第四章 Java 并发机制的底层实现

2.1 volatile 的应用

Volatile 是轻量级的 synchronized，它在多处理器开发中保证了共享变量的“可见性”。即当一个线程修改了一个被 volatile 修饰共享变量的值，新值总数可以被其他线程立即得知。Volatile 不会引起线程上下文的切换和调度。

1. volatile 的特性

1) 可见性：当读一个 volatile 变量时，JMM 会把该线程对应的本地内存置为无效。线程从主内存中读取共享变量。

(1) 如果对声明了 volatile 的变量进行写操作，JVM 会向处理器发送一条 LOCK 前缀的指令，将这个变量所在缓存行的数据写回到系统内存。

(2) 一个处理器的缓存回写到内存会导致其他处理器里缓存了该内存地址的数据无效。

2)原子性：对任意单个 volatile 变量的读/写具有原子性，但类似于 volatile++这种复合操作不具有原子性。

3)禁止指令重排序优化【1.5 才完全恢复】。

2.2 synchronized 的应用

JavaSE1.6 对 synchronized 进行了各种优化，有些情况下他就并不那么重了。

1、Java 中的每一个对象都可以作为锁

a) 对于普通同步方法，锁是当前实例对象

b) 对于静态同步方法，锁是当前类的 class 对象

c) 对于同步方法块，锁是 synchronized 括号里配置的对象

2、JVM 实现代码块同步

JVM 基于进入和退出 Monitor 对象来实现方法同步和代码块同步。代码块同步是使用 monitorenter 和 monitorexit 指令实现的，而同步方法是依靠方法修饰符上的 ACC_SYNCHRONIZED 来完成的，实质是对一个对象的监视器（monitor）进行获取。Monitorenter 指令是在编译后插入到同步代码块的开始位置，而 monitorexit 是插入到方法结束处和异常处。任何对象都有一个 monitor 预知关联，当且一个 monitor 被持有后，它将处于锁定状态。线程指向到 monitorenter 指令时，将会尝试获取对象所对应的 monitor 的所有权，即尝试获得对象的锁。

3、Java 对象头

Synchronized 用的锁是存在 Java 对象头里。如果对象是数组类型，则用 3 个字宽存储对象头，其他使用 2 字宽存储对象头。

Java 对象头里的 mark word 里默认储存对象的 hashcode、分代年龄和锁标记位。mark word 里存储的数据会随着锁标志位的变化而变化。锁标志位分为：轻量级锁 00、重量级锁 10、GC 标记 11、偏向锁 01。

4、锁的升级与对比

在 JavaSE1.6 中，**引入了轻量级锁和偏向锁**。锁一共有 4 中状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁、重量级锁。锁可以升级但不能降级。

d) 偏向锁：偏向锁就是在**无竞争的情况下把整个同步都消除掉**，连 CAS 都不做了。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时 Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的

性能。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，毕竟极有可能连续多次是同一个线程申请相同的锁。但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。

【当锁对象第一次被线程获取的时候，虚拟机将会把对象头中的标志位设为“01”，即偏向模式。同时使用 CAS 操作把获取到这个锁的线程的 ID 记录在对象的 mark word 之中，如果 CAS 操作成功，持有偏向锁的线程以后每次进入这个锁相关的同步块，虚拟机都可以不再进行任何同步操作，只需简单测试一下对象头的 mark word 里是否存储着指向当前线程的偏向锁】

e) 轻量级锁：轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

代码进入同步块的时候，如果此同步对象没有被锁定，虚拟机首先将在当前线程的栈帧中创建用于存储锁记录的空间，并将对象头中的 mark word 复制到锁记录中。然后，虚拟机将使用 CAS 操作将对象的 mark word 更新为指向锁记录的指针。如果更新成功，那么这个线程就拥有了该对象的锁，并且对象的锁标记位转变为 00.如果更新失败，虚拟机首先会检查对象的 mark word 是够指向当前线程的栈帧，如果是说明当前线程已经拥有了这个对象的锁，否则说明这个锁对象已经被其他线程抢占，当前线程便尝试使用自旋来获取锁。自旋获取失败，这时轻量级锁就不再有效，要膨胀为重量级锁，锁标志变为“10”。

f) 轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是 50 个循环或 100 循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

g) 锁的优缺点对比

锁	优点	缺点	使用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，适用自旋会消耗 CPU	追求响应时间，同步块执行速度非常快
重量级锁	线程竞争不适用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢	追求吞吐量 同步块指向速度较长

2.3 原子操作的实现原理

无锁执行者 CAS 的核心算法原理，然后分析 Java 执行 CAS 的实践者 Unsafe 类，该类中的方

法都是 native 修饰的，因此我们会以说明方法作用为主介绍 Unsafe 类，最后再介绍并发包中的 Atomic 系统使用 CAS 原理实现的并发类。

1、Java 中实现原子操作

在 Java 中可以通过锁和循环 CAS 的方式来实现原子操作。从 Java1.5 开始，JDK 的并发包里提供了一些来支持原子操作，如 AtomicBoolean、AtomicInteger 和 AtomicLong。

2、 CAS

1) Compare and Swap，翻译成比较并交换。

java.util.concurrent 包中借助 CAS 实现了区别于 synchronouse 同步锁的一种乐观锁。

CAS 有 3 个操作数，内存值 V，旧的预期值 A，要修改的新值 B。当且仅当预期值 A 和内存值 V 相同时，将内存值 V 修改为 B，否则什么都不做。

CAS 通过调用 JNI 的代码实现的。JNI:Java Native Interface 为 JAVA 本地调用，允许 java 调用其他语言。而 compareAndSwapInt 就是借助 C 来调用 CPU 底层指令实现的。因为 CAS 是一种系统原语，原语属于操作系统用语范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许被中断，也就是说 CAS 是一条 CPU 的原子指令，不会造成所谓的数据不一致问题。

2) 执行函数：CAS(V,E,N)

其包含 3 个参数：V 表示要更新的变量,E 表示预期值,N 表示新值.

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，原理图如下

3、CAS 实现原子操作的三大问题

1) ABA 问题：

CAS 需要检查值有没有发生变化，如果没有变化则更新，但是如果一个值**原来是 A，变成 B，又变成了 A**，那么使用 CAS 进行检查时会发现它的值没有发生变化，实际上变化了。

ABA 问题的解决思路就是：使用版本号。在变量前面追加上版本号。

从 JDK1.5 开始，JDK 的 Atomic 包里提供了一个类 **AtomicStampedReference 来解决 ABA 问题**。这个类的 compareAndSet 方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式设置更新值。

2)循环时间长开销大

自旋 CAS 如果长时间不成功，会给 CPU 带来非常大的执行开销。

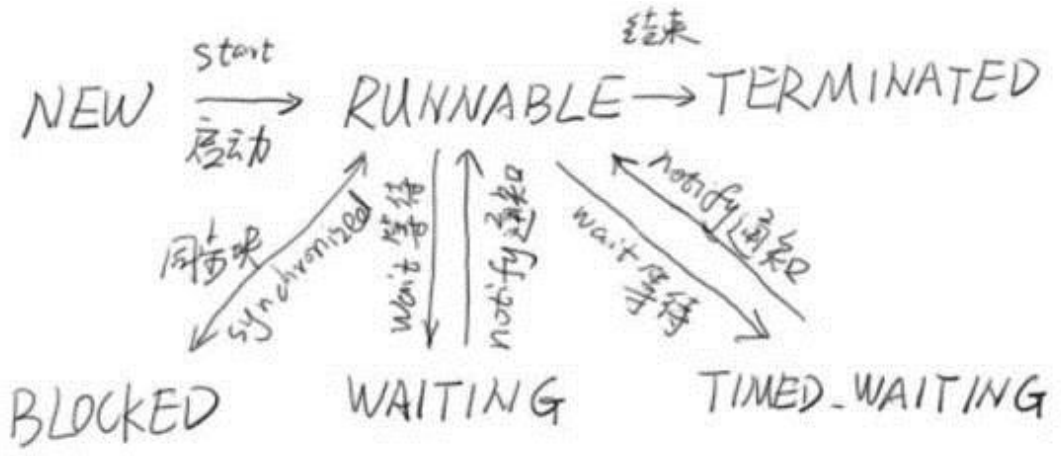
3)只能保证一个共享变量的原子操作

对多个共享变量操作时，循环 CAS 无法保证操作的原子性，这个时候可以用锁。从 JDK1.5 开始，JDK 提供了 **AtomicReference 类来保证引用对象之间的原子性**，可以把多个变量放在一个对象里来进行 CAS 操作。

第三章 Java 并发编程基础

5.1 线程的状态

Java 线程在运行的生命周期中可能处于 6 种不同的状态：new、runnable、blocked、waiting、time_waiting、terminated。



1. **Deamon 线程**

当一个 Java 虚拟机中不存在非 Deamon 线程的时候，Java 虚拟机将会退出。

2. 新建线程

1)实现 runnable 接口

2)继承 Thread 类

3)实现 Callable 接口，实现 call()

3. 启动和终止线程

启动线程：start()

中断线程 interrupt() :线程并不会立即退出，而是给线程发送一个通知，告知目标线程，有人希望你退出，至于目标接到通知后如何处理，则完全由目标线程自行决定。

安全地终止线程：①使用中断；② 使用 Boolean 变量

4. 过期的 挂起（suspend）、继续执行（resume）和 stop

Suspend：线程不会释放任何资源，而是占有着资源进入睡眠状态，容易引起死锁。

Resume：被挂起的线程直到遇到 resume 操作，才能继续。

Stop：在终结一个线程时不会保证线程的资源正常释放。

5. 等待（wait）和通知（notify）

线程 A 调用 wait()，A 就转为等待状态；

线程 A 调用 notify，其他线程被唤醒争抢资源；

6. 等待线程结束（join）和谦让（yield）

Join：在线程 A 中调用 B.join() 表示祖师 A 线程，知道线程 B 完成或超时。

Yield：调用 Thread.yidld()，给予其它重要线程更多的机会。

7、线程交替打印例子

```
public static void main(String[] args) {
    Thread A = new Thread(new ThreadA());
    Thread B = new Thread(new ThreadB());
    A.start();
    B.start();
}

static class ThreadA implements Runnable{
  @Override
  public void run() {
    synchronized(lock) {
      for (int i = 0; i < 26; i++ ) {
          System.out.print(ch ++);
          lock.notify();
          try {
            lock.wait();
          } catch (InterruptedException e) {}
      }
    }
  }
}

static class ThreadB implements Runnable{
  @Override
  public void run() {
    synchronized(lock) {
      for (int i = 0; i < 26; i++ ) {
          System.out.print(array ++);
          lock.notify();
          try {
            lock.wait();
          } catch (InterruptedException e) {}
      }
    }
  }
}
```

第六章 Java 中的锁

CAS 的操作在 ReetrantLock 的实现原理中可是随处可见。
无锁操作：CAS
轻级的隐式锁：volatile
重量级的隐式锁：synchronized
显示锁：锁的持有和释放都必须手动编写

6.1 Lock 接口

```
Lock lock = new ReentrantLock();
lock.lock();
try{
    //临界区......
}finally{
    lock.unlock();
}
```

当前线程使用 lock()方法与 unlock()对临界区进行包围，其他线程由于无法持有锁将无法进入临界区直到当前线程释放锁，注意 **unlock()操作必须在 finally 代码块中**，这样可以确保即使临界区执行抛出异常，线程最终也能正常释放锁。

```
public interface Lock {
    //加锁
    void lock();

    //解锁
    void unlock();

    //可中断获取锁，与lock()不同之处在于可响应中断操作，即在获
    //取锁的过程中可中断，注意synchronized在获取锁时是不可中断的
    void lockInterruptibly() throws InterruptedException;

    //尝试非阻塞获取锁，调用该方法后立即返回结果，如果能够获取则返回true，否则返回false
    boolean tryLock();

    //根据传入的时间段获取锁，在指定时间内没有获取锁则返回false，如果在指定时间内当前线程未被中并断获取到锁则返回true
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

    //获取等待通知组件，该组件与当前锁绑定，当前线程只有获得了锁
    //才能调用该组件的wait()方法，而调用后，当前线程将释放锁。
    Condition newCondition();
```

在 Java 1.5 中，官方在 concurrent 并发包中加入了 Lock 接口，该接口中提供了 lock()方法和 unLock()方法对显式加锁和显式释放锁操作进行支持。Lock 对象锁还提供了 synchronized 所不具备的其他同步特性，如尝试非阻塞地获取锁，能被中断锁的获取(synchronized 在等待获取锁时是不可中的)，超时锁的获取，等待唤醒机制的多条件变量 Condition 等，这也使得 Lock 锁在使用上具有更大的灵活性。

6.2 队列同步器 AbstractQueuedSynchronizer

队列同步器 AbstractQueuedSynchronized 是用来构建锁或者其他同步组件的基础框架。它使用了一个 int 成员变量表示同步状态，并且提供了 3 个方法(getState()、setState(int newState)和 compareAndSetState(int expect, int update))来操作同步状态，同时它们能保证状态的改变是安全的。

1. AQS 工作原理概要

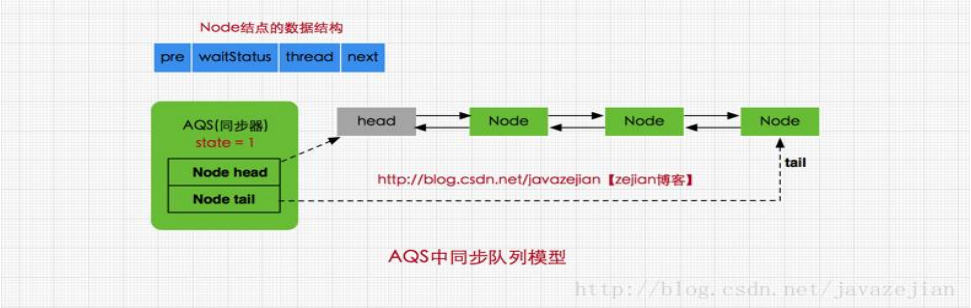
AbstractQueuedSynchronizer 内部通过一个 **int 类型的成员变量 state 来控制同步状态**,当 **state=0 时，则说明没有任何线程占有共享资源的锁**，当 **state=1** 时，则说明有线程目前正在使用共享变量，其他线程必须**加入同步队列进行等待**。

AQS 内部**通过内部类 Node 构成 FIFO 的同步队列**来完成线程获取锁的排队工作，同时利用**内部类 ConditionObject 构建等待队列**，当 Condition 调用 await()方法后，线程将会加入等待队列中，而当 Condition 调用 signal()方法后，线程将从等待队列转移动同步队列中进行锁竞争。注意这里涉及到两种队列，一种的同步队列，当线程请求锁而等待的后将加入同步

队列等待，而另一种则是等待队列(可有多个)，通过 Condition 调用 await()方法释放锁后，将加入等待队列。

AQS 中的同步队列模型，如下：

```
/**
 * AQS抽象类
 */
public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer{
//指向同步队列队头
private transient volatile Node head;

//指向同步的队尾
private transient volatile Node tail;

//同步状态，0代表锁未被占用，1代表锁已被占用
private volatile int state;

//省略其他代码.....
}
```



head 和 tail 分别是 AQS 中的变量，**注意 head 为空结点，不存储信息**。而 tail 则是同步队列的队尾，同步队列采用的是**双向链表的结构**这样可方便队列进行结点增删操作。state 变量则是代表同步状态，执行当线程调用 lock 方法进行加锁后，如果此时 state 的值为 0，则说明当前线程可以获取到锁，同时将 state 设置为 1，表示获取成功。**如果 state 已为 1，也就是当前锁已被其他线程持有，那么当前执行线程将被封装为 Node 结点加入同步队列的尾部等待**。同步器提供了一个机遇 CAS 的设置为节点的方法：compareAndSetTail(Node expect, Node updte)。首节点是获取同步状态成功的节点，首节点的线程在释放同步状态时，将会唤醒后继节点，后继节点将会在获取同步状态成功时将自己设置为首节点。由于只有一个线程能够成功获取到同步状态，因此设置头节点的方法并不需要使用 CAS 来保证。

其中 Node 结点是 AQS 的内部类，是对每一个访问同步代码的线程的封装，其包含了需要同步的线程本身以及线程的状态，如**是否被阻塞，是否等待唤醒，是否已经被取消**等。每个 Node 结点内部关联其前继结点 prev 和后继结点 next，这样可以方便线程释放锁后快速唤醒下一个在等待的线程。

```
//等待状态,存在CANCELLED、SIGNAL、CONDITION、PROPAGATE 4种
volatile int waitStatus;

//同步队列中前驱结点
volatile Node prev;

//同步队列中后继结点
volatile Node next;

//请求锁的线程
volatile Thread thread;

//等待队列中的后继结点，这个与Condition有关，稍后会分析
Node nextWaiter;
```

其中 **SHARED 和 EXCLUSIVE 常量分别代表共享模式和独占模式**，所谓共享模式是一个锁允许多条线程同时操作，如信号量 Semaphore 采用的就是基于 AQS 的共享模式实现的，而独占模式则是同一个时间段只 能有一个线程对共享资源进行操作，多余的请求线程需要排队等待，如 ReentranLock。

2. 独占式同步状态获取

1、通过调用 aqs 的 acquire(int arg)方法获取同步状态。

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

完成同步状态获取、节点构造、加入同步队列以及在同步队列中自旋等待的工作，主要是：①先调用实现者的 tryAcquire(int arg)方法，该方法保证线程安全的获取同步状态。②如果获取失败，则构造同步节点，并通过 addWaiter(Node node)方法加入到同步队列的尾部；③调用 acquireQueued(node, int)方法，以死循环的方式获取同步状态，并会阻塞节点的线程，直到前驱节点的出兑或阻塞线程被中断实现。

2、

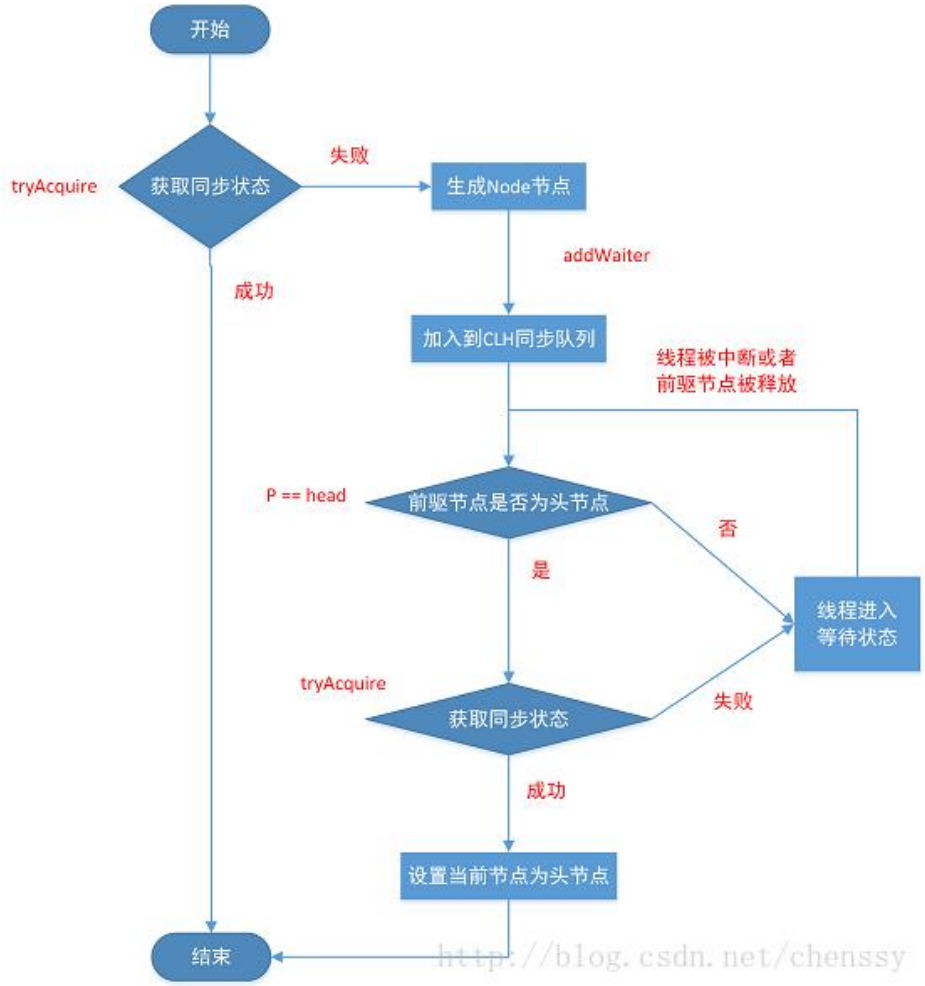
```
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);
    return node;
}
```

