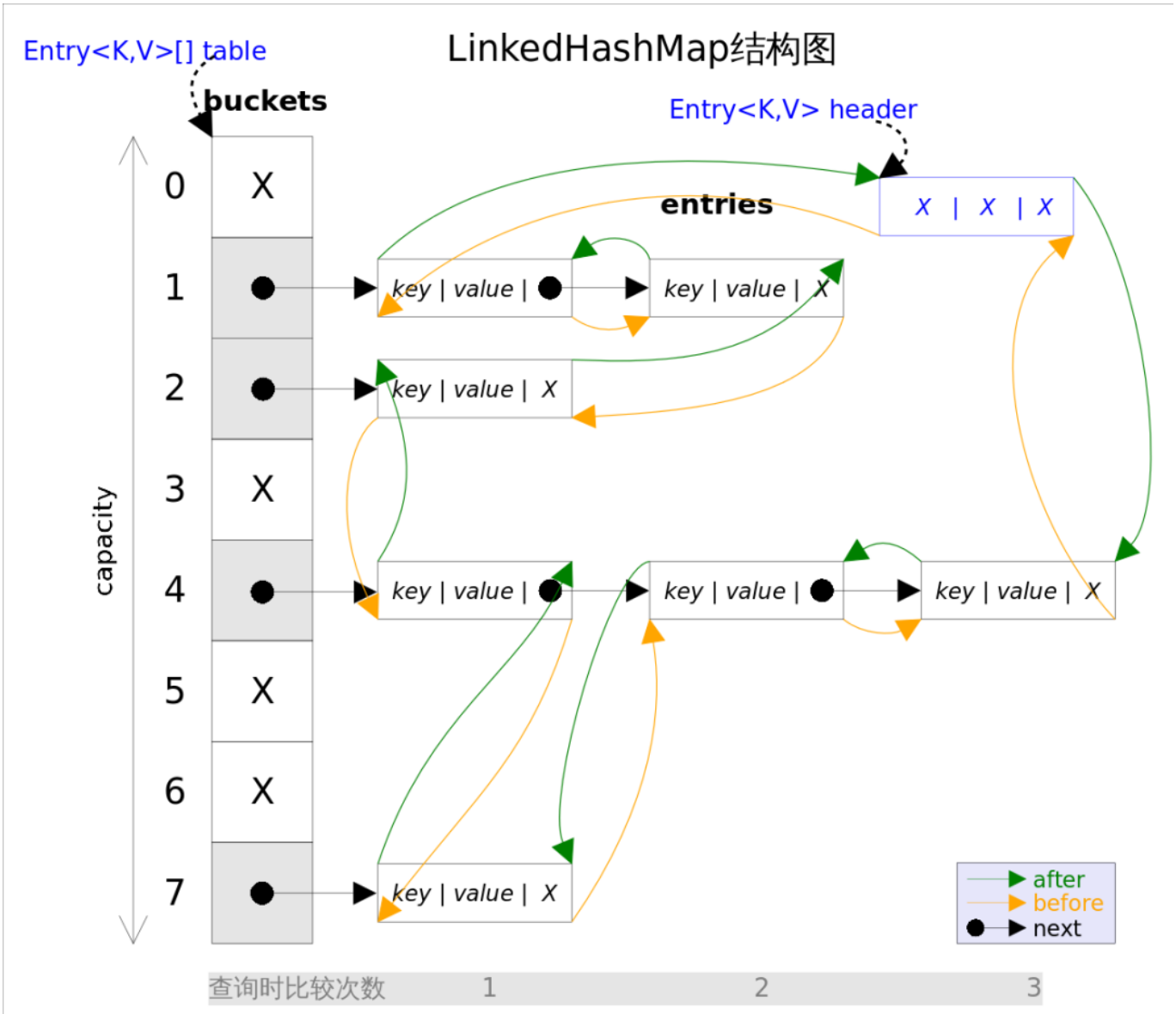
Object

{registerNatives()}

1. hash
2. equal
3. tostring
4. wait
5. notify
6. notifyall
7. finalize

LinkedHashMap

```
hashmap + 双向链表
保持顺序的核心方法 afterNodeAccess
```

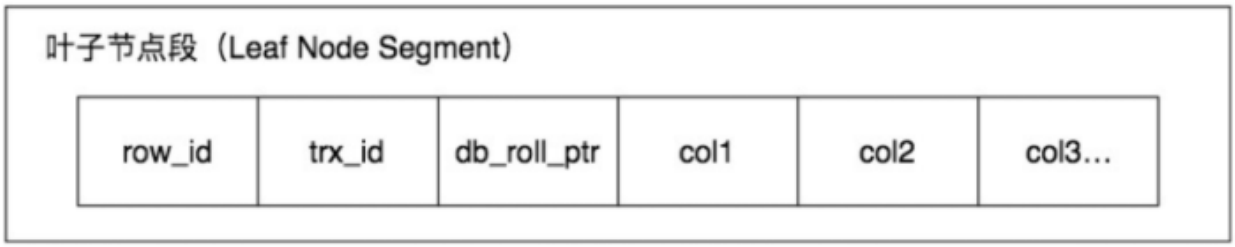


MVCC 多版本并发控制协议

```
1. 行记录的隐藏列
