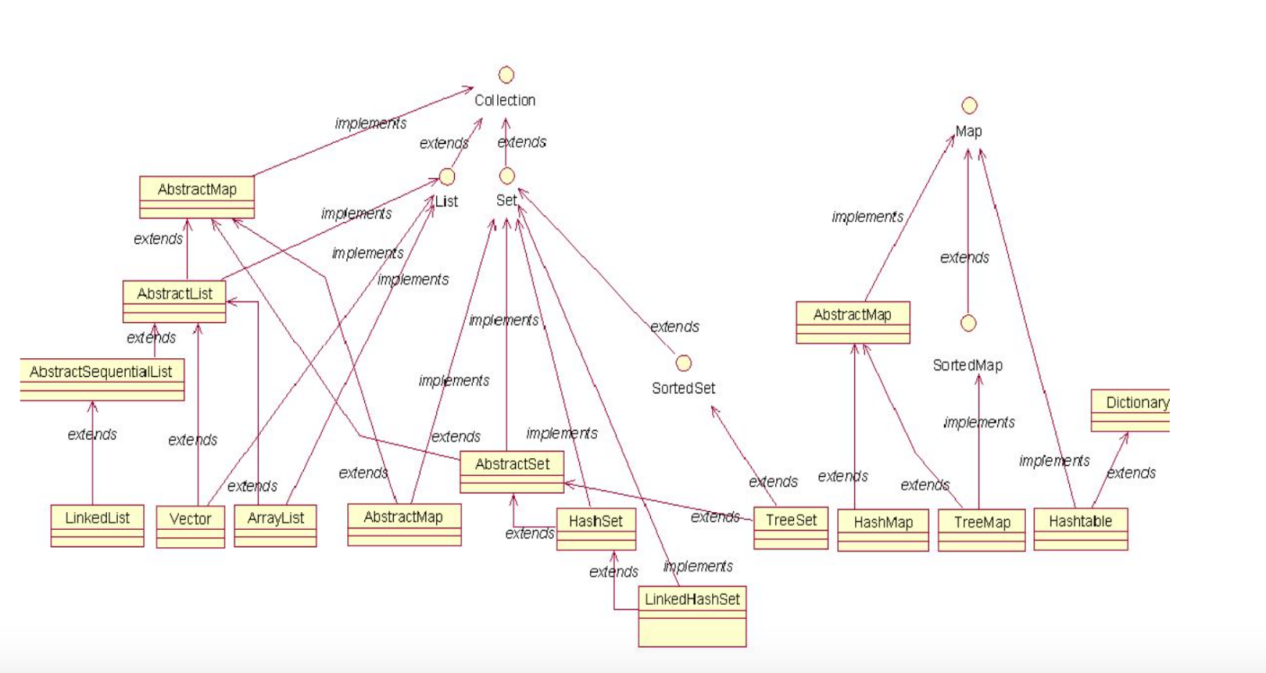
Java集合

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Collection** |  | **是否有序** | **是否允许元素重复** | **是否允许null值** | **实现** | **优点** | **缺点** | **默认容量** | **扩容因子** | **扩容** |
| **List** | **ArrayList** | **是** | **是** | **是** | **数组实现** | **无容量限制，查询快，轻量级** | **增删慢，线程不安全** | **10** | **1** | **1.5倍** |
|  | **LinkedList** | **是** | **是** | **是** | **链表实现** | **增删快** | **查询慢，双向链表，线程不安全** | **10** | **1** | **1.5倍** |
|  | **Vector** | **是** | **是** | **否** | **数组实现** | **线程安全，查询快** | **重量级，存在性能问题，增删慢** | **10** | **1** | **2倍** |
| **Set** | **HashSet** | **否** | **否** | **是** | **基于hashMap实现** | **效率高** | **非线程安全** | **16** | **0.75** | **2倍** |
|  | **TreeSet** | **是（用二叉树实现排序）** | **否** | **否** | **基于TreeMap实现** | **可以指定比较方法进行排序** | **非线程安全** |  |  |  |
|  | **LinkedHashSet** | **是（用二叉树实现排序）** | **否** | **是** | **继承hashSet，底层使用LinkedHashMap** | **链表保证元素有序，hash表保证元素唯一** | **双重链接列表** | **16** | **0.75** | **2倍** |
| **Map** | **HashMap** | **否** | **使用key-value来映射和存储数据，Key必须唯一，value可以重复** | **最多只允许一条记录的key为null** | **根据key的hashCode值存储数据** | **使用hash算法去重，效率高** | **非线程安全** | **16** | **0.75** | **2倍** |
|  | **TreeMap** | **是（用二叉树排序）** | **同上** | **否** | **基于红黑树实现，继承AbstractMap，并且实现了NavigableMap接口** | **可以用指定比较方法进行排序** | **非线程安全** |  |  |  |
|  | **LinkedHashMap** | **是** | **同上** | **是** | **在hashMap基础上采用双向链表** | **元素迭代顺序与插入顺序一致** | **双重链接链表，非同步，遍历比hashMap慢** | **16** | **0.75** | **2倍** |
|  | **WeakHashMap** | **否** | **同上** | **是** | **KV对存储，“弱键”自动被GC回收删除，继承AbstractMap** |  |  |  |  |  |
|  | **HashTale** | **否** | **同上** | **否** | **在hashMap基础上加synchronized** | **线程安全** | **效率低** | **11** | **0.75** | **2倍+1** |



