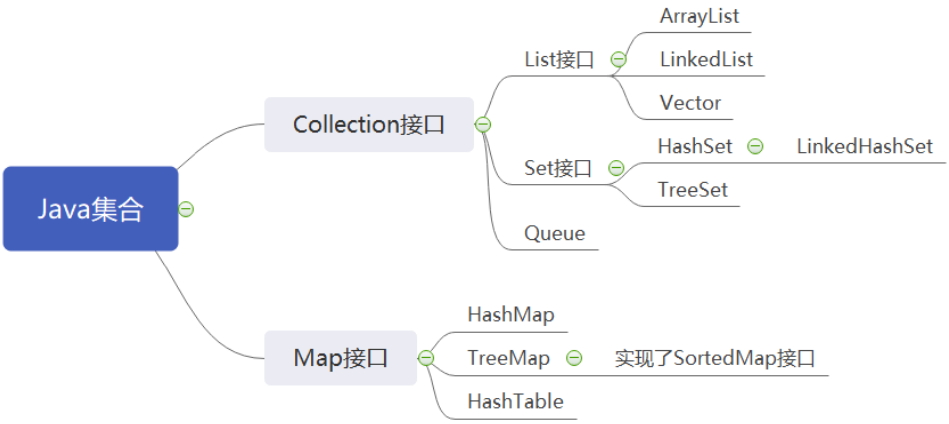
Java集合

java.lang

- Collection接口
  - List接口
  - Set接口
- Map接口



Collection

List

- 重复
- 有序

1 ArrayList

- List接口的主要实现类，底层用数组实现
- 优点
  - 访问速度快
- 缺点
  - 插入和删除开销大：增加和删除元素时，需要对整个数组进行遍历、定位和移动
  - 线程不安全

源码分析:

- JDK7
  - 创建
    - 底层创建一个长度为10的数组
  - 扩容
    - 设置新的存储空间为原来的1.5倍
      - 如果新存储空间仍然不够，则将要求的最小存储空间设置成新存储空间

        - 将原数组里的元素复制到新数组里

- JDK8
    - 创建
        - 初始时，底层数组的容量为0，当添加第一个元素时，才将数组容量设置为10
    - add：

```
    private void rangeCheckForAdd(int index) {
        if (index > size || index < 0)
            throw new
IndexOutOfBoundsException(outOfBoundsMsg(index));
    }

    private void ensureCapacityInternal(int minCapacity) {
        ensureExplicitCapacity(calculateCapacity(elementData,
minCapacity));
    }

    private void ensureExplicitCapacity(int minCapacity) {
        // 记录list被结构化修改的次数
        modCount++;

        // overflow-conscious code
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }

    /* void add(index, element) */
    public void add(int index, E element) {
        // 如果要插入的位置超过现有数组元素长度，或者小于0，抛出异常
        rangeCheckForAdd(index);

        // 判断是否要扩容，增加modCount
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        // 把子数组移动到后面
        System.arraycopy(elementData, index, elementData, index +
1,
                         size - index);
        // 插入数组
        elementData[index] = element;
        // 数组元素大小++
        size++;
    }

    /* boolean add(Element) */
    public boolean add(E e) {
        // 判断是否要扩容，modCount++
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        // 把新元素插入到现有元素后面
        elementData[size++] = e;
        // 返回 true
        return true;
    }
```