通过调用 compareAndSetTail(Node, Node)：线程安全添加到不同队列尾部；
Enq：在“死循环”中通过 CAS 将节点设置成为尾节点后，才能返回，并并发添加节点的请求通过 CAS 变得“串行化”。

3、acquireQueued

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

节点进入同步队列之后，就进入了一个自旋过程，当条件满足，获得同步状态。当前线程在“死循环”中尝试获取同步状态，而只有前去节点是头结点才能够尝试获取同步状态，因为：
① 头结点是成功获取到同步状态的节点，而头结点的线程释放了同步状态之后，将唤醒起后继节点，后继节点的线程呗唤醒后需要检查自己的前驱节点是否是头结点。
② 维护同步队列的 FIFO 原则。



3. 共享式同步状态获取

通过调用同步器的 acquireShared(int arg)方法可以共享式地获取同步状态。在acquireShared(int arg)方法中，同步器调用 tryAcquireShared(int arg)方法尝试获取同步状态。当返回值大于等于 0 时，表示能够获取到同步状态。因此，在共享式获取的自旋过程中，成功获取到同步状态并退出自旋的条件就是 tryAcquireShared(int arg)方法返回值大于等于 0。即当前节点的前驱为头结点时，尝试获取同步状态，如果返回值大于等于 0，表示该次获取同步状态成功并从自旋过程中退出。

```
protected int tryAcquireShared(int acquires) {
    for (;;) {
        if (hasQueuedPredecessors())
            return -1;
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
```

```
        }
    }
```

Semaphore 将 aqs 的同步状态用于保存当前可用许可的数量。tryAcquireShared 首先计算剩余许可的数量，如果没有足够的许可，那么会返回一个值表示获取操作失败。如果还有剩余的许可，那么通过 compareAndSetState 以原子方式来降低许可的计数。

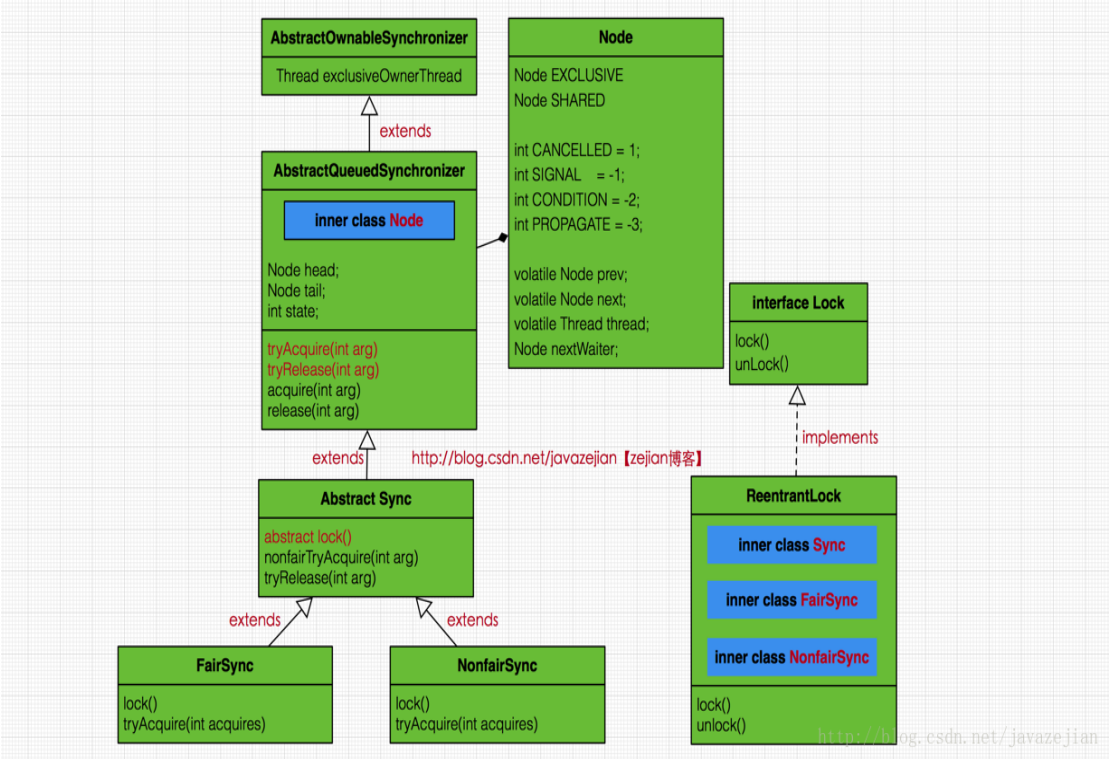
CountDownLatch 使用 AQS 像是：在同步状态中保存的是当前的计数值。CountDown 方法调用 release。从而导致计数值递减，并且当计数值为零时，解除所有等待线程的阻塞。Await 调用 acquire，当计数器为零时，acquire 将立即返回，否则将阻塞。

4. 独占式超时获取同步状态

增强版的方法：tryAcquireNanos(int arg,long nanos)。该方法为 acquireInterruptibly 方法的进一步增强，它除了响应中断外，还有超时控制。

针对超时控制，程序**首先记录上次唤醒时间** lastTime,当前唤醒时间 now。如果获取同步状态失败，则需要计算出**需要休眠的时间间隔** nanosTimeout -= now - lastTime，如果 nanosTimeout <= 0 表示已经超时了，返回 false，如果大于 spinForTimeoutThreshold（1000L）则需要休眠 nanosTimeout ，如果 nanosTimeout <= spinForTimeoutThreshold ，就不需要休眠了，直接进入快速自旋的过程。原因在于 spinForTimeoutThreshold 已经非常小了，非常短的时间等待无法做到十分精确，如果这时再次进行超时等待，相反会让 nanosTimeout 的超时从整体上面表现得不是那么精确，所以在超时非常短的场景中，AQS 会进行无条件的快速自旋。

6.3 ReentrantLock



```
//默认构造，创建非公平锁NonfairSync
public ReentrantLock() {
    sync = new NonfairSync();
}
//根据传入参数创建锁类型
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}

//加锁操作
public void lock() {
     sync.lock();
}
```

1. **ReetrantLock 中非公平锁**

```
/**
 * 非公平锁实现
 */
static final class NonfairSync extends Sync {
    //加锁
    final void lock() {
        //执行CAS操作，获取同步状态
        if (compareAndSetState(0, 1))
       //成功则将独占锁线程设置为当前线程
          setExclusiveOwnerThread(Thread.currentThread());
        else
            //否则再次请求同步状态
            acquire(1);
    }
}
```

首先对同步状态执行 CAS 操作，尝试把 state 的状态从 0 设置为 1。如果返回 true 则代表获取同步状态成功，也就是当前线程获取锁成，可操作临界资源。如果返回 false，则表示已有线程持有该同步状态(其值为 1)，获取锁失败，执行 acquire(1)方法，该方法是 AQS 中的方法，它对中断不敏感，即使线程获取同步状态失败，进入同步队列，后续对该线程执行中断操作也不会从同步队列中移出，方法如下

```
public final void acquire(int arg) {
    //再次尝试获取同步状态
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

这里传入参数 arg 表示要获取同步状态后设置的值(即要设置 state 的值)，因为要获取锁，而 status 为 0 时是释放锁，1 则是获取锁，所以这里一般传递参数为 1，进入方法后首先会执行 tryAcquire(arg)方法，在前面分析过该方法在 AQS 中并没有具体实现，而是交由子类实现，因此该方法是由 ReetrantLock 类内部实现的。

```
//NonfairSync类
static final class NonfairSync extends Sync {

    protected final boolean tryAcquire(int acquires) {
        return nonfairTryAcquire(acquires);
    }
}

//Sync类
abstract static class Sync extends AbstractQueuedSynchronizer {

  //nonfairTryAcquire方法
  final boolean nonfairTryAcquire(int acquires) {
      final Thread current = Thread.currentThread();
      int c = getState();
      //判断同步状态是否为0，并尝试再次获取同步状态
      if (c == 0) {
          //执行CAS操作
          if (compareAndSetState(0, acquires)) {
              setExclusiveOwnerThread(current);
              return true;
          }
      }
      //如果当前线程已获取锁，属于重入锁，再次获取锁后将status值加1
      else if (current == getExclusiveOwnerThread()) {
          int nextc = c + acquires;
          if (nextc < 0) // overflow
              throw new Error("Maximum lock count exceeded");
          //设置当前同步状态，当前只有一个线程持有锁，因为不会发生线程安全问题，可以直接执行 setState(nextc);
          setState(nextc);
          return true;
      }
      return false;
  }
```

一是尝试**再次获取同步状态**，如果获取成功则将当前线程设置为 OwnerThread，否则失败，二是**判断当前线程 current 是否为 OwnerThread**，如果是则属于重入锁，state 自增 1，并获取锁成功，返回 true，反之失败，返回 false。

接着看之前的方法 acquire(int arg)：如果 tryAcquire(arg)返回 true，acquireQueued 自然不会执行，如果 tryAcquire(arg)返回 false，则会执行 addWaiter(Node.EXCLUSIVE)进行入队操作,由于 ReentrantLock 属于独占锁，因此结点类型为 Node.EXCLUSIVE，下面看看 addWaiter 方法具体实现：

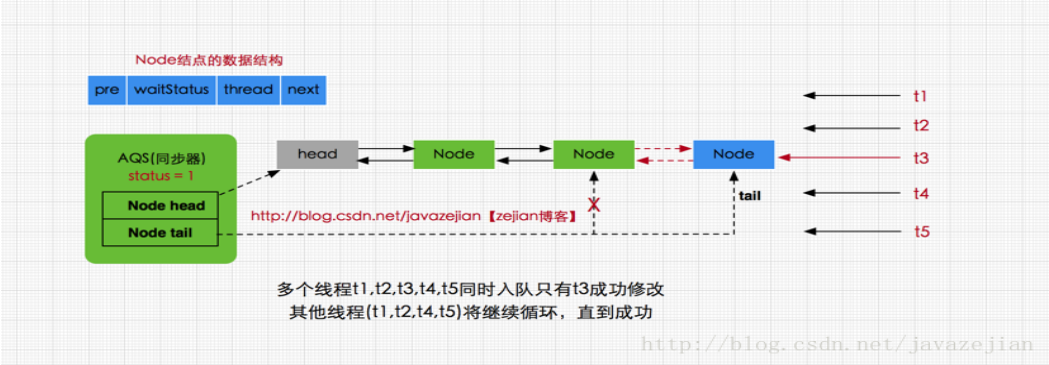
```
private Node addWaiter(Node mode) {
    //将请求同步状态失败的线程封装成结点
    Node node = new Node(Thread.currentThread(), mode);

    Node pred = tail;
    //如果是第一个结点加入肯定为空，跳过。
    //如果非第一个结点则直接执行CAS入队操作，尝试在尾部快速添加
    if (pred != null) {
        node.prev = pred;
        //使用CAS执行尾部结点替换，尝试在尾部快速添加
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    //如果第一次加入或者CAS操作没有成功执行enq入队操作
    enq(node);
    return node;
}
```

创建了一个 Node.EXCLUSIVE 类型 Node 结点用于封装线程及其相关信息，其中 tail 是 AQS 的成员变量，指向队尾。如果是第一个结点，则为 tail 肯定为空，那么将执行 enq(node)操作，如果非第一个结点即 tail 指向不为 null，直接尝试执行 CAS 操作加入队尾，如果 CAS 操作失败还是会执行 enq(node)，继续看 enq(node)：

```
private Node enq(final Node node) {
    //死循环
    for (;;) {
        Node t = tail;
        //如果队列为null，即没有头结点
        if (t == null) { // Must initialize
            //创建并使用CAS设置头结点
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {//队尾添加新结点
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
```

这个方法使用一个**死循环进行 CAS 操作**，可以解决多线程并发问题。这里做了两件事，一是如果还没有初始同步队列则创建新结点并使用 compareAndSetHead 设置头结点，tail 也指向 head，二是队列已存在，则将新结点 node 添加到队尾。

注意这两个步骤都存在同一时间多个线程操作的可能，**如果有一个线程修改 head 和 tail 成功，那么其他线程将继续循环，直到修改成功，**这里使用 CAS 原子操作进行头结点设置和尾结点 tail 替换可以保证线程安全，**从这里也可以看出 head 结点本身不存在任何数据，**它只是作为一个牵头结点，而 tail 永远指向尾部结点(前提是队列不为 null)。

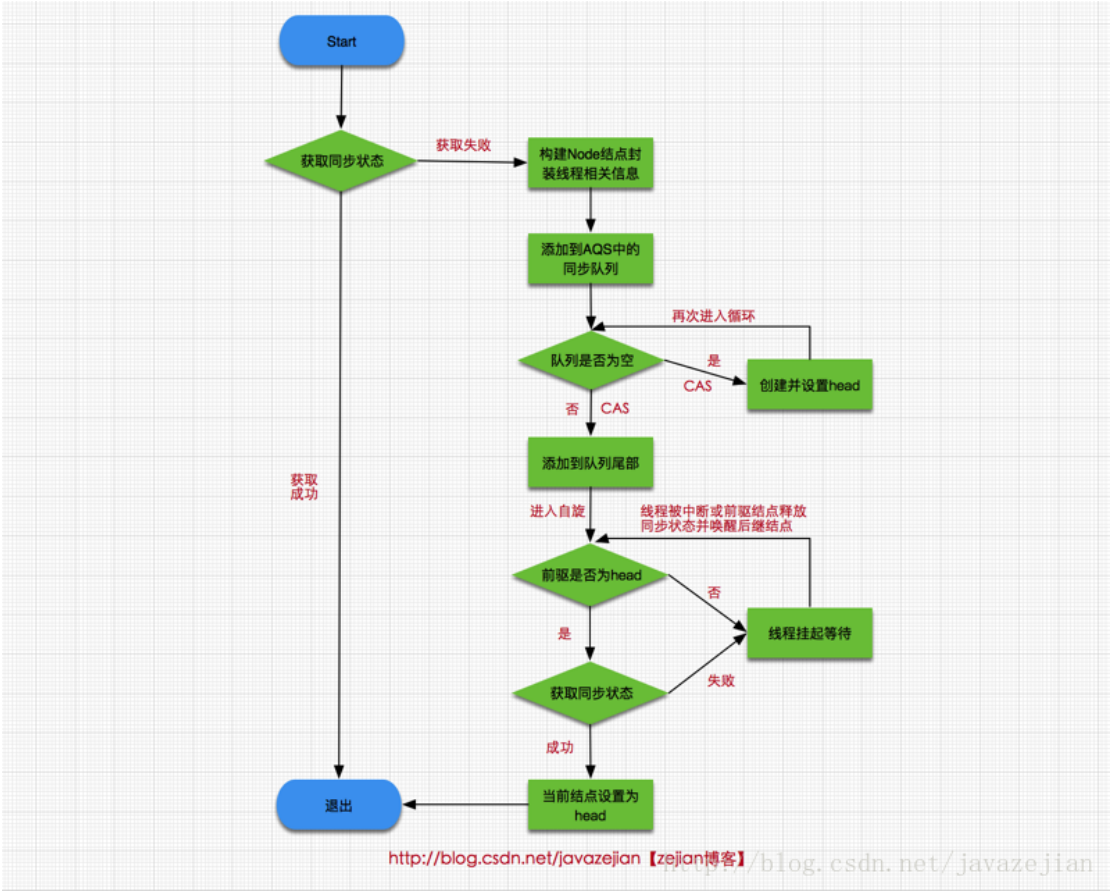


添加到同步队列后，结点就会进入一个自旋过程，即每个结点都在观察时机待条件满足获取同步状态，然后从同步队列退出并结束自旋，回到之前的 acquire()方法，自旋过程是在 acquireQueued(addWaiter(Node.EXCLUSIVE), arg))方法中执行的，代码如下

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        //自旋，死循环
        for (;;) {
            //获取前驱结点
            final Node p = node.predecessor();
            当且仅当p为头结点才尝试获取同步状态
            if (p == head && tryAcquire(arg)) {
                //将node设置为头结点
                setHead(node);
                //清空原来头结点的引用便于GC
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            //如果前驱结点不是head，判断是否挂起线程
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            //最终都没能获取同步状态，结束该线程的请求
            cancelAcquire(node);
    }
}
```

当前线程在自旋(死循环)中获取同步状态，当且仅当前驱结点为头结点才尝试获取同步状态，这符合 FIFO 的规则，即先进先出，其次 head 是当前获取同步状态的线程结点，只有当 head 释放同步状态唤醒后继结点，后继结点才有可能获取到同步状态，因此后继结点在其前继结点为 head 时，才进行尝试获取同步状态，其他时刻将被挂起。进入 if 语句后调用 setHead(node)方法，将当前线程结点设置为 head。

ReetrantLock 内部间接通过 AQS 的 FIFO 的同步队列就完成了 lock()操作，这里我们总结成逻辑流程图：


Start
获取同步状态
获取失败
构建Node结点封装线程相关信息
添加到AQS中的同步队列
再次进入循环
队列是否为空
是
CAS
创建并设置head
否
CAS
添加到队列尾部
获取成功
进入自旋
线程被中断或前驱结点释放同步状态并唤醒后继结点
前驱是否为head
否
是
线程挂起等待
获取同步状态
失败
成功
退出
当前结点设置为head
http://blog.csdn.net/javazejian【zejian博客】
http://blog.csdn.net/javazejian

```java
//ReentrantLock类的unlock
public void unlock() {
    sync.release(1);
}

//AQS类的release()方法
public final boolean release(int arg) {
    //尝试释放锁
    if (tryRelease(arg)) {

        Node h = head;
        if (h != null && h.waitStatus != 0)
            //唤醒后继结点的线程
            unparkSuccessor(h);
        return true;
    }
    return false;
}

//ReentrantLock类中的内部类Sync实现的tryRelease(int releases)
protected final boolean tryRelease(int releases) {

      int c = getState() - releases;
      if (Thread.currentThread() != getExclusiveOwnerThread())
          throw new IllegalMonitorStateException();
      boolean free = false;
      //判断状态是否为0，如果是则说明已释放同步状态
      if (c == 0) {
          free = true;
          //设置Owner为null
          setExclusiveOwnerThread(null);
      }
      //设置更新同步状态
      setState(c);
      return free;
  }
```

2. ReetrantLock 中公平锁

```java
//公平锁FairSync类中的实现
protected final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
            //注意！！这里先判断同步队列是否存在结点
                if (!hasQueuedPredecessors() &&
                    compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) {
                int nextc = c + acquires;
                if (nextc < 0)
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
        }
```

唯一不同的位置为判断条件多了 hasQueuedPredecessors()方法，即加入了同步队列中当前

节点是够有前驱节点的判断，如果该方法返回 true，则表示有线程比当前线程更早地请求获取锁，因此需要等待前驱线程获取并释放锁之后才能继续获取锁。

3. 总结

非公平性锁可能是线程“饥饿”，为什么它又被设定成默认的实现呢？因为非公平性锁的开销更小，因为减少了上下文切换的次数。

公平性锁保证了锁的获取按照 FIFO 原则，则代价是进行大量的线程切换，非公平性锁虽然可能曹成线程“饥饿”，但极少的线程切换，保证了其更大的吞吐量。

6.4 读写锁 ReentrantReadWriteLock

读写锁【jdk1.5】在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。读写锁维护了一堆锁，一个读锁和一个写锁，写操作对读操作保证可见性。把 State 状态作为一个读写锁的计数器，包括了重入的次数。

当写锁被获取到时，非当前写操作线程的读写操作都会被阻塞，写锁释放之后，所有操作继续执行。

1. 特性

公平性选择：支持非公平（默认）和公平的锁获取方式。

重入性：该锁支持重进入。读线程在获取了读锁之后，能够再次获取读锁。而写线程在获取了写锁之后，能够再次获取写锁，同时也可以获取读锁。

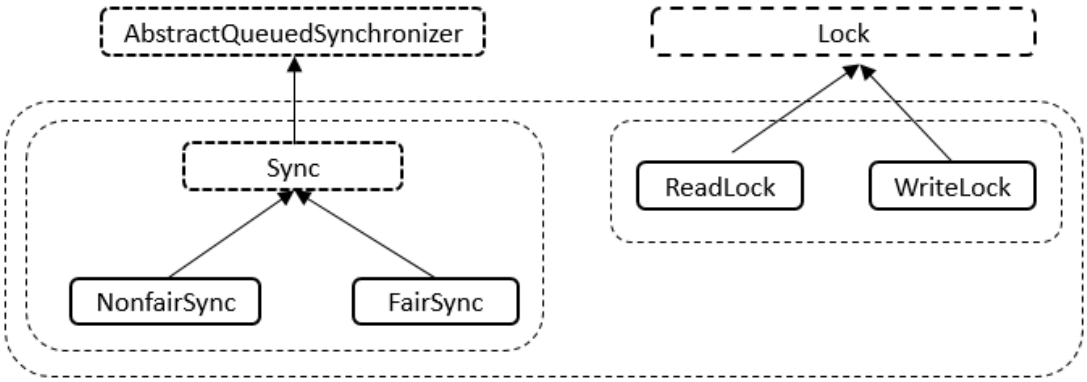
锁降级：遵循获取写锁、获取读锁再释放写锁的次序，写锁能够降级成为读锁。

2. 读写锁的接口

ReentrantReadWriteLock:

内部类：Sync、NonfairSync、FairSync、ReadLock、WriteLock

方法：Lock readLock();、Lock writeLock();



2.1 Sync 类

1、类的内部类

Sync 类内部存在两个内部类，分别为 HoldCounter 和 ThreadLocalHoldCounter，其中

HoldCounter 主要与读锁配套使用，其中，HoldCounter 源码如下。【tid 1.7 1.8 不一样】

static final class HoldCounter {
// 计数
int count = 0;
// Use id, not reference, to avoid garbage retention
// final long tid = Thread.currentThread().getId(); 1.7
final long tid = getThreadId(Thread.currentThread()); }
	其中 count 表示某个读线程重入的次数，tid 表示该线程的 tid 字段的值，该字段可以用来唯一标识一个线程。

```
// 本地线程计数器
 static final class ThreadLocalHoldCounter
    extends ThreadLocal<HoldCounter> {
   // 重写初始化方法，在没有进行 set 的情况下，获取的都是该 HoldCounter 值
    public HoldCounter initialValue() {
         return new HoldCounter();
    }
}
```

2、类的属性

```
abstract static class Sync extends AbstractQueuedSynchronizer {
   private static final long serialVersionUID = 6317671515068378041L;
   static final int SHARED_SHIFT   = 16; // 高 16 位为读锁，低 16 位为写锁
   static final int SHARED_UNIT    = (1 << SHARED_SHIFT);    // 读锁单位
   static final int MAX_COUNT      = (1 << SHARED_SHIFT) - 1;// 读锁最大数量
   static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;// 为了便于与运算去掉高 16 位
   private transient ThreadLocalHoldCounter readHolds;
  // 本地线程计数器，保存当前线程持有的重入读锁的数目，在读锁重入次数为 0 时移除
   private transient HoldCounter cachedHoldCounter; // 缓存的计数器
   private transient Thread firstReader = null;  // 第一个读线程
   private transient int firstReaderHoldCount;  // 第一个读线程的计数
}
```

把 State 状态作为一个读写锁的计数器，包括了重入的次数。state 是 32 位的 int 值，所以把高位 16 位作为读锁的计数器，**低位的 16 位作为写锁的计数器**，所以读锁增加 1 就相当于增加了 2*16

3、核心函数分析

I. sharedCount 函数

表示**占有读锁的线程数量**，源码如下

```
static int sharedCount(int c)    { return c >>> SHARED_SHIFT; }
```

说明：**直接将 state 无符号右移 16 位，就可以得到读锁的线程数量**，因为 state 的高 16 位表示读锁，对应的第十六位表示写锁数量。当读状态增加 1 时，即加 0x00010000.

II. exclusiveCount 函数

表示占有写锁的线程数量，源码如下

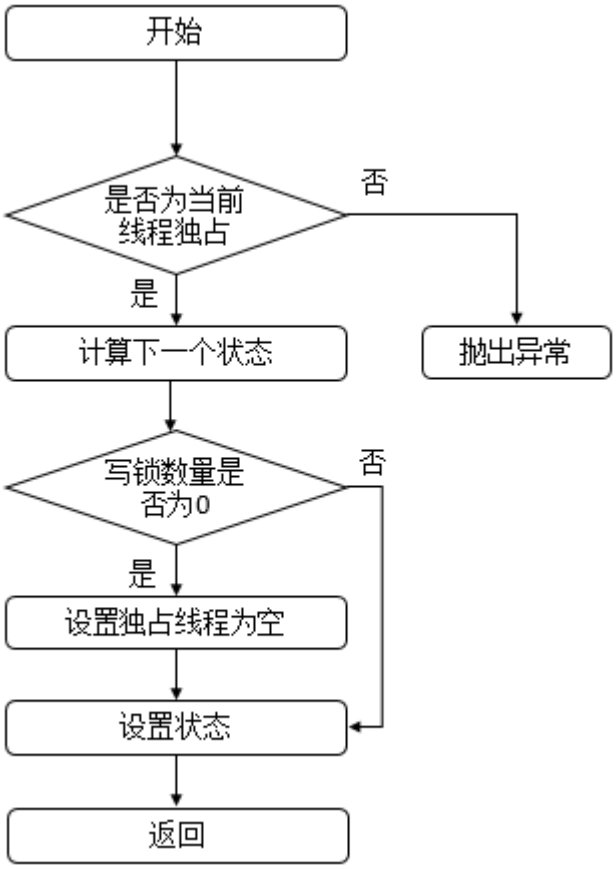
```
static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; }
```

说明：直接**将状态 state 和（2^16 – 1，即 0x0000FFFF）做与运算**，其等效于将 state 模上 2^16。写锁数量由 state 的低十六位表示。

III. tryRelease 函数

```
protected final boolean tryRelease(int releases) {
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    int nextc = getState() - releases;
    boolean free = exclusiveCount(nextc) == 0;
    if (free)
        setExclusiveOwnerThread(null);
    setState(nextc);
    return free;
}
```

此函数用于释放写锁资源，首先会判断该线程**是否为独占线程**，若不为独占线程，则抛出异常，否则，**计算释放资源后的写锁的数量，若为0，表示成功释放**，资源不将被占用，否则，表示资源还被占用。其函数流程图如下。



IV. tryAcquire 函数

```
protected final boolean tryAcquire(int acquires) {
        Thread current = Thread.currentThread();// 获取当前线程
```

```
            int c = getState();   // 获取状态
            int w = exclusiveCount(c); // 写线程数量
            if (c != 0) { // 状态不为0
            // (Note: if c != 0 and w == 0 then shared count != 0)
             if (w == 0 || current != getExclusiveOwnerThread()) // 写线
程数量为0或者当前线程没有占有独占资源
                 return false;
             if (w + exclusiveCount(acquires) > MAX_COUNT) // 判断是否超
过最高写线程数量
                 throw new Error("Maximum lock count exceeded");
             // Reentrant acquire
             // 设置AQS状态
             setState(c + acquires);
             return true;
           }
           if (writerShouldBlock() ||
                   !compareAndSetState(c, c + acquires)) // 写线程是否应
该被阻塞
           return false;
           // 设置独占线程
           setExclusiveOwnerThread(current);
           return true;
       }
```

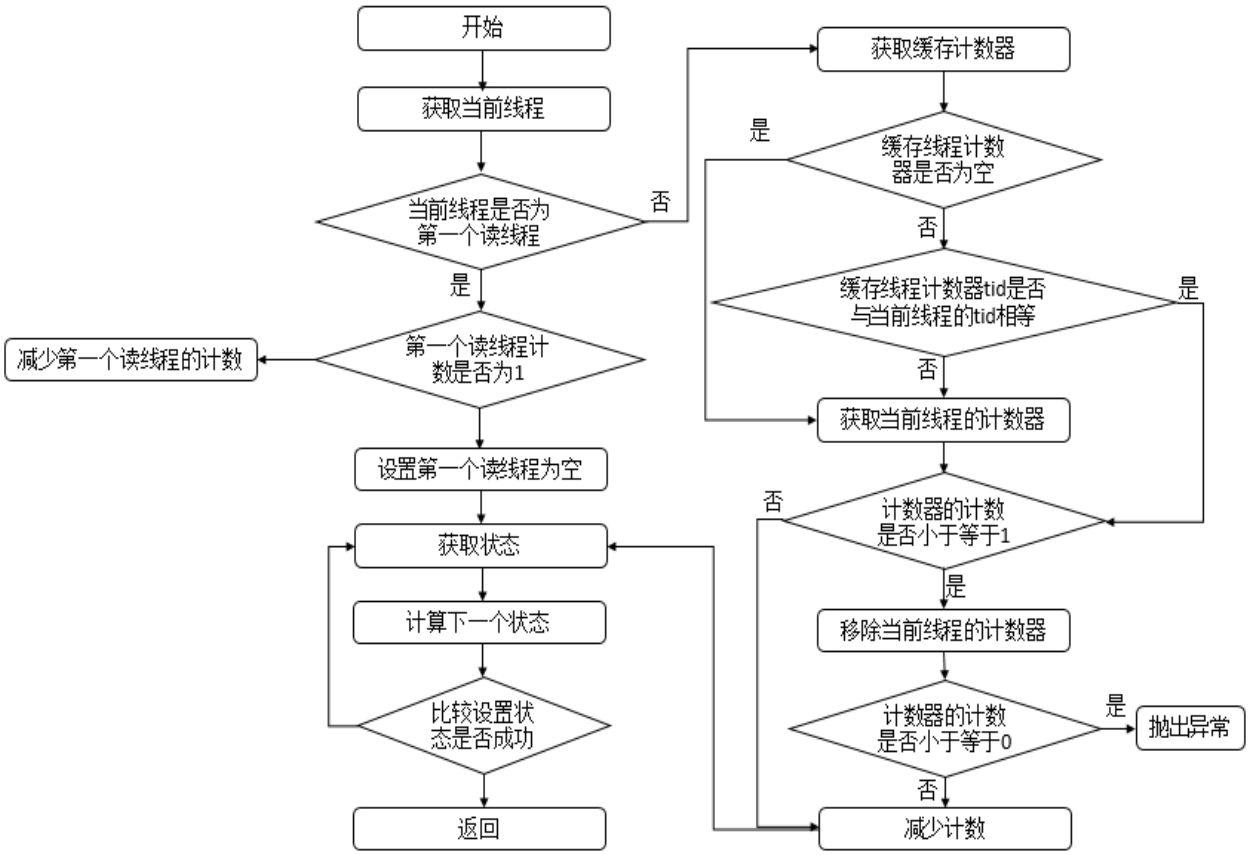
此函数用于获取写锁，首先会获取 state，判断是否为 0，若为 0，表示此时没有读锁线程，再判断写线程是否应该被阻塞，而在非公平策略下总是不会被阻塞，在公平策略下会进行判断（判断同步队列中是否有等待时间更长的线程，若存在，则需要被阻塞，否则，无需阻塞），之后在设置状态 state，然后返回 true。若 state 不为 0，则表示此时存在读锁或写锁线程，若写锁线程数量为 0 或者当前线程为独占锁线程，则返回 false，表示不成功，否则，判断写锁线程的重入次数是否大于了最大值，若是，则抛出异常，否则，设置状态 state，返回 true，表示成功。

V. tryReleaseShared 函数

```
protected final boolean tryReleaseShared(int unused) {
     // 获取当前线程
     Thread current = Thread.currentThread();
     if (firstReader == current) { // 当前线程为第一个读线程
       // assert firstReaderHoldCount > 0;
        if (firstReaderHoldCount == 1) // 读线程占用的资源数为1
             firstReader = null;
        else // 减少占用的资源
             firstReaderHoldCount--;
     } else { // 当前线程不为第一个读线程
         // 获取缓存的计数器
         HoldCounter rh = cachedHoldCounter;
```

```
        if (rh == null || rh.tid != getThreadId(current)) // 计数器为空或者计数器的 tid 不为当前正在运行的线程的 tid
        // 获取当前线程对应的计数器
            rh = readHolds.get();
       // 获取计数
        int count = rh.count;
        if (count <= 1) { // 计数小于等于 1
             // 移除
             readHolds.remove();
             if (count <= 0) // 计数小于等于 0，抛出异常
                  throw unmatchedUnlockException();
         }
               // 减少计数
         --rh.count;
           }
           for (;;) { // 无限循环
               // 获取状态
               int c = getState();
               // 获取状态
               int nextc = c - SHARED_UNIT;
               if (compareAndSetState(c, nextc)) // 比较并进行设置
                  return nextc == 0;
           }
       }
```

说明：**此函数表示读锁线程释放锁**。首先判断当前线程是否为第一个读线程 firstReader，若是，则判断第一个读线程占有的资源数是否为 1，若是，则设置第一个读线程 firstReader 为空，否则，将第一个读线程占有的资源数 firstReaderHoldCount 减 1；若当前线程不是第一个读线程，那么首先会获取缓存计数器（上一个读锁线程对应的计数器 ），若计数器为空或者 tid 不等于当前线程的 tid 值，则获取当前线程的计数器，如果计数器的计数 count 小于等于 1，则移除当前线程对应的计数器，如果计数器的计数 count 小于等于 0，则抛出异常，之后再减少计数即可。无论何种情况，都会进入无限循环，该循环可以确保成功设置状态 state。其流程图如下

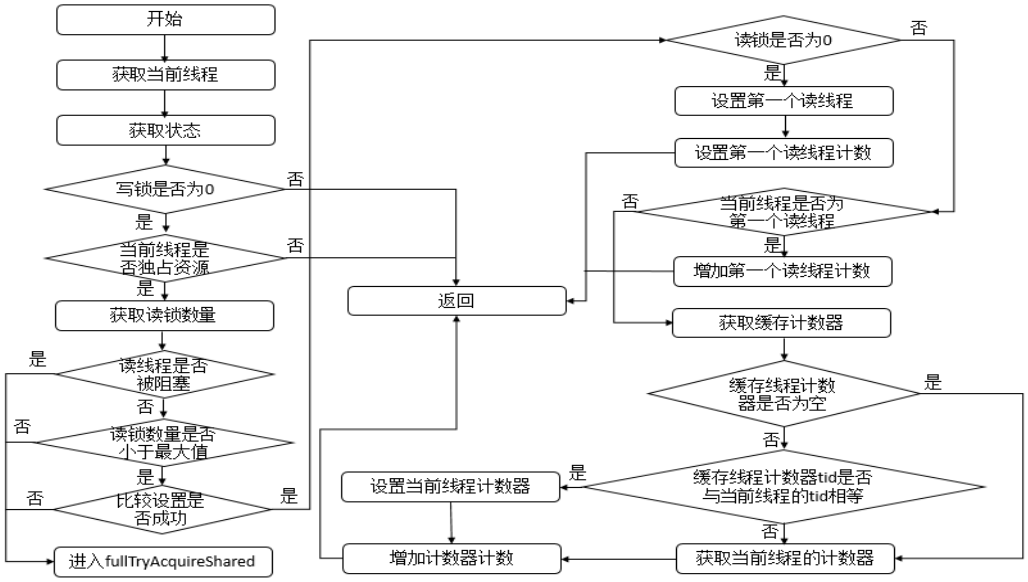


VI. tryAcquireShared 函数

```
// 共享模式下获取资源
protected final int tryAcquireShared(int unused) {
    Thread current = Thread.currentThread();// 获取当前线程
    int c = getState();  // 获取状态
    if (exclusiveCount(c) != 0 && getExclusiveOwnerThread() != current)
          // 写线程数不为 0 并且占有资源的不是当前线程
        return -1;
    int r = sharedCount(c);   // 读锁数量
    if (!readerShouldBlock() && r < MAX_COUNT && compareAndSetState(c, c + SHARED_UNIT)) {
         // 读线程不应该被阻塞、并且小于最大值、并且比较设置成功
        if (r == 0) { // 读锁数量为 0
           firstReader = current;   // 设置第一个读线程
           firstReaderHoldCount = 1;  // 读线程占用的资源数为 1
        } else if (firstReader == current) { // 当前线程为第一个读线程
           firstReaderHoldCount++;  // 占用资源数加 1
        } else { // 读锁数量不为 0 并且不为当前线程
           HoldCounter rh = cachedHoldCounter; // 获取计数器
           if (rh == null || rh.tid != getThreadId(current))
           // 计数器为空或者计数器的 tid 不为当前正在运行的线程的 tid
                cachedHoldCounter = rh = readHolds.get();// 获取当前线程对应的计数器
           else if (rh.count == 0) // 计数为 0
```

```
                    // 设置
                    readHolds.set(rh);
                    rh.count++;
                }
                return 1;
            }
            return fullTryAcquireShared(current);
        }
```

说明：此函数表示读锁线程获取读锁。首先判断写锁是否为 0 并且当前线程不占有独占锁，直接返回；否则，判断读线程是否需要被阻塞并且读锁数量是否小于最大值并且比较设置状态成功，若当前没有读锁，则设置第一个读线程 firstReader 和 firstReaderHoldCount；若当前线程线程为第一个读线程，则增加 firstReaderHoldCount；否则，将设置当前线程对应的 HoldCounter 对象的值。流程图如下。



3. 读写状态的设计

读写锁的自定义同步器需要在同步状态（一个整型变量）上维护**多个读线程和一个写线**程的状态。需要“按位切割使用”这个变量，读写锁将变量切分成了两个部分，高 16 位表示读，低 16 位表示写。

假设当前同步状态值为 S，读状态为 S>>>16（无符号右移 16 位）。写状态为 S & 0x0000FFFF(将高 16 位全部抹去)。当读状态增加 1 时，等于 S+(1<<16)，也就是 S+0x00010000.

4. 读锁的读取与释放

读锁能够被多个线程同时获取，**在没有其他写线程访问**（或者写状态为 0）**时，读锁总会被成功地获取**，而所做的也只是线程安全的增加读状态。

如果当前线程在获取读锁时，写锁已被其他线程获取，则进入等待状态。

写线程数不为 0 并且占有资源的不是当前线程

5. 写锁的读取与释放

如果当前线程在获取写锁时，**读锁已经被获取（读状态不为 0）或者该线程不是已经获取写锁的线程**，则当前线程进入等待状态。

该方法除了重入条件（当前线程为获取了写锁的线程）之外，增加了一个读锁是否存在的判断。如果存在读锁，则写锁不能被获取。原因在于：**读写锁要确保写锁的操作对读锁可见，如果允许在已被获取的情况下对写锁的获取，那么正在运行的其他读线程就无法感知到当前写线程的操作。因此，只有等待其他读线程都释放了读锁，写锁才能被当前线程获取，而写锁一旦被获取，则其他读写线程的后续访问均被阻塞。**

写锁的释放：每次缩放均减少写状态。当写状态为 0 时表示写锁已被释放，从而等待的读写线程能够继续访问读写锁，同时前次写线程的修改对后续写线程可见。

6. 锁降级

锁降级是指把持住（当前拥有的）写锁，再获取到读锁，随后释放（先前拥有的）写锁的过程。

只有一个线程能够获取到写锁，其他线程会被阻塞在读锁和写锁的 lock()方法上，当前线程获取写锁完成数据准备之后，可以再获取读锁，随后释放写锁，完成锁降级。

锁降级中读锁的获取是否必要呢？答案是必要的，**主要是为了保证数据的可见性**，如果当前线程不获取读锁而是直接释放写锁没，假设此刻另一个线程（基座 T）获取了写锁并修改了数据，那么当前线程无法感知线程 T 的数据更新。如果当前线程获取读锁，即遵循锁降级的步骤，则线程 T 将会被阻塞，直到当前线程使用数据并释放读锁之后，线程 T 才能获取写锁进行数据更新。

读写锁不支持锁升级（把持读锁，获取写锁，最后释放读锁的过程）。目的是保证数据可见性。如果读锁已被多个线程获取。其中任意线程成功获取了写锁并更新了数据，则其更新对其他获取到读锁的线程是不可见的。

7. 总结

经过分析 ReentrantReadWriteLock 的源码，可知其可以实现多个线程同时读，此时，写线程会被阻塞。并且，写线程获取写入锁后可以获取读取锁，然后释放写入锁，这样写入锁变成了读取锁。

6.5 Condition 接口

1. 接口方法

```
public interface Condition {
 /**
  * 使当前线程进入等待状态直到被通知(signal)或中断
  * 当其他线程调用 singal()或 singalAll()方法时，该线程将被唤醒
  * 当其他线程调用 interrupt()方法中断当前线程
  * await()相当于 synchronized 等待唤醒机制中的 wait()方法
  */
 void await() throws InterruptedException;

 //当前线程进入等待状态，直到被唤醒，该方法不响应中断要求
 void awaitUninterruptibly();

 //调用该方法，当前线程进入等待状态，直到被唤醒或被中断或超时
```

