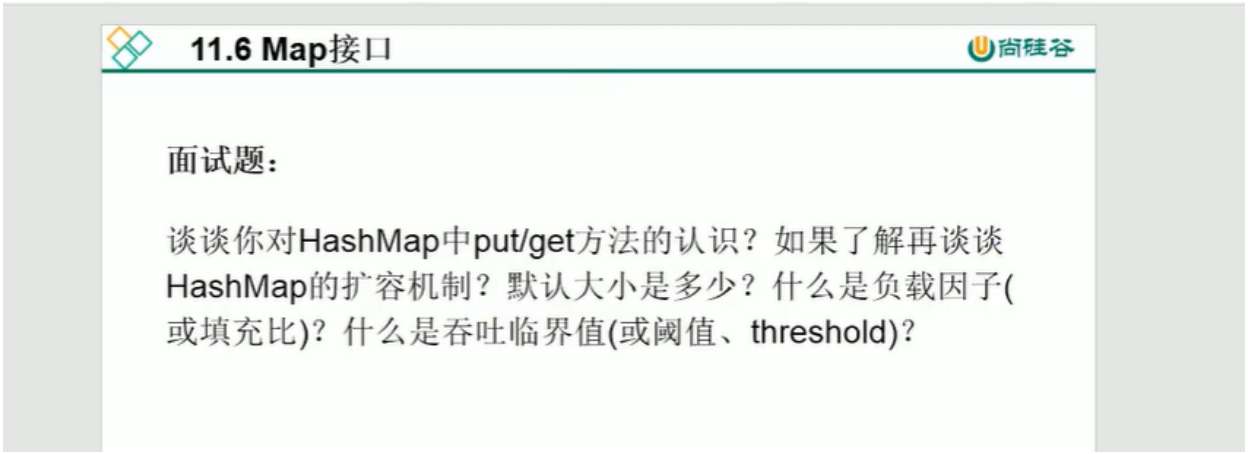
HashMap和HashTable的区别

```
1. 
HashMap允许存储null键和null值，由于key不能重复，所以键null只有一个
Hashtable不允许存储null键和null值
2. 
Hashtable是线程安全的，HashMap不是线程安全的
多线程环境下使用HashMap可以使用Collections.synchronizedMap()获得一个线程安全的HashMap

```



HashMap JDK7源码分析

```
public HashMap() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR);
}
```

1.默认构造器

默认容量16，默认加载因子0.75f

```
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);

    // Find a power of 2 >= initialCapacity
    int capacity = 1;
    while (capacity < initialCapacity)
        capacity <<= 1;

    this.loadFactor = loadFactor;
    threshold = (int)Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
    table = new Entry[capacity];
    useAltHashing = sun.misc.VM.isBooted() &&
            (capacity >= Holder.ALTERNATIVE_HASHING_THRESHOLD);
    init();
```

默认情况下都进入到

`while(capacity < initialCapacity)`

`capacity <<=1;`

例：

当你声明一个长度为15的HashMap

进入这里，经过capacity的移位，也会变成16（2的n次幂）

put()方法

```
public V put(K key, V value) {
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key);
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess( m: this);
            return oldValue;
        }
    }

    modCount++;
    addEntry(hash, key, value, i);
    return null;
}
```

```
    */
    static int indexFor(int h, int length) {
        return h & (length-1);
    }
```

```
1 通过hash值和数组长度的与运算，获得元素的位置
2
```

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    if ((size >= threshold) && (null != table[bucketIndex])) {
        resize( newCapacity: 2 * table.length);
        hash = (null != key) ? hash(key) : 0;
        bucketIndex = indexFor(hash, table.length);
    }

