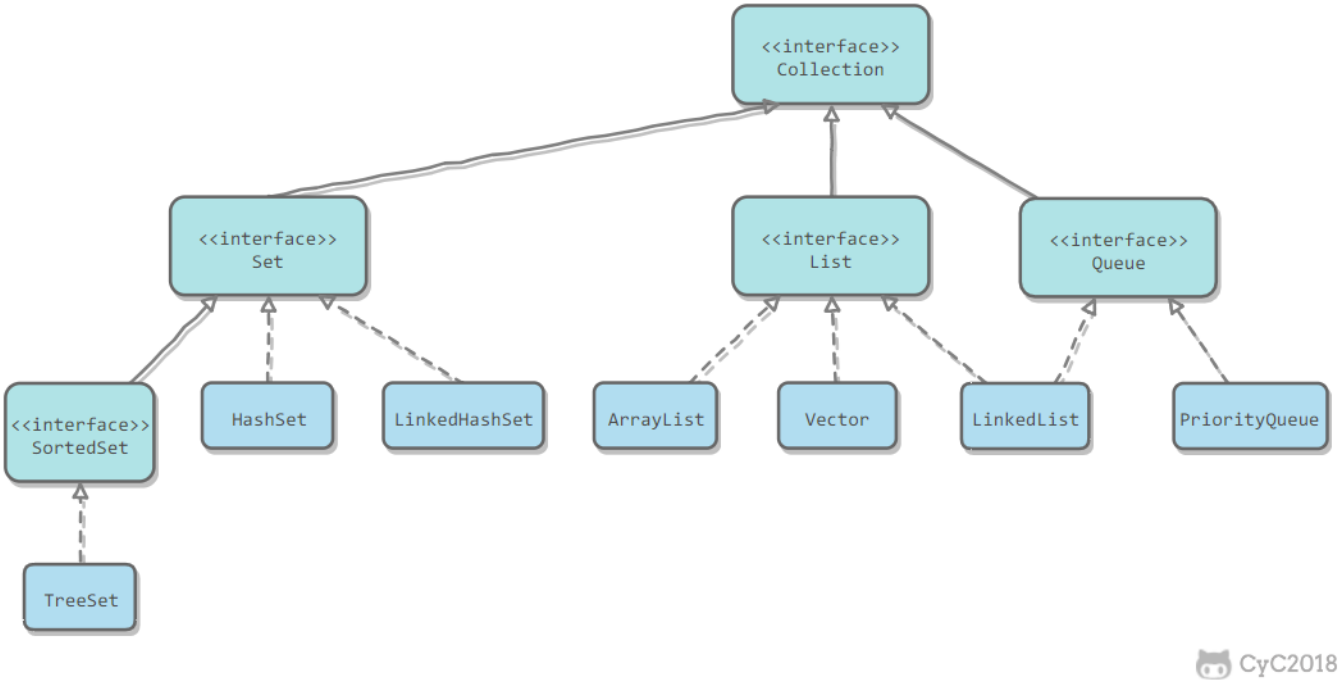
概览

容器主要包括 Collection 和 Map 两种，Collection 存储着对象的集合，而 Map 存储着键值对（两个对象）的映射表。

Collection



Set

- TreeSet：基于红黑树实现，支持有序性操作，例如根据一个范围查找元素的操作。但是查找效率不如 HashSet，HashSet 查找的时间复杂度为 O(1)，TreeSet 则为 O(logN)。
- HashSet：基于哈希表实现，支持快速查找，但不支持有序性操作。并且失去了元素的插入顺序信息，也就是说使用 Iterator 遍历 HashSet 得到的结果是不确定的。
- LinkedHashSet：具有 HashSet 的查找效率，且内部使用双向链表维护元素的插入顺序。