- 扩容：和JDK7类似

```
    private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = elementData.length;
        // 设置新的存储能力为原来的1.5倍
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        // 扩容之后仍小于最低存储要求minCapacity
        if (newCapacity - minCapacity < 0)
            //
            newCapacity = minCapacity;
        // 扩容后超过了最大容量 MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8,
        if (newCapacity - MAX_ARRAY_SIZE > 0) //private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
            newCapacity = hugeCapacity(minCapacity);
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

2 LinkedList

- 用双向链表存储元素

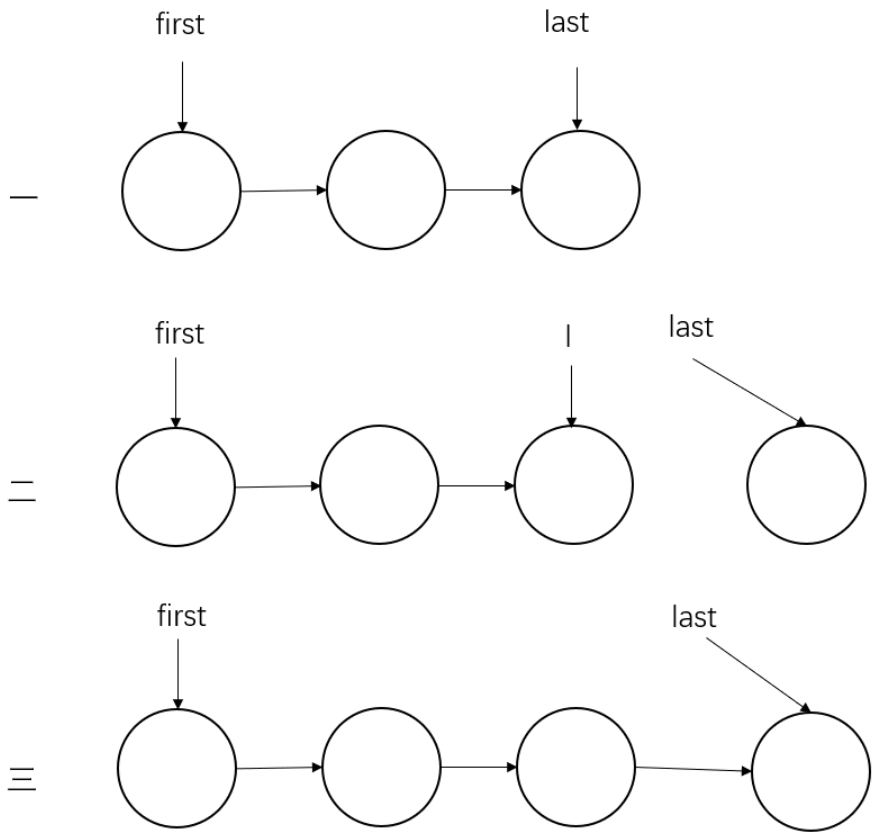
```
    transient int size = 0;
    // 头节点
    transient Node<E> first;
    // 尾节点
    transient Node<E> last;
```

- add():

```
    void linkLast(E e) {
        // 获取尾节点
        final Node<E> l = last;
        // 创建新节点
        final Node<E> newNode = new Node<>(l, e, null);
        // 把最后一个节点指向新插入的节点
        last = newNode;
        // 如果新节点前面一个节点为空，说明现在的LinkedList是空的
        if (l == null)
            // 把头节点指向新节点
            first = newNode;
        else
            // 否则，把新节点插入到前一个节点后面
            l.next = newNode;
        // size++
        size++;
        // 记录列表改变的次数
```

```
        modCount++;
    }

    public boolean add(E e) {
        linkLast(e);
        return true;
    }
```



- 优点：
  - 插入和删除数据快
- 缺点：
  - 遍历和查找慢
  - 线程不安全

3 Vector

- 底层用数组实现
- 初始化时，默认容量为10
- 扩容时，如果增量>0，则新容量为旧容量+增量；否则，新容量为原来的2倍。最后将数组里的值，复制到新数组里
  - 如果新容量不够，则以要求的容量为新容量

```java
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);
}
```

- 优点
  - 线程安全的，同一时间只能有一个线程写Vector
- 缺点
  - 访问速度慢

Set

- 无序
  - 无序不代表没有顺序，而是不像数组一样按索引值顺序添加的，set是由**HashCode**来决定存储位置的
- 不可以重复

1 HashSet

- Set接口的主要实现类，是线程不安全的
- 可以存储null值
- 添加
  - 当向HashSet添加元素a时，会调用HashCode()方法计算元素a的哈希值
  - 然后通过a的哈希值根据某种算法获得a的索引位置
    - 如果索引位置处没有元素，则元素a添加成功
    - 如果索引位置处有元素b，则a和b通过equals()方法比较
      - 两值相等，则添加失败
      - 两值不等，则添加成功

        旧元素"七上八下"：jdk7新元素放到数组，指向旧元素；jdk8旧元素指向新元素
- 扩容

底层也是数组，初始容量为16，当如果使用率超过0.75，（16*0.75=12）就会扩大容量为原来的2倍。（16扩容为32，依次为64,128....等）

2 LinkedHashSet

- 添加数据的时候，会添加两个引用，引向前一个数据和后一个数据
- 底层用LinkedHashMap实现

3 TreeSet

- 要求添加的必须是同一个类的对象
- 会对添加的元素排序

Map

- Entry表示一对key和value
- Entry由Set结构保存，是无序、不可重复的

1 HashMap （⭐）

- Map的主要实现类
- 可以存放null的key和value，但最多只能允许一条key为null的entry，可以有多条value为null的entry

