**写时复制的List和Set**

写时复制(CopyOnWrite)的实现原理就是写时复制，它是线程安全的，支持并发访问，支持同时读写，它的迭代器不支持修改操作，也不会抛出ConcurrentModifationException，以原子方式实现一些复合操作。

1 CopyOnWriteArrayList

支持两个原子方法:

//不存在才添加，如果添加了，返回true，否则返回false，达到了去重效果

public boolean addIfAbsent(E e);

//同上，返回实际添加的元素个数

public int addAllAbsend(Collection<? extends E> c);

CopyOnWriteArrayList的实现原理很简单，内部使用ReentrantLock维护一个数组，读数据时直接通过下标返回数组元素，修改时通过ReentrantLock进行保护，然后通过原数组复制一个新数组，在新数组上进行修改，修改完后，再将老数组指向新数组。这样的好处是:如果在修改的时候，有别的线程访问，内容还是正确的，因为修改的是新数组，此时别的线程访问的是老数组。说白了，通过写时复制，将**读取和修改分为两个目标**，这是一种伟大的思想，不像synchronized和CAS对同一个目标进行保护，而是直接拆分为两个目标，巧妙的避开了资源竞争。我们来简单看下add()和get()源码:

public boolean add(E e) {

//加锁

final ReentrantLock lock = this.lock;

lock.lock();

try {

Object[] elements = getArray();

int len = elements.length;

//复制内容到新数组，新数组长度比老数组大1，因为要添加元素

Object[] newElements = Arrays.copyOf(elements, len + 1);

//修改新数组

newElements[len] = e;

//将新数组内容设置给老数组

setArray(newElements);

return true;

} finally {

//解锁

lock.unlock();

}

}

可以看到，add()是基于CAS的写时复制，高并发的时候也是安全的，接下来看get():

public E get(int index) {

//返回数组对应下标的元素

return get(getArray(), index);

}

//返回当前数组

final Object[] getArray() {

return array;

}

//直接根据下标返回数组元素

private E get(Object[] a, int index) {

return (E) a[index];

}

可以看到，get()是不加锁的，因为它并没用修改数组，所以不存在安全问题，在一个for()循环内部，不断调用get()获取数据，可能每次得到的结果都不同，因为可能期间其他线程修改了数据，这也是完全合理的。

2 CopyOnWriteArraySet

这个实现比较简单，直接包装了CopyOnWriteArrayList实现，我们看下代码:

//构造函数，直接创建一个CopyOnWriteArrayList

public CopyOnWriteArraySet() {

al = new CopyOnWriteArrayList<E>();

}

//add()方法直接调用内部CopyOnWriteArrayList的addIfAbsent()，从而达到去重效果

public boolean add(E e) {

return al.addIfAbsent(e);

}

CopyOnWriteArrayList/Set使用了写时复制的思想，避免了资源竞争，但是复制本身效率较低，所以适用于读远大于写，读多写少，集合不大的场景。

**ConcurrentHashMap**

并发安全，以原子方式支持一些复合操作，读操作完全并行，写操作支持一定粒度的并行，具有弱一致性，且不会抛出ConcurrentModificationException。

以下这些操作都是原子的:

//功能等价于HashMap的put(key,value)，不过这个是原子的

V putIfAbsent(K key, V value);

//删除键值对，删除成功则返回ture，否则返回false

boolean remove(Object key, Object value);

//替换，成功返回true，否则返回false

boolean repalce(K key, V oldValue, V newValue)

//替换，返回原来key对应的value，如果原来没有则返回null

V replace(K key, V value)

ConcurrentHashMap的使用跟HashMap基本一样，但是它是高并发的，采用了**分段锁**技术，也就是将数据分割为多个段，每段有一个独立的锁，每段等价于一个独立的Hash表，分段的依据也是Hash值，默认情况是16个段，不过可以通过构造函数设置:

//第三个参数就表示支持并行更新的线程个数，也就是分段锁的个数，Concurrent会将它转换为2的n次幂。

public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrentLevel);

ConcurrentHashMap的读操作是完全并行的，不需要锁，写操作是支持"段个数并行的"，比如有4个分段锁，那就是支持4个线程同时并行写，比如现在4个段分别为ABCD，那么就对应LockA，LockB，LockC，LockD，四把锁，写A的时候，LockA生效，同一时刻写B的时候，LockB生效，互不干扰，可以并行写，这也是一种思想，跟CopyOnWriteArrayList相似的道理，将目标拆分，只不过CopyOnWriteArrayList是将读/写操作的目标拆分，ConcurrentHashMap是将写操作的目标拆分，但是思想是一样的，就是"目标拆分"，或者叫"目标细化"，这是一种NB的思想，值得学习。

ConcurrentHashMap的弱一致性只指的是: 在遍历的过程中，如果同时另一个线程也在修改，如果修改的部分发生在已遍历过的地方，那么迭代器就不会反映出来，如果发生在尚未遍历的部分，那么迭代器就能反映出来，这就是弱一致性，这是理所当然的。