1. ArrayList源码分析

默认容量为10。

private static final int *DEFAULT\_CAPACITY* = 10;

创建了一个大小为0的数组，在后面用到。

private static final Object[] *EMPTY\_ELEMENTDATA* = {};

声明一个数组。

transient Object[] elementData;

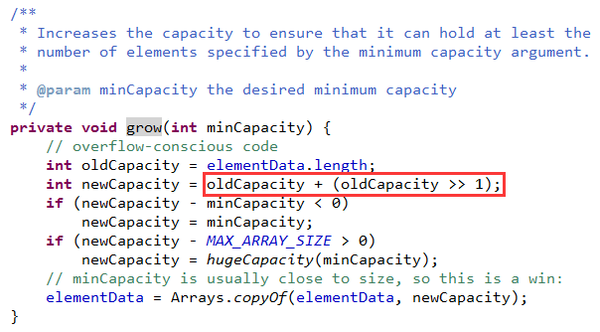
ArrayList的无参构造方法，将前面声明创建的大小为0的数组赋给elementData数组。

public ArrayList() {  
 this.elementData = *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*;  
}

这是ArrayList的有参构造方法，传入一个int类型的变量，相当于我们在使用arrayList的时候指定list的大小。如果传入的参数大于0，则新建一个initialCapacity大小的数组。传入值等于0的话，将这个空数组给elementData。

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);  
 }  
}

**扩容机制：**



第一次添加元素扩充容量为10，之后的扩充算法：原来数组大小+原来数组一半。

步骤：

1. 得到当前ArrayList的容量（oldCapacity）。为elementData.length;
2. 计算出扩容后的新容量（newCapacity），其值（oldCapacity+(oldCapacity>>1)）约是oldCapacity的1.5倍。
3. 当newCapacity小于所需最小容量，那么将所需最小容量赋给newCapacity。
4. newCapacity大于ArrayList的所允许的最大容量，将处理。
5. 进行数据复制，完成向ArrayList实例添加元素操作。

2.HashMap源码分析

HashMap初始默认大小:

static final int *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4; // aka 16

增长因子：

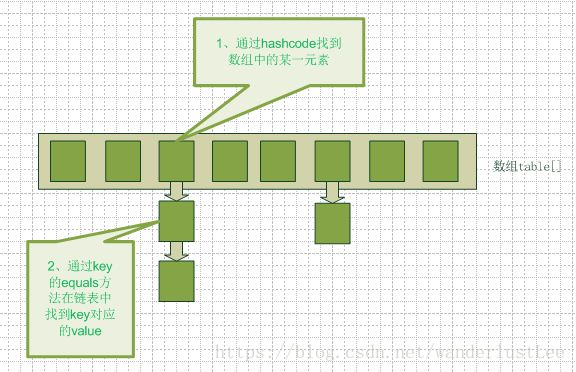
static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f;

最大大小限制：

static final int *MAXIMUM\_CAPACITY* = 1 << 30;

**JDK1.7：**

HashMap底层以数组+链表实现。



Put操作时：

对Key做hash算法得到int hashCode=hash(key);

int index(数组下标)=hashCode%table.length;

