一、并发容器之写时拷贝

1. CopyOnWriteArrayList

(1) 基本介绍

CopyOnWriteArrayList实现了List接口，用法与其他List如ArrayList基本是一样的，区别是：

① 线程安全的，可以被多个线程并发访问。

② 迭代器不支持修改操作，但也不会抛出ConcurrentModificationException。

③ 以原子方式支持一些复合操作。

(2) 注意

① 在Java 1.8之前的实现中，CopyOnWriteArrayList的迭代器不支持修改操作，也不支持一些依赖迭代器修改方法的操作，比如Collections的sort方法。

② 基于synchronized的同步容器的另一个问题是复合操作，比如先检查再更新，也需要调用方加锁，而CopyOnWriteArrayList直接支持两个原子方法：

//不存在才添加，如果添加了，返回true，否则返回false

public boolean addIfAbsent(E e)

//批量添加c中的非重复元素，不存在才添加，返回实际添加的个数

public int addAllAbsent(Collection<? extends E> c)

1.1 基本原理

(1) 修改操作

CopyOnWriteArrayList的内部也是一个数组，但这个数组是以原子方式被整体更新的。每次修改操作，都会新建一个数组，复制原数组的内容到新数组，在新数组上进行需要的修改，然后以原子方式设置内部的数组引用，这就是写时拷贝。

(2) 读操作

所有的读操作，都是先拿到当前引用的数组，然后直接访问该数组，在读的过程中，可能内部的数组引用已经被修改了，但不会影响读操作，依旧访问原数组内容。换句话说，数组内容是只读的，写操作都是通过新建数组，然后原子性的修改数组引用来实现的。

1.2 总结

(1) 可以同时执行读操作（没加锁），不能同时执行写操作（加锁）。

(2) CopyOnWriteArrayList不适用于数组很大，且修改频繁的场景。它是以优化读操作为目标的，读不需要同步，性能很高，但在优化读的同时就牺牲了写的性能。

(3) 保证线程安全的三种思路：

① 加锁，使用synchronized或ReentrantLock。

② 一种是循环CAS。

③ 写时加锁拷贝，读时不加锁，优化读的性能。

2. CopyOnWriteArraySet

(1) 基本介绍

CopyOnWriteArraySet实现了Set接口，不包含重复元素，使用比较简单，我们就不赘述了。内部，它是通过CopyOnWriteArrayList实现的。

(2) 特点

① 由于CopyOnWriteArraySet是基于CopyOnWriteArrayList实现的，所以与之前介绍过的Set的实现类如HashSet/TreeSet相比，它的性能比较低，不适用于元素个数特别多的集合。如果元素个数比较多，可以考虑ConcurrentHashMap或ConcurrentSkipListSet。

② ConcurrentHashMap与HashMap类似，适用于不要求排序的场景，ConcurrentSkipListSet与TreeSet类似，适用于要求排序的场景。Java并发包中没有与HashSet对应的并发容器，但可以很容易的基于ConcurrentHashMap构建一个，利用Collections.newSetFromMap方法即可。

二、并发容器之ConcurrentHashMap

1. 基本介绍

(1) ConcurrentHashMap是HashMap的并发版本，与HashMap相比，有如下特点：

① 并发安全

② 直接支持一些原子复合操作

③ 支持高并发、读操作完全并行、写操作支持一定程度的并行

④ 与同步容器Collections.synchronizedMap相比，迭代不用加锁，不会抛出

ConcurrentModificationException

⑤ 弱一致性

(2) 类定义 – 实现了ConcurrentMap接口

public class ConcurrentHashMap<K,V> extends AbstractMap<K,V>

implements ConcurrentMap<K,V>, Serializable

2. 并发安全

2.1 HashMap的并发问题

在并发更新的情况下，HashMap的链表结构可能形成环，出现死循环，占满CPU。

(1) 原因：

死循环出现在多个线程同时扩容哈希表的时候，不是同时更新一个链表的时候，那种情况可能会出现更新丢失，但不会死循环。

2.2 并发容器

(1) 使用Collections.synchronizedMap方法可以生成一个同步容器（HashMap自带）：

final Map<Integer, Integer> map =

Collections.synchronizedMap(new HashMap<Integer, Integer>());