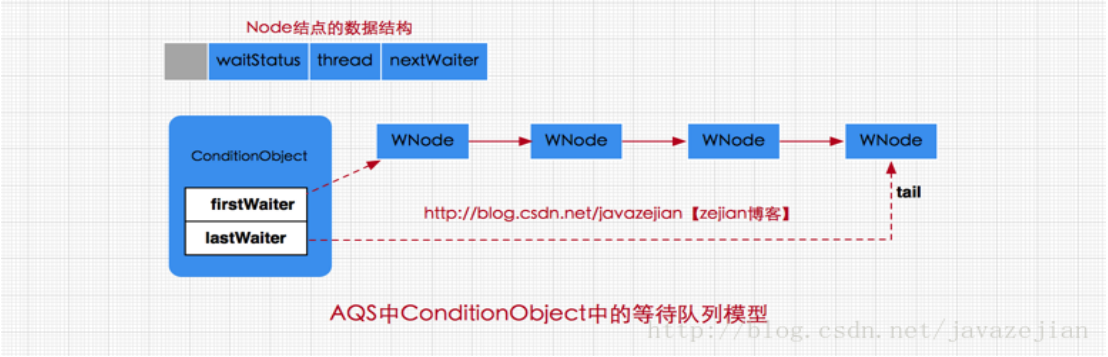
```
 //其中 nanosTimeout 指的等待超时时间，单位纳秒
 long awaitNanos(long nanosTimeout) throws InterruptedException;

  //同 awaitNanos，但可以指明时间单位
  boolean await(long time, TimeUnit unit) throws InterruptedExcepti
on;

 //调用该方法当前线程进入等待状态，直到被唤醒、中断或到达某个时
 //间期限(deadline),如果没到指定时间就被唤醒，返回 true，其他情况返回 false
  boolean awaitUntil(Date deadline) throws InterruptedException;

 //唤醒一个等待在 Condition 上的线程，该线程从等待方法返回前必须
 //获取与 Condition 相关联的锁，功能与 notify()相同
  void signal();

 //唤醒所有等待在 Condition 上的线程，该线程从等待方法返回前必须
 //获取与 Condition 相关联的锁，功能与 notifyAll()相同
  void signalAll();
}
```

2. 对比 Object 监视器

对比项	Object 监视器方法	Condition
前置条件	获取对象锁	调用 Lock.lock()获取锁 调用 Lock.newCondition()获取 condition 对象
调用方法	直接调用 例如：object.wait()	直接调用 例如：condition.await()
等待队列个数	一个	多个
当前线程释放锁并进入等待状态	支持	支持
在等待状态中不响应中断	不支持	支持
当前线程释放锁并进入超时等待状态	支持	支持
当前线程释放锁并进入等待状态到将来的某个时间	不支持	支持
唤醒等待队列中的一个/全部线程	支持	支持

3.Condition 实现分析

AQS 中存在两种队列，一种是同步队列，一种是等待队列，而等待队列就相对于 Condition 而言的。**每个 Condition 都对应着一个等待队列**，也就是说如果一个锁上创建了多个 Condition 对象，那么也就存在多个等待队列。等待队列是一个 FIFO 的队列，在队列中每一个节点都包含了一个线程的引用，而该线程就是 Condition 对象上等待的线程。当一个线程调用了 await()相关的方法，那么该线程将会释放锁，并构建一个 Node 节点封装当前线程的相关信息加入到等待队列中进行等待，直到被唤醒、中断、超时才从队列中移出。



在等待队列中使用的变量与同步队列是不同的，**Condtion 中等待队列的结点只有直接指向的后继结点并没有指明前驱结点**，而且**使用的变量是 nextWaiter 而不是 next**。firstWaiter 指向等待队列的头结点，lastWaiter 指向等待队列的尾结点，等待队列中结点的状态只有两种即 CANCELLED 和 CONDITION，前者表示线程已结束需要从等待队列中移除，后者表示条件结点等待被唤醒。再次强调每个 Codition 对象对于一个等待队列，也就是说 AQS 中只能存在一个同步队列，但可拥有多个等待队列。下面从代码层面看看被调用 await()方法(其他 await()实现原理类似)的线程是如何加入等待队列的，而又是如何从等待队列中被唤醒的。

3.1 等待

```
public final void await() throws InterruptedException {
      //判断线程是否被中断
      if (Thread.interrupted())
          throw new InterruptedException();
      //创建新结点加入等待队列并返回
      Node node = addConditionWaiter();
      //释放当前线程锁即释放同步状态
      int savedState = fullyRelease(node);
      int interruptMode = 0;
      //判断结点是否同步队列(SyncQueue)中,即是否被唤醒
      while (!isOnSyncQueue(node)) {
          //挂起线程
          LockSupport.park(this);
          //判断是否被中断唤醒，如果是退出循环。
          if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
              break;
      }
      //被唤醒后执行自旋操作争取获得锁，同时判断线程是否被中断
      if(acquireQueued(node,savedState) && interruptMode != THROW_IE)
          interruptMode = REINTERRUPT;
       // clean up if cancelled
      if (node.nextWaiter != null)
          //清理等待队列中不为 CONDITION 状态的结点
          unlinkCancelledWaiters();
      if (interruptMode != 0)
          reportInterruptAfterWait(interruptMode);
      }
```

执行 addConditionWaiter()添加到等待队列。

```
private Node addConditionWaiter() {
   Node t = lastWaiter;
     // 判断是否为结束状态的结点并移除
     if (t != null && t.waitStatus != Node.CONDITION) {
         unlinkCancelledWaiters();
         t = lastWaiter;
     }
     //创建新结点状态为 CONDITION
     Node node = new Node(Thread.currentThread(), Node.CONDITION);
     //加入等待队列
     if (t == null)
         firstWaiter = node;
     else
         t.nextWaiter = node;
     lastWaiter = node;
     return node;
       }
```

await()方法主要做了 3 件事，一是**同步队列的首节点调用 addConditionWaiter()方法将当前线程封装成 node 结点加入等待队列**，二是调用 **fullyRelease(node)方法释放同步状态并唤醒后继结点的线程，当前线程会进入等待状态**。三是调用 isOnSyncQueue(node)方法判断结点是否在同步队列中，注意是个 while 循环，如果同步队列中没有该结点就直接挂起该线程，需要明白的是如果线程被唤醒后就调用 acquireQueued(node, savedState)执行自旋操作争取锁，即当前线程结点从等待队列转移到同步队列并开始努力获取锁。

3.2 通知

调用 Condition 的 signal ()方法，将会唤醒在等待队列中等待时间最长的节点（首节点），在唤醒节点之前，会将节点移动到同步队列中。

```
public final void signal() {
    //判断是否持有独占锁，如果不是抛出异常【当前线程必须是获取了锁的线程】
  if (!isHeldExclusively())
         throw new IllegalMonitorStateException();
     Node first = firstWaiter;
     //唤醒等待队列第一个结点的线程
     if (first != null)
         doSignal(first);
}
```

这里 signal()方法做了两件事，一是**判断当前线程是否持有独占锁**，没有就抛出异常，从这点也可以看出只有独占模式先采用等待队列，而共享模式下是没有等待队

列的，也就没法使用 Condition。二是唤醒等待队列的第一个结点，即执行 doSignal(first)，首节点移动到同步队列并使用 LockSupport 唤醒节点中的线程。

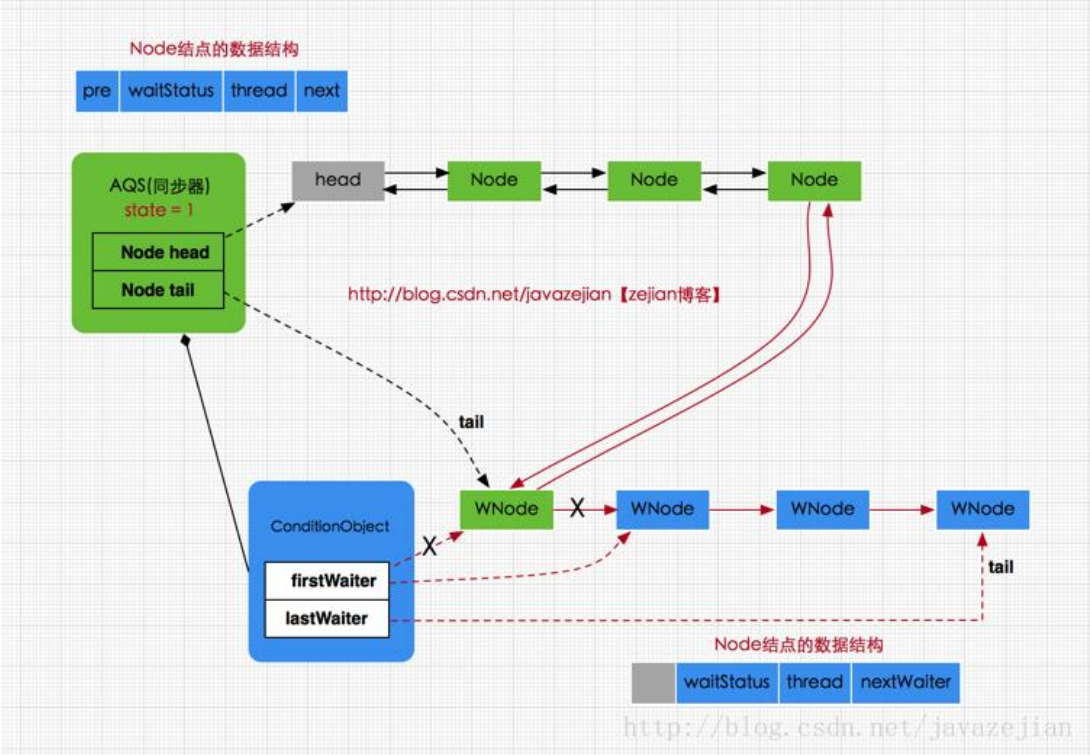
```
 private void doSignal(Node first) {
     do {
         //移除条件等待队列中的第一个结点，
         //如果后继结点为 null，那么说没有其他结点将尾结点也设置为 null
         if ( (firstWaiter = first.nextWaiter) == null)
              lastWaiter = null;
         first.nextWaiter = null;
          //如果被通知节点没有进入到同步队列并且条件等待队列还有不为空的节点，则继续循环通知后续结点
     } while (!transferForSignal(first) &&
          (first = firstWaiter) != null);
     }
//transferForSignal 方法
final boolean transferForSignal(Node node) {
    //尝试设置唤醒结点的 waitStatus 为 0，即初始化状态
    //如果设置失败，说明当期结点 node 的 waitStatus 已不为
    //CONDITION 状态，那么只能是结束状态了，因此返回 false
    //返回 doSignal()方法中继续唤醒其他结点的线程，注意这里并
    //不涉及并发问题，所以 CAS 操作失败只可能是预期值不为 CONDITION，
    //而不是多线程设置导致预期值变化，毕竟操作该方法的线程是持有锁的。
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
         return false;
        //加入同步队列并返回前驱结点 p
        Node p = enq(node);
        int ws = p.waitStatus;
        //判断前驱结点是否为结束结点(CANCELLED=1)或者在设置
        //前驱节点状态为 Node.SIGNAL 状态失败时，唤醒被通知节点代表的线程
        if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
            //唤醒 node 结点的线程
            LockSupport.unpark(node.thread);
        return true;
    }
```

doSignal(first)方法中做了两件事，一是从条件等待队列移除被唤醒的节点，然后重新维护条件等待队列的 firstWaiter 和 lastWaiter 的指向。二是将从等待队列移除的结点加入同步队列(在 transferForSignal()方法中完成的)，如果进入到同步队列失败并且条件等待队列还有不为空的节点，则继续循环唤醒后续其他结点的线程。

整个 signal()的唤醒过程，即 signal()被调用后，先判断当前线程是否持有独占锁，如果有，那么唤醒当前 Condition 对象中等待队列的第一个结点的线程，并通过调用同步器的 enq 方法，等待队列中的头结点线程安全地移动到同步队列中，如果

加入同步队列失败，那么继续循环唤醒等待队列中的其他结点的线程，如果成功加入同步队列，那么如果其前驱结点是否已结束或者设置前驱节点状态为 Node.SIGNAL 状态失败，则通过 `LockSupport.unpark()`唤醒被通知节点代表的线程，到此 `signal()`任务完成。注意被唤醒后的线程，将从前面的 `await()`方法中的 while 循环中退出，因为此时该线程的结点已在同步队列中，那么 `while (!isOnSyncQueue(node))`将不在符合循环条件，进而调用 AQS 的 `acquireQueued()`方法加入获取同步状态的竞争中。

Condition 的 signalAll()方法，相当于对等待队列中的每个节点均执行一次 signal()方法，效果就是讲等待队列中所有节点全部移动待同步队列中，并唤醒每个节点的线程。



6.6 信号量-Semaphore

信号量维护了一个许可集，我们在初始化 Semaphore 时需要为这个许可集传入一个数量值，**该数量值代表同一时间能访问共享资源的线程数量**。线程可以通过 acquire()方法获取到一个许可，然后对共享资源进行操作，注意如果许可集已分配完了，那么线程将进入等待状态，直到其他线程释放许可才有机会再获取许可，线程释放一个许可通过 release()方法完成。

1．实现内部原理概要

信号量 Semaphore 的类结构与 ReetrantLock 的类结构几乎如出一辙。Semaphore 内部同样存在继承自 AQS 的内部类 Sync 以及继承自 Sync 的公平锁(FairSync)和非公平锁(NofairSync)，从这点也足以说明 Semaphore 的内部实现原理也是基于 AQS 并发组件的，AQS 是基础组件，

只负责核心并发操作，如加入或维护同步队列，控制同步状态，等，而具体的**加锁和解锁操作交由子类完成**，因此子类 Semaphore 共享锁的获取与释放需要自己实现，这两个方法分别是获取锁的 tryAcquireShared(int arg)方法和释放锁的 tryReleaseShared(int arg)方法. Semaphore 的内部类公平锁(FairSync)和非公平锁(NoFairSync)各自实现不同的获取锁方法即 tryAcquireShared(int arg)，毕竟公平锁和非公平锁的获取稍后不同，而释放锁 tryReleaseShared(int arg)的操作交由 Sync 实现，因为释放操作都是相同的

2. 非公平锁中的共享锁

```
    //默认创建公平锁，permits 指定同一时间访问共享资源的线程数
    public Semaphore(int permits) {
        sync = new NonfairSync(permits);
    }
```

在 AQS 中存在一个变量 state，当我们创建 Semaphore 对象传入许可数值时，最终会赋值给 state，state 的数值代表同一个时刻可同时操作共享数据的线程数量，每当一个线程请求(如调用 Semaphored 的 acquire()方法)获取同步状态成功，state 的值将会减少 1，直到 state 为 0 时，表示已没有可用的许可数，也就是对共享数据进行操作的线程数已达到最大值，其他后来线程将被阻塞，此时 AQS 内部会将线程封装成共享模式的 Node 结点，加入同步队列中等待并开启自旋操作。只有当持有对共享数据访问权限的线程执行完成任务并释放同步状态后，同步队列中的对于的结点线程才有可能获取同步状态并被唤醒执行同步操作，注意在同步队列中获取到同步状态的结点将被设置成 head 并清空相关线程数据(毕竟线程已在执行也就没有必要保存信息了)。

```
//Semaphore 的 acquire()
public void acquire() throws InterruptedException {
     sync.acquireSharedInterruptibly(1);
  }
public void acquireUninterruptibly(int permits) {
        if (permits < 0) throw new IllegalArgumentException();
        sync.acquireShared(permits);
    }
/**
*  注意 Sync 类继承自 AQS
*  AQS 的 acquireSharedInterruptibly()方法
*/
public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {
    //判断是否中断请求
    if (Thread.interrupted())
        throw new InterruptedException();
    //如果 tryAcquireShared(arg)不小于 0，则线程获取同步状态成功
    if (tryAcquireShared(arg) < 0)
        //未获取成功加入同步队列等待
        doAcquireSharedInterruptibly(arg);
    }
```

Semaphore 的 acquire()方法也是可中断的。**首先进行线程中断的判断**，如果没有中断，那么先尝试调用实现者 Sync 的 tryAcquireShared(arg)【首先获取 state，减去 1，直到信号量

是否已小于 0 或者 CAS 执行成功返回 int 信号量】方法**获取同步状态是否小于 0**。如果获取成功【大于 0】，那么方法执行结束，如果获取失败，则创建一个共享模式（Node.SHARED）的结点并加入同步队列，加入完成后，当前线程进入自旋状态，首先判断前驱结点是否为 head：如果是，那么尝试获取同步状态并返回信号量 r 值，如果 r 大于 0，则说明获取同步状态成功，将当前线程设置为 head，并通知后续结点继续获取同步状态。如果前驱结点不为 head 或前驱结点为 head 并尝试获取同步状态失败，那么判断前驱结点的 waitStatus 值是否为 SIGNAL【等待被唤醒状态】并调整同步队列中的 node 结点状态，如果返回 true，那么将当前线程挂起并返回是否中断线程的 flag。

不可中断的获取锁

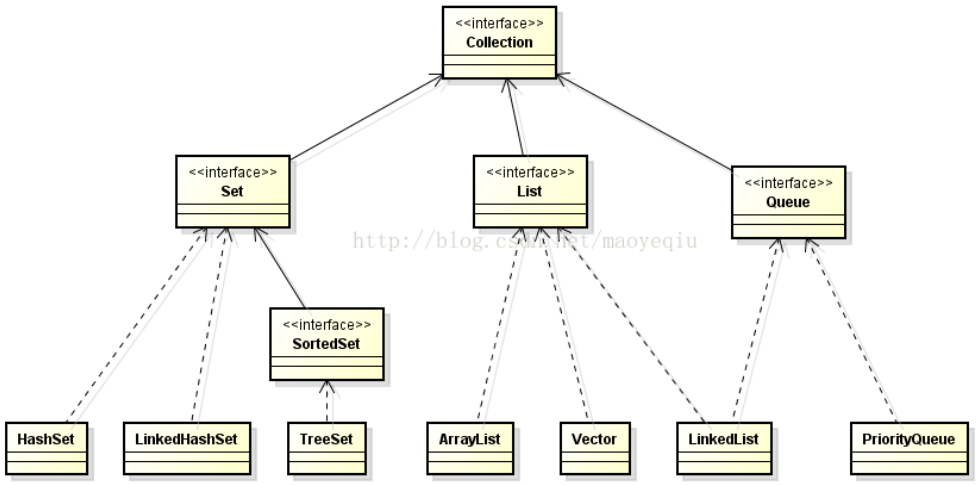
与带中断请求方法不同的是少了线程中断的判断以及异常抛出，其他操作都一样。

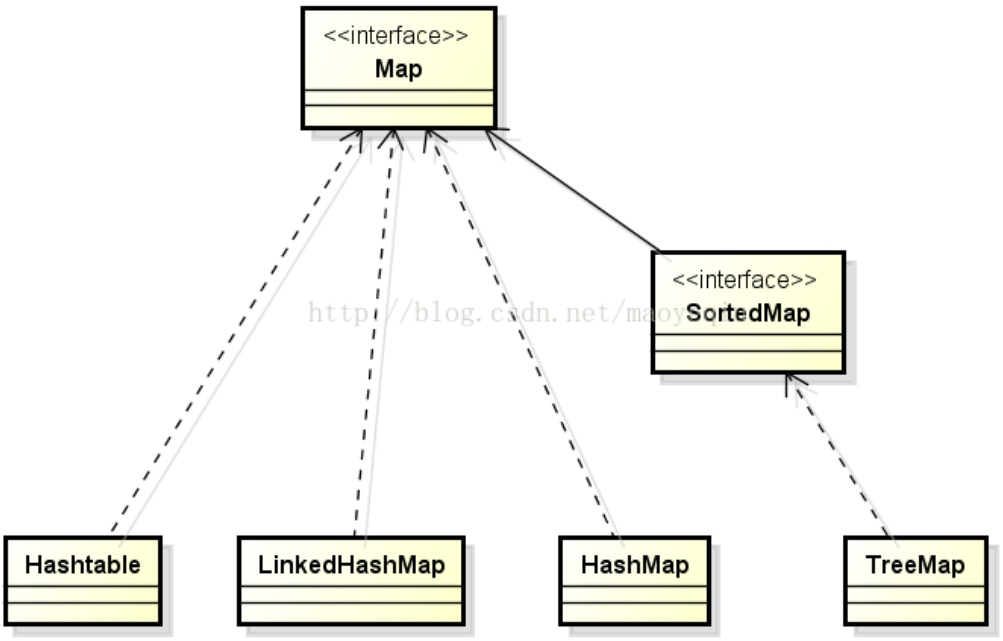
释放锁

Semaphore 调用了 AQS 中的 releaseShared(int arg)方法。通过 tryReleaseShared(arg)方法【释放同步状态，更新 state 的值，这里是无锁操作，即 for 死循环和 CAS 操作来保证线程安全问题】尝试释放同步状态，如果释放成功，那么将调用 doReleaseShared()唤醒同步队列中后继结点。

doReleaseShared()方法中通过调用 unparkSuccessor(h)方法唤醒 head 的后继结点对应的线程。注意这里把 head 的状态设置为 Node.PROPAGATE 是【在共享模式中使用表示获得的同步状态会被传播】为了保证唤醒传递。