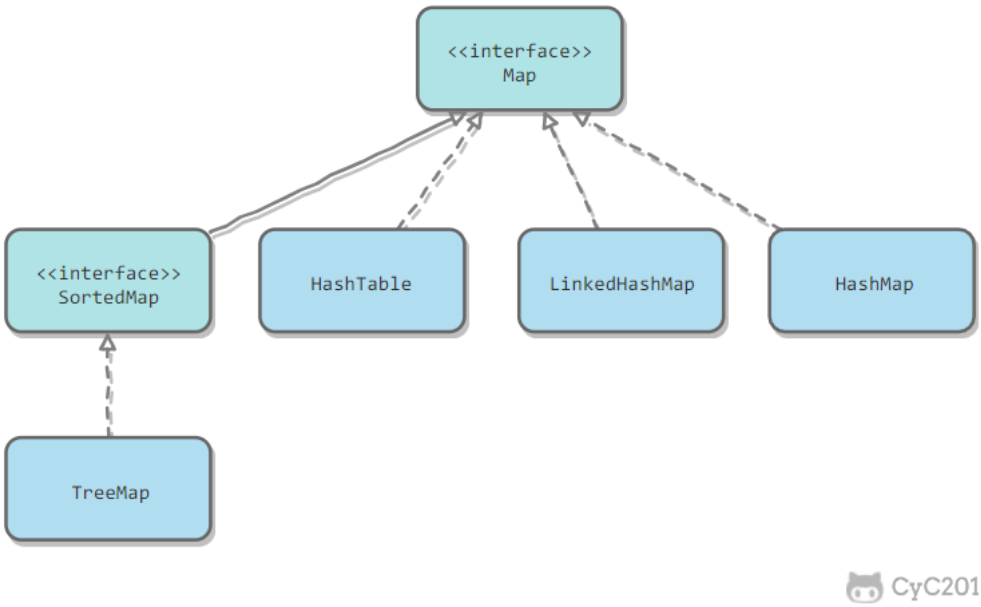
List

- ArrayList：基于动态数组实现，支持随机访问。
- Vector：和 ArrayList 类似，但它是线程安全的。
- LinkedList：基于双向链表实现，只能顺序访问，但是可以快速地在链表中间插入和删除元素。不仅如此，LinkedList 还可以用作栈、队列和双向队列。

Queue

- LinkedList：可以用它来实现双向队列。
- PriorityQueue：基于堆结构实现，可以用它来实现优先队列。

Map



- TreeMap：基于红黑树实现。
- HashMap：基于哈希表实现。
- HashTable：和 HashMap 类似，但它是线程安全的，这意味着同一时刻多个线程可以同时写入 HashTable 并且不会导致数据不一致。它是遗留类，不应该去使用它。现在可以使用 ConcurrentHashMap 来支持线程安全，并且 ConcurrentHashMap 的效率会更高，因为 ConcurrentHashMap 引入了分段锁。
- LinkedHashMap：使用双向链表来维护元素的顺序，顺序为插入顺序或者最近最少使用（LRU）顺序。

源码分析

ArrayList

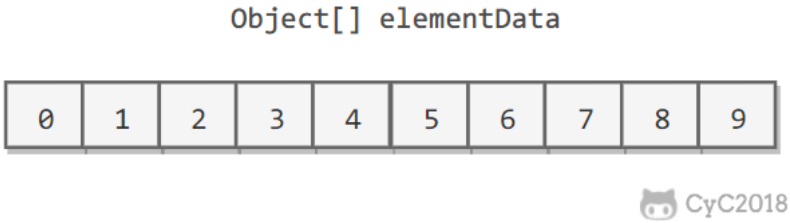
1. 概览

因为 ArrayList 是基于数组实现的，所以支持快速随机访问。RandomAccess 接口标识着该类支持快速随机访问。

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

数组的默认大小为 10

```
private static final int DEFAULT_CAPACITY = 10;
```



1. 扩容

添加元素时使用 ensureCapacityInternal() 方法来保证容量足够，如果不够时，需要使用 grow() 方法进行扩容，新容量的大小为 oldCapacity + (oldCapacity >> 1)，也就是旧容量的 1.5 倍。

扩容操作需要调用 Arrays.copyOf() 把原数组整个复制到新数组中，这个操作代价很高，因此最好在创建 ArrayList 对象时就指定大概的容量大小，减少扩容操作的次数。

```
    /**
     * Appends the specified element to the end of this list.
     *
     * @param e element to be appended to this list
     * @return <tt>true</tt> (as specified by {@link Collection#add})
     */
    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 void ensureExplicitCapacity(int minCapacity) {
        modCount++;

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

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

1. 删除元素

需要调用 System.arraycopy() 将 index+1 后面的元素都复制到 index 位置上，该操作的时间复杂度为 O(N)，可以看出 ArrayList 删除元素的代价是非常高的。

