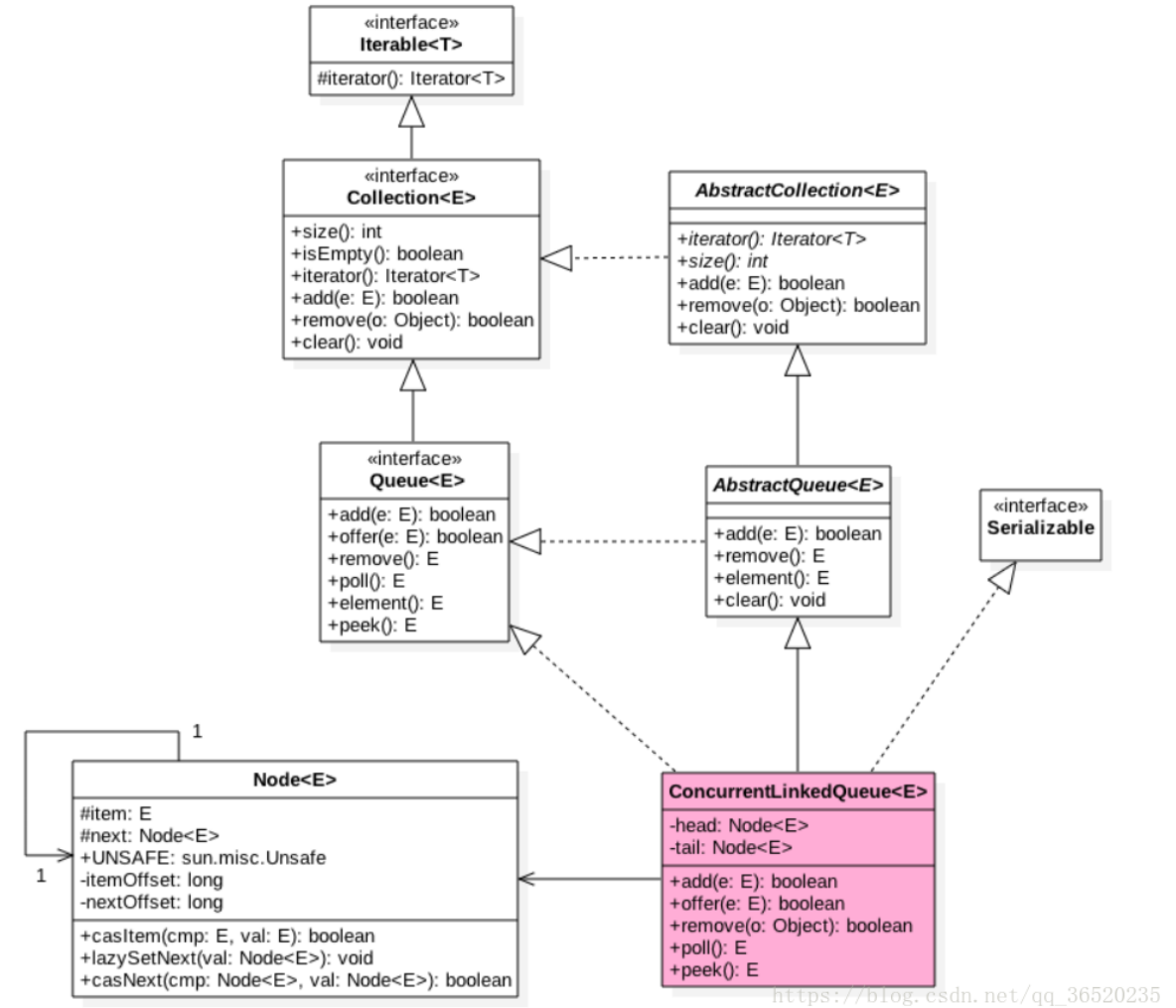
ConcurrenLinkedQueue:

1.concurrentlinkedqueue的简单了解和用途

> 继承的是AbstractQueue接口，实现的是Queue接口。
>
> 使用的是单向链表的形式来实现队列

```
 * @since 1.5
 * @author Doug Lea
 * @param <E> the type of elements held in this collection
 */
public class ConcurrentLinkedQueue<E> extends AbstractQueue<E>
        implements Queue<E>, java.io.Serializable {
    private static final long serialVersionUID = 196745693267521676L;
```