第七章 Java 并发容器和框架





HashSet:

HashSet 继承 AbstractSet 类，实现 Set、Cloneable、Serializable 接口。 Set 接口是一种不包括重复元素的 Collection，它维持它自己的内部排序。

```
1.   * 默认构造函数
2.           * 初始化一个空的 HashMap，并使用默认初始容量为 16 和加载因子 0.75。
3.           */
4.          public HashSet() {
5.              map = new HashMap<>();
6.          }
```

LinkedHashSet:

TreeSet:

ArrayList:

LinkedList:

HashMap:

LinkedHashMap:

HashTable:

TreeMap:

7.1 HashMap【1.7】

1. HashMap 的数据结构

HashMap 实际上是一个“链表散列”的数据结构，即数组和链表的结合体。HashMap 底层就是一个数组结构，数组中的每一项又是一个链表。**Entry 就是数组中的元素**，每个 Map.Entry 其实就是一个 key-value 对，它持有一个指向下一个元素的引用，这就构成了链表。**HashMap 允许存放 null 键和 null 值**。

2. HashMap 的存取实现

```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;
    final int hash;  }
```

HashMap 中 put 元素的时候，首先根据该 key 的 hashCode() 返回值决定该 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() 返回值相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry 的 value，但 key 不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，那么在这个位置上的元素将以链表的形式存放，**新加入的放在链头**，最先加入的放在链尾。新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部。

3. HashMap 的 resize（rehash）

当 HashMap 中的元素个数超过数组大小*loadFactor 时，就会进行数组扩容，loadFactor 的默认值为 0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为 16，那么当 HashMap 中元素个数超过 16*0.75=12 的时候，就把数组的大小扩展为 2*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作。

4. HashMap 的死循环

多线程同时 put 时，如果同时触发了 rehash 操作，会导致 HashMap 中的链表中出现循环节点，进而使得后面 get 的时候，会死循环。

Put一个Key,Value对到Hash表中：

```
public V put(K key, V value)
{
    ......
    //算Hash值
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    //如果该key已被插入，则替换掉旧的value （链接操作）
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    //该key不存在，需要增加一个结点
    addEntry(hash, key, value, i);
    return null;
}
```

检查容量是否超标

```
void addEntry(int hash, K key, V value, int bucketIndex)
{
    Entry<K,V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    //查看当前的size是否超过了我们设定的阈值threshold，如果超过，需要resize
    if (size++ >= threshold)
        resize(2 * table.length);
}
```

新建一个更大尺寸的hash表，然后把数据从老的Hash表中迁移到新的Hash表中。

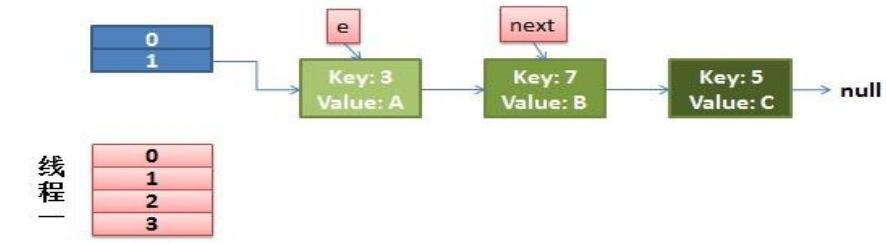
```
void resize(int newCapacity)
{
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    ......
    //创建一个新的Hash Table
    Entry[] newTable = new Entry[newCapacity];
    //将Old Hash Table上的数据迁移到New Hash Table上
    transfer(newTable);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);
}
```

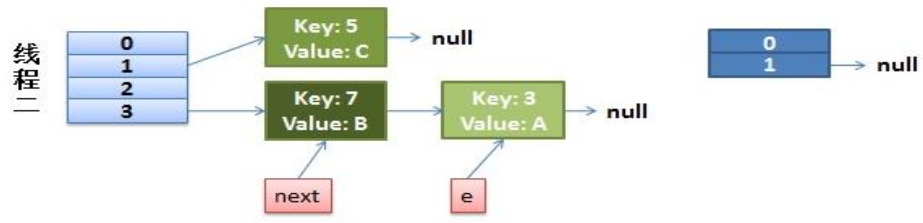
```java
void transfer(Entry[] newTable)
{
    Entry[] src = table;
    int newCapacity = newTable.length;
    //下面这段代码的意思是：
    //  从OldTable里摘一个元素出来，然后放到NewTable中
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
            src[j] = null;
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
                e = next;
            } while (e != null);
        }
    }
}
```

(1)线程 1、2 都进入 transfer 第三行,此时 map 为：

(2)线程 1 执行到【int i = indexFor(e.hash, newCapacity);】，线程 2 执行到【src[j] = null;】，此时 map 为：

(3) 直接把 Thread2 执行完毕，此时 map 的内容为：

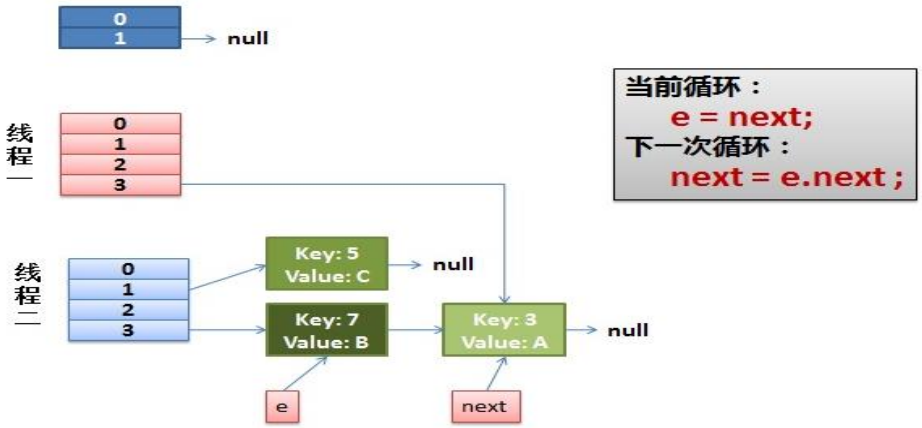




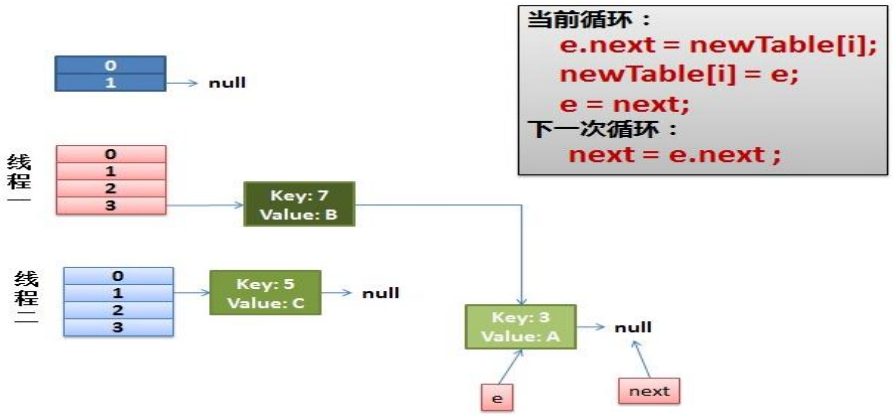
注意：因为 Thread1 的 **e 指向了 key(3)，而 next 指向了 key(7)**，其在线程二 rehash 后，指

向了线程二重组后的链表。我们可以看到**链表的顺序被反转后**。

(4)切换回 Thread1，先是执行 newTalbe[i] = e;然后是 e = next，导致了 **e 指向了 key(7)**，而下一次循环的 next = e.next 导致了 **next 指向了 key(3)**，此时 map 的内容为：



(5) Thread1 第 2 次循环后，把 key(7)摘下来，放到 newTable[i]的第一个，然后把 e 和 next 往下移。

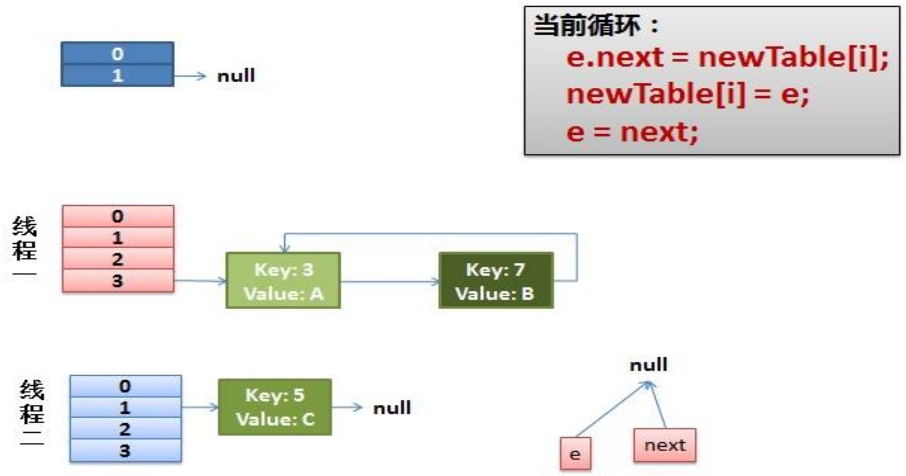


(5)环形链接出现。

e.next = newTable[i] 导致 key(3).next 指向了 key(7)

注意：此时的 key(7).next 已经指向了 key(3)， 环形链表就这样出现了。

于是，当我们的线程一调用到，HashTable.get(11)时，悲剧就出现了——Infinite Loop。



7.2 HashMap【1.8】

1.8 中 hashmap **声明两对指针，维护两个连链表**，依次在末端添加新的元素。因此不会因为多线程 put 导致死循环，但是依然有其他的弊端，比如数据丢失等等。因此多线程情况下还是建议使用 concurrenthashmap。

HashMap 采用的是**数组+链表+红黑树**的形式。数组是可以扩容的，链表也是转化为红黑树的。用户可以设置的参数：初始总容量默认 16，默认的加载因子 0.75。初始的数组个数默认是 16 容量 X 加载因子=阈值。一旦目前容量超过该阈值，则执行扩容操作。

HashMap 类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个 Node 的数组。Node 是 HashMap 的一个内部类，实现了 Map.Entry 接口，本质是就是一个映射(键值对)。

什么时候扩容？

1 当前容量超过阈值

2 当链表中元素个数超过默认设定（8 个），当数组的大小还未超过 64 的时候，此时进行数组的扩容，如果超过则将链表转化成红黑树

当**数组大小已经超过 64 并且链表中的元素个数超过默认设定（8 个）时，将链表转化为红黑树**。

1. Put 过程

- 根据 key 计算出 hash 值
- hash 值&（数组长度-1）得到所在数组的 index
  - 如果该 index 位置的 Node 元素不存在，则直接创建一个新的 Node
  - 如果该 index 位置的 Node 元素是 TreeNode 类型即红黑树类型了，则直接按照红黑树的插入方式进行插入
  - 如果该 index 位置的 Node 元素是非 TreeNode 类型则，则按照链表的形式进行插入操作

    链表插入操作完成后，判断是否超过阈值 TREEIFY_THRESHOLD（默认是 8），超过则要么数组扩容要么链表转化成红黑树
- 判断当前总容量是否超出阈值，如果超出则执行扩容

2. 扩容过程

按照 **2 倍**扩容的方式。这里为啥是 2 倍？因为 2 倍的话，更加容易计算他们所在的桶，并且各自不会相互干扰。桶 0 中的元素会被分到桶 0 和桶 4 中。

扩充 HashMap 的时候，不需要像 JDK1.7 的实现那样重新计算 hash，只需要看看原来的 hash 值新增的那个 bit 是 1 还是 0 就好了，是 0 的话索引没变，是 1 的话索引变成“原索引+oldCap”，JDK1.7 中 rehash 的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8 不会倒置。

3. get 过程

根据 key 计算 hash 值：计算出 hashCode 值，hash=hashCode^(hashCode>>>16);
hash 值&（数组长度-1）得到所在数组的 index

如果要找的 key 就是上述数组 index 位置的元素，直接返回该元素的值

如果该数组 index 位置元素是 TreeNode 类型，则按照红黑树的查询方式来进行查找 O(logn)

如果该数组 index 位置元素非 TreeNode 类型，则按照链表的方式来进行遍历查询 O(n)

4. 性能

随着 size 的变大，JDK1.7 的花费时间是增长的趋势，而 JDK1.8 是明显的降低趋势，并且呈现对数增长稳定。当一个**链表太长的时候，HashMap 会动态的将它替换成一个红黑树**，这样的话会将时间复杂度从 O(n)降为 O(logn)。

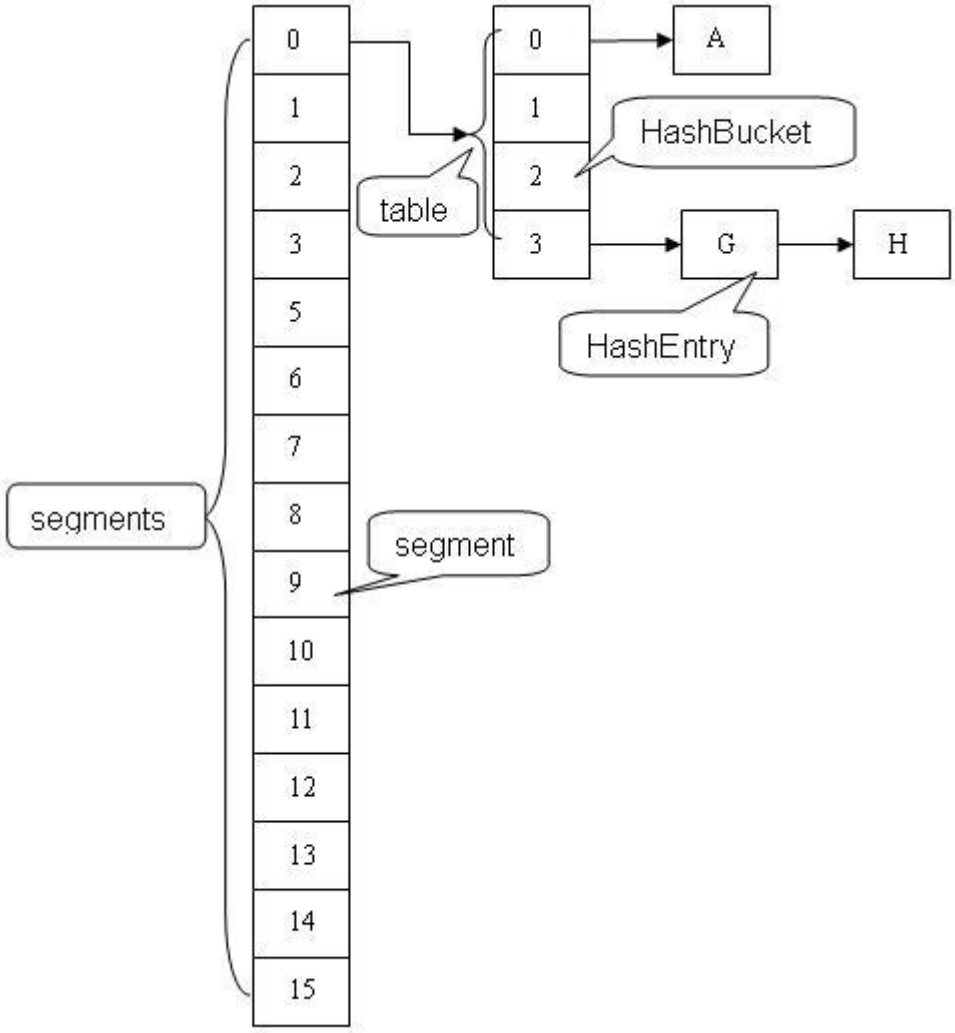
7.3 HashTable

HashTable 是一个线程安全的类，它使用 synchronized 来锁住整张 Hash 表来实现线程安全，即每次锁住整张表让线程独占。如线程 1 使用 put 进行元素添加，线程 2 不但不能使用 put 添加元素，也不能使用 get 获取元素。hashTable 不允许 key 和 value 为 null。

ConcurrentHashMap 允许多个修改操作并发进行，其关键在于使用了锁分离技术。它使用了多个锁来控制对 hash 表的不同部分进行的修改。ConcurrentHashMap 内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的 Hashtable，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。

有些方法需要跨段，比如 size()和 containsValue()，它们可能需要锁定整个表而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在 ConcurrentHashMap 内部，段数组是 final 的，并且其成员变量实际上也是 final 的，但是，仅仅是将数组声明为 final 的并不保证数组成员也是 final 的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

7.4 ConcurrentHashMap【1.7】



1. 结构

ConcurrentHashMap 采用了非常精妙的"分段锁"策略，ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成，主干是个 Segment 数组。**Segment 继承了 ReentrantLock，**

所以它就是一种可重入锁（ReentrantLock)。HashEntry 则用于存储键值对数据。

在 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和 HashMap 类似，是一种数组和链表结构。一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，Segment 里维护了一个 HashEntry 数组。当对 HashEntry 数组的数据进行修改时，必须首先获得与它对应的 Segment 锁。(就按默认的 ConcurrentLeve 为 16 来讲，理论上就允许 16 个线程并发执行)。

2.实现原理

ConcurrentHashMap 使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。

ConcurrentHashMap 内部分为很多个 Segment，Segment 继承了 ReentrantLock，表明每个 segment 都可以当做一个锁。对于一个 key，需要经过三次（）hash 操作，才能最终定位这个元素的位置，这三次 hash 分别为：

- 对于一个 key，先进行一次 hash 操作，得到 hash 值 h1，也即 h1 = hash1(key)；
- 将得到的 h1 的高几位进行第二次 hash，得到 hash 值 h2，也即 h2 = hash2(h1 高几位)，通过 **h2 能够确定该元素的放在哪个 Segment**；
- 将得到的 h1 进行第三次 hash，得到 hash 值 h3，也即 h3 = hash3(h1)，通过 h3 能够**确定该元素放置在哪个 HashEntry**。

ConcurrentHashMap 实现技术是保证 HashEntry 几乎是不可变的。HashEntry 代表每个 hash 链中的一个节点，其结构如下所示：

```
static final class HashEntry<K,V> {
    final K key;
    final int hash;
    volatile V value;
    volatile HashEntry<K,V> next;
 }
```

2.1 初始化 Segment

```
Segment(float lf, int threshold, HashEntry<K,V>[] tab) {
    this.loadFactor = lf;//负载因子
    this.threshold = threshold;//阈值
    this.table = tab;//主干数组即 HashEntry 数组
 }
```

2.2 初始化 ConcurrentHashMap

```
public ConcurrentHashMap(int initialCapacity, float loadFactor, int
concurrencyLevel) {
    … …
    //MAX_SEGMENTS 为 1<<16=65536，也就是最大并发数为 65536
    if (concurrencyLevel > MAX_SEGMENTS)
       concurrencyLevel = MAX_SEGMENTS;
    //2 的 sshif 次方等于 ssize，例:ssize=16,sshift=4;ssize=32,sshif=5
    int sshift = 0;
```

```
    int ssize = 1;
    while (ssize < concurrencyLevel) {
        ++sshift;
        ssize <<= 1;
    }
    this.segmentShift = 32 - sshift;
    this.segmentMask = ssize - 1;
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    //计算 cap 的大小，即 Segment 中 HashEntry 的数组长度，cap 也一定为 2 的 n 次方.
    int c = initialCapacity / ssize;
    if (c * ssize < initialCapacity)
        ++c;
    int cap = MIN_SEGMENT_TABLE_CAPACITY;
    while (cap < c)
      cap <<= 1;
    … …
}
```

