1. Lock 简介

1.1. synchronized

1、Java 中每一个对象都可以作为锁，这是 synchronized 实现同步的基础

1) 普通同步方法，锁是当前实例对象
2) 静态同步方法，锁是当前类的 class 对象
3) 同步方法块，锁是括号里面的对象

2、synchronized 可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性

- 如何保证内存可见性???对于变量的操作会直接刷新到主内存还是说同步代码块结束后从工作内存刷新到主内存中

3、同步方法：synchronized 方法则会被翻译成普通的方法调用和返回指令如:invokevirtual、areturn 指令，在 VM 字节码层面并没有任何特别的指令来实现被 synchronized 修饰的方法，而是在 Class 文件的方法表中将该方法的 access_flags 字段中的 synchronized 标志位置 1，表示该方法是同步方法并使用调用该方法的对象或该方法所属的 Class 在 JVM 的内部对象表示 Klass 做为锁对象

4、synchronized 是重量级锁，重量级锁通过对象内部的监视器(monitor)实现，其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock 实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高

5、当退出或者抛出异常时必须要释放锁，synchronized 代码块能自动保证这一点

1.1.1. HotSpotJVM 底层实现

1、同步代码块是使用 monitorenter 和 monitorexit 指令实现的，同步方法依靠的是方法修饰符上的 ACC_SYNCHRONIZED 实现

2、同步代码块：monitorenter 指令插入到同步代码块的开始位置，monitorexit 指令插入到同步代码块的结束位置，JVM 需要保证每一个 monitorenter 都有一个 monitorexit 与之相对应(这就是保证在任何情况下退出 synchronized 代码块释放锁的原因)。任何对象都有一个 monitor 与之相关联，当且一个 monitor 被持有之后，他将处于锁定状态。线程执行到 monitorenter 指令时，将会尝试获取对象所对应的 monitor 所有权，即尝试获取对象的锁

1.1.2. 对象头、monitor

1.1.2.1. 对象头

1、synchronized 用的锁是存在 Java 对象头里的，那么什么是 Java 对象头呢？Hotspot 虚拟机的对象头主要包括两部分数据： Mark Word(标记字段)、Klass Pointer(类型指针)。其中 Klass Point 是是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，Mark Word 用于存储对象自身的运行时数据，它是实现轻量级锁和偏向锁的关键

2、Mark Word 用于存储对象自身的运行时数据，如哈希码(HashCode)、GC 分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等等。

3、Java 对象头里的 Mark Word 里默认存储对象的 HashCode，分代年龄和锁标记位。32 位 JVM 的 Mark Word 的默认存储结构如下

25Bit	4bit	1bit	2bit
对象的hashCode	对象的分代年龄	是否是偏向锁	锁标志位

3、在运行期间 Mark Word 里存储的数据会随着锁标志位的变化而变化(约了节省空间)。Mark Word 可能变化为存储以下 5 种数据

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁状态	对象hashcode、对象分代年龄				01
轻量级锁	指向锁记录的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空，不需要记录信息				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01

- 其中偏向锁和无锁状态的锁标志位相同，于是需要用是否偏向锁的标志位来判断是哪一种状态

1.1.1.1. monitor

1、什么是 Monitor

- 我们可以把它理解为一个同步工具，也可以描述为一种同步机制，它通常被描述为一个对象
- 与一切皆对象一样，所有的 Java 对象是天生的 Monitor，每一个 Java 对象都有成为 Monitor 的潜质，因为在 Java 的设计中，每一个 Java 对象自打娘胎里出来就带了一把看不见的锁，它叫做内部锁或者 Monitor 锁

2、Monitor 是线程私有的数据结构，每一个线程都有一个可用 monitor record 列表，同时还有一个全局的可用列表

3、每一个被锁住的对象都会和一个 monitor 关联(对象头的 MarkWord 中的 LockWord 指向 monitor 的起始地址)，同时 monitor 中有一个 Owner 字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。其结构如下

1) Owner：初始时为 NULL 表示当前没有任何线程拥有该 monitor record，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为 NULL

2) EntryQ:关联一个系统互斥锁(semaphore)，阻塞所有试图锁住 monitor record 失败的线程
3) RcThis:表示 blocked 或 waiting 在该 monitor record 上的所有线程的个数
4) Nest:用来实现重入锁的计数
5) HashCode:保存从对象头拷贝过来的 HashCode 值(可能还包含 GC age)
6) Candidate:用来避免不必要的阻塞或等待线程唤醒，因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换(从阻塞到就绪然后因为竞争锁失败又被阻塞)从而导致性能严重下降。Candidate 只有两种可能的值 0 表示没有需要唤醒的线程 1 表示要唤醒一个继任线程来竞争锁

1.2. 锁优化

1、jdk1.6 对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销
2、锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率