注：HashMap还有一个同步版本Hashtable，它与使用synchronizedMap生成的Map基本是一样的，也是在每个方法调用上加了synchronized

(2) 同步容器的问题：

① 每个方法都需要同步，支持的并发度比较低

② 对于迭代和复合操作，需要调用方加锁，使用比较麻烦，且容易忘记。

2.3 并发容器类 - ConcurrentHashMap

(1) ConcurrentHashMap没有这些问题，它同样实现了Map接口，也是基于哈希表实现的：

final Map<Integer, Integer> map = new ConcurrentHashMap<>();

3. 原子复合操作 – 由接口ConcurrentMap定义

public interface ConcurrentMap<K, V> extends Map<K, V> {

//条件更新，如果Map中没有key，设置key为value，返回原来key对应的值，如果没有，返回null

V putIfAbsent(K key, V value);

//条件删除，如果Map中有key，且对应的值为value，则删除，如果删除了，返回true，否则false

boolean remove(Object key, Object value);

//条件替换，如果Map中有key，且对应的值为oldValue，则替换为newValue，如果替换了，返回ture，否则false

boolean replace(K key, V oldValue, V newValue);

//条件替换，如果Map中有key，则替换值为value，返回原来key对应的值，如果原来没有，返回null

V replace(K key, V value);

}

(1) 使用ConcurrentMap，调用方也没有办法进行加锁，它没有暴露锁接口，也不使用synchronized。相当于封装了原子操作方法。

4. 高并发

4.1 基本介绍

ConcurrentHashMap是为高并发设计的，主要通过两点实现：分段锁，读不需要锁。

(1) 分段锁

同步容器使用synchronized，所有方法，竞争同一个锁，而ConcurrentHashMap采用分段锁技术，将数据分为多个段，而每个段有一个独立的锁，每一个段相当于一个独立的哈希表，分段的依据也是哈希值，无论是保存键值对还是根据键查找，都先根据键的哈希值映射到段，再在段对应的哈希表上进行操作。

① 采用分段锁，可以大大提高并发度，多个段之间可以并行读写。默认情况下，段是16个，不过，这个数字可以通过构造方法进行设置，如下所示：

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

在对每个段的数据进行读写时，ConcurrentHashMap也不是简单的使用锁进行同步，内部使用了CAS、对一些写采用原子方式。

(2) 优点：

对于写操作，需要获取锁，不能并行，但是读操作可以，多个读可以并行，写的同时也可以读。

5. 迭代

使用同步容器，在迭代中需要加锁，否则可能会抛出ConcurrentModificationException。ConcurrentHashMap没有这个问题，在迭代器创建后，在迭代过程中，如果另一个线程对容器进行了修改，迭代会继续，不会抛出异常。

6. 弱一致性

(1) 基本介绍

ConcurrentHashMap的迭代器创建后，就会按照哈希表结构遍历每个元素，但在遍历过程中，内部元素可能会发生变化，如果变化发生在已遍历过的部分，迭代器就不会反映出来，而如果变化发生在未遍历过的部分，迭代器就会发现并反映出来，这就是弱一致性。

(2) 强一致性定义

强一致性可以理解为在任意时刻，所有节点中的数据是一样的。同一时间点，你在节点A中获取到key1的值与在节点B中获取到key1的值应该都是一样的。

弱一致性可以相应的理解为，当数据发生改变时，在两个时间点或两个地方获取的值是不一定是一样的。

7. 小结

(1) Java中没有并发版的HashSet，但可以通过Collections.newSetFromMap方法基于ConcurrentHashMap构建一个。（同步容器）