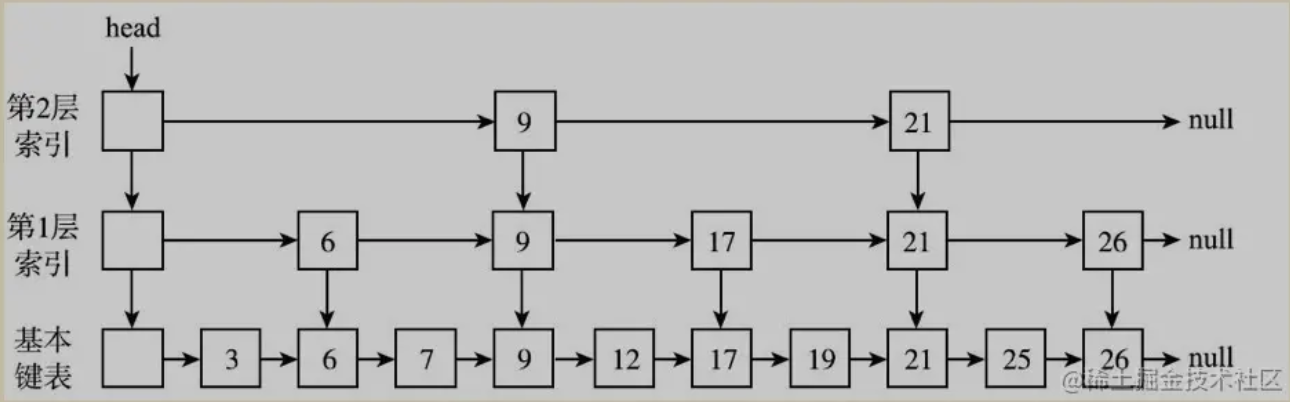
**跳表**

基于跳表的ConcurrentSkipListMap 和 ConcurrentSkipListSet，没有使用锁，所有操作都可以并行，不会抛出ConcurrentModificationException，具有弱一致性，有序，默认按自然排序。

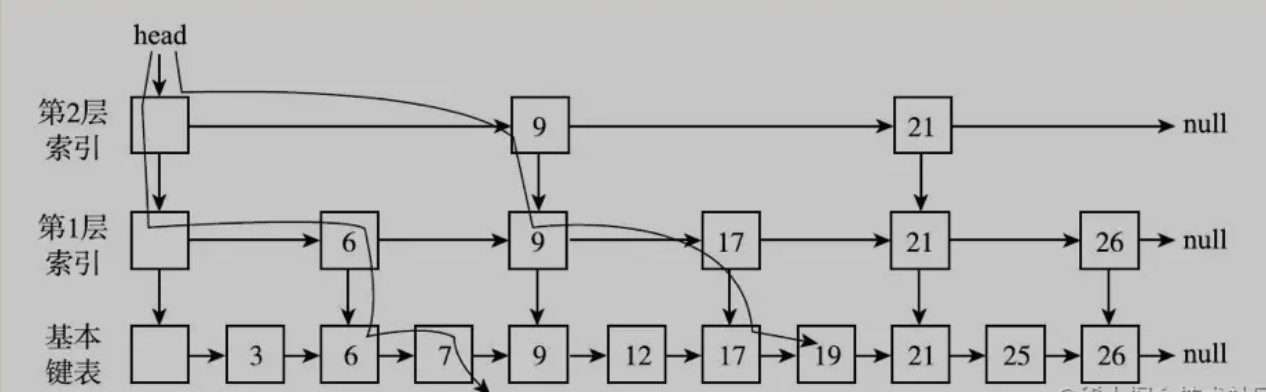
ConcurrentSkipListSet是包装了ConcurrentSkipListMap实现的，我们只需要了解后者即可。

ConcurrentSkipListMap的size()函数的时间复杂度是O(n)，它需要遍历所有元素，而且遍历后，元素个数可能已经发生了变化，这是因为它没有使用锁的缺点之一。

跳表是基于链表的，在链表的基础上添加了多层索引，有点像多叉树，准确点说，应该是一个有向图，比如一个集合: {3,6,7,9,12,17,19,21,25,26}的结构如下:



其中每一层都是有序的，高一层的是低一层的1/2，以此类推，右指针指向同一层的后一个节点，下指针指向下一层的相同节点，在这个结构上，就可以进行**二分查找**了。查找的过程很简单，从最上层开始比较，如果大于则右移，否则就下移，如下图的查找8和19的例子:



这个查找的时间复杂度是O(log(n))的，

非阻塞并发队列: ConcurrentLinkedQueue和ConcurrentLiknedDeque

都采用了CAS，都是基于链表实现的，所以是没有大小限制的，size()不是常量运算，需要遍历整个集合。其中Queue是一个单向链表，一端入队一端出队，Deque是一个双端队列，两端都可以入队出队。

普通阻塞队列: ArrayBlockingQueue(基于数组)和LinkedBlockingQueue/LinkedBolckingDeque(基于链表)

内部都采用了ReentrantLock和Condition，其中ArrayBlockingQueue是基于循环数组实现的，其他两个是基于链表实现的，也都没有大小限制。

优先级阻塞队列: PriorityBlockingQueue

用法类似于PriorityQueue，按照优先级入队出队，没有大小限制，也是使用了ReentrantLock和Condition实现。

延时阻塞队列: DelayQueue

基于PriorityQueue实现，没有大小限制，可以用于实现定时任务，按元素的延时时间出队。

其他阻塞队列: SynchronousQueue和LinkedTransferQueue

SynchronousQueue没有存储元素的空间，它的入队操作需要等待另一个线程的出队操作，出队也是一样的，否则put和take都会等待，适用于两个线程之间直接传递消息。线程池:Executors.newCachedThreadPool()采用的就是SynchronousQueue。  
LinkedTransferQueue是基于链表实现的，没有大小限制，适用于生产者消费者模型。