```
    /**
     * Removes the element at the specified position in this list.
     * Shifts any subsequent elements to the left (subtracts one from their
     * indices).
     *
     * @param index the index of the element to be removed
     * @return the element that was removed from the list
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    public E remove(int index) {
        rangeCheck(index);

        modCount++;
        E oldValue = elementData(index);

        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

        return oldValue;
    }
```

1. Fail-Fast

modCount 用来记录 ArrayList 结构发生变化的次数。结构发生变化是指添加或者删除至少一个元素的所有操作，或者是调整内部数组的大小，仅仅只是设置元素的值不算结构发生变化。

在进行序列化或者迭代等操作时，需要比较操作前后 modCount 是否改变，如果改变了需要抛出 ConcurrentModificationException。

```
    /**
     * Save the state of the <tt>ArrayList</tt> instance to a stream (that
     * is, serialize it).
     *
     * @serialData The length of the array backing the <tt>ArrayList</tt>
     *             instance is emitted (int), followed by all of its elements
     *             (each an <tt>Object</tt>) in the proper order.
     */
    private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException{
        // Write out element count, and any hidden stuff
        int expectedModCount = modCount;
        s.defaultWriteObject();

        // Write out size as capacity for behavioural compatibility with clone()
        s.writeInt(size);

        // Write out all elements in the proper order.
        for (int i=0; i<size; i++) {
            s.writeObject(elementData[i]);
        }

        if (modCount != expectedModCount) {
            throw new ConcurrentModificationException();
        }
    }
```

1. 序列化

ArrayList 基于数组实现，并且具有动态扩容特性，因此保存元素的数组不一定都会被使用，那么就没必要全部进行序列化。

保存元素的数组 elementData 使用 transient 修饰，该关键字声明数组默认不会被序列化。

```
    /**
     * The array buffer into which the elements of the ArrayList are stored.
     * The capacity of the ArrayList is the length of this array buffer. Any
     * empty ArrayList with elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA
     * will be expanded to DEFAULT_CAPACITY when the first element is added.
     */
    transient Object[] elementData; // non-private to simplify nested class access
```

ArrayList 实现了 writeObject() 和 readObject() 来控制只序列化数组中有元素填充那部分内容。

```
/**
     * Reconstitute the <tt>ArrayList</tt> instance from a stream (that is,
     * deserialize it).
     */
    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        elementData = EMPTY_ELEMENTDATA;

        // Read in size, and any hidden stuff
        s.defaultReadObject();

        // Read in capacity
        s.readInt(); // ignored

        if (size > 0) {
            // be like clone(), allocate array based upon size not capacity
            int capacity = calculateCapacity(elementData, size);
            SharedSecrets.getJavaOISAccess().checkArray(s, Object[].class, capacity);
            ensureCapacityInternal(size);

            Object[] a = elementData;
            // Read in all elements in the proper order.
            for (int i=0; i<size; i++) {
                a[i] = s.readObject();
            }
        }
    }
```

序列化时需要使用 ObjectOutputStream 的 writeObject() 将对象转换为字节流并输出。而 writeObject() 方法在传入的对象存在 writeObject() 的时候会去反射调用该对象的 writeObject() 来实现序列化。反序列化使用的是 ObjectInputStream 的 readObject() 方法，原理类似。

```
ArrayList list = new ArrayList();
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(file));
oos.writeObject(list);
```

Vector

1. 同步

它的实现与 ArrayList 类似，但是使用了 synchronized 进行同步。

```
    /**
     * Appends the specified element to the end of this Vector.
     *
     * @param e element to be appended to this Vector
     * @return {@code true} (as specified by {@link Collection#add})
     * @since 1.2
     */
    public synchronized boolean add(E e) {
        modCount++;
        ensureCapacityHelper(elementCount + 1);
        elementData[elementCount++] = e;
        return true;
    }


    /**
     * Returns the element at the specified position in this Vector.
     *
     * @param index index of the element to return
     * @return object at the specified index
     * @throws ArrayIndexOutOfBoundsException if the index is out of range
     *            ({@code index < 0 || index >= size()})
     * @since 1.2
     */
    public synchronized E get(int index) {
        if (index >= elementCount)
            throw new ArrayIndexOutOfBoundsException(index);

        return elementData(index);
    }
```

1. 与 ArrayList 的比较

- Vector 是同步的，因此开销就比 ArrayList 要大，访问速度更慢。最好使用 ArrayList 而不是 Vector，因为同步操作完全可以由程序员自己来控制
- Vector 每次扩容请求其大小的 2 倍空间，而 ArrayList 是 1.5

1. 替代方案

可以使用 Collections.synchronizedList(); 得到一个线程安全的 ArrayList。