(2) HashMap/HashSet基于哈希，不能对元素排序，对应的可排序的容器类是TreeMap/TreeSet，并发包中可排序的对应版本不是基于树，而是基于Skip List（跳跃表）的，类分别是ConcurrentSkipListMap和ConcurrentSkipListSet。

三、并发容器之基于SkipList的Map和Set

1. 基本介绍

1.1 概念

ConcurrentSkipListMap是基于SkipList实现的，SkipList称为跳跃表或跳表，是一种数据结构，跳表更易于实现高效并发算法。、

1.2 特点

(1) 没有使用锁，所有操作都是无阻塞的，所有操作都可以并行，包括写，多个线程可以同时写。

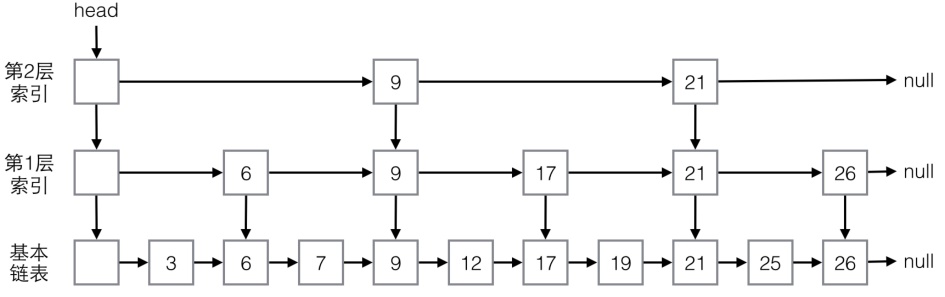
(2) 与ConcurrentHashMap类似，迭代器不会抛出ConcurrentModificationException，是弱一致的，迭代可能反映最新修改也可能不反映，一些方法如putAll, clear不是原子的。