    row_id, trx_id, roll_ptr
2. Read View
```

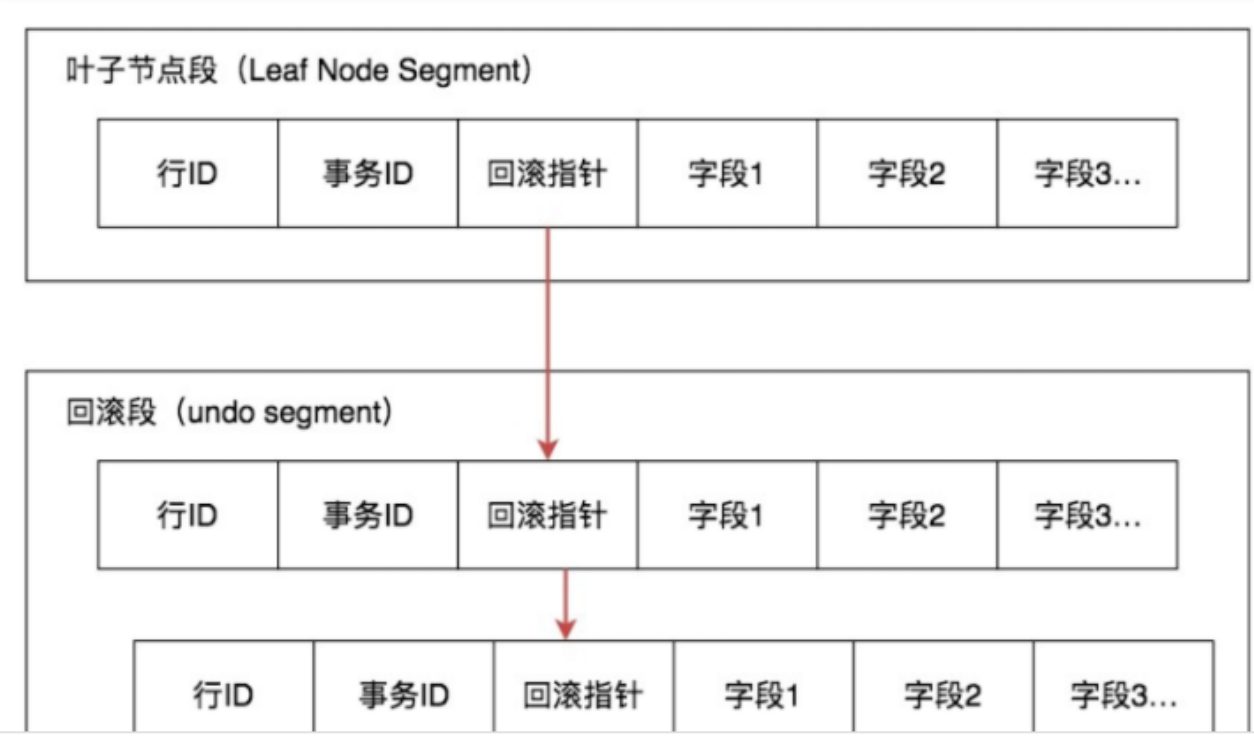
行记录的隐藏列

1. row_id :隐藏的行 ID ,用来生成默认的聚集索引。如果创建数据表时没指定聚集索引，这时 InnoDB 就会用这个隐藏 ID 来创建聚集索引。采用聚集索引的方式可以提升数据的查找效率。
2. trx_id: 操作这个数据事务 ID ，也就是最后一个对数据插入或者更新的事务 ID 。
3. roll_ptr:回滚指针，指向这个记录的 Undo Log 信息。