它是一个Entry类型的数组，Entry的源码：

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

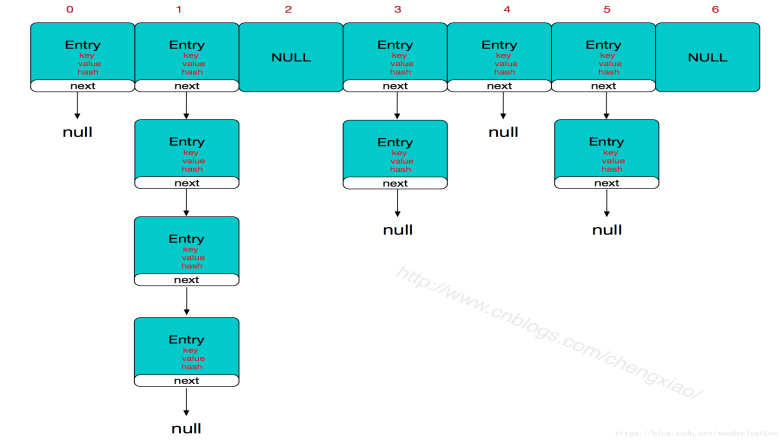
final int hash;

Entry<K,V> next;

}

其中存放了Key，Value，hash值，还有指向下一个元素的引用。

当向HashMap中put(key,value)时，会首先通过hash算法计算出存放到数组中的位置，比如位置索引为i，将其放入到Entry[i]中，如果这个位置上面已经有元素了，那么就将新加入的元素放在链表的头上，最先加入的元素在链表尾。比如，第一个键值对A进来，通过计算其key的hash得到的index=0，记做:Entry[0] = A。一会后又进来一个键值对B，通过计算其index也等于0，现在怎么办？HashMap会这样做:B.next = A,Entry[0] = B,如果又进来C,index也等于0,那么C.next = B,Entry[0] = C；这样我们发现index=0的地方其实存取了A,B,C三个键值对,他们通过next这个属性链接在一起,也就是说数组中存储的是最后插入的元素。



HashMap的get(key)方法是：首先计算key的hashcode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。从这里我们可以想象得到，如果每个位置上的链表只有一个元素，那么hashmap的get效率将是最高的。所以我们需要让这个hash算法尽可能的将元素平均的放在数组中每个位置上。

**扩容机制：**

缺点：在并发情况下会造成死循环。

当HashMap中的元素越来越多的时候，hash冲突的几率也就越来越高，因为数组的长度是固定的。所以为了提高查询的效率，就要对HashMap的数组进行扩容。

HashMap的容量由一下几个值决定：

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // HashMap初始容量大小(16)  
static final int MAXIMUM\_CAPACITY = 1 << 30; // HashMap最大容量  
transient int size; // The number of key-value mappings contained in this map   
static final float DEFAULT\_LOAD\_FACTOR = 0.75f; // 负载因子  
HashMap的容量size乘以负载因子[默认0.75] = threshold; // threshold即为开始扩容的临界值  
transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY\_TABLE; // HashMap的基本构成Entry数组

当HashMap中的元素个数超过数组大小(数组总大小length,不是数组中个数size)\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为16，那么当HashMap中元素个数超过16\*0.75=12（这个值就是代码中的threshold值，也叫做临界值）的时候，就把数组的大小扩展为 2\*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置。

0.75这个值成为负载因子，那么为什么负载因子为0.75呢？这是通过大量实验统计得出来的，如果过小，比如0.5，那么当存放的元素超过一半时就进行扩容，会造成资源的浪费；如果过大，比如1，那么当元素满的时候才进行扩容，会使get,put操作的碰撞几率增加。