(3) 与ConcurrentHashMap类似，同样实现了ConcurrentMap接口，直接支持一些原子复合操作。

(4) 与TreeMap一样，可排序，默认按键自然有序，可以传递比较器自定义排序，实现了SortedMap和NavigableMap接口。

2. 基本实现原理 – 跳表

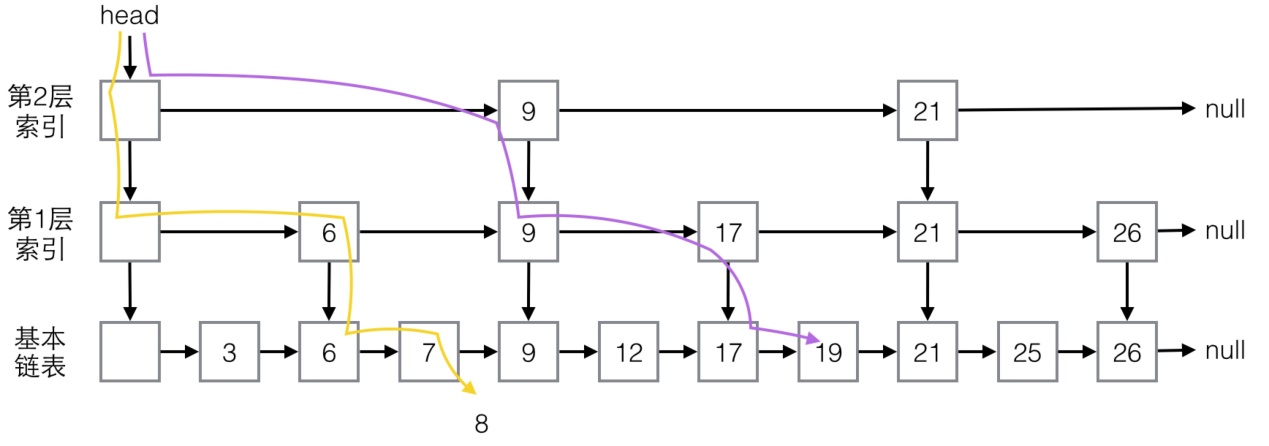
2.1 跳表结构



(1) 为了快速查找，跳表有多层索引结构，这个例子中有两层，第一层有5个节点，第二层有2个节点。高层的索引节点一定同时是低层的索引节点，比如9和21。

(2) 对于每个索引节点，有两个指针，一个向右，指向下一个同层的索引节点，另一个向下，指向下一层的索引节点或基本链表节点。有了这个结构，就可以实现类似二分查找了，查找元素总是从最高层开始，将待查值与下一个索引节点的值进行比较，如果大于索引节点，就向右移动，继续比较，如果小于，则向下移动到下一层进行比较。

2.2 查找过程



(1) 对于19，查找过程是：

与9相比，大于9；

向右与21相比，小于21；

向下与17相比，大于17；

向右与21相比，小于21；

向下与19相比，找到。

(2) 对于8，查找过程是：

与9相比，小于9

向下与6相比，大于6

向右与9相比，小于9

向下与7相比，大于7

向右与9相比，小于9，不能再向下，没找到

(3) 这个结构是有序的，查找的性能与二叉树类似，复杂度是O(log(N))。

2.3 跳表结构的建立

(1) 跳表结构是在更新过程中进行保持的，保存元素的基本思路是：

① 先保存到基本链表，找到待插入的位置，找到位置后，先插入基本链表

② 更新索引层。

(2) 对于索引更新，随机计算一个数，表示为该元素最高建几层索引，一层的概率为1/2，二层为1/4，三层为1/8，依次类推。然后从最高层到最低层，在每一层，为该元素建立索引节点，建的过程也是先查找位置，再插入。

(3) 对于删除元素，ConcurrentSkipListMap不是一下子真的进行删除，为了避免并发冲突，有一个复杂的标记过程，在内部遍历元素的过程中会真正删除。

四、并发容器之各种队列

1. 基本介绍（队列类）

无锁非阻塞并发队列：ConcurrentLinkedQueue和ConcurrentLinkedDeque

普通阻塞队列：基于数组的ArrayBlockingQueue，基于链表的LinkedBlockingQueue和

LinkedBlockingDeque

优先级阻塞队列：PriorityBlockingQueue

延时阻塞队列：DelayQueue

其他阻塞队列：SynchronousQueue和LinkedTransferQueue

(1) 无锁非阻塞是这些队列不使用锁，所有操作总是可以立即执行，主要通过循环CAS实现并发安全，阻塞队列是指这些队列使用锁和条件，很多操作都需要先获取锁或满足特定条件，获取不到锁或等待条件时，会等待(即阻塞)，获取到锁或条件满足再返回。

(2) 队列迭代都不会抛出ConcurrentModificationException，都是弱一致的。

2. 无锁非阻塞并发队列

(1) 两个无锁非阻塞队列：ConcurrentLinkedQueue和ConcurrentLinkedDeque，它们适用于多个线程并发使用一个队列的场合，都是基于链表实现的，都没有限制大小，是无界的，与ConcurrentSkipListMap类似，它们的size方法不是一个常量运算，不过这个方法在并发应用中用处也不大。

(2) ConcurrentLinkedQueue实现了Queue接口，表示一个先进先出的队列，从尾部入队，从头部出队，内部是一个单向链表。ConcurrentLinkedDeque实现了Deque接口，表示一个双端队列，在两端都可以入队和出队，内部是一个双向链表。

(3) 基本原理是通过CAS实现的。

3. 普通阻塞队列

(1) 主要方法

//入队，如果队列满，等待直到队列有空间

void put(E e) throws InterruptedException;

//出队，如果队列空，等待直到队列不为空，返回头部元素

E take() throws InterruptedException;

//入队，如果队列满，最多等待指定的时间，如果超时还是满，返回false

boolean offer(E e, long timeout, TimeUnit unit) throws InterruptedException;

