Atomic*

CAS 算法：

CAS 的全称是 Compare And Swap 即比较交换，其算法核心思想如下

函数：CAS(V,E,N) 参数：V 表示要更新的变量 E 预期值 N 新值

如果 V 值等于 E 值，则将 V 的值设为 N。若 V 值和 E 值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。通俗的理解就是 CAS 操作需要我们提供一个期望值，当期望值与当前线程的变量值相同时，说明还没线程修改该值，当前线程可以进行修改，也就是执行 CAS 操作，但如果期望值与当前线程不符，则说明该值已被其他线程修改，此时不执行更新操作，但可以选择重新读取该变量再尝试再次修改该变量，也可以放弃操作

Java 的 CAS 操作通过 Unsafe 类来完成里面基本都是 native，即通过 JNI 调用 c/c++等代码

基本类型：

AtomicBoolean：原子更新布尔类型。

AtomicInteger：原子更新整型。

AtomicLong：原子更新长整型。

数组类：

- AtomicIntegerArray：原子更新整型数组里的元素。
- AtomicLongArray：原子更新长整型数组里的元素。
- AtomicReferenceArray：原子更新引用类型数组里的元素。

引用类型：

AtomicReference：原子更新引用类型。

AtomicReferenceFieldUpdater：原子更新引用类型里的字段。

AtomicMarkableReference：原子更新带有标记位的引用类型。可以原子的更新一个布尔类型的标记位和引用类型。构造方法是 AtomicMarkableReference(V initialRef, boolean initialMark)

更新字段：

AtomicIntegerFieldUpdater：原子更新整型的字段的更新器。

AtomicLongFieldUpdater：原子更新长整型字段的更新器。

AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于原子的更数据和数据的版本号，可以解决使用 CAS 进行原子更新时，可能出现的 ABA 问题。

CAS 优缺点

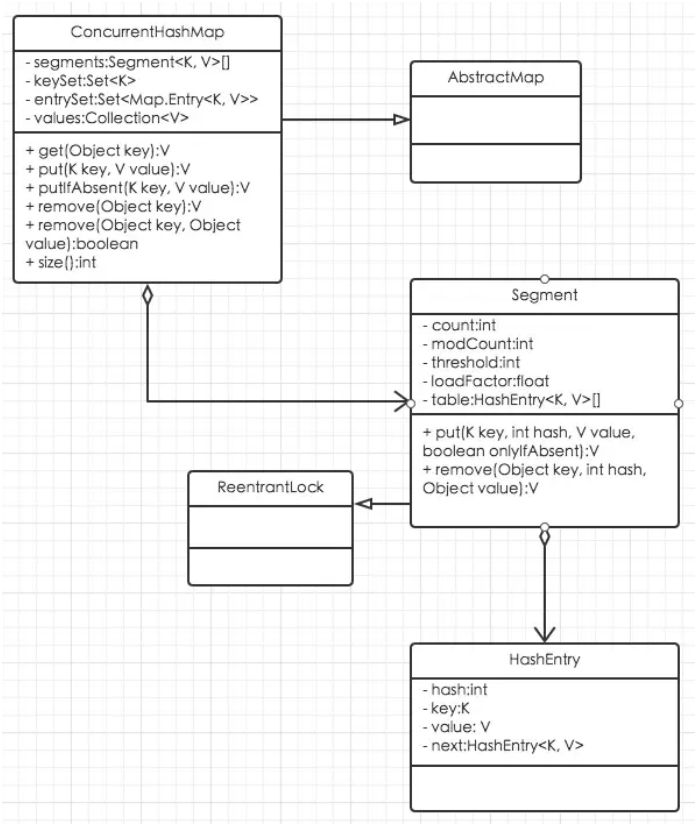
非阻塞算法 、ABA 问题、循环开销大、只保证一个共享变量原子操作

ConcurrentHashMap

线程安全的 map 有 Hashtable 和 SynchronizedMap

以及 concurrentHashMap

ConcurrentHashMap 所使用的锁分段技术通过细化锁的粒度来降低锁的竞争。



不足：算 size 的结果需要遍历

CopyOnWrite

介绍：Copy-On-Write 简称 COW，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 容器介绍：

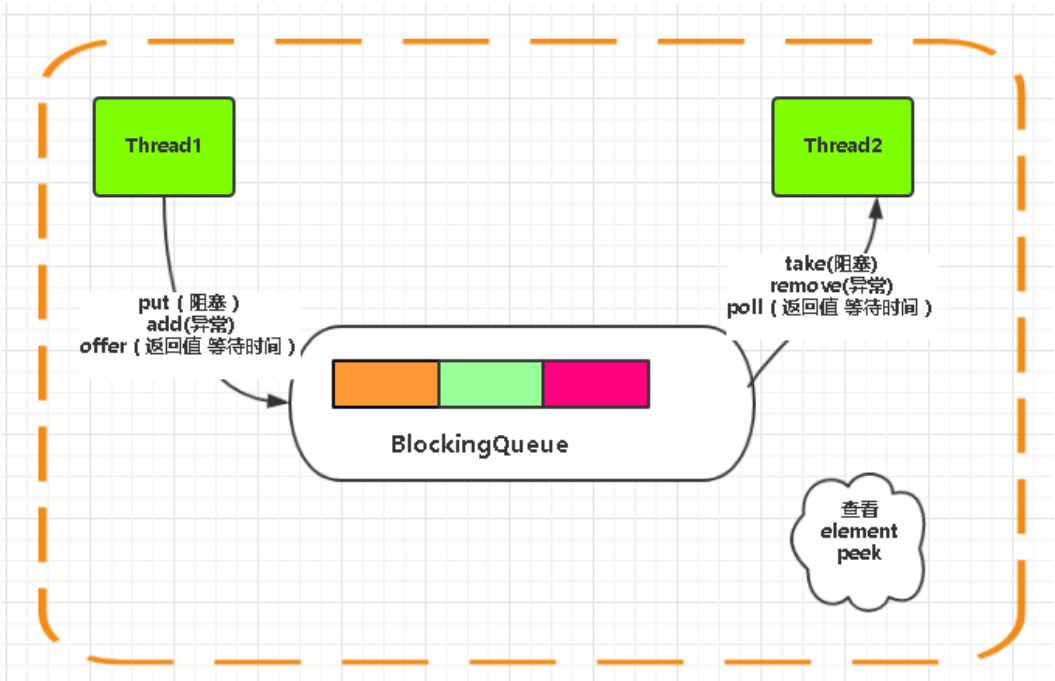
CopyOnWrite 容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行 Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我