**HashMap中扩容是调用resize()方法，方法源码：**

void resize(int newCapacity) {  
 Entry[] oldTable = table;  
 int oldCapacity = oldTable.length;  
 //如果当前的数组长度已经达到最大值，则不在进行调整  
 if (oldCapacity == MAXIMUM\_CAPACITY) {  
 threshold = Integer.*MAX\_VALUE*;  
 return;  
 }  
 //根据传入参数的长度定义新的数组  
 Entry[] newTable = new Entry[newCapacity];  
 //按照新的规则，将旧数组中的元素转移到新数组中  
 transfer(newTable);  
 table = newTable;  
 //更新临界值  
 threshold = (int)(newCapacity \* loadFactor);  
}  
//旧数组中元素往新数组中迁移  
void transfer(Entry[] newTable) {  
 //旧数组  
 Entry[] src = table;  
 //新数组长度  
 int newCapacity = newTable.length;  
 //遍历旧数组  
 for (int j = 0; j < src.length; j++) {  
 Entry<K,V> e = src[j];  
 if (e != null) {  
 src[j] = null;  
 do {  
 Entry<K,V> next = e.next;  
 int i = indexFor(e.hash, newCapacity);//放在新数组中的index位置  
 e.next = newTable[i];//实现链表结构，新加入的放在链头，之前的的数据放在链尾  
 newTable[i] = e;  
 e = next;  
 } while (e != null);  
 }  
 }  
}

可以看到HashMap不是无限扩容的，当达到了实现预定的MAXIMUM\_CAPACITY，就不再进行扩容。

Hashmap为什么大小是2的幂次？

因为在计算元素该存放的位置的时候，用到的算法是将元素的hashcode与当前map长度-1进行与运算。源码：

static int indexFor(int h, int length) {  
 // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";  
 return h & (length-1);  
}

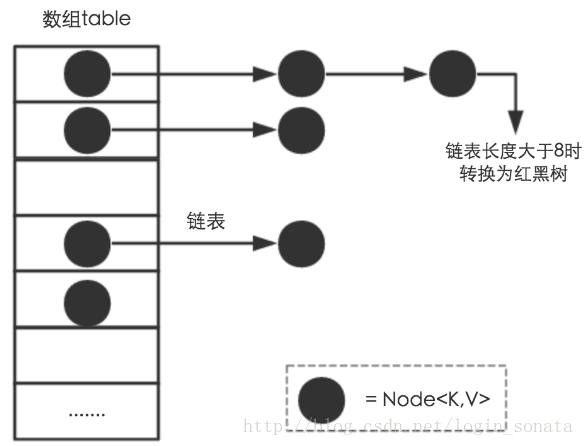
如果map长度为2的幂次，那长度-1的二进制一定为11111...这种形式，进行与运算就看元素的hashcode，但是如果map的长度不是2的幂次，比如为15，那长度-1就是14，二进制为1110，无论与谁相与最后一位一定是0，0001，0011，0101，1001，1011，0111，1101这几个位置就永远都不能存放元素了，空间浪费相当大。也增加了添加元素是发生碰撞的机会。减慢了查询效率。所以Hashmap的大小是2的幂次。

get方法实现

Hashmap get一个元素是，是计算出key的hashcode找到对应的entry，这个时间复杂度为O(1)，然后通过对entry中存放的元素key进行equal比较，找出元素，这个的时间复杂度为O(m)，m为entry的长度。