传入的参数有 initialCapacity，loadFactor，concurrencyLevel 这三个。

- initialCapacity 表示新创建的这个 ConcurrentHashMap 的初始容量，也就是上面的结构图中的 **Entry 数量**。默认值为 static final int DEFAULT_INITIAL_CAPACITY = 16;
- loadFactor 表示**负载因子**，就是当 ConcurrentHashMap 中的元素个数大于 loadFactor * 最大容量时就需要 rehash，扩容。默认值为 static final float DEFAULT_LOAD_FACTOR = 0.75f;
- concurrencyLevel 表示**并发级别**，这个值用**来确定 Segment 的个数**。
- **Segment 数组的大小 ssize** 是由 concurrentLevel 来决定的，但是却不一定等于 concurrentLevel，**ssize 一定是大于或等于 concurrentLevel 的最小的 2 的 N 次幂。**比如：默认情况下 concurrentLevel 是 16，则 ssize 为 16；若 concurrentLevel 为 14，ssize 为 16；若 concurrentLevel 为 17，则 ssize 为 32。为什么 Segment 的数组大小一定是 2 的次幂？其实主要是便于通过按位与的散列算法来定位 Segment 的 index。
- segmentMask：段掩码，假如 **ssize** 为 16，则段掩码为 16-1=15；segments 长度为 32，段掩码为 32-1=31。这样得到的所有 bit 位都为 1，可以更好地保证散列的均匀性 segmentShift：2 的 sshift 次方等于 ssize，segmentShift=32-sshift。若 segments 长度为 16，segmentShift=32-4=28。这里之所以用 32 是因为 ConcurrentHashMap 里的 hash()方法输出的最大数是 32 位的。无符号右移 segmentShift，则意味着只保留高几位（其余位是没用的），然后与段掩码 segmentMask 位运算来定位 Segment。
- Cap 就是 **segment 里 HashEntry 数组的长度**。计算每个 Segment 平均应该放置多少个元素，**这个值 c 是向上取整的值**。比如初始容量为 15，Segment 个数为 4，则每个 Segment 平均需要放置 4 个元素。Cap 最小是 2 或者大于等于 C 的最小的 2 的 N 次幂。
- Segment 的容量 threshold=(int)cap*loadFactor,默认情况下，并发级别 concurrentLevel=16，ssize=16，初始容量 initialCapacity=16，负载因子 loadFactor=0.75，通过运算 cap=1，threshold = 0。

2.3 定位 segment

在插入和获取数据元素的时候，必须先通过散列算法定位到 segment。首先使用 Wang/Jenkins hash 的变种算法对元素的 hashCode 进行一次再散列。之所以进行在散列，目的是减少散列冲突，使元素能够均匀地分布在不同的 Segment 上，从而提高容器的存取效率。如果不适用再散列，散列冲突会非常严重，只要地位一样，无论高位是什么数，起散列值总是一样。使用在散列，可以把每一位的数据都散列开了，通过这种再散列能让数字的每一位都参加到散列运算中，从而减少散列冲突。

3 put 操作

```
public V put(K key, V value) {
   Segment<K,V> s;
   if (value == null)
       throw new NullPointerException();
   int hash = hash(key);
   int j = (hash >>> segmentShift) & segmentMask;
   if ((s = (Segment<K,V>)UNSAFE.getObject (segments, (j << SSHIFT) + SBASE)) == null)
      s = ensureSegment(j);
   return s.put(key, hash, value, false);
}
```

操作步骤如下：

- 判断 value 是否为 null，如果为 null，直接抛出异常。
- key 通过一次 hash 运算得到一个 hash 值。
- 将得到 hash 值向右按位移动 segmentShift 位，然后再与 segmentMask 做&运算得到 segment 的索引 j。默认 segmentShift=28，segmentMask=15，hash 值向右移动 28 位就变成这个样子：0000 0000 0000 0000 0000 0000 0000 xxxx，然后再用这个值与 segmentMask 做&运算，也就是**取最后四位的值。这个值确定 Segment 的索引**。即 hash 值的最高 sshift 位。
- 使用 **Unsafe 的方式**从 Segment 数组中获取该索引对应的 Segment 对象。
- 调用 Segment 的 put 方法【put 操作是要加锁的】向这个 Segment 对象中 put 值，这个 put 操作也基本是一样的步骤（通过&运算获取 HashEntry 的索引，然后 set）。

4. get 操作

```
    public V get(Object key) {
        Segment<K,V> s;
        HashEntry<K,V>[] tab;
        int h = hash(key);
        long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;
        if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
            (tab = s.table) != null) {
            for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile
                    (tab, ((long)(((tab.length - 1) & h)) << TSHIFT) + TBASE);
                 e != null; e = e.next) {
                K k;
                if ((k = e.key) == key || (e.hash == h && key.equals(k)))
                    return e.value;
            }
```

```
        }
        return null;
    }
```

操作步骤为：

- 和 put 操作一样，先通过 key 进行**两次 hash 确定应该去哪个 Segment** 中取数据。
- 使用 **Unsafe 获取对应的 Segment**，然后再进行一次&运算得到 HashEntry 链表的位置，然后从链表头开始遍历整个链表（因为 Hash 可能会有碰撞，所以用一个链表保存），如果找到对应的 key，则返回对应的 value 值，如果链表遍历完都没有找到对应的 key，则说明 Map 中不包含该 key，返回 null。get 操作是不需要加锁的（如果 value 为 null，会调用 readValueUnderLock，只有这个步骤会加锁），通过前面提到的 volatile 和 final 来确保数据安全。

5. size 操作

size 操作与 put 和 get 操作最大的区别在于，**size 操作需要遍历所有的 Segment 才能算出整个 Map 的大小**，而 put 和 get 都只关心一个 Segment。假设我们当前遍历的 Segment 为 SA，那么在遍历 SA 过程中其他的 Segment 比如 SB 可能会被修改，于是这一次运算出来的 size 值可能并不是 Map 当前的真正大小。牛逼的作者还有一个更好的 Idea：**先给 3 次机会，不 lock 所有的 Segment**，遍历所有 Segment，累加各个 Segment 的大小得到整个 Map 的大小，如果**某相邻的两次计算获取的所有 Segment 的更新的次数**（每个 Segment 都有一个 modCount 变量，这个变量在 Segment 中的 Entry 被修改时会加一，通过这个值可以得到每个 Segment 的更新操作的次数）**是一样的**，说明计算过程中没有更新操作，则直接返回这个值。如果这三次不加锁的计算过程中 Map 的更新次数有变化，则之后的计算先对所有的 Segment 加锁，再遍历所有 Segment 计算 Map 大小，最后再解锁所有 Segment。

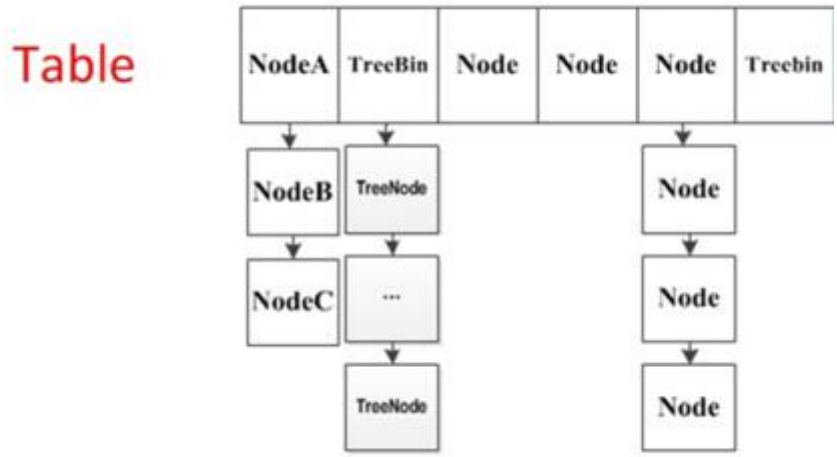
```
for (;;) {
  if (retries++ == RETRIES_BEFORE_LOCK) { //2
      for (int j = 0; j < segments.length; ++j)
           ensureSegment(j).lock(); // force creation
  }
  sum = 0L;
  size = 0;
  overflow = false;
  for (int j = 0; j < segments.length; ++j) {
       Segment<K,V> seg = segmentAt(segments, j);
       if (seg != null) {
           sum += seg.modCount;
           int c = seg.count;
           if (c < 0 || (size += c) < 0)
                overflow = true;
        }
  }
  if (sum == last)
      break;
  last = sum;
}
```

containsValue 操作采用了和 size 操作一样的想法。

6. 注意事项

- ConcurrentHashMap 中的 key 和 value 值都不能为 null，HashMap 中 key 可以为 null，HashTable 中 key 不能为 null。
- ConcurrentHashMap 是线程安全的类并不能保证使用了 ConcurrentHashMap 的操作都是线程安全的！
- ConcurrentHashMap 的 get 操作不需要加锁，put 操作需要加锁

7.5 ConcurrentHashMap【1.8】



1.8 的实现已经抛弃了 Segment 分段锁机制，利用 CAS+Synchronized 来保证并发更新的安全，底层依然采用数组+链表+红黑树的存储结构。

主要设计上的变化有以下几点:

- JDK1.8 的实现已经摒弃了 Segment 的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，锁住 node 来实现减小锁粒度。如果还是采用单向列表方式，那么查询某个节点的时间复杂度为 O(n)；因此，对于个数超过 8(默认值)的列表，jdk1.8 中采用了红黑树的结构，那么查询的时间复杂度可以降低到 O(logN)，可以改进性能。
- 设计了 MOVED 状态 当 resize 的中过程中 线程 2 还在 put 数据，线程 2 会帮助 resize。
- 使用 3 个 CAS 操作来确保 node 的一些操作的原子性，这种方式代替了锁。
- sizeCtl 的不同值来代表不同含义，起到了控制的作用。
- 并发控制使用 Synchronized 和 CAS 来操作。至于为什么 JDK8 中使用 synchronized 而不是 ReentrantLock，可能是 JDK8 中对 synchronized 有了足够的优化吧。
- 在其他方面也有一些小的改进，比如新增字段 transient volatile CounterCell[] counterCells; 可方便的计算 hashmap 中所有元素的个数，性能大大优于 jdk1.7 中的 size() 方法。

1. 属性

下面看一下基本属性：

```
// node 数组最大容量
private static final int MAXIMUM_CAPACITY = 1 << 30;
// 默认初始值，必须是 2 的幕数
private static final int DEFAULT_CAPACITY = 16;
// 负载因子
private static final float LOAD_FACTOR = 0.75f;
// 链表转红黑树阀值,> 8 链表转换为红黑树
static final int TREEIFY_THRESHOLD = 8;
//树转链表阀值，小于等于 6
static final int UNTREEIFY_THRESHOLD = 6;
private transient volatile int sizeCtl;
```

2. 重要概念

在开始之前，有些重要的概念需要介绍一下：

1. **table**：默认为 null，**初始化发生在第一次插入操作**，默认大小为 16 的数组，用来存储 Node 节点数据，扩容时大小总是 2 的幂次方。
2. **nextTable**：默认为 null，扩容时新生成的数组，其大小为原数组的两倍。
3. **sizeCtl** ：默认为 0，用来控制 table 的初始化和扩容操作，具体应用在后续会体现出来。
   - -1 **代表 table 正在初始化**
   - -N **表示有 N-1 个线程正在进行扩容操作**
   - 其余情况：

     1、如果 table 未初始化，表示 table 需要初始化的大小。

     2、如果 table 初始化完成，表示 table 的容量，默认是 table 大小的 0.75 倍，居然用这个公式算 0.75（n - (n >>> 2)）。
4. **Node**：保存 key，value 及 key 的 hash 值的数据结构。

```
class Node<K,V> implements Map.Entry<K,V> {
  final int hash;
  final K key;
  volatile V val;
  volatile Node<K,V> next;
  ... 省略部分代码
 }
```

其中 value 和 next 都用 volatile 修饰，保证并发的可见性。

5. **ForwardingNode**：一个特殊的 Node 节点，hash 值为-1，其中存储 nextTable 的引用。只有 table 发生扩容的时候，ForwardingNode 才会发挥作用，作为一个占位符放在 table 中表示当前节点为 null 或则已经被移动。

ConcurrentHashMap 在构造函数中只会初始化 sizeCtl 值，并不会直接初始化 table，而是延缓到第一次 put 操作。

3. 实例初始化

实例化 ConcurrentHashMap 时带参数时，会根据参数调整 table 的大小，假设参数为 **100，最终会调整成 256**，确保 **table 的大小总是 2 的幂次方**.

```
tableSizeFor(initialCapacity + a + 1));
private static final int tableSizeFor(int c) {
```

```
    int n = c - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    int resule = (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
    return resule;
}
```

4. table 初始化

table 初始化操作**会延缓到第一次 put 行为**。但是 put 是可以并发执行的，Doug Lea 是如何实现 table 只初始化一次的？sizeCtl 默认为 0，执行第一次 put 操作的线程会执行 Unsafe.compareAndSwapInt 方法修改 sizeCtl 为-1，有且只有一个线程能够修改成功，其它线程通过 Thread.yield()让出 CPU 时间片等待 table 初始化完成。如果一个线程发现 sizeCtl<0，意味着另外的线程执行 CAS 操作成功，当前线程只需要让出 cpu 时间片。

5. **put 操作**

put 操作采用 CAS+synchronized 实现并发插入或更新操作，具体实现如下。

- **hash 算法**

  ```
  int hash = spread(key.hashCode());
  ```

- **table 中定位索引位置，n 是 table 的大小**

  ```
  int index = (n - 1) & hash
  ```

- **获取 table 中对应索引的元素 f。**

Doug Lea 采用 Unsafe.getObjectVolatile 来获取。虽然 table 是 volatile 修饰的，但不能保证线程每次都拿到 table 中的最新元素，Unsafe.getObjectVolatile 可以直接获取指定内存的数据，保证了每次拿到数据都是最新的。

- 如果 f 为 null，说明 table 中这个位置第一次插入元素，利用 **Unsafe.compareAndSwapObject** 方法插入 Node 节点。
  - 如果 CAS 成功，说明 Node 节点已经插入，随后 addCount(1L, binCount)方法会检查当前容量是否需要进行扩容。
  - 如果 CAS 失败，说明有其它线程提前插入了节点，自旋重新尝试在这个位置插入节点。
- **如果 f 的 hash 值为-1，**说明当前 f 是 ForwardingNode 节点，**意味有其它线程正在扩容，**则一起进行扩容操作。
- 其余情况把新的 Node 节点**按链表或红黑树的方式插入到合适的位置**，这个过程采用同步内置锁 synchronized 实现并。
- 在节点 f 上进行同步，节点插入之前，再次利用 tabAt(tab, i) == f 判断，防止被其它线程修改。
  - 如果 f.hash >= 0，说明 f 是链表结构的头结点，遍历链表，如果找到对应的 node 节点，则修改 value，否则在链表尾部加入节点。
  - 如果 f 是 TreeBin 类型节点，说明 f 是红黑树根节点，则在树结构上遍历元素，更新或增加节点。
  - 如果链表中节点数 binCount >= TREEIFY_THRESHOLD(默认是 8)，则把链表转化为红黑树结构。

6. table 扩容

当 table 容量不足的时候，即 table 的元素数量达到容量阈值 sizeCtl，需要对 table 进行扩容。整个扩容分为两部分：

- 构建一个 nextTable，大小为 table 的两倍。
- 把 table 的数据复制到 nextTable 中。

ConcurrentHashMap 是支持并发插入的，扩容操作自然也会有并发的出现，这种情况下，第二步可以支持节点的并发复制，这样性能自然提升不少，但实现的复杂度也上升了一个台阶。先看第一步，构建 nextTable，毫无疑问，这个过程只能只有单个线程进行 nextTable 的初始化，通过 Unsafe.compareAndSwapInt 修改 sizeCtl 值，保证只有一个线程能够初始化 nextTable，扩容后的数组长度为原来的两倍，但是容量是原来的 1.5。

节点从 table 移动到 nextTable，大体思想是遍历、复制的过程。

遍历过所有的节点以后就完成了复制工作，把 table 指向 nextTable，并更新 sizeCtl 为新数组大小的 0.75 倍 ，扩容完成。

7. 红黑树构造

注意：如果链表结构中元素超过 **TREEIFY_THRESHOLD 阈值，默认为 8 个**，则把链表转化为红黑树，提高遍历查询效率。

可以看出，生成树节点的代码块是同步的（synchronized），进入同步代码块之后，再次验证 table 中 index 位置元素是否被修改过。主要根据 Node 节点的 hash 值大小构建二叉树。

8. **get 操作**

判断 table 是否为空，如果为空，直接返回 null。

计算 key 的 hash 值，并获取指定 table 中指定位置的 Node 节点，通过遍历链表或则树结构找到对应的节点，返回 value 值。

9. 总结

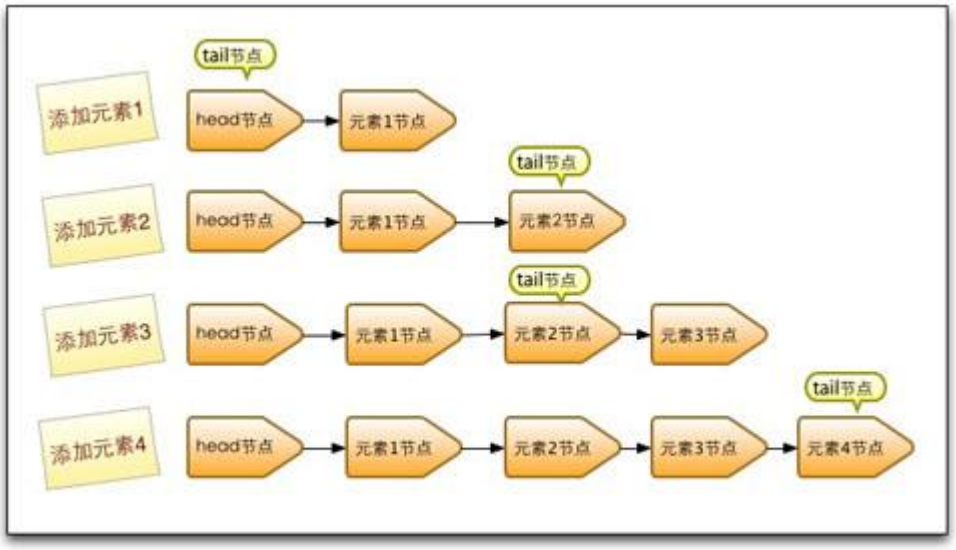
ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于 HashTable 和同步包装器包装的 HashMap，使用一个全局的锁来同步不同线程间的并发访问，同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器，这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成串行化的了。

7.6 ConcurrentLinkedQueue

ConcurrentLinkedQueue 是非阻塞队列。ConcurrentLinkedQueue 由 head 节点和 tail 节点组成，每个节点 Node 由节点元素 item 和指向下一个节点 next 的引用组成，节点和节点之间就是通过这个 next 关联起来，从而组成一张链表结构的队列。默认情况下 head 节点存储的元素为空，tail 节点并不总是尾节点。

1. 入队列【CAS 添加到队列尾部直至成功，永远返回 true】

入队列就是讲入队节点添加到队列的尾部。



入队主要做两件事情，第一是**将入队节点设置成当前队列尾节点的下一个节点**。第二是**更新 tail 节点**，如果 tail 节点的 **next 节点不为空，则将入队节点设置成 tail 节点**，如果 **tail 节点的 next 节点为空，则将入队节点设置成 tail 的 next 节点**，所以 **tail 节点不总是尾节点。**

从源代码角度来看整个入队过程主要做二件事情。第一是**定位出尾节点**，第二是使用 **CAS** 算法能将入队节点**设置成尾节点的 next 节点**，如不成功则重试。

第一步定位尾节点。判断 tail 是否有 next 节点，这个队列刚初始化，p 节点和 p 的 next 节点都等于空，需要返回 head 节点。

```
if (p == q){
    p = (t != (t = tail)) ? t : head;
}
```

第二步设置入队节点为尾节点。`p.casNext(null, newNode)`方法用于将入队节点设置为当前队列尾节点的 next 节点，p 如果是 null 表示 p 是当前队列的尾节点，如果不为 null 表示有其他线程更新了尾节点，则需要重新获取当前队列的尾节点。

HOPS 的设计意图

使用 hops 变量来控制并减少 tail 节点的更新频率，并不是每次节点入队后都将 tail 节点更新为尾节点，而是当 tail 节点和尾节点的距离大于等于常量 hops 的值（默认等于 1）时才更新 tail 节点。

入队方法返回永远是 true。

2. 出队列

出队列的就是从队列里返回一个节点元素，并清空该节点对元素的引用。首先**获取头节点**的元素，然后判断头节点元素**是否为空**，如果为空，表示另外一个线程已经进行了一次出队操作将该节点的元素取走，如果**不为空，则使用 CAS** 的方式将头节点的引用**设置成 null**，如果 CAS 成功，则直接返回头节点的元素，如果不成功，表示另外一个线程已经进行了一次出队操作更新了 head 节点，导致元素发生了变化，需要重新获取头节点。