Undo Log

InnoDB 将行记录快照保存在 Undo Log 里。

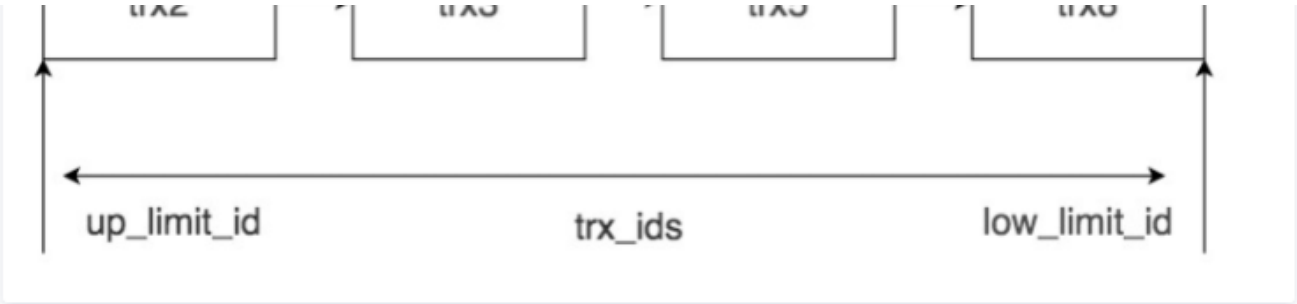


Read View 是啥?

如果一个事务要查询行记录，需要读取哪个版本的行记录呢？Read View 就是来解决这个问题的。Read View 可以帮助我们解决可见性问题。Read View 保存了**当前事务开启时所有活跃的事务列表。**换个角度，可以理解为: **Read View 保存了不应该让这个事务看到的其他事务 ID 列表。**

1. trx_ids 系统当前正在活跃的事务ID集合。
2. low_limit_id ,活跃事务的最大的事务 ID。
3. up_limit_id 活跃的事务中最小的事务 ID。
4. creator_trx_id，创建这个 ReadView 的事务ID。

ReadView



Read View

如果当前事务的 creator_trx_id 想要读取某个行记录，这个行记录ID 的trx_id ，这样会有以下的情况：

- 如果 trx_id < 活跃的最小事务ID（up_limit_id）,也就是说这个行记录在**这些活跃的事务创建前就已经提交了，那么这个行记录对当前事务是可见的。**
- 如果trx_id > 活跃的最大事务ID（low_limit_id），这个说明行记录在这些活跃的事务之后才创建，说明**这个行记录对当前事务是不可见的。**
- 如果 up_limit_id < trx_id <low_limit_id,说明该记录需要在 trx_ids 集合中，可能还处于活跃状态，因此我们需要在 trx_ids 集合中遍历，如果trx_id 存在于 trx_ids 集合中，证明这个事务 trx_id 还处于活跃状态，不可见，否则，trx_id 不存在于 trx_ids 集合中，说明事务trx_id 已经提交了，这行记录是可见的。

如何查询一条记录

java concurrent

```
1. 三种核心
    concurrent, copyOnWrite, Blocking

concurrent ->  concurrentmap, concurrentSKipListsMap,
            特点: 修改操作相对copyonwrite轻量级一些
                遍历一致性较低，弱一致性(size), 读取性能不确定



Blocking ->  LinkedBlockingQueue, PriorityBlockingQueue, ArrayBlockingQueue,
SynchronousQueue(队列容量为1)
            特点: block, 多线程操作不用加锁, 常用于 消费者模型
            核心方法: offer(不阻塞), put(阻塞),  take(会阻塞), poll(不阻塞),  peek(取出第一个但是不删除)
            核心实现方式:  ReentrantLock  与 conditon



copyOnWrite -> CopyOnWriteArrayList, CopyOnWriteArraySet(引用CopyOnWriteArrayList)
            特点: 优化读不加锁，所有update都加 reentrantlock
```