    createEntry(hash, key, value, bucketIndex);
}

/**
```

```
1 扩容：
2 当容量大于临界值的时候并且要插入的位置不为空时扩容
3
```

```
 */
void createEntry(int hash, K key, V value, int bucketIndex) {
    Entry<K,V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<>(hash, key, value, e);
    size++;
}
```

1 将插入的元素放在i的位置上，原来的元素放在新插入元素的后面，叫头插法

get()方法：

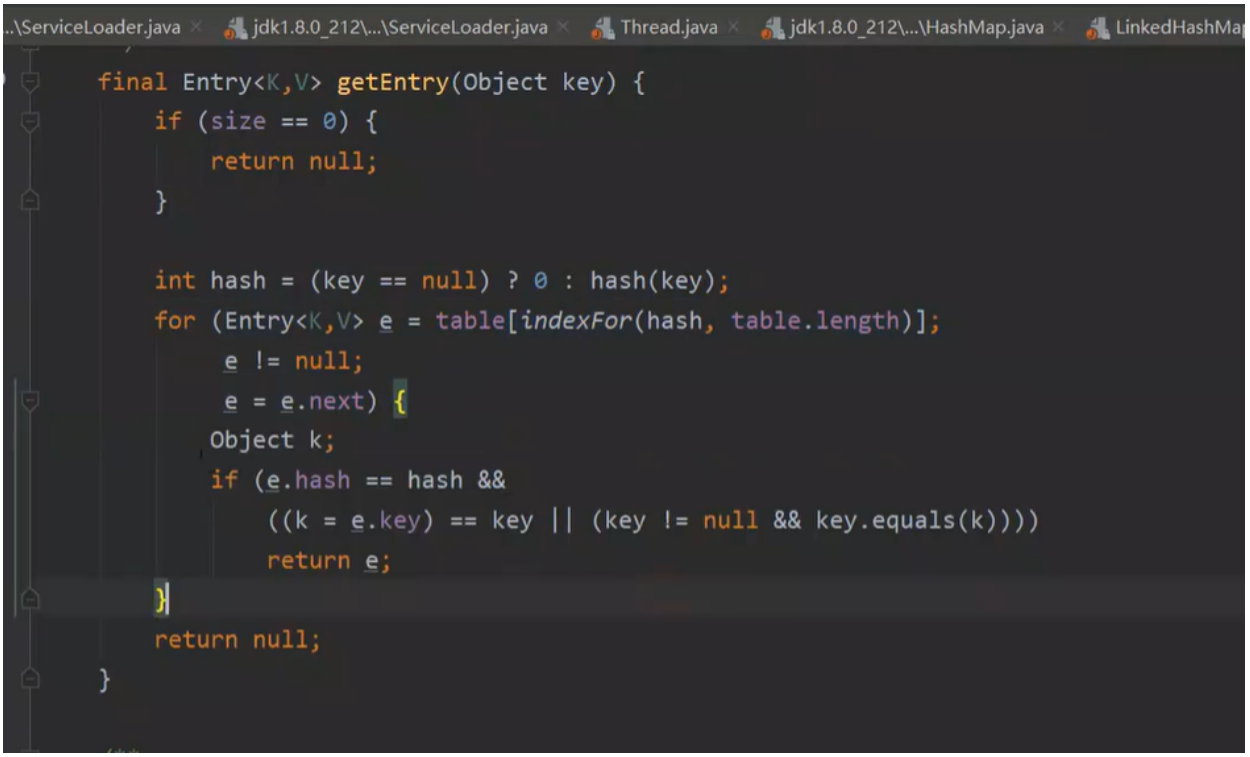
```
 */
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    Entry<K,V> entry = getEntry(key);

    return null == entry ? null : entry.getValue();
}

/**
```

```
     */
    private V getForNullKey() {
        if (size == 0) {
            return null;
        }
        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
            if (e.key == null)
                return e.value;
        }
        return null;
    }

    /**
```



HashMap JDK8源码

默认初始化

```
public HashMap() {
 this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}
默认初始化只赋值了加载因子
底层Node[]数组没有初始化
```

put方法

```
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;
 //第一次table==null,table通过resize进行初始化，初始化值为16
 if ((tab = table) == null || (n = tab.length) == 0)
 n = (tab = resize()).length;
 //如果当前位置的元素为空，就放入newNode
 if ((p = tab[i = (n - 1) & hash]) == null)
```