并不是每次出队时都更新 head 节点，当 head 节点里有元素时，直接弹出 head 节点里的元素，而不会更新 head 节点。只有当 head 节点里没有元素时，出队操作才会更新 head

节点。这种做法也是通过 hops 变量来减少使用 CAS 更新 head 节点的消耗，从而提高出队效率。

27.7 阻塞队列

阻塞的插入方法：当队列满时，队列会阻塞插入元素的线程，直到队列不满。
阻塞的移除方法：当队列空时，队列会阻塞移除元素的线程，直到队列非空。
阻塞队列常用于生产者和消费者的场景。

1. BlockingQueue 接口方法

方法/处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	Offer(e)	Put(e)	Offer(e,time,unit)
移除方法	Remove()	Poll()	Take()	Poll(time,unit)
检查方法	Element()	Peek()	不可用	不可用

✧ 抛出异常:当队列满时，再往队列里插入元素，抛出 IllegalStateException 异常
当队列空时；从队列里获取元素会抛出 NoSuchElementException 异常；

✧ 返回特殊值:当队列插入元素时，会返回元素是否插入成功，成功返回 true；
如果是移除方法，则是从队列里取出一个元素，如果没有则返回 null。

✧ 一直阻塞:一直阻塞，直到队列可用或者相应中断退出。

✧ 超时退出:队列会阻塞生产者线程一段时间；如果超过了指定时间，生产者线程退出。

如果是无界阻塞队列，队列不可能会出现满的情况，所以使用 put 和 offer 方法永远不被阻塞，而是 offer 方法时，返回的永远是 true。

2. ArrayBlockingQueue

2.1 基本使用

ArrayBlockingQueue 是一个用**数组**实现的**有界**阻塞队列，其内部按**先进先出**的原则对元素进行排序，其中 **put 方法**和 **take 方法**为添加和删除的阻塞方法，下面我们通过 ArrayBlockingQueue 队列实现一个生产者消费者的案例，Consumer 消费者和 Producer 生产者，通过 ArrayBlockingQueue 队列获取和添加元素，其中消费者调用了 take()方法获取元素当队列没有元素就阻塞，生产者调用 put()方法添加元素，当队列满时就阻塞，通过这种方式便实现生产者消费者模式。比直接使用等待唤醒机制或者 Condition 条件队列来得更加简单。

```
public class ArrayBlockingQueueDemo {
    private final static ArrayBlockingQueue<Apple> queue= new
ArrayBlockingQueue<>(1);
    public static void main(String[] args){
        new Thread(new Producer(queue)).start();
        new Thread(new Producer(queue)).start();
        new Thread(new Consumer(queue)).start();
        new Thread(new Consumer(queue)).start();
    }
}

 class Apple {
    public Apple(){
    }
```

```
}

/**
 * 生产者线程
 */
class Producer implements Runnable{
    private final ArrayBlockingQueue<Apple> mAbq;
    Producer(ArrayBlockingQueue<Apple> arrayBlockingQueue){
        this.mAbq = arrayBlockingQueue;
    }

    @Override
    public void run() {
        while (true) {
            Produce();
        }
    }

    private void Produce(){
        try {
            Apple apple = new Apple();
            mAbq.put(apple);
            System.out.println("生产:"+apple);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

/**
 * 消费者线程
 */
class Consumer implements Runnable{

    private ArrayBlockingQueue<Apple> mAbq;
    Consumer(ArrayBlockingQueue<Apple> arrayBlockingQueue){
        this.mAbq = arrayBlockingQueue;
    }

    @Override
    public void run() {
        while (true){
            try {
                TimeUnit.MILLISECONDS.sleep(1000);
```

```
                comsume();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private void comsume() throws InterruptedException {
        Apple apple = mAbq.take();
        System.out.println("消费 Apple="+apple);
    }
    }
```

ArrayBlockingQueue 内部的阻塞队列是通过重入锁 ReenterLock 和 Condition 条件队列实现的，所以 ArrayBlockingQueue 中的元素存在公平访问与非公平访问的区别，对于公平访问队列，被阻塞的线程可以按照阻塞的先后顺序访问队列，即先阻塞的线程先访问队列。而非公平队列，当队列可用时，阻塞的线程将进入争夺访问资源的竞争中，也就是说谁先抢到谁就执行，没有固定的先后顺序。创建公平与非公平阻塞队列代码如下：

```
//默认非公平阻塞队列

ArrayBlockingQueue queue = new ArrayBlockingQueue(2);

//公平阻塞队列

ArrayBlockingQueue queue1 = new ArrayBlockingQueue(2,tru
e);



//构造方法源码

public ArrayBlockingQueue(int capacity) {

    this(capacity, false);

 }
```

```
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
 }
```

2.2 原理概要

ArrayBlockingQueue 的内部是通过一个可重入锁 ReentrantLock 和两个 Condition 条件对象来实现阻塞。

3. LinkedBlockingQueue

Offer()方法做了两件事，第一件事是判断队列是否满，满了就直接释放锁，没满就将节点封装成 Node 入队，然后再次判断队列添加完成后是否已满，不满就继续唤醒在条件对象 notFull 上的添加线程。第二件事是，判断是否需要唤醒等到在 notEmpty 条件对象上的消费线程。

为什么添加完成后是继续唤醒在条件对象 notFull 上的添加线程而不是像 ArrayBlockingQueue 那样直接唤醒 notEmpty 条件对象上的消费线程？而又为什么要当 if (c == 0)时才去唤醒消费线程呢？

唤醒添加线程的原因，在添加新元素完成后，会判断队列是否已满，不满就继续唤醒在条件对象 notFull 上的添加线程，这点与前面分析的 ArrayBlockingQueue 很不相同，在 ArrayBlockingQueue 内部完成添加操作后，会直接唤醒消费线程对元素进行获取，这是因为 **ArrayBlockingQueue 只用了一个 ReenterLock 同时对添加线程和消费线程进行控制，这样如果在添加完成后再次唤醒添加线程的话，消费线程可能永远无法执行**，而对于 LinkedBlockingQueue 来说就不一样了，其内部对添加线程和消费线程分别使用了各自的 ReenterLock 锁对并发进行控制，也就是说添加线程和消费线程是不会互斥的，所以添加锁只要管好自己的添加线程即可，添加线程自己直接唤醒自己的其他添加线程，如果没有等待的添加线程，直接结束了。如果有就直到队列元素已满才结束挂起，当然 offer 方法并不会挂起，而是直接结束，只有 put 方法才会当队列满时才执行挂起操作。注意消费线程的执行过程也是如此。这也是为什么 LinkedBlockingQueue 的吞吐量要相对大些的原因。

为什么要判断 if (c == 0)时才去唤醒消费线程呢，这是因为消费线程一旦被唤醒是一直在消费

的（前提是有数据），所以 c 值是一直在变化的，c 值是添加完元素前队列的大小，此时 c 只可能是 0 或 c>0，如果是 c=0，那么说明之前消费线程已停止，条件对象上可能存在等待的消费线程，添加完数据后应该是 c+1，那么有数据就直接唤醒等待消费线程，如果没有就结束啦，等待下一次的消费操作。如果 c>0 那么消费线程就不会被唤醒，只能等待下一个消费操作（poll、take、remove）的调用，那为什么不是条件 c>0 才去唤醒呢？我们要明白的是消费线程一旦被唤醒会和添加线程一样，一直不断唤醒其他消费线程，如果添加前 c>0，那么很可能上一次调用的消费线程后，数据并没有被消费完，条件队列上也就不存在等待的消费线程了，所以 c>0 唤醒消费线程得意义不是很大，当然如果添加线程一直添加元素，那么一直 c>0，消费线程执行的换就要等待下一次调用消费操作了（poll、take、remove）。

7.8 Fork/Join 框架

Fork/Join 框架是 Java7 提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小人物，最终汇合每个小任务结果后得到大任务结果的框架。

1. **工作窃取算法**

工作窃取算法（work-stealing）算法是指某个线程从其让队列里窃取任务来执行。干完活的线程去其他线程的队列里窃取一个任务来执行。

通常使用双端队列，被窃取任务线程永远从双端队列的头部那任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

2. **Fork/join 框架设计**

步骤一：分割任务。Fork 类来把大任务分割成子任务，知道分割出的子任务足够小

步骤二：执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程获取任务执行，执行完的结果都同一放在一个队列中，启动一个线程从队列里拿数据合并。

使用步骤：① 创建一个 forkjoin 任务，需要继承 RecursiveTask（有返回结果）或者 RecursiveAction（没有返回结果），重写 compute 方法；② ForkJoinTask 通过 ForkJoinPool 来执行。

3. 使用 Fork/join

```
protected Integer compute() {
        int sum = 0;

        // 如果任务足够小就计算任务
        boolean canCompute = (end - start) <= THRESHOLD;
        if (canCompute) {
                for (int i = start; i <= end; i++) {
                        sum += i;
                }
        } else {
                // 如果任务大于阈值，就分裂成两个子任务计算
                int middle = (start + end) / 2;
                CountTask leftTask = new CountTask(start, middle);
                CountTask rightTask = new CountTask(middle + 1, end);
                //执行子任务
                leftTask.fork();
                rightTask.fork();
                //等待子任务执行完，并得到其结果

                int leftResult=leftTask.join();
                int rightResult=rightTask.join();
                //合并子任务
                sum = leftResult  + rightResult;
        }
        return sum;
}

public static void main(String[] args) {
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        // 生成一个计算任务，负责计算1+2+3+4
        CountTask task = new CountTask(1, 4);
        // 执行一个任务
        Future<Integer> result = forkJoinPool.submit(task);
        try {
                System.out.println(result.get());
        } catch (InterruptedException e) {
        } catch (ExecutionException e) {
        }
}
```

4. 实现原理

ForkJoinPool 由 ForkJoinTask 数组（存放任务）和 ForkJoinWorkerThread 数组（执行任务）组成。

执行任务：ForkJoinTask.fork() →ForkJoinWorkerThread.pushTask()异步执行

→ForkJoinPool.signalWork()唤醒/创建线程执行

等待任务执行完：ForkJoinTask.join()→dojoin()返回结果

1） Normal:完成；②cancelled:取消；③signal:信号；④exceptional:异常

第八章 13 个原子操作类

1、Java 中实现原子操作

在 Java 中可以通过锁和循环 CAS 的方式来实现原子操作。从 Java1.5 开始，JDK 的并发包

里提供了一些来支持原子操作，如 AtomicBoolean、AtomicInteger 和 AtomicLong。

2、CAS 实现原子操作的三大问题

1) ABA 问题：

CAS 需要检查值有没有发生变化，如果没有变化则更新，但是如果一个值**原来是 A，变成 B，又变成了 A**，那么使用 CAS 进行检查时会发现它的值没有发生变化，实际上变化了。

ABA 问题的解决思路就是：使用版本号。在变量前面追加上版本号。

从 JDK1.5 开始，JDK 的 Atomic 包里提供了一个类 **AtomicStampedReference 来解决 ABA 问题**。这个类的 compareAndSet 方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式设置更新值。

2)循环时间长开销大

自旋 CAS 如果长时间不成功，会给 CPU 带来非常大的执行开销。

3)只能保证一个共享变量的原子操作

对多个共享变量操作时，循环 CAS 无法保证操作的原子性，这个时候可以用锁。从 JDK1.5 开始，JDK 提供了 **AtomicReference 类来保证引用对象之间的原子性**，可以把多个变量放在一个对象里来进行 CAS 操作。

3. CAS 原理

CAS 有 3 个操作数，内存值 V，旧的预期值 A，要修改的新值 B。当且仅当预期值 A 和内存值 V 相同时，将内存值 V 修改为 B，否则什么都不做。

compareAndSet 利用 **JNI 来完成 CPU 指令的操作**。

```
public final boolean compareAndSet(int expect, int update) {
return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
}
```

CAS 通过调用 JNI 的代码实现的。JNI:Java Native Interface 为 JAVA 本地调用。而 compareAndSwapInt 就是借助 C 来调用 CPU 底层指令实现的。下面从分析比较常用的 CPU（intel x86）来解释 CAS 的实现原理。

程序会根据当前处理器的类型来决定是否为 cmpxchg 指令添加 lock 前缀。如果程序是在多处理器上运行，就为 cmpxchg 指令加上 lock 前缀（lock cmpxchg）。反之，如果程序是在单处理器上运行，就省略 lock 前缀。

lock 前缀的说明如下：

①确保对内存的读-改-写操作原子执行。

②禁止该指令与之前和之后的读和写指令重排序。

2） 把写缓冲区中的所有数据刷新到内存中。

第九章 并发工具类

9.1 等待多线程完成的 CountDownLatch

CountDownLatch 允许一个或多个线程等待其他线程完成操作。Join 用于让当前执行线程等待 join 线程执行结束。其实现原理是不停检查 join 线程是否存活，如果线程存活则让当前线程永远等待。

CountDownLatch 通过构造函数传入一个初始计数值，调用者可以通过调用 CounDownLatch 对象的 **cutDown()方法，来使计数减 1**；如果调用对象上的 **await()方法，那么调用者就会一**

直阻塞在这里，直到别人通过 cutDown 方法，将计数减到 0，才可以继续执行。

```
public CountDownLatch(int count) {  };
//参数 count 为计数值
public void await() throws InterruptedException { };
//调用 await()方法的线程会被挂起，它会等待直到 count 值为 0 才继续执行
public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };
//和 await()类似，只不过等待一定的时间后 count 值还没变为 0 的话就会继续执行
public void countDown() { };  //将 count 值减 1
```

9.2 同步屏障 CyclicBarrier

CyclicBarrier 让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。

CyclicBarrier 的默认构造方法会传 int 型参数，表示屏障拦截的线程数量，每个线程调用 await 方法表示已经到达了屏障，当前线程被阻塞。还提供了一个更高级的构造函数

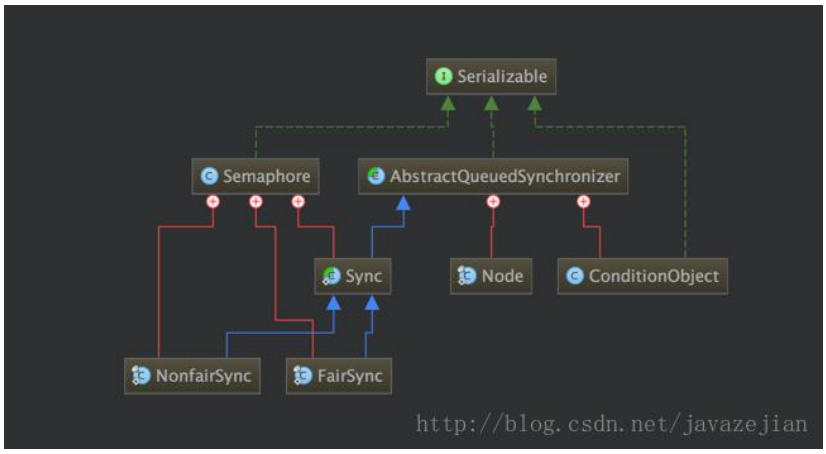
```
public CyclicBarrier(int parties, Runnable barrierAction)
```

用于在线程到达屏障时，优先执行 barrierAction。

CyclicBarrier 和 CountDownLatch 的区别：CountDownLatch 的计数器只能使用一次，而 CyclicBarrier 的计数器可以使用 reset 方法重置。同时 CyclicBarrier 提供了获取阻塞的线程数量的方法和阻塞线程是否被中断的方法。

9.3 控制并发线程数的 Semaphore

信号量(Semaphore)，维护了一个许可集，我们在初始化 Semaphore 时需要为这个许可集传入一个数量值，该数量值代表同一时间能访问共享资源的线程数量。线程可以通过 **acquire()**方法获取到一个许可，然后对共享资源进行操作，注意如果许可集已分配完了，那么线程将进入等待状态，直到其他线程释放许可才有机会再获取许可，线程释放一个许可通过 **release()**方法完成。



Semaphore 内部同样存在继承自 AQS 的内部类 Sync 以及继承自 Sync 的公平锁(FairSync)和非公平锁(NofairSync)。Semaphore 的内部类公平锁(FairSync)和非公平锁(NoFairSync)各自实现不同的获取锁方法即 tryAcquireShared(int arg)，而释放锁 tryReleaseShared(int arg)的操作交由 Sync 实现。

Reentrank 获取/释放锁都返回 true 或 false。获取锁，状态值和 0 比较，不为 0 时，比较是否为当前线程，状态值加 1。释放锁，状态值和 0 比较。Semaphore 释放锁，返回 true 和 false。获取锁返回剩余许可数量。如果 state 值代表的许可数已为 0，则请求线程将被加

入到同步队列并阻塞。

调用 Semaphore 的 acquire()方法后将会调用到 AQS 的 acquireSharedInterruptibly()，也就是说 Semaphore 的 acquire()方法也是可中断的。在 acquireSharedInterruptibly()方法内部先进行了线程中断的判断，如果没有中断，那么先尝试调用 tryAcquireShared(arg)方法获取同步状态，如果获取失败调用 doAcquireSharedInterruptibly(arg);方法加入同步队列等待。这里的 tryAcquireShared(arg)是个**模板方法**，AQS 内部没有提供具体实现，由子类实现，也就是有 Semaphore 内部自己实现。

非公平锁：

```
protected int tryAcquireShared(int acquires) {
    return nonfairTryAcquireShared(acquires);
}
abstract static class Sync extends AbstractQueuedSynchronizer {
    final int nonfairTryAcquireShared(int acquires) {
        //使用死循环
        for (;;) {
            int available = getState();
            int remaining = available - acquires;
            //判断信号量是否已小于 0 或者 CAS 执行是否成功
            if (remaining < 0 || compareAndSetState(available, remaining))
                return remaining;
        }
    }
}
```

先**获取 state 的值**，并**执行减法操作**，得到 remaining 值，如果 remaining **不小于 0**，那么线程**获取同步状态成功**，可访问共享资源，并**更新 state 的值**；如果 remaining **小于 0**，那么线程获取同步状态失败，将被**加入同步队列**(通过 doAcquireSharedInterruptibly(arg))，注意 Semaphore 的 acquire()可能存在并发操作，因此 nonfairTryAcquireShared()方法体内部采用**无锁(CAS)**并发的操作保证对 state 值修改的安全性。如果尝试获取同步状态失败，那么将会执行 **doAcquireSharedInterruptibly**(int arg)方法. 由于当前线程没有获取同步状态，因此创建一个**共享模式（Node.SHARED）的结点**并通过 addWaiter(Node.SHARED)**加入同步队列**，加入完成后，当前线程进入自旋状态，首先判断前驱结点是否为 head，如果是，那么尝试获取同步状态并返回 r 值，如果 r 大于 0，则说明获取同步状态成功，将当前线程设置为 head 并传播，传播指的是，同步状态剩余的许可数值不为 0，通知后续结点继续获取同步状态，到此方法将会 return 结束，获取到同步状态的线程将会执行原定的任务。但如果前驱结点不为 head 或前驱结点为 head 并尝试获取同步状态失败，那么调用 shouldParkAfterFailedAcquire(p, node)方法判断前驱结点的 waitStatus 值是否为 SIGNAL 并调整同步队列中的 node 结点状态，如果返回 true，那么执行 parkAndCheckInterrupt()方法，将当前线程挂起并返回是否中断线程的 flag。

到此，加入同步队列的整个过程完成。这里小结一下，在 AQS 中存在一个变量 state，当我们创建 Semaphore 对象**传入许可数值**时，**最终会赋值给 state**，state 的数值代表同一个时刻可同时操作共享数据的线程数量，每当一个线程请求(如调用 Semaphored 的 acquire()方法)获取同步状态成功，state 的值将会减少 1，直到 state 为 0 时，表示已没有可用的许可数，也就是对共享数据进行操作的线程数已达到最大值，其他后来线程将被阻塞，此时 AQS 内部会将线程封装成共享模式的 Node 结点，加入同步队列中等待并开启自旋操作。只有当

持有对共享数据访问权限的线程执行完成任务并释放同步状态后，同步队列中的对于的结点线程才有可能获取同步状态并被唤醒执行同步操作，注意在同步队列中获取到同步状态的结点将被设置成 head 并清空相关线程数据(毕竟线程已在执行也就没有必要保存信息了)，AQS 通过这种方式便实现共享锁。