```
List<String> list = new ArrayList<>();
List<String> synList = Collections.synchronizedList(list);
```

也可以使用 concurrent 并发包下的 CopyOnWriteArrayList 类。

```
List<String> list = new CopyOnWriteArrayList<>();
```

CopyOnWriteArrayList

读写分离

写操作在一个复制的数组上进行，读操作还是在原始数组中进行，读写分离，互不影响。

写操作需要加锁，防止并发写入时导致写入数据丢失。

写操作结束之后需要把原始数组指向新的复制数组。

```
    /**
     * Appends the specified element to the end of this list.
     *
     * @param e element to be appended to this list
     * @return {@code true} (as specified by {@link Collection#add})
     */
    public boolean add(E e) {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            Object[] elements = getArray();
            int len = elements.length;
            Object[] newElements = Arrays.copyOf(elements, len + 1);
            newElements[len] = e;
            setArray(newElements);
            return true;
        } finally {
            lock.unlock();
        }
    }


    /**
     * Sets the array.
     */
    final void setArray(Object[] a) {
        array = a;
    }


    /**
     * {@inheritDoc}
     *
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    public E get(int index) {
        return get(getArray(), index);
    }
```

使用场景

CopyOnWriteArrayList 在写操作的同时允许读操作，大大提高了读操作的性能，因此很适合读多写少的应用场景。

但是 CopyOnWriteArrayList 有其缺陷：

- 内存占用：在写操作时需要复制一个新的数组，使得内存占用为原来的两倍左右
- 数据不一致：读操作不能读取实时性的数据，因为部分写操作的数据还未同步到读数组中 所以 CopyOnWriteArrayList 不适合内存敏感以及对实时性要求很高的场景。

LinkedList

1. 概览

基于双向链表实现，使用 Node 存储链表节点信息。

```
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;
        }
    }
```

每个链表存储了 first 和 last 指针：

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

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

1. 与 ArrayList 的比较

- ArrayList 基于动态数组实现，LinkedList 基于双向链表实现
- ArrayList 支持随机访问，LinkedList 不支持(未实现标记接口RandomAccess)(总是从头开始变遍历)

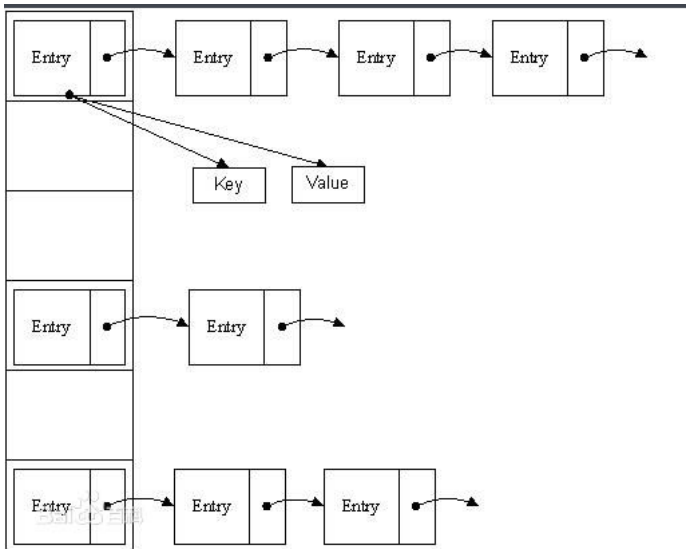
```
/**
 * Returns the (non-null) Node at the specified element index.
 */
Node<E> node(int index) {
    // assert isElementIndex(index);