**JDK1.8实现：**

HashMap是****数组+链表+红黑树****（JDK1.8增加了红黑树部分）实现的。



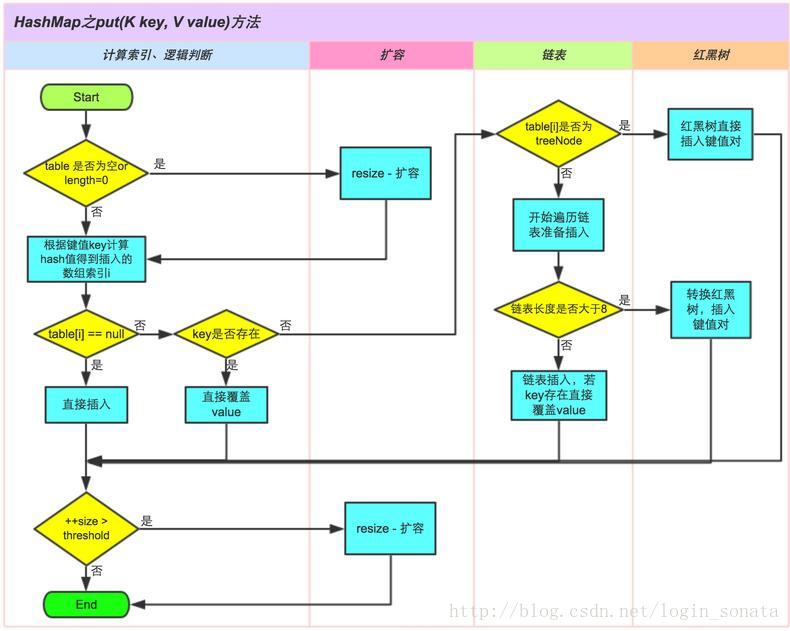
其他的与jdk1.7相同

为了使得链表的长度不要那么长，当链表的长度大于8时，转为红黑树，不同于jdk1.7的是jdk1.7中增加链表元素时在链表头部增加然后整体向下移动一位，而jdk1.8直接添加到链表的尾部，在扩容的时候就会避免死循环。

扩容后的index=index+oldTable.length.

****threshold就是在此Load factor和length(数组长度)对应下允许的最大元素数目，超过这个数目就重新resize(扩容)，扩容后的HashMap容量是之前容量的两倍****。默认的负载因子0.75是对空间和时间效率的一个平衡选择，

HashMap的put方法：



3.ConcurrentHashMap源码分析

**jDK1.7中concurrentHashMap的底层分析:**

HashTable是对整个HashMap加了锁，作用范围太大，导致性能下降，所以考虑给其中的一部分加锁。也就是对元素进行分组，然后给每一组分别加锁，这样就可以让多个元素共用一把锁（即分段锁:segment），元素在put的时候只需要看它是否能获取当前分组的锁即可，不会受到其他组的操作而受影响。每一个segment可以理解为是一个HashMap。即先有一个segment数组，每个segment数组中又存了一个数组是保存HashEntry的。