Semaphore， AQS 中通过 state 值来控制对共享资源访问的线程数，每当线程请求同步状态成功，state 值将会减 1，如果超过限制数量的线程将被封装共享模式的 Node 结点加入同步队列等待，直到其他执行线程释放同步状态，才有机会获得执行权，而每个线程执行完成任务释放同步状态后，state 值将会增加 1，这就是共享锁的基本实现模型。至于公平锁与非公平锁的不同之处在于公平锁会在线程请求同步状态前，判断同步队列是否存在 Node，如果存在就将请求线程封装成 Node 结点加入同步队列，从而保证每个线程获取同步状态都是先到先得的顺序执行的。非公平锁则是通过竞争的方式获取，不管同步队列是否存在 Node 结点，只有通过竞争获取就可以获取线程执行权。

第十章 线程池-Executor 框架

Eexecutor 作为灵活且强大的**异步执行框架**，其支持多种不同类型的任务执行策略，提供了一种标准的方法将任务的提交过程和执行过程解耦开发，基于生产者-消费者模式，其提交任务的线程相当于生产者，执行任务的线程相当于消费者，并用 Runnable 来表示任务，Executor 的实现还提供了对生命周期的支持，以及统计信息收集，应用程序管理机制和性能监视等机制。

10.1 线程池的实现原理

当提交一个新任务到线程池时，线程池的处理流程：

1）线程池判断**核心线程池**里的线程都在执行任务。如果不是，创建一个新的工作线程执行任务，如果是，进入 2);

2）线程池判断**工作队列**是否已满。没满，存储到这个工作队列中，否则，进入 3);

3）线程池判断**线程池的线程**是否都在执行任务，若满，则采用拒绝策略来处理这个任务。

ThreadPoolExecutor 执行 execute 方法分为以下 4 种情况：

1) 如果当前运行的线程少于 corePoolSize，则创建新线程来执行任务【获取全局锁】;
2) 如果运行的线程等于或多于 corePoolSize，则将任务加入 BlockingQueue。
3) 如果不能将任务加入 BlockingQueue(队列已满)，创建新的线程来处理任务；
4) 如果当前运行的线程超过 maximumPoolSize，任务将被拒绝。

10.2 线程池的使用

1. 创建线程池

通过 ThreadPoolExecutor 来创建一个线程池：

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler )
```

- corePoolSize：线程池的基本大小；
- maximumPoolSize：线程池的最大数量；
- keepAliveTime：多余线程活动保持的时间
- workQueue：任务队列，用于保存等待执行的任务的阻塞队列，可以选择：
  - ArrayBlockingQueue：基于数组的有界阻塞队列，FIFO。
  - LinkedBlockingQueue：基于链表的阻塞队列，Executors.newFixedThreadPool()
  - SynchronousQueue：不存储元素的阻塞队列。Executors.newCachedThreadPool()

- PriorityBlockingQueue：具有优先级的无线阻塞队列。

✧ Handler：拒绝策略。

- AbortPolicy：直接抛出异常。
- CallerRunsPolicy：只用调用者所在线程来运行任务。
- DiscardOldestPolicy：丢弃队列里最近的一个任务，并指向当前任务。
- DiscardPolicy：不处理，丢弃掉。

2. 提交任务

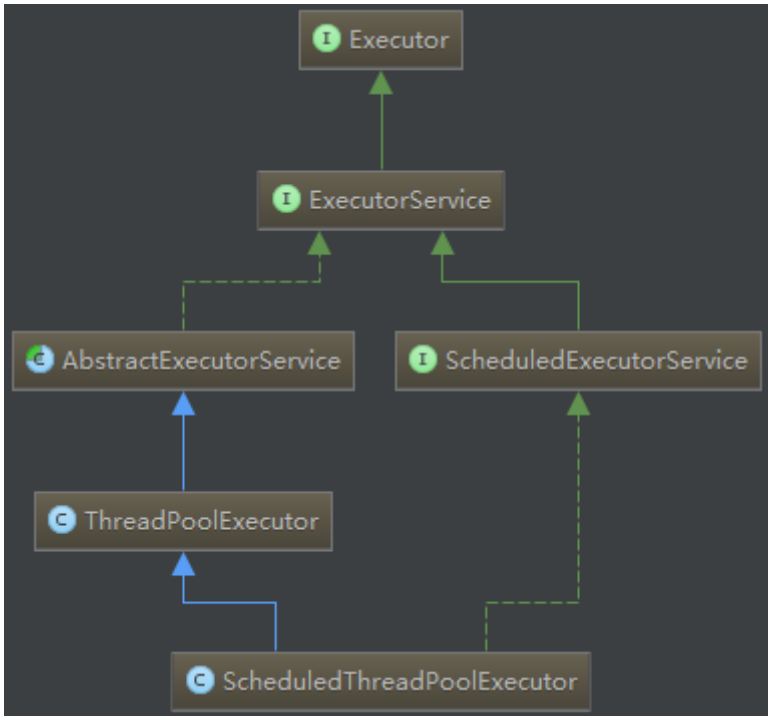
- execute()：提交不需要返回值的任务
- submit()：提交需要返回值的任务，返回一个 future 类型的对象，通过 get()获取返回值。

3. 关闭线程池

遍历线程池中的工作线程，然后逐个调用线程的 interrupt 方法来中断线程，所以无法响应中断的任务可能永远无法终止。

- shutdown()：只是将线程池的状态设置为 shutdown 状态，然后中断所有没有正在执行任务的线程。
- shutdownNow()：首先将线程池的状态设置为 STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表。

10.1 Executor 的 UML 图



Executor：一个接口，其定义了一个接收 Runnable 对象的方法 executor(Runnable command)。

ExecutorService：是一个比 Executor 使用更广泛的子类接口，其提供了生命周期管理的方法，以及可跟踪一个或多个异步任务执行状况返回 Future 的方法

AbstractExecutorService： ExecutorService 执行方法的默认实现

ScheduledExecutorService： 一个可定时调度任务的**接口**

ScheduledThreadPoolExecutor： ScheduledExecutorService 的实现，一个可定时调度任务的线程池。

ThreadPoolExecutor：线程池，可以通过调用 Executors 以下静态工厂方法来创建线程池并返

回一个 ExecutorService 对象。

10.2 ThreadPoolExecutor 构造函数

```
public ThreadPoolExecutor(int corePoolSize,   int maximumPoolSize,
          long keepAliveTime,   TimeUnit unit,   BlockingQueue<Runnable> workQueue,
          ThreadFactory threadFactory,   RejectedExecutionHandler handler)
                              //后两个参数为可选参数
```

参数说明：

corePoolSize：核心线程数，如果运行的线程少于 corePoolSize，则创建新线程来执行新任务，即使线程池中的其他线程是空闲的。

maximumPoolSize：最大线程数，可允许创建的线程数。

keepAliveTime：如果线程数多于 corePoolSize,多余的线程的存活时间。

Unit: keepAliveTime 参数的时间单位。

workQueue：保存任务的阻塞队列，被提交但尚未被执行的任务。

threadFactory:使用 ThreadFactory 创建新线程，默认使用 defaultThreadFactory 创建线程

handle:定义处理被拒绝任务的策略，默认使用 ThreadPoolExecutor.AbortPolicy。

10.3 .Executors

提供了一系列静态工厂方法用于创建各种线程池。

newFixedThreadPool:创建可重用且**固定线程数**的线程池。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

newSingleThreadExecutor:创建一个单线程的 Executor

```
public static ExecutorService newSingleThreadExecutor() {
   return new FinalizableDelegatedExecutorService
             (new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>()));
}
```

newScheduledThreadPool:创建一个可延迟执行或定期执行的线程池。模拟心跳机制

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
   super(corePoolSize, Integer.MAX_VALUE, 0, TimeUnit.NANOSECONDS,
              new DelayedWorkQueue());
}
```

newCachedThreadPool:创建可缓存的线程池，如果线程池中的线程在 60 秒未被使用就将被移除。

```
public static ExecutorService newCachedThreadPool() {
   return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit. SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

10.4 Executor 的生命周期

ExecutorService 提供了管理 Eecutor 生命周期的方法，ExecutorService 的生命周期包括了：运行、关闭和终止三种状态。

ExecutorService 在初始化创建时处于运行状态。
shutdown 方法等待提交的任务执行完成并不再接受新任务，在完成全部提交的任务后关闭
shutdownNow 方法将强制终止所有运行中的任务并不再允许提交新任务

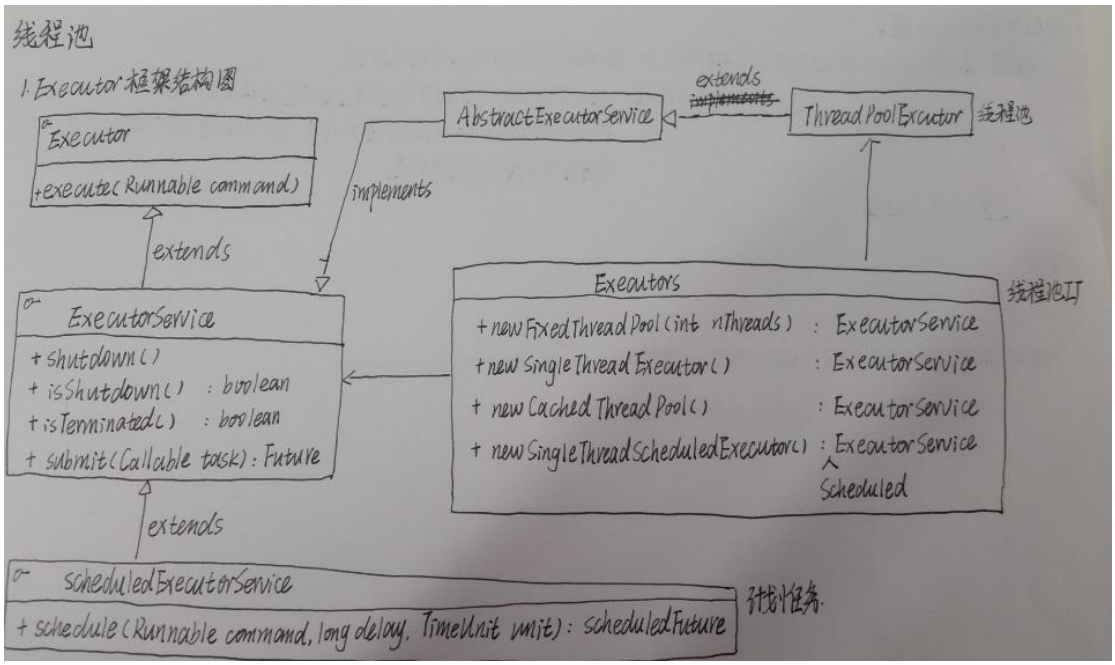
排队有三种通用策略：

1.直接提交队列。默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。提交的任务不会被真实的保存，总是加你个新任务提交给线程执行，如果没有空闲的进程，则尝试创建新的进程。达到最大值，则拒绝。

2.无界队列。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize 的值也就无效了。）

3. 有界队列。当使用有限的 maximumPoolSizes 时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。

10.5 Executor 框架



- Executor：是一个接口，将任务的提交和任务的执行分离开来；
- ThreadPoolExecutor：线程池的核心实现类，用来执行被提交的任务；
- ScheduledThreadPoolExecutor：实现类，在给定的延迟后运行命令，或者定时执行；
- Future 接口和实现 Future 接口的 FutureTask 类，代表异步计算的结果。

1. **ThreadPoolExecutor**

ThreadPoolExecutor 通常使用工厂类 Executors 来创建 3 种类型的 ThreadPoolExecutor：

- newFixedThreadPool:创建可重用且固定线程数的线程池。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

- newSingleThreadExecutor:创建一个单线程的 Executor

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
```

(new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
new LinkedBlockingQueue<Runnable>()));
}

- newCachedThreadPool:创建可缓存的线程池，如果线程池中的线程在 60 秒未被使用就将被移除。

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit. SECONDS,
        new SynchronousQueue<Runnable>()));
}
```

2. ScheduledThreadPoolExecutor

通常使用工厂类 Executors 来创建 2 种类型的 ScheduledThreadPoolExecutor：

- newScheduledThreadPool:创建一个可延迟执行或定期执行的线程池。模拟心跳机制

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, TimeUnit.NANOSECONDS,
        new DelayedWorkQueue());
}
```

- 下面提供了四种预定义的处理程序策略：

1. 在默认的 ThreadPoolExecutor.AbortPolicy 中，处理程序遭到拒绝将抛出运行时 RejectedExecutionException。
2. 在 ThreadPoolExecutor.CallerRunsPolicy 中，线程调用运行该任务的 execute 本身。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度。
3. 在 ThreadPoolExecutor.DiscardPolicy 中，不能执行的任务将被删除。
4. 在 ThreadPoolExecutor.DiscardOldestPolicy 中，如果执行程序尚未关闭，

第十一章 JAVA1.8 版本增了哪些新功能

① 接口的默认方法

Java 8 允许给接口添加一个非抽象的方法实现，只需要使用 default 关键字即可。

```
public interface TestInterfaceDefault {
    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

② Lambda 表达式：

```
Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> e1.compareTo( e2 ) );
```

③ 函数式接口

将 lambda 表达式当作任意只包含一个抽象方法的接口类型，确保你的接口一定达到这个要求，你只需要给你的接口添加 @FunctionalInterface 注解，编译器如果发现你标注了这个注解的接口有多于一个抽象方法的时候会报错的。

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

```
//测试 BiFunction
BiFunction<Integer, Integer , Integer> biFunction  =  (a, b) ->  a * b;
System.out.println(biFunction.apply(10, 2));

//测试 Predicate
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
eval(list, a -> true);
eval(list, a -> a % 2 == 0);

public static void eval(List<Integer> list, Predicate<Integer> predicate){
    list.forEach(integer -> {
        if(predicate.test(integer))
            System.out.println(integer);
    });
}
```

④函数式接口里的默认方法和静态方法

允许添加 default 方法，和 static 方法。静态方法不可以被集成。默认方法可以被继承，默认方法可以被重写，重新声明后，则变成了普通方法

⑤方法的引用

构造器引用：语法是 Class::new。请注意构造器没有参数。

静态方法引用 ：语法是 Class::static_method。这个方法接受一个 Class 类型的参数。

特定类的任意对象的方法引用：语法是 Class::method。这个方法没有参数。

特定对象的方法引用：语法是 instance::method。接受一个 instance 对应的 Class 类型的参数。

```
names.forEach(System.out::println);     //3、方法引用，没有参数
HashSet::new                            //1、构造器引用，没有参数
Person::compareByAge                    // 2、静态方法引用，接收参数
person::compareByAge                    //4、对象的方法引用，接收参数
```

⑥ 重复注解

允许同一个位置声明多次注解。重复注解机制本身必须用@Repeatable 注解

⑦Optional：Optional 的引入是为了解决臭名昭著的空指针异常问题.

- **ofNullable：** 如果不为 null，则返回一个描述指定值的 Optional；否则返回空的 Optional
- **of：** 如果不为 null，则返回一个描述指定值的 Optional；否则报异常
- **isPresent：** 如果有值存在，则返回 TRUE，否则返回 false。
- **orElse：** 如果有值，则返回当前值；如果没有，则返回 other
- **get：** 如果有值，则返回当前值；否则返回 NoSuchElementException

```
Optional<Integer> a = Optional.ofNullable(null);  //1、ofNullable 允许参数为 null
```

```
Optional<Integer> b = Optional.of(value2);        //2、of，参数为空，抛异常
a.isPresent();                                    //3、 false， isPresent 判断值是否存在
Integer value1 = a.orElse(new Integer(0));        //4、返回值，没有返回参数
Integer value2 = b.get();                         //5、返回值，没有抛异常
```

⑧Stream

- **stream** 返回顺序流，集合作为其源。
- **parallelStream** 返回并行数据流， 集合作为其源。
- **filter** 方法用于消除基于标准元素
- **map** 方法用于映射每个元素对应的结果
- **forEach** 方法遍历该流中的每个元素
- **limit** 方法用于减少流的大小
- **sorted** 方法用来流排序
- **collect** 方法是终端操作，这是通常出现在管道传输操作结束标记流的结束。

```
//把空指针过滤掉，返回前三个：
    public void test() {
        List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd", "", "jkl");
        List<String> list = strings
                .stream()
                .filter(n -> !"".equals(n))
                .limit(4)
                .collect(Collectors.toList());
        list.forEach(System.out::println);
    }

    //计算集合每个元素的平方，并且去重，然后将值为 81 的去掉，输出排序后的数据
    public void test2() {
        List<Integer> ints = Arrays.asList(1, 5, 9, 6, 5, 4, 2, 5, 9);
        ints
                .stream()
                .map(n -> n * n)
                .distinct()
                .filter(n -> n != 81)
                .sorted()
                .collect(Collectors.toList())
                .forEach(n -> System.out.println(n));
    }

    //将上面的实例改为并行流，求和。
    public void test3() {
        List<Integer> ints = Arrays.asList(1, 5, 9, 6, 5, 4, 2, 5, 9);
        System.out.println(ints
                .parallelStream()
                .filter(n -> n > 2)
```

```
                .distinct()
                .count());
        Integer sum = ints
                .parallelStream()
                .map(n -> n * n)
                .filter(n -> n != 81)
                .reduce(Integer::sum)
                .get();
        System.out.println(sum);
    }

    //统计，这里用到了 mapToInt 将其转换为可以进行统计的数值型。类似的还有mapToLong、mapToDouble
    public void test5(){
        List<Integer> ints = Arrays.asList(1,5,9,6,5,4,2,5,9);
        IntSummaryStatistics statistics = ints.stream().mapToInt(n -> n).summaryStatistics();
        System.out.println(statistics.getAverage());
        System.out.println(statistics.getCount());
        System.out.println(statistics.getMax());
        System.out.println(statistics.getMin());
    }
```

⑨日期/时间

Clock 类，它通过指定一个时区，然后就可以获取到当前的时刻，日期与时间。

LocalDateTime/LocalDateTime/LocalDateTime：

ZonedDateTime：带时区日期时间处理。

ChronoUnit：代替 Calendar 的日期操作。

DateTimeFormatter：可以替代以前的 dateformat，使用起来方便一些。

Period/Duration：处理时间差。

(10) 并行（parallel）数组

parallelSort()方法，因为它可以在多核机器上极大提高数组排序的速度。

```
public static void main( String[] args ) {
    LocalTime begin = LocalTime.now();
    long[] arrayOfLong = new long [ 20000 ];
    Arrays.parallelSetAll(arrayOfLong, index -> ThreadLocalRandom.current().nextInt(1000000));
    Arrays.stream(arrayOfLong).limit(10).forEach(i -> System.out.print(i + " "));
    System.out.println();

    Arrays.parallelSort( arrayOfLong );
    Arrays.stream(arrayOfLong).limit(10).forEach(i -> System.out.print(i + " "));
    System.out.println();
    System.out.println(Duration.between(begin, LocalTime.now()));
}
```

第十二章 线上问题定位

1）Top 查看每个进程的情况：关注 command 为 Java 的%CPU

```
top - 10:07:30 up 274 days, 5 min,  5 users,  load average: 0.01, 0.03, 0.05
Tasks: 153 total,   1 running, 152 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.1%us,  0.2%sy,  0.0%ni, 99.5%id,  0.2%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8193560k total,  5351456k used,  2842104k free,   248372k buffers
Swap:  2097148k total,  1146112k used,   951036k free,   755256k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
10831 nobody    20   0 4937m 715m  19m S  0.7  8.9   3:07.50 java
26322 nobody    20   0 5475m 442m 5112 S  0.7  5.5 149:02.14 java
 2200 root      20   0 3307m  43m 2308 S  0.3  0.5 402:01.56 java
 2251 nobody    20   0 5470m 647m 5192 S  0.3  8.1 846:52.20 java
```

2）使用 top 的交互命令数字 1 查看每个 CPU 的性能数据。
如果某个 CPU 的利用率到达 100%，可能写了一个死循环。

3）使用 top 的交互命令 H 查看每个线程的性能数据。

- 某个线程 CPU 利用率一直 100%，则说明是这个线程有可能有死循环。
- 某个线程一直在 TOP10，这个线程可能有性能问题。
- CPU 利用率高的几个线程在不断变化，说明并不是由某一个线程导致的。

对于第一种情况，也可能是 GC 造成的：

- jstat –gcutil 线程 pid 1000 5

```
C:\Documents and Settings\liuzhenxu>jstat -gcutil 1192
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00  28.40   0.00  15.60      0    0.000     0    0.000    0.000
```

S0、S1 代表两个 Survivor 区；
E 代表 Eden 区；
O（Old）代表老年代；
P（Permanent）代表永久代；
YGC（Young GC）代表 Minor GC；
YGCT 代表 Minor GC 耗时；
FGC（Full GC）代表 Full GC 耗时；
GCT 代表 Minor & Full GC 共计耗时。

- 还可 dump 下来： jstack pid > /home/lhelper/dump17