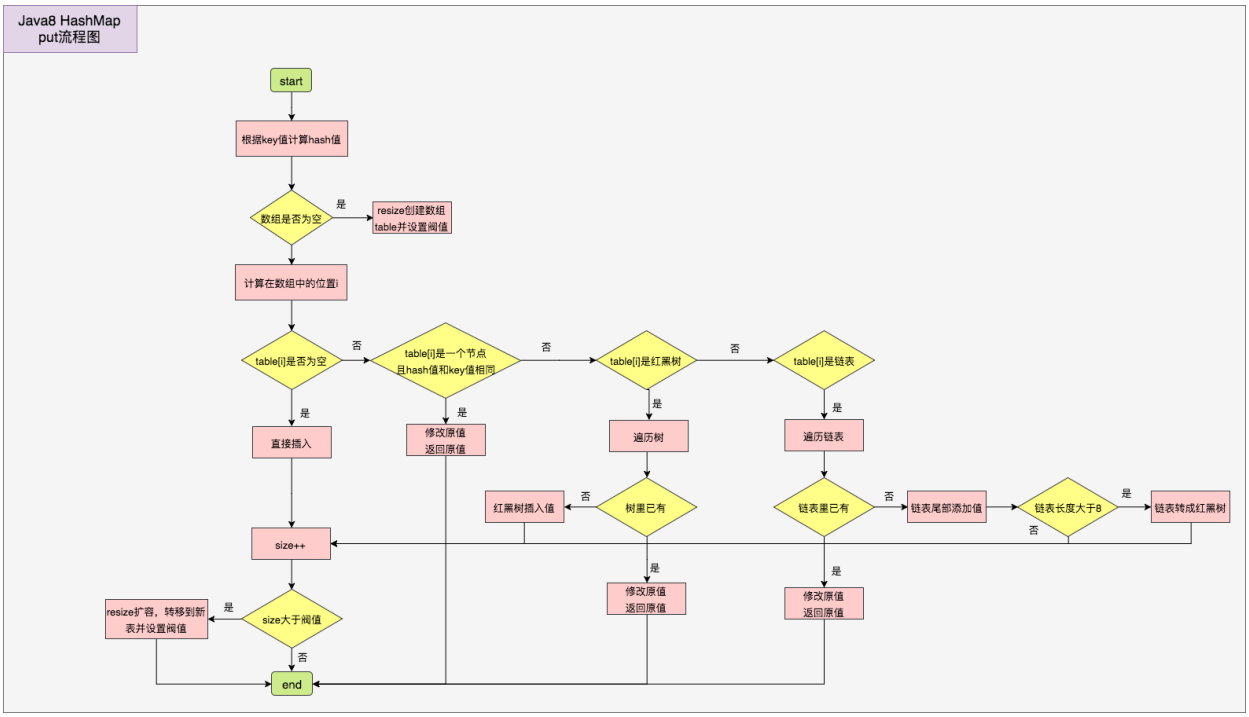


用链表是来解决数组下标会覆盖的问题，冲突的问题。为什么hashmap 用两个数据结构。两个数据结构。JDK8 红黑树？？？

因为链表查询的时候链表过长了查询效率非常低，所以需要用红黑树

```
/**
 * 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.
 */
```

源码分析：



ConcurrentHashMap（并发安全map）：

特点：

并发安全的HashMap，比Hashtable效率更高

存储结构：

底层采用数组、链表、红黑树 内部大量采用CAS操作。并发控制使用**synchronized 和 CAS 来操作**来实现的。

原理讲解：

源码分析：

初始化方法：

java.util.concurrent.ConcurrentHashMap#initTable

是在put方法进行的

transient volatile Node<K,V>[] table;