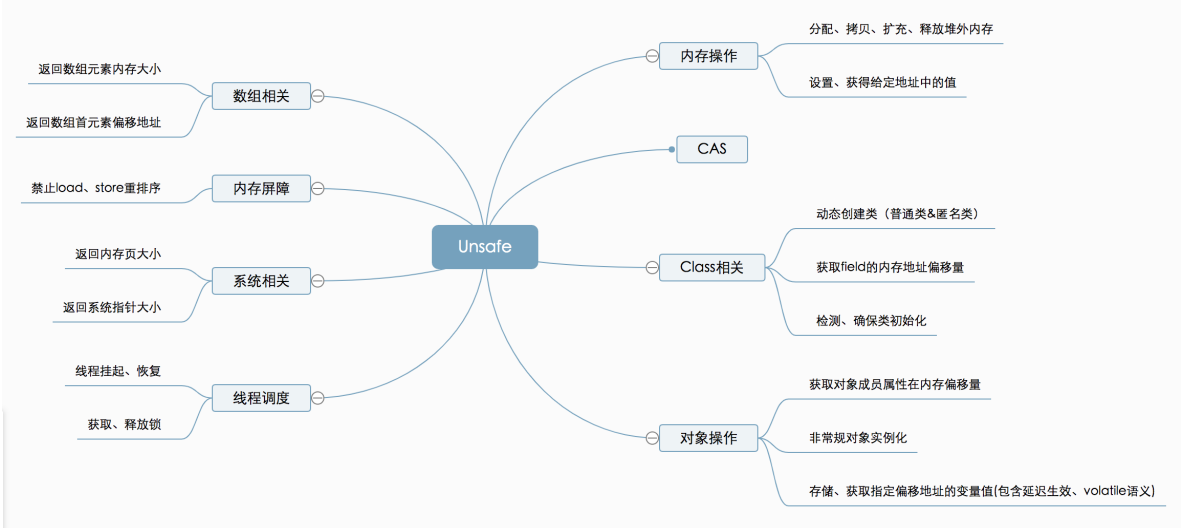
2.ConcurrentLinkedQueue的部分源码解读

> 定义一个Node方法体，使用volatile对元素item和next域进行修饰，这样就可以修改到主内存，全部线程可见。

```java
private static class Node<E> {
    volatile E item;
    volatile Node<E> next;

    /**
     * Constructs a new node.  Uses relaxed write because item can
     * only be seen after publication via casNext.
     */
    Node(E item) { UNSAFE.putObject( o: this, itemOffset, item); }

    boolean casItem(E cmp, E val) { return UNSAFE.compareAndSwapObject( o: this, itemOffset, cmp, val); }

    void lazySetNext(Node<E> val) { UNSAFE.putOrderedObject( o: this, nextOffset, val); }

    boolean casNext(Node<E> cmp, Node<E> val) { return UNSAFE.compareAndSwapObject( o: this, nextOffset, cmp, val); }

    // Unsafe mechanics

    private static final sun.misc.Unsafe UNSAFE;
    private static final long itemOffset;
    private static final long nextOffset;

    static {
        try {
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            Class<?> k = Node.class;
            itemOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField( name: "item"));
            nextOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField( name: "next"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
}
```

从上图可以看到，在Node构造函数中，使用了unsafe类，分别设置、比较并替换了item和next域的值。但是对next域使用了**unsafe.putOrderedObject**方法，实现了非堵塞的写入，并且写入不会被指令重排序，能够实现快速的**存储-存储**屏障，而不是较慢的**存储-加载**屏障。虽然导致对next域的修改并不会对其他线程立即可见(但只会在纳秒级别)。使用了unsafe类的cas算法保证了出入队列的一致性。CAS其实是一条CPU原子指令，其作用是让CPU先进行比较两个值是否相等，然后原子的更新某个位置的值，**即cas是基于硬件平台的，JVM封装了汇编调用，AtomicInteger类则使用了这些封装后的接口(atomic底层提供的是volatile和cas来实现的对数据的更改**，volatile实现了数据修改的主内存可见，禁止重排序，cas比较并替换实现了数据更新的原子性)。