```
tab[i] = newNode(hash, key, value, null);
//否则说明这个点有元素
else {
Node<K,V> e; K k;
//先判断如果两个的hash值相等并且key相等就替换当前节点
if (p.hash == hash &&
((k = p.key) == key || (key != null && key.equals(k))))
e = p;
else if (p instanceof TreeNode)
e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
else {
//就看其他节点的hash值和key
for (int binCount = 0; ; ++binCount) {
//如果i元素的下一个为空，就直接赋值
if ((e = p.next) == null) {
p.next = newNode(hash, key, value, null);
//当链表的个数达到8个时，进入红黑树
if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
treeifyBin(tab, hash);
break;
}
if (e.hash == hash &&
((k = e.key) == key || (key != null && key.equals(k))))
break;
p = e;
}
}
//替换节点，并且把新的value赋值给节点，返回旧的value
if (e != null) { // existing mapping for key
V oldValue = e.value;
if (!onlyIfAbsent || oldValue == null)
e.value = value;
afterNodeAccess(e);
return oldValue;
}
}
++modCount;
if (++size > threshold)
resize();
afterNodeInsertion(evict);
```

```
 return null;
}

红黑树
final void treeifyBin(Node<K,V>[] tab, int hash) {
 int n, index; Node<K,V> e;
 //如果当前数组的长度小于64，只进行扩容
 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);
 }
}
```

LinkedHashMap原理

```
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);
 }
}
底层的节点entry记录了前后两个的元素
```

HashTable锁住整个对象，

ConcurrentHashMap：

默认初始化：

```
public ConcurrentHashMap(int initialCapacity,
                         float loadFactor, int concurrencyLevel) {
    if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    if (initialCapacity < concurrencyLevel)   // Use at least as many bins
        initialCapacity = concurrencyLevel;   // as estimated threads
    long size = (long)(1.0 + (long)initialCapacity / loadFactor);
    int cap = (size >= (long)MAXIMUM_CAPACITY) ?
        MAXIMUM_CAPACITY : tableSizeFor((int)size);
    this.sizeCtl = cap;
}
```

```
1 concurrencyLevel：并发级别，segment的数量
```

put()方法：

```
public V put( @NotNull K key,  @NotNull V value) {
    Segment<K,V> s;
    if (value == null)
        throw new NullPointerException();
    int hash = hash(key);
    int j = (hash >>> segmentShift) & segmentMask;
    if ((s = (Segment<K,V>)UNSAFE.getObject          // nonvolatile; recheck
         (segments,  l: (j << SSHIFT) + SBASE)) == null) //  in ensureSegment
        s = ensureSegment(j);
    return s.put(key, hash, value,  onlyIfAbsent: false);
}
```

```
1  ConcurrentHashMap 1.7和1.8的区别
2  1、整体结构
3  1.7：Segment + HashEntry + Unsafe
4  1.8：移除Segment，使锁的粒度更小，Synchronized + CAS + Node + Unsafe
5  2、put（）
6  1.7：先定位Segment，再定位桶，put全程加锁，没有获取锁的线程提前找桶的位置，并最多自旋64次获取锁，超过则挂起。
7  1.8：由于移除了Segment，类似HashMap，可以直接定位到桶，拿到first节点后进行判断，1、为空则CAS插入；2、为-1则说明在扩容，则跟着一起扩容；3、else则加锁put（类似1.7）
8  3、get（）
9  基本类似，由于value声明为volatile，保证了修改的可见性，因此不需要加锁。
10  4、resize（）
11  1.7：跟HashMap步骤一样，只不过是搬到单线程中执行，避免了HashMap在1.7中扩容时死循环的问题，保证线程安全。
```

1.8：支持并发扩容，HashMap扩容在1.8中由头插改为尾插（为了避免死循环问题），ConcurrentHashmap也是，迁移也是从尾部开始，扩容前在桶的头部放置一个hash值为-1的节点，这样别的线程访问时就能判断是否该桶已经被其他线程处理过了。

5、size（）

1.7：很经典的思路：计算两次，如果不变则返回计算结果，若不一致，则锁住所有的Segment求和。

1.8：用baseCount来存储当前的节点个数，这就设计到baseCount并发环境下修改的问题（说实话我没看懂-_-!）。