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 容器也是一种读写分离的思想，读和写不同的容器。

场景：黑白名单,读多写少(很少改动)场景 商品 sku 商品类目

优缺点：占内存（写时复制 new 两个对象）、不能保证数据实时一致性.

BlockingQueue



BlockingQueue 5 种实现:

ArrayBlockingQueue：基于数组实现的有界阻塞队列，创建后不能修改队列的大小；是一个有边界的阻塞队列，它的内部实现是一个数组。有边界的意思是它的容量是有限的，我们必须在其初始化的时候指定它的容量大小，容量大小一旦指定就不可改变。

LinkedBlockingQueue：基于链表实现的无界(可以指定)阻塞队列，默认大小为 Integer.MAX_VALUE，有较好的吞吐量，但可预测性差。

PriorityBlockingQueue：具有优先级的无界阻塞队列，不允许插入 null，所有元素都必须可比较（即实现 Comparable 接口）。顺序：非先进先出

SynchronousQueue：只有一个元素的同步队列。若队列中有元素插入操作将被阻塞，直到队列中的元素被其他线程取走。

DelayQueue：无界阻塞队列，每个元素都有一个延迟时间，在延迟时间之后才释放元素。阻塞的是其内部元素，DelayQueue 中的元素必须实现 java.util.concurrent.Delayed 接口，这个接口的定义非常简单

ConcurrentLinkedQueue:基于链表实现的非阻塞队列

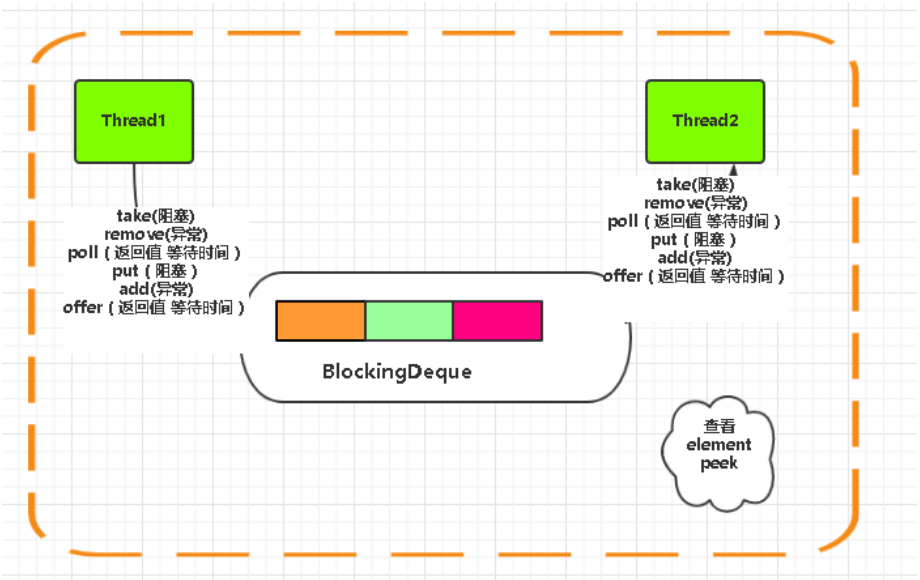
使用：

BlockingQueue 的队列长度受限，用以保证生产者与消费者的速度不会相差太远，避免内存耗尽。队列长度设定后不可改变。当入队时队列已满，或出队时队列已空，不同函数的效果见下表：

	可能报异常	返回布尔值	可能阻塞等待	可设定等待时间
入队	add(e)	offer(e)	put(e)	offer(e, timeout, unit)
出队	remove()	poll()	take()	poll(timeout, unit)
查看	element()	peek()	无	无

典型：生产与消费

BlockingDeque



一个线程生产元素并将元素插入到队列的两端。如果当前队列是满的，插入线程将会被阻塞直到一个移除元素的线程从队列中取出一个元素。同样，如果队列当前是空的，移除元素的线程会被阻塞直到一个插入元素的线程向队列中插入了一个元素。

ThreadLocal 线程本地变量

Java 中的 ThreadLocal 类允许我们创建只能被同一个线程读写的变量。因此，如果一段代码含有一个 ThreadLocal 变量的引用，即使两个线程同时执行这段代码，它们也无法访问到对方的 ThreadLocal 变量。