> 而java的原子类是通过Unsafe类实现的，所以简单了解一下unsafe类。主要提供的是执行级别较低、且不安全操作的方法(java没有为我们提供unsafe类的对外API),例如：访问和管理系统内存资源。



> ConcurrentLinkedQueue中头尾节点的定义：concurrentlinkedqueue持有head和tail头尾指针管理队列，用来存放队首和队尾的节点信息。

```
private transient volatile Node<E> head;
private transient volatile Node<E> tail;
```

如此，我们便可以在O(l)的时间内获取到头结点和尾节点的信息。head和tail除了使用**volatile**进行原子修饰以外，还使用了**transient**进行修饰。transient的作用主要是使其修饰的变量可以不被序列化(transient只能修饰变量，同时其所在的类需要继承serializable)。

> 类的构造函数，对head和tail进行初始化，指向一个空的域。

```
/**
 * Creates a {@code ConcurrentLinkedQueue} that is initially empty.
 */
@Contract(pure = true)
public ConcurrentLinkedQueue() {
    head = tail = new Node<E>( item: null);
}
```

> 包含给定集合的构造函数，用于创建一个包含该集合给定元素的concurrentLinkedqueue

```
public ConcurrentLinkedQueue(Collection<? extends E> c) {
    Node<E> h = null, t = null;
    for (E e : c) {
        checkNotNull(e);
        Node<E> newNode = new Node<E>(e);
        if (h == null)
            h = t = newNode;
        else {
            t.lazySetNext(newNode);
            t = newNode;
        }
    }
    if (h == null)
        h = t = new Node<E>( item: null);
    head = h;
    tail = t;
```

如上图，使用h，t分别表示临时的头尾节点，遍历集合元素，每次都会使用checkNotNull方法判断元素是否存在，为空就会抛出空指针异常。每次取出的对象会包装成一个新的node节点，如果ht都是指向空域的话，就会同时赋值，否则，使用lazysetnext，延迟设置，将t永远指向最新加入的一个节点。再之后，将head和tail分别指向h和t。

> 其中，lasySetNext这里直接调用了UNSAFE对象的putorderedObject方法，三个参数分别是当前node，偏移量，下一个node对象。**实现了低延迟代码的非阻塞写入。**

```
void lazySetNext(Node<E> val) {
    UNSAFE.putOrderedObject(this, nextOffset, val);
}
```

和另一个putObjectVolatile相比，少了内存屏障，提升了性能，相当于其的内存非立即可见版本。（此处打上？，对于lazySetNext的具体原理还是不清楚）

2.concurrentlinkedqueue的一些用法

> add方法，其本质还是调用队列的offer方法

```
public boolean add(E e) {
    return offer(e);
}
```

> **offfer方法是核心方法，由于队列是无边界的，所以不会返回false**

```
public boolean offer(E e) {
    //检查元素不为空
    checkNotNull(e);
    //为该元素新生一个节点
    final Node<E> newNode = new Node<E>(e);

    for (Node<E> t = tail, p = t;;) {//无限循环
        Node<E> q = p.next;//q作为p的下一个节点
        if (q == null) { //q为空，说明当前p节点为尾节点
            // p is last node
            if (p.casNext(null, newNode)) {//比较并替换p的next域为新的节点
                // Successful CAS is the linearization point
                // for e to become an element of this queue,
                // and for newNode to become "live".
                if (p != t) // hop two nodes at a time 如果p与t的值不等，不一致
                    casTail(t, newNode);  // Failure is OK. 比较并替换尾节点
                return true;
            }
            // Lost CAS race to another thread; re-read next
        }
        else if (p == q) //如果p和q节点相等
            // We have fallen off list.  If tail is unchanged, it
            // will also be off-list, in which case we need to
            // jump to head, from which all live nodes are always
            // reachable.  Else the new tail is a better bet.
            //原来的尾结点与现在的尾结点是否相等，若相等，则p赋值为head，否则，赋值为现在的尾结点
            p = (t != (t = tail)) ? t : head;
        else
            // Check for tail updates after two hops.
            //重新赋值p节点
            p = (p != t && t != (t = tail)) ? t : q;
    }
}
```