LinkedBlockingQueue

```
核心依赖是 lock 和 conditong
conditon ： 用来做block通知
reentryLocklock     :  用来做线程通知 和通知
```

```
private final AtomicInteger count = new AtomicInteger();

/**
 * Head of linked list.
 * Invariant: head.item == null
 */
transient Node<E> head;

/**
 * Tail of linked list.
 * Invariant: last.next == null
 */
private transient Node<E> last;

/** Lock held by take, poll, etc */
private final ReentrantLock takeLock = new ReentrantLock();

/** Wait queue for waiting takes */
private final Condition notEmpty = takeLock.newCondition();

/** Lock held by put, offer, etc */
private final ReentrantLock putLock = new ReentrantLock();

/** Wait queue for waiting puts */
private final Condition notFull = putLock.newCondition();

/**
 * Signals a waiting take. Called only from put/offer (which do not
 * otherwise ordinarily lock takeLock.)
 */
private void signalNotEmpty() {
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
```

```
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        while (count.get() == 0) {
            notEmpty.await();
        }
        x = dequeue();
        c = count.getAndDecrement();
        if (c > 1)
```

```
            if (c > 1)
                notEmpty.signal();
        } finally {
            takeLock.unlock();
        }
        if (c == capacity)
            signalNotFull();
        return x;
    }
```

```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    // Note: convention in all put/take/etc is to preset local var
    // holding count negative to indicate failure unless set.
    int c = -1;
    Node<E> node = new Node<E>(e);
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly();
    try {
        /*
         * Note that count is used in wait guard even though it is
         * not protected by lock. This works because count can
         * only decrease at this point (all other puts are shut
         * out by lock), and we (or some other waiting put) are
         * signalled if it ever changes from capacity. Similarly
         * for all other uses of count in other wait guards.
         */
        while (count.get() == capacity) {
            notFull.await();
        }
        enqueue(node);
        c = count.getAndIncrement();
        if (c + 1 < capacity)
            notFull.signal();
    } finally {
        putLock.unlock();
    }
    if (c == 0)
        signalNotEmpty();
}
```

ConcurrentLinkedQueue

核心依赖是 cas， 体现在代码的
ConcurrentHashMap 做了优化， 用了cas + synchronize

```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) {...}
    }
```

```java
public boolean offer(E e) {
    checkNotNull(e);
    final Node<E> newNode = new Node<E>(e);

    for (Node<E> t = tail, p = t;;) {
        Node<E> q = p.next;
        if (q == null) {
            // p is last node
            if (p.casNext( cmp: null, newNode)) {
                // 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
                    casTail(t, newNode);  // Failure is OK.
                return true;
            }
            // Lost CAS race to another thread; re-read next
        }
        else if (p == q)
            // 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.
            p = (t != (t = tail)) ? t : head;
```

```
        else
            // Check for tail updates after two hops.
            p = (p != t && t != (t = tail)) ? t : q;
    }
}
```

```
public E poll() {
    restartFromHead:
    for (;;) {
        for (Node<E> h = head, p = h, q;;) {
            E item = p.item;

            if (item != null && p.casItem(item, val: null)) {
                // Successful CAS is the linearization point
                // for item to be removed from this queue.
                if (p != h) // hop two nodes at a time
                    updateHead(h, ((q = p.next) != null) ? q : p);
                return item;
            }
            else if ((q = p.next) == null) {
                updateHead(h, p);
                return null;
            }
            else if (p == q)
                continue restartFromHead;
            else
                p = q;
        }
    }
}
```

CopyOnWriteArrayList

```
*/
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    private static final long serialVersionUID = 8673264195747942595L;

    /** The lock protecting all mutators */
    final transient ReentrantLock lock = new ReentrantLock();

    /** The array, accessed only via getArray/setArray. */
    private transient volatile Object[] array;

    /**
     * Gets the array.  Non-private so as to also be accessible
     * from CopyOnWriteArraySet class.
     */
    final Object[] getArray() { return array; }

    /**
     * Sets the array.
     */
    final void setArray(Object[] a) {
        array = a;
    }
```

```java
/**
 * Replaces the element at the specified position in this list with the
 * specified element.
 *
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public E set(int index, E element) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        E oldValue = get(elements, index);

        if (oldValue != element) {
            int len = elements.length;
            Object[] newElements = Arrays.copyOf(elements, len);
            newElements[index] = element;
            setArray(newElements);
        } else {
            // Not quite a no-op; ensures volatile write semantics
            setArray(elements);
        }
        return oldValue;
    } finally {
        lock.unlock();
    }
}
```