```
1 map.put(null, null);
```

- 线程不安全
  - 但是可以使用`synchronizedMap()`方法和`ConcurrentHashMap()`方法使其变成线程安全的

JDK7

- 底层用数组+链表实现的，数组默认容量为16

- 插入数据时，首先计算key的哈希值，然后找到数组相应要存放的位置
  - 如果该位置是空的，则直接插入即可
  - 如果该位置是非空的，则先比较key和位置上所有key值的哈希值
    - 如果哈希值不相等，则插入key-value
    - 如果哈希值相等，则需要调用key的equals方法和位置上存在的key1进行比较
      - 如果相等，则将key1的value1替换成新的value
      - 如果不相等，则以链表形式插入新的key-value

JDK8

- 底层用数组+链表+红黑树实现，初始化时，数组容量为0
  - 当第一次put数据时，初始化16容量的数组

```
    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;
        // 如果table数组为空，初始化一个容量为16的数组
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        // （n-1 & hash）处Node为p， 如果该索引位置p为null，直接放入
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        // 如果p不为空
        else {
            //
            Node<K,V> e; K k;
            // 如果p的key和要存入的对象且hash值相同，并且两个key的equals方法相同或两个key相同
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                // 令e = p
                e = p;
            // 如果p是红黑树的节点，则放入红黑树
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            // 如果p的key和要存入的对象且hash值不同，且两个key的equals方法不同，并且p不是红黑树的节点
            else {
```

```
                // 在链表中遍历比较，遍历元素指定为e
                for (int binCount = 0; ; ++binCount) {
                    // 如果链表中没有找到hash值相同或者key的equals结果相同的值
                    if ((e = p.next) == null) {
                        // 在链表后面插入节点
                        p.next = newNode(hash, key, value, null);
                        // 如果存放后，链表中的元素大于等于8，则进入treeifyBin方法来判断是否要转换成红黑树
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    // 如果找到hash值相同的node对象，或者两个key的equals方法相同，则推出循环
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
                }
            }
            // 循环中找到的那个Node，替换value
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        // 如果达到16*0.75，扩容
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

- **当链表元素 > 8且哈希表长度 > MIN_TREEIFY_CAPACITY（即64）**，则转换成红黑树

```
// 链表元素>=8时，进入treeifyBin方法
if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
    treeifyBin(tab, hash);
break;

/**
     * Replaces all linked nodes in bin at index for given hash unless
     * table is too small, in which case resizes instead.
     */
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    // 如果数组元素小于64，则做扩容操作
```

```
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        TreeNode<K,V> hd = null, tl = null;
        do {
            TreeNode<K,V> p = replacementTreeNode(e, null);
            if (tl == null)
                hd = p;
            else {
                p.prev = tl;
                tl.next = p;
            }
            tl = p;
        } while ((e = e.next) != null);
        if ((tab[index] = hd) != null)
            hd.treeify(tab);
    }
}
```

扩容

```
    /**
     * Initializes or doubles table size.  If null, allocates in
     * accord with initial capacity target held in field threshold.
     * Otherwise, because we are using power-of-two expansion, the
     * elements from each bin must either stay at same index, or move
     * with a power of two offset in the new table.
     *
     * @return the table
     */
    final Node<K,V>[] resize() {
        Node<K,V>[] oldTab = table;
        int oldCap = (oldTab == null) ? 0 : oldTab.length;
        int oldThr = threshold;
        int newCap, newThr = 0;
        if (oldCap > 0) {
            if (oldCap >= MAXIMUM_CAPACITY) {
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY)
                newThr = oldThr << 1; // double threshold
        }
        else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;
        else {               // zero initial threshold signifies using
defaults
            newCap = DEFAULT_INITIAL_CAPACITY;
            newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
        }
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor;
```

```
            newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
                      (int)ft : Integer.MAX_VALUE);
        }
        threshold = newThr;
        @SuppressWarnings({"rawtypes","unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        table = newTab;
        if (oldTab != null) {
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                if ((e = oldTab[j]) != null) {
                    oldTab[j] = null;
                    if (e.next == null)
                        newTab[e.hash & (newCap - 1)] = e;
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                    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;
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        if (loTail != null) {
                            loTail.next = null;
                            newTab[j] = loHead;
                        }
                        if (hiTail != null) {
                            hiTail.next = null;
                            newTab[j + oldCap] = hiHead;
                        }
                    }
                }
            }
        }
        return newTab;
    }
```