ConcurrentHashMap中的构造函数：

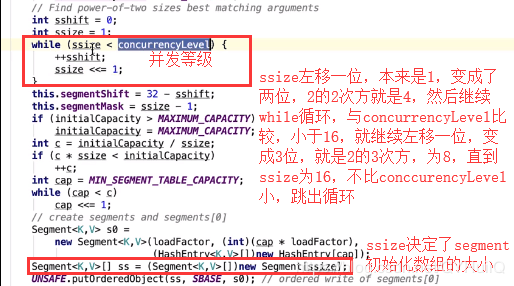
IMG_256

参数一默认值是16，指的是真正存数据的存储容量，也就是segment下存储的数组的容量。

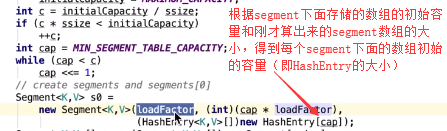
参数二为加载因子，是真正存元素的数组的加载因子。

参数三是并发等级。

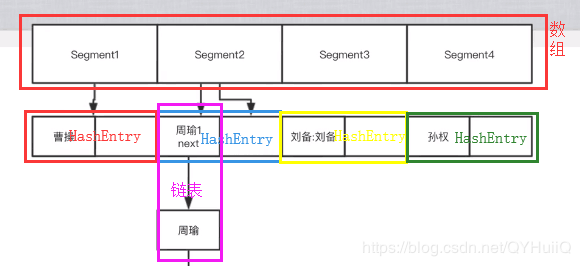
在上述的this()方法中初始化segment数组：

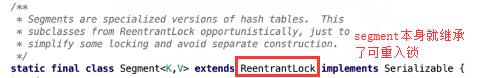


初始化每个segment下面存储的数组的大小：

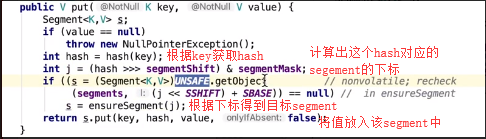


结构图：

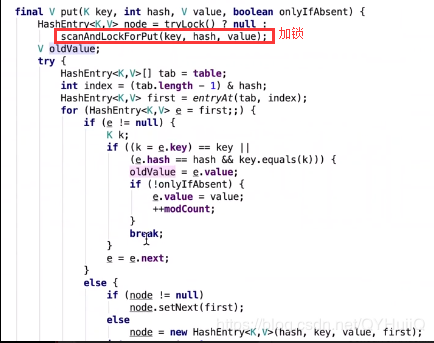




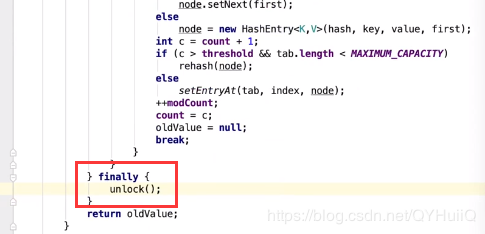
在执行put方法时：



最后调用put()放入HashEntry中：

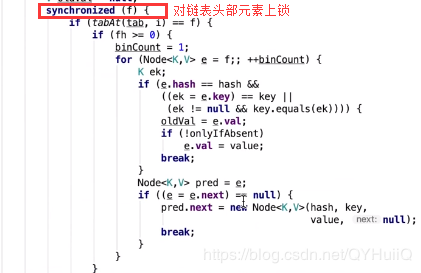


使用完之后解锁：



在java8中继续优化：

每次在put()的时候其实我们都是要去修改HashEntry数组中存的那个元素，即链表头部元素。所以我们可以只对数组中存的那个元素加锁即可。所以，在java8中不存在segment的概念。



Java8中的ConcurrentHashMap的底层结构：数组+链表+红黑树。

线程安全，key/value不为空。

1. CopyOnWriteArrayList源码分析

1、CopyOnWrite容器（并发容器）

Copy-On-Write简称COW，是一种用于程序设计中的优化策略。

其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容Copy出去形成一个新的内容然后再改，这是一种延时懒惰策略。

从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器,它们是CopyOnWriteArrayList和CopyOnWriteArraySet。

CopyOnWrite容器即写时复制的容器。

通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。

这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。

所以CopyOnWrite容器是一种读写分离的思想，读和写不同的容器、最终一致性 以及使用另外开辟空间的思路，来解决并发冲突的思想。

2、CopyOnWriteArrayList数据结构：

public class CopyOnWriteArrayList<E>  
 implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

CopyOnWriteArrayList实现了List接口，List接口定义了对列表的基本操作；

同时实现了RandomAccess接口，表示可以随机访问（数组具有随机访问的特性）；

同时实现了Cloneable接口，表示可克隆；

同时也实现了Serializable接口，表示可被序列化。

CopyOnWriteArrayList底层使用数组来存放元素。

3、CopyOnWriteArrayList Add方法：

CopyOnWriteArrayList容器是Collections.synchronizedList(List list)的替代方案，是一个ArrayList的线程安全的变体。

基本原理：

初始化的时候只有一个容器，很常一段时间，这个容器数据、数量等没有发生变化的时候，大家(多个线程)，都是读取(假设这段时间里只发生读取的操作)同一个容器中的数据，所以这样大家读到的数据都是唯一、一致、安全的，但是后来有人往里面增加了一个数据，这个时候CopyOnWriteArrayList 底层实现添加的原理是先copy出一个容器(可以简称副本)，再往新的容器里添加这个新的数据，最后把新的容器的引用地址赋值给了之前那个旧的的容器地址，但是在添加这个数据的期间，其他线程如果要去读取数据，仍然是读取到旧的容器里的数据。

CopyOnWriteArrayList中add方法的实现（向CopyOnWriteArrayList里添加元素），可以发现在添加的时候是需要加锁的，否则多线程写的时候会Copy出N个副本出来。

public void add(int index, E element) {  
 synchronized (lock) {  
 Object[] elements = getArray();  
 int len = elements.length;  
 if (index > len || index < 0)  
 throw new IndexOutOfBoundsException(*outOfBounds*(index, len));  
 Object[] newElements;  
 int numMoved = len - index;  
 if (numMoved == 0)  
 newElements = Arrays.*copyOf*(elements, len + 1);  
 else {  
 newElements = new Object[len + 1];  
 System.*arraycopy*(elements, 0, newElements, 0, index);  
 System.*arraycopy*(elements, index, newElements, index + 1,  
 numMoved);  
 }  
 newElements[index] = element;  
 setArray(newElements);  
 }  
}

读的时候不需要加锁，如果读的时候有多个线程正在向CopyOnWriteArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的CopyOnWriteArrayList。

public E get(int index) {  
 return *elementAt*(getArray(), index);  
}

remove方法：

删除元素，很简单，就是判断要删除的元素是否最后一个，如果最后一个直接在复制副本数组的时候，复制长度为旧数组的length-1即可；

但是如果不是最后一个元素，就先复制旧的数组的index前面元素到新数组中，然后再复制旧数组中index后面的元素到数组中，最后再把新数组复制给旧数组的引用。最后在finally语句块中将锁释放。

5.无论我们用哪一个构造方法创建一个CopyOnWriteArrayList对象，都会创建一个Object类型的数组，然后赋值给成员array。  
  
6、copyOf函数：

该函数用于复制指定的数组，截取或用 null 填充（如有必要），以使副本具有指定的长度.