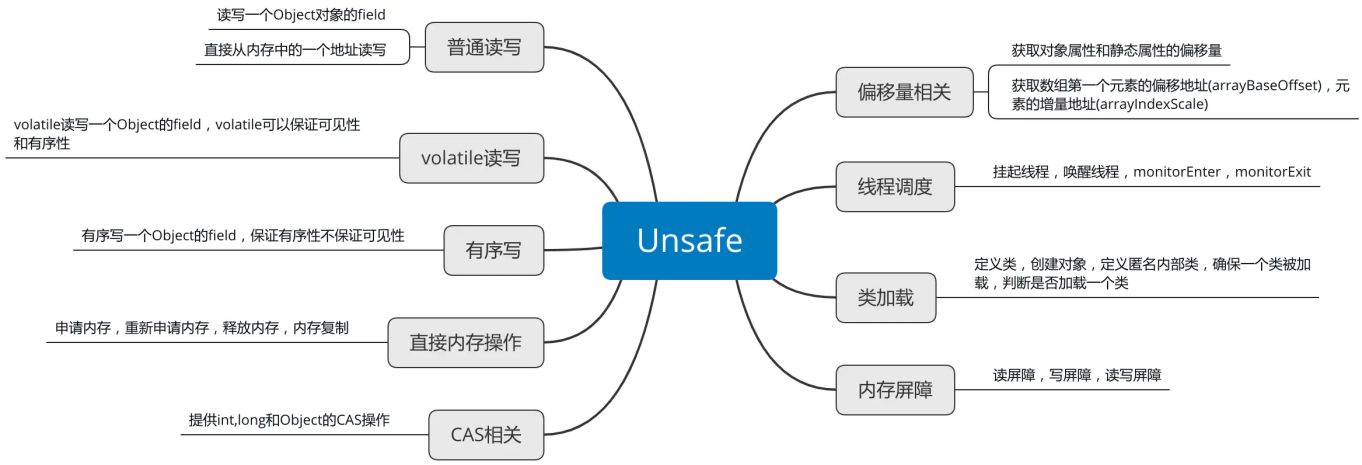
Atomic*

核心使用的是 cas 操作，
核心支持的数据结构是 unsafe

unsafe

https://www.jianshu.com/p/db8dce09232d

compareAndSwapInt(this, valueOffset, expect, update);
其中第一个参数为需要改变的对象，第二个为偏移量(即之前求出来的valueOffset的值)，第三个参数为期待的值，第四个为更新后的值



LockSupport

```
park
unpark

blocker(用于排查问题)
```

关键词

```
synchronize
transint
volitile
```

FutureTask

```
1. get 阻塞，waitNode（unsafe 加入），Locksupprot.park()
2. run(), 执行, set(), finishConpletion(), 获取waitnode,  LockSupport.unpark(), 唤醒线程
```