- 默认容量为16，负载因子为0.75

    - 当超过容量*负载因子时，就扩容，扩容成原来的两倍
- 为什么负载因子为0.75?
    - Ideally, under random hashCodes, the frequency of nodes in bins follows a Poisson distribution.
    - 负载因子是0.75的时候，空间利用率比较高，而且避免了相当多的Hash冲突，使得底层的链表或者是红黑树的高度比较低，提升了空间效率。
- **为什么扩容的值是原来的2的幂次数**
    - 为了进行 (n-1)&hash 来计算索引位置
    - 扩容的值是原来的2的幂次数，这样 n-1 就是开头为0，后面全是1的值
    - 和 hash 进行 & 运算，会保留 hash 中后 x 位，
        - 这样可以保证索引值肯定在capacity中
        - 并且满足公式 (n-1)&hash = hash%n

```
比如：
hash = 10，即1010
n = 8，即1000
则hash&(n-1) = 0010
  hash%n = 2 = 0010
```

2 TreeMap

- 插入的key必须是同一个类的对象
- 会根据key值进行排序
- 底层用红黑树实现

3 HashTable

- 线程安全的，效率低
- 不能存储值为null的key和value
- 有一个子类Property，用于处理配置文件

4 ConcurrentHashMap

1. put

```
    /**
     * Maps the specified key to the specified value in this table.
     * Neither the key nor the value can be null.
     *
     * <p>The value can be retrieved by calling the {@code get} method
     * with a key that is equal to the original key.
     *
     * @param key key with which the specified value is to be associated
     * @param value value to be associated with the specified key
     * @return the previous value associated with {@code key}, or
```

```
     *         {@code null} if there was no mapping for {@code key}
     * @throws NullPointerException if the specified key or value is
null
     */
    public V put(K key, V value) {
        return putVal(key, value, false);
    }

    /** Implementation for put and putIfAbsent */
    final V putVal(K key, V value, boolean onlyIfAbsent) {
        if (key == null || value == null) throw new
NullPointerException();
        int hash = spread(key.hashCode());
        int binCount = 0;
        for (Node<K,V>[] tab = table;;) {
            Node<K,V> f; int n, i, fh;
            if (tab == null || (n = tab.length) == 0)
                tab = initTable();
            else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
                if (casTabAt(tab, i, null,
                             new Node<K,V>(hash, key, value, null)))
                    break;                   // no lock when adding to
empty bin
            }
            else if ((fh = f.hash) == MOVED)
                tab = helpTransfer(tab, f);
            else {
                V oldVal = null;
                synchronized (f) {
                    if (tabAt(tab, i) == f) {
                        if (fh >= 0) {
                            binCount = 1;
                            for (Node<K,V> e = f;; ++binCount) {
                                K ek;
                                if (e.hash == hash &&
                                    ((ek = e.key) == key ||
                                     (ek != null && key.equals(ek)))) {
                                    oldVal = e.val;
                                    if (!onlyIfAbsent)
                                        e.val = value;
                                    break;
                                }
                                Node<K,V> pred = e;
                                if ((e = e.next) == null) {
                                    pred.next = new Node<K,V>(hash, key,
                                                              value,
null);
                                    break;
                                }
                            }
                        }
                        else if (f instanceof TreeBin) {
                            Node<K,V> p;
                            binCount = 2;
```

```
61                              if ((p = ((TreeBin<K,V>)f).putTreeVal(hash,
key,
62                                                                    value)) !=
null) {
63                                  oldVal = p.val;
64                                  if (!onlyIfAbsent)
65                                      p.val = value;
66                              }
67                          }
68                      }
69                  }
70                  if (binCount != 0) {
71                      if (binCount >= TREEIFY_THRESHOLD)
72                          treeifyBin(tab, i);
73                      if (oldVal != null)
74                          return oldVal;
75                      break;
76                  }
77              }
78          }
79          addCount(1L, binCount);
80          return null;
81      }
```