@HotSpotIntrinsicCandidate  
public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType) {  
 @SuppressWarnings("unchecked")  
 T[] copy = ((Object)newType == (Object)Object[].class)  
 ? (T[]) new Object[newLength]  
 : (T[]) Array.newInstance(newType.getComponentType(), newLength);  
 System.arraycopy(original, 0, copy, 0,  
 Math.min(original.length, newLength));  
 return copy;  
}

CopyOnWrite的应用场景  
 CopyOnWrite并发容器用于读多写少的并发场景。

8、CopyOnWrite的缺点：

CopyOnWrite容器有很多优点（解决开发工作中的多线程的并发问题），但是同时也存在两个问题，即内存占用问题和数据一致性问题。

1.内存占用问题。

因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。

如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。

针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全是10进制的数字，可以考虑把它压缩成36进制或64进制。

或者不使用CopyOnWrite容器，而使用其他的并发容器，如ConcurrentHashMap。

2.数据一致性问题。

CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

9、总结：

1.CopyOnWriteArrayList适用于读多写少的场景

2.在并发操作容器对象时不会抛出ConcurrentModificationException，并且返回的元素与迭代器创建时的元素是一致的

3.容器对象的复制需要一定的开销，如果对象占用内存过大，可能造成频繁的YoungGC和Full GC

4.CopyOnWriteArrayList不能保证数据实时一致性，只能保证最终一致性

5.在需要并发操作List对象的时候优先使用CopyOnWriteArrayList

6.随着CopyOnWriteArrayList中元素的增加，CopyOnWriteArrayList的修改代价将越来越昂贵，因此，CopyOnWriteArrayList适用于读操作远多于修改操作的并发场景中。

# 5.CAS与ConcurrentLinkedQueue的实现

CAS

CAS，compare and swap的缩写，意思为比较并交换。

从JDK 5 开始，Doug lea 给我们提供了Coucurrent包，让我们解决并发问题，CAS就是实现这个包的基础。

CAS包含三个操作数--内存位置，预期位置和新值，当且仅当内存位置的值与预期位置的值相同的时候，处理器才会把该位置的值修改为新值，否则，处理器不做任何操作。无论如何，处理器都会在CAS指令之前返回该位置的值。

CAS指令允许算法执行读-修改-写操作，不用担心其他线程同时修改变量，因为该变量只能同时只有一个线程可以修改，其他线程失败后可以重新读取在操作。

利用CAS指令，可以完成java的非组阻塞算法，synchronized是阻塞算法，整个并发包都是在CAS基础之上完成的，相比synchronized，J.U.C在性能上有了很大的提高，可以很高效的完成原子操作。

**ConcurrentLinkedQueue**

ConcurrentLinkedQueue是基于CAS实现的非阻塞的线程安全的队列，它遵循先入先出的原则。当添加一个元素的时候，它被添加到队列的尾部，当获取一个元素的时候，它会返回队列头部的元素。

ConcurrentLinkedQueue是由head和tail节点组成，每个节点都有item和指向下一个节点的指针，组成了链表结构的队列，默认的构造函数是head和tail为null。

## **add方法**

添加元素的过程主要做了两件事，第一是把要添加的节点作为当前队列尾节点的next节点，第二是把更新尾节点，如果尾节点的next节点不为空，就把新添的节点作为尾节点，如果尾节点的next为空，就把新添节点作为尾节点的next节点。

1. 找到当前队列的尾节点，利用CAS把新增节点作为尾节点的下一个节点,保证最后一个节点是新添加的节点,其他失败的线程会更新p指向这个新节点.

2. 如果tail节点与p相距2个节点,修改tail指向.

3. 如果p的next节点指向自己的话,说明节点p已经被删除.