readwritelock

```
核心是 场景问题 和 锁降级 升级问题
```

AQS

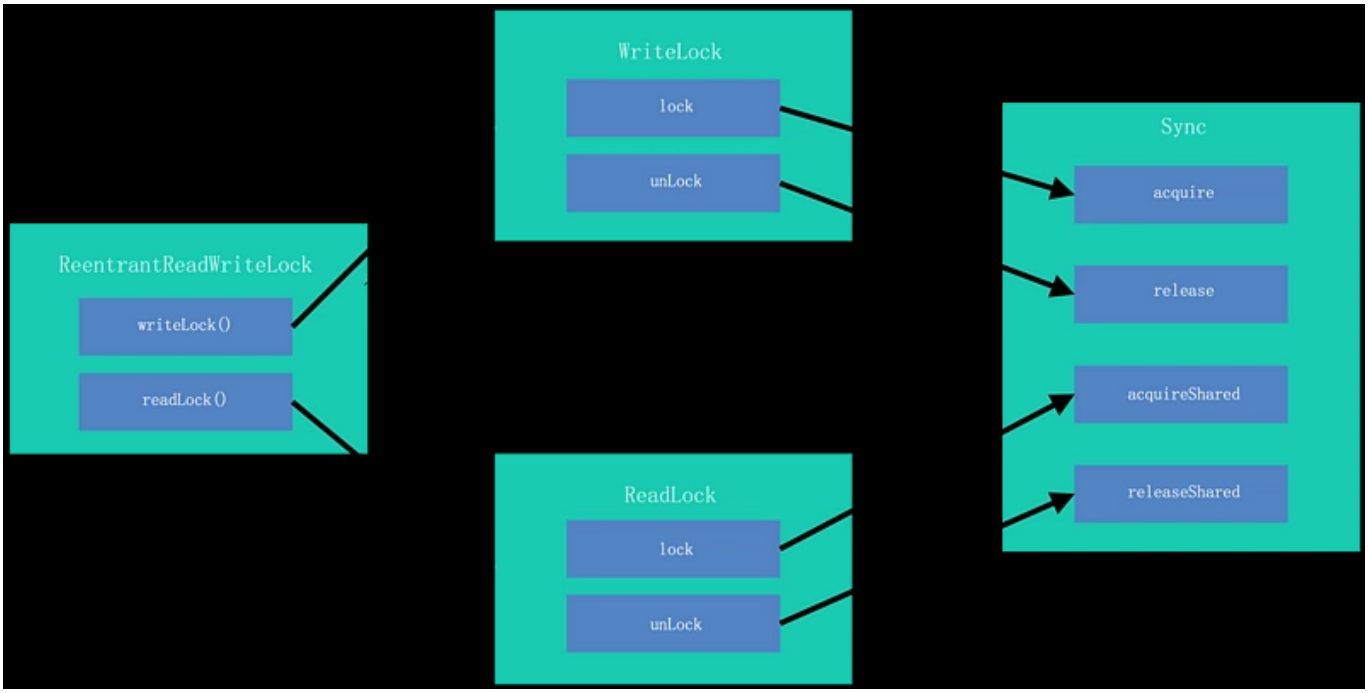
```
1. voltile status
2. Node 双向链表，
```

3. Node 包含属性：模式(share, exclusive)，状态(取消，signal, condition，等等), 节点指针，thread对象

4. 提供了一个 ConditionObject, 本时是一个 该锁关联的【条件锁】， 提供的方法是 notify(唤醒队列的头节点) 和 wait(释放当前线程的锁)
    以 AbstractQueuedSynchronizer.Node 链表为线程管理结构

reentaintReadWriteLock

https://segmentfault.com/a/1190000015768003



happen before

https://cloud.tencent.com/developer/article/1628493

JMM 内部是怎样实现 happen-before 原则的？
1. 内存屏障
2. 禁止重排序

volatile: 保证有序性与可见性
    对改变量的写操作之后，编译器插入一个写屏障
    对改变量的读操作之前，编译器会插入一个读屏障

什么是内存屏障：
    线程写入，写屏障会通过类似强迫刷出处理器缓存的方式，让其他线程能够拿到最新数值。
    1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；
    2）它会强制将对缓存的修改操作立即写入主存；
    3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

有序性，原子性，可见性是线程安全的基本保障。

线程安全

https://cloud.tencent.com/developer/article/1626521

https://cloud.tencent.com/developer/article/1625437

原子性()，可见性()，有序性

synchronized -> monitorenter/monitorexit

```
D:\>javac SynchronizedDemo.java

D:\>javap -c SynchronizedDemo.class
Compiled from "SynchronizedDemo.java"
public class SynchronizedDemo {
  public SynchronizedDemo();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static synchronized void doSth();
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Hello World
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return

  public static void doSth1();
    Code:
       0: ldc           #5                  // class SynchronizedDemo
       2: dup
       3: astore_0
       4: monitorenter
       5: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       8: ldc           #6                  // String Hello World 1
      10: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      13: aload_0
      14: monitorexit
      15: goto          23
      18: astore_1
      19: aload_0
      20: monitorexit
      21: aload_1
      22: athrow
      23: return
    Exception table:
       from    to  target type
           5    15    18   any
          18    21    18   any

  public static void main(java.lang.String[]);
    Code:
       0: invokestatic  #7                  // Method doSth1:()V
       3: return
}
```

并发包

汇总
https://cloud.tencent.com/developer/article/1625435

AQS
https://www.jianshu.com/p/497a8cfeef63
https://monkeysayhi.github.io/2017/12/04/AQS%E7%9A%84%E5%9F%BA%E6%9C%AC%E5%8E%9F%E

```
7%90%86/

ReentraitLock
https://monkeysayhi.github.io/2017/12/05/%E6%BA%90%E7%A0%81%7C%E5%B9%B6%E5%8F%91%E
4%B8%80%E6%9E%9D%E8%8A%B1%E4%B9%8BReentrantLock%E4%B8%8EAQS%EF%BC%881%EF%BC%89%EF%
BC%9Alock%E3%80%81unlock/
```

分布式事务与MQ

```
https://cloud.tencent.com/developer/article/1632632
https://cloud.tencent.com/developer/article/1625430
```

JVM

```
https://cloud.tencent.com/developer/article/1584579
https://cloud.tencent.com/developer/article/1649768
https://cloud.tencent.com/developer/article/1610259
https://cloud.tencent.com/developer/article/1625431
```

定时器

```
```