    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;
    }
}
```

- LinkedList 在任意位置添加删除元素更快

HashMap

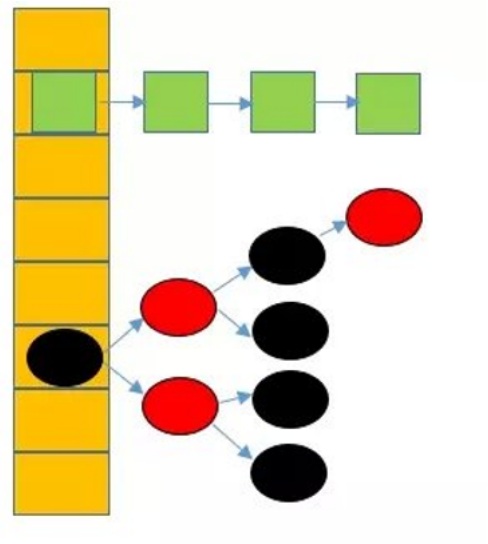
以下源码分析以 JDK 1.7 为主

1. 存储结构



左边部分代表Hash表，数组的每一个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中。

JDK1.8之前的HashMap都采用上图的结构，都是基于一个数组和多个单链表，hash值冲突的时候，就将对应节点以链表形式存储。如果在一个链表中查找一个节点时，将会花费O(n)的查找时间，会有很大的性能损失。到了JDK1.8，当同一个Hash值的节点数不小于8时，不再采用单链表形式存储，而是采用红黑树，如下图所示：



内部包含一个Node类型的数据table

```
/**
 * 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;
```

Node 存储着键值对。它包含了四个字段，从 next 字段我们可以看出 Node 是一个链表。即数组中的每个位置被当成一个桶，一个桶存放一个链表。HashMap 使用拉链法来解决冲突，同一个链表中存放哈希值和散列桶取模运算结果相同的 Node

```
/**
 * 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;

/**
 * Basic hash bin node, used for most entries.  (See below for
 * TreeNode subclass, and in LinkedHashMap for its Entry subclass.)
 */
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

    Node(int hash, K key, V value, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }

    public final K getKey()        { return key; }
    public final V getValue()      { return value; }
    public final String toString() { return key + "=" + value; }

    public final int hashCode() {
        return Objects.hashCode(key) ^ Objects.hashCode(value);
    }

    public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

    public final boolean equals(Object o) {
        if (o == this)
            return true;
        if (o instanceof Map.Entry) {
            Map.Entry<?,?> e = (Map.Entry<?,?>)o;
            if (Objects.equals(key, e.getKey()) &&
                Objects.equals(value, e.getValue()))
                return true;
        }
        return false;
    }
}
```

1. 拉链法的工作原理

```
HashMap<String, String> map = new HashMap<>();
map.put("K1", "V1");
```

```
map.put("K2", "V2");
map.put("K3", "V3");
```

- 新建一个 HashMap，默认大小为 16

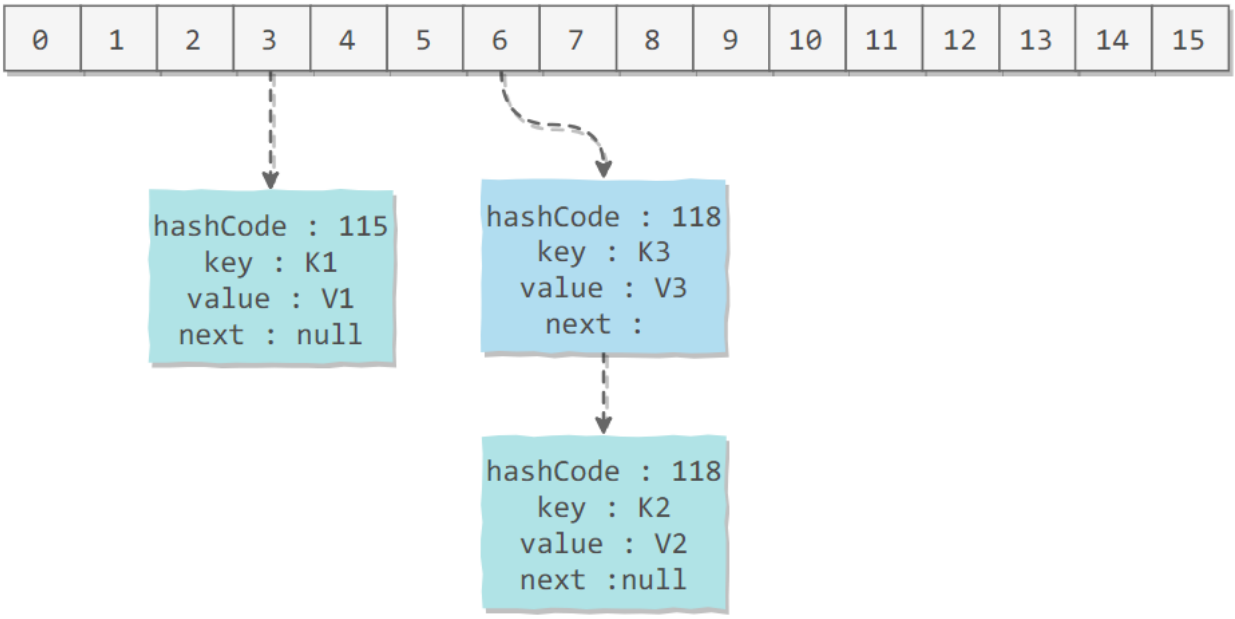
```
/**
 * The default initial capacity - MUST be a power of two.
 */
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