//出队，如果队列空，最多等待指定的时间，如果超时还是空，返回null

E poll(long timeout, TimeUnit unit) throws InterruptedException;

3.1 特点

(1) 普通阻塞队列是常用的队列，常用于生产者/消费者模式。ArrayBlockingQueue和LinkedBlockingQueue都是实现了Queue接口，表示先进先出的队列，尾部进，头部出，而LinkedBlockingDeque实现了Deque接口，是一个双端队列。

(2) ArrayBlockingQueue是基于循环数组实现的，有界，创建时需要指定大小，且在运行过程中不会改变，这与我们在容器类中介绍的ArrayDeque是不同的，ArrayDeque也是基于循环数组实现的，但是是无界的，会自动扩展。

(3) LinkedBlockingQueue是基于单向链表实现的，在创建时可以指定最大长度，也可以不指定，默认是无限的，节点都是动态创建的。

(4) LinkedBlockingDeque与LinkedBlockingQueue一样，最大长度也是在创建时可选的，默认无限，不过，它是基于双向链表实现的。

(5) 内部，它们都是使用显式锁ReentrantLock和显式条件Condition实现的。

4. 优先级阻塞队列 -- PriorityBlockingQueue

(1) 普通阻塞队列是先进先出的，而优先级队列是按优先级出队的，优先级高的先出，基本数据结构是堆。

(2) PriorityBlockingQueue是PriorityQueue的并发版本，与PriorityQueue一样，它没有大小限制，是无界的，内部的数组大小会动态扩展，要求元素要么实现Comparable接口，要么创建PriorityBlockingQueue时提供一个Comparator对象。

(3) PriorityBlockingQueue实现了BlockingQueue接口，在队列为空时，take方法会阻塞等待。

另外，PriorityBlockingQueue是线程安全的，它的基本实现原理与PriorityQueue是一样的，也是基于堆，但它使用了一个锁ReentrantLock保护所有访问，使用了一个条件协调阻塞等待。

5. 延时阻塞队列（可以用来执行定时任务）

(1) 延时阻塞队列DelayQueue是一种特殊的优先级队列，它也是无界的，它要求每个元素都实现Delayed接口，该接口的声明为：

public interface Delayed extends Comparable<Delayed> {

long getDelay(TimeUnit unit);

}

(2) DelayQueue也是优先级队列，它按元素的延时时间出队，它的特殊之处在于，只有当元素的延时过期之后才能被从队列中拿走，也就是说，take方法总是返回第一个过期的元素，如果没有，则阻塞等待。

(3) DelayedTask表示延时任务，只有延时过期后任务才会执行，任务按延时时间排序，延时一样的按照入队顺序排序。内部，DelayQueue是基于PriorityQueue实现的，它使用一个锁ReentrantLock保护所有访问，使用一个条件available表示头部是否有元素，当头部元素的延时未到时，take操作会根据延时计算需睡眠的时间，然后睡眠，如果在此过程中有新的元素入队，且成为头部元素，则阻塞睡眠的线程会被提前唤醒然后重新检查。

6. 其他阻塞队列

6.1 SynchronousQueue

SynchronousQueue与一般的队列不同，它不算一种真正的队列，它没有存储元素的空间，存储一个元素的空间都没有。它的入队操作要等待另一个线程的出队操作，反之亦然。如果没有其他线程在等待从队列中接收元素，put操作就会等待。take操作需要等待其他线程往队列中放元素，如果没有，也会等待。SynchronousQueue适用于两个线程之间直接传递信息、事件或任务。

6.2 LinkedTransferQueue

LinkedTransferQueue实现了TransferQueue接口，TransferQueue是BlockingQueue的子接口，但增加了一些额外功能，生产者在往队列中放元素时，可以等待消费者接收后再返回，适用于一些消息传递类型的应用中。LinkedTransferQueue是基于链表实现的、无界的TransferQueue。