假设多个线程同时都要添加元素,第一次添加元素的时候,队列为空,全部线程都能执行到q=p.next=null,此时,只有一个线程才能利用CAS把p的next节点设为newNode,这时判断p==t,所以并不会利用CAS把newNode更新为tail节点,casNext方法执行成功,直接返回true.其它失败的线程走到最后一个else分支,把p指向newNode.这时,p的next肯定为空,第二个竞争成功执行的线程利用CAS把自己的节点放到newNode的next,由于t指向一个空节点,p指向newNode节点,所有p!=t,利用CAS把新添加的节点设为tail节点,依次类推,每隔两个节点更新一次tail节点,这样通过增加对volatile的读操作减少对volatile的写操作,写操作的开销要高于读操作,所以入队效率总体是提升的.

## **总结**

ConcurrentLinkedQueue是基于链表线程安全的非阻塞队列，采用先入先出的顺序对链表排序，当添加一个元素的时候直接加到队尾，当获取一个元素的时候，直接返回头部元素。通过CAS操作，在线程安全的前提下，提高了队列操作效率。

**问题：**

## **1.为什么不使用AVL树而使用红黑树？**

红黑树和AVL树都是****最常用的平衡二叉搜索树****，它们的查找、删除、修改都是O(lgn) time

AVL树和红黑树有几点比较和区别：  
（1）AVL树是更加严格的平衡，因此可以提供更快的查找速度，一般读取查找密集型任务，适用AVL树。  
（2）红黑树更适合于插入修改密集型任务。  
（3）通常，AVL树的旋转比红黑树的旋转更加难以平衡和调试。

2.[Java 中 Comparable 和 Comparator 比较](https://www.cnblogs.com/skywang12345/p/3324788.html)

Comparable 简介

Comparable 是排序接口。

若一个类实现了Comparable接口，就意味着“该类支持排序”。  即然实现Comparable接口的类支持排序，假设现在存在“实现Comparable接口的类的对象的List列表(或数组)”，则该List列表(或数组)可以通过 Collections.sort（或 Arrays.sort）进行排序。

此外，“实现Comparable接口的类的对象”可以用作“有序映射(如TreeMap)”中的键或“有序集合(TreeSet)”中的元素，而不需要指定比较器。

Comparable 定义

Comparable 接口仅仅只包括一个函数，它的定义如下：

package java.lang;import java.util.\*;

public interface Comparable<T> {

public int compareTo(T o);

}

说明：  
假设我们通过 x.compareTo(y) 来“比较x和y的大小”。若返回“负数”，意味着“x比y小”；返回“零”，意味着“x等于y”；返回“正数”，意味着“x大于y”。

Comparator 简介

Comparator 是比较器接口。

我们若需要控制某个类的次序，而该类本身不支持排序(即没有实现Comparable接口)；那么，我们可以建立一个“该类的比较器”来进行排序。这个“比较器”只需要实现Comparator接口即可。

也就是说，我们可以通过“实现Comparator类来新建一个比较器”，然后通过该比较器对类进行排序。

Comparator 定义

Comparator 接口仅仅只包括两个个函数，它的定义如下：

package java.util;

public interface Comparator<T> {

int compare(T o1, T o2);

boolean equals(Object obj);

}

说明：  
(01) 若一个类要实现Comparator接口：它一定要实现compareTo(T o1, T o2) 函数，但可以不实现 equals(Object obj) 函数。

        为什么可以不实现 equals(Object obj) 函数呢？ 因为任何类，默认都是已经实现了equals(Object obj)的。 Java中的一切类都是继承于java.lang.Object，在Object.java中实现了equals(Object obj)函数；所以，其它所有的类也相当于都实现了该函数。

(02) int compare(T o1, T o2) 是“比较o1和o2的大小”。返回“负数”，意味着“o1比o2小”；返回“零”，意味着“o1等于o2”；返回“正数”，意味着“o1大于o2”。

Comparator 和 Comparable 比较

Comparable是排序接口；若一个类实现了Comparable接口，就意味着“该类支持排序”。  
而Comparator是比较器；我们若需要控制某个类的次序，可以建立一个“该类的比较器”来进行排序。

我们不难发现：Comparable相当于“内部比较器”，而Comparator相当于“外部比较器”。