- 插入 键值对，先计算 K1 的 hashCode 为 115，使用除留余数法得到所在的桶下标 115%16=3
- 插入 键值对，先计算 K2 的 hashCode 为 118，使用除留余数法得到所在的桶下标 118%16=6。
- 插入 键值对，先计算 K3 的 hashCode 为 118，使用除留余数法得到所在的桶下标 118%16=6，插在 前面。

应该注意到链表的插入是以头插法方式进行的，例如上面的 不是插在 后面，而是插入在链表头部

查找需要分成两步进行：

- 计算键值对所在的桶；
- 在链表上顺序查找，时间复杂度显然和链表的长度成正比



1. put操作

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

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        //如果哈希表为空或长度为0，调用resize()方法创建哈希表
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        //如果哈希表中K对应的桶为空，那么该K，V对将成为该桶的头节点
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        //该桶处已有节点，即发生了哈希冲突
        else {
            Node<K,V> e; K k;
            //如果添加的值与头节点相同，将e指向p
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            //如果与头节点不同，并且该桶目前已经是红黑树状态，调用putTreeVal()方法
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            //桶中仍是链表阶段
            else {
                //遍历，要比较是否与已有节点相同
                for (int binCount = 0; ; ++binCount) {
                    //将e指向下一个节点，如果是null，说明链表中没有相同节点，添加到链表尾部即可
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        //如果此时链表个数达到了8，那么需要将该桶处链表转换成红黑树，treeifyBin()方法将hash处的桶转成红黑树
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    //如果与已有节点相同，跳出循环
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
```

```
                }
            }
            //如果有重复节点，那么需要返回旧值
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                //子类实现
                afterNodeAccess(e);
                return oldValue;
            }
        }
        //是一个全新节点，那么size需要+1
        ++modCount;
        //如果超过了阈值，那么需要resize()扩大容量
        if (++size > threshold)
            resize();
        //子类实现
        afterNodeInsertion(evict);
        return null;
    }
```

putVal流程

1. 判断哈希表是否为空，如果为空，调用resize()方法进行创建哈希表
2. 根据hash值得到哈希表中桶的头节点，如果为null，说明是第一个节点，直接调用newNode()方法添加节点即可
3. 如果发生了哈希冲突，那么首先会得到头节点，比较是否相同，如果相同，则进行节点值的替换返回
4. 如果头节点不相同，但是头节点已经是TreeNode了，说明该桶处已经是红黑树了，那么调用putTreeVal()方法将该结点加入到红黑树中
5. 如果头节点不是TreeNode，说明仍然是链表阶段，那么就需要从头开始遍历，一旦找到了相同的节点就跳出循环或者直到了链表尾部，那么将该节点插入到链表尾部
6. 如果插入到链表尾部后，链表个数达到了阈值8，那么将会将该链表转换成红黑树，调用treeifyBin()方法
7. 如果是新加一个数据，那么将size+1，此时如果size超过了阈值，那么需要调用resize()方法进行扩容
8. 扩容

```
final Node<K,V>[] resize() {
        Node<K,V>[] oldTab = table;
        int oldCap = (oldTab == null) ? 0 : oldTab.length;//旧表容量
        int oldThr = threshold;//旧表与之
        int newCap, newThr = 0;
        //旧表存在
        if (oldCap > 0) {
            //旧表已经达到了最大容量，不能再大，直接返回旧表
            if (oldCap >= MAXIMUM_CAPACITY) {
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }
            //否则，新容量为旧容量2倍，新阈值为旧阈值2倍
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY)
                newThr = oldThr << 1; // double threshold
        }
        //如果就阈值>0，说明构造方法中指定了容量
        else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;
        //初始化时没有指定阈值和容量，使用默认的容量16和阈值16*0.75=12
        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);
        }
        //更新阈值
        threshold = newThr;
        //创建表,初始化或更新表
        @SuppressWarnings({"rawtypes","unchecked"})
            Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        table = newTab;
        //如果属于容量扩展，rehash操作
        if (oldTab != null) {
            //遍历旧表
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                //如果该桶处存在数据
                if ((e = oldTab[j]) != null) {
                    //将旧表数据置为null，帮助gc
                    oldTab[j] = null;
                    //如果只有一个节点，直接在新表中赋值
                    if (e.next == null)
                        newTab[e.hash & (newCap - 1)] = e;
                    //如果该节点已经为红黑树
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                    //如果该桶处仍为链表
                    else { // preserve order
                    //下面这段暂时没有太明白，通过e.hash & oldCap将链表分为两队，参考知乎上的一段解释
                        /**
* 把链表上的键值对按hash值分成lo和hi两串，lo串的新索引位置与原先相同[原先位
* j]，hi串的新索引位置为[原先位置j+oldCap]；
* 链表的键值对加入lo还是hi串取决于 判断条件if ((e.hash & oldCap) == 0)，因为* capacity是2的幂，所以oldCap为10...0的二进制形式，若判断条件为真，意味着
* oldCap为1的那位对应的hash位为0，对新索引的计算没有影响（新索引
* =hash&(newCap-*1)，newCap=oldCap<<2）；若判断条件为假，则 oldCap为1的那位* 对应的hash位为1，
```

```
* 即新索引=hash&( newCap-1 )= hash&( (oldCap<<2) - 1)，相当于多了10...0，
* 即 oldCap

* 例子：
* 旧容量=16，二进制10000；新容量=32，二进制100000
* 旧索引的计算：
* hash = xxxx xxxx xxxy xxxx
* 旧容量-1 1111
* &运算 xxxx
* 新索引的计算：
* hash = xxxx xxxx xxxy xxxx
* 新容量-1 1 1111
* &运算 y xxxx
* 新索引 = 旧索引 + y0000，若判断条件为真，则y=0(lo串索引不变)，否则y=1(hi串
* 索引=旧索引+旧容量10000)
   */
                        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;
    }
```

1. 与 HashTable 的比较

- HashTable 使用 synchronized 来进行同步。
- HashMap 可以插入键为 null 的 Entry。
- HashMap 的迭代器是 fail-fast 迭代器。
- HashMap 不能保证随着时间的推移 Map 中的元素次序是不变的。

ConcurrentHashMap

1. 存储结构

```
/**
     * Key-value entry.  This class is never exported out as a
     * user-mutable Map.Entry (i.e., one supporting setValue; see
     * MapEntry below), but can be used for read-only traversals used
     * in bulk tasks.  Subclasses of Node with a negative hash field
     * are special, and contain null keys and values (but are never
     * exported).  Otherwise, keys and vals are never null.
     */
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        volatile V val;
        volatile Node<K,V> next;

        Node(int hash, K key, V val, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.val = val;
            this.next = next;
        }

        public final K getKey()       { return key; }
        public final V getValue()     { return val; }
        public final int hashCode()   { return key.hashCode() ^ val.hashCode(); }
        public final String toString(){ return key + "=" + val; }
        public final V setValue(V value) {
            throw new UnsupportedOperationException();
        }

        public final boolean equals(Object o) {
            Object k, v, u; Map.Entry<?,?> e;
            return ((o instanceof Map.Entry) &&
                    (k = (e = (Map.Entry<?,?>)o).getKey()) != null &&
                    (v = e.getValue()) != null &&
                    (k == key || k.equals(key)) &&
```

```
                (v == (u = val) || v.equals(u)));
        }

        /**
         * Virtualized support for map.get(); overridden in subclasses.
         */
        Node<K,V> find(int h, Object k) {
            Node<K,V> e = this;
            if (k != null) {
                do {
                    K ek;
                    if (e.hash == h &&
                        ((ek = e.key) == k || (ek != null && k.equals(ek))))
                        return e;
                } while ((e = e.next) != null);
            }
            return null;
        }
    }
```

ConcurrentHashMap 和 HashMap 实现上类似，最主要的差别是 ConcurrentHashMap 采用了分段锁（Segment），每个分段锁维护着几个桶（HashEntry），多个线程可以同时访问不同分段锁上的桶，从而使其并发度更高