1.2.1. 自旋锁

1、线程的阻塞和唤醒需要 CPU 从用户态转为核心态，频繁的阻塞和唤醒对 CPU 来说是一件负担很重的工作，势必会给系统的并发性能带来很大的压力。同时我们发现在许多应用上面，对象锁的锁状态只会持续很短一段时间，为了这一段很短的时间频繁地阻塞和唤醒线程是非常不值得的。所以引入自旋锁
2、所谓自旋锁，就是让该线程等待一段时间，不会被立即挂起，看持有锁的线程是否会很快释放锁。怎么等待呢？执行一段无意义的循环即可(自旋)
3、自旋等待不能替代阻塞，虽然它可以避免线程切换带来的开销，但是它占用了处理器的时间
1) 如果持有锁的线程很快就释放了锁，那么自旋的效率就非常好
2) 反之，自旋的线程就会白白消耗掉处理的资源，它不会做任何有意义的工作，典型的占着茅坑不拉屎，这样反而会带来性能上的浪费
- 所以说，自旋等待的时间(自旋的次数)必须要有一个限度，如果自旋超过了定义的时间仍然没有获取到锁，则应该被挂起。

4、自旋锁在 JDK 1.4.2 中引入，默认关闭，但是可以使用-XX:+UseSpinning 开开启，在 JDK1.6 中默认开启。同时自旋的默认次数为 10 次，可以通过参数-XX:PreBlockSpin 来调整；
- 如果通过参数-XX:preBlockSpin 来调整自旋锁的自旋次数，会带来诸多不便。假如我将参数调整为 10，但是系统很多线程都是等你刚刚退出的时候就释放了锁(假如你多自旋一两次就可以获取锁)，你是不是很尴尬
- 于是 JDK1.6 引入自适应的自旋锁，让虚拟机会变得越来越聪明

1.2.1.1. 另一篇博客中的描述

http://www.cnblogs.com/wade-luffy/p/5969418.html
1、线程的阻塞和唤醒需要 CPU 从用户态转为核心态，频繁的阻塞和唤醒对 CPU

来说是一件负担很重的工作。同时我们可以发现，很多对象锁的锁定状态只会持续很短的一段时间，例如整数的自加操作，在很短的时间内阻塞并唤醒线程显然不值得，为此引入了自旋锁

2、所谓"自旋"，就是让线程去执行一个无意义的循环，循环结束后再去重新竞争锁，如果竞争不到继续循环，循环过程中线程会一直处于 running 状态，但是基于 JVM 的线程调度，会出让时间片，所以其他线程依旧有申请锁和释放锁的机会

3、自旋锁省去了阻塞锁的时间空间(队列的维护等)开销，但是长时间自旋就变成了"忙式等待"，忙式等待显然还不如阻塞锁。所以自旋的次数一般控制在一个范围内，例如 10,100 等，在超出这个范围后，自旋锁会升级为阻塞锁

1. 2. 2. 适应自旋锁

1、JDK 1.6 引入了更加聪明的自旋锁，即自适应自旋锁。所谓自适应就意味着自旋的次数不再是固定的，它是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定

1) 线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能会再次成功，那么它就会允许自旋等待持续的次数更多
2) 反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要或者这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源

2、有了自适应自旋锁，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测会越来越准确，虚拟机会变得越来越聪明

1. 2. 3. 锁消除

1、为了保证数据的完整性，我们在进行操作时需要对这部分操作进行同步控制，但是在有些情况下，JVM 检测到不可能存在共享数据竞争，这是 JVM 会对这些同步锁进行锁消除。锁消除的依据是逃逸分析的数据支持(逃逸分析的另一用处就是让对象在栈上而非堆中分配空间以提高效率)

2、如果不存在竞争，为什么还需要加锁呢？所以锁消除可以节省毫无意义的请求锁的时间。变量是否逃逸，对于虚拟机来说需要使用数据流分析来确定，但是对于我们程序员来说这还不清楚么？我们会在明明知道不存在数据竞争的代码块前加上同步吗？但是有时候程序并不是我们所想的那样？我们虽然没有显示使用锁，但是我们在使用一些 JDK 的内置 API 时，如 StringBuffer、Vector、HashTable 等，这个时候会存在隐形的加锁操作

1. 2. 4. 锁粗化

1、我们知道在使用同步锁的时候，需要让同步块的作用范围尽可能小—仅在共享数据的实际作用域中才进行同步，这样做的目的是为了使需要同步的操作数量尽可能缩小，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。

2、在大多数的情况下，上述观点是正确的。但是如果一系列的连续加锁解锁操作，可能会导致不必要的性能损耗，所以引入锁粗话的概念

3、锁粗话概念比较好理解，就是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁

- 例如：vector 每次 add 的时候都需要加锁操作，JVM 检测到对同一个对象(vector)连续加锁、解锁操作，会合并一个更大范围的加锁、解锁操作，即加锁解锁操作会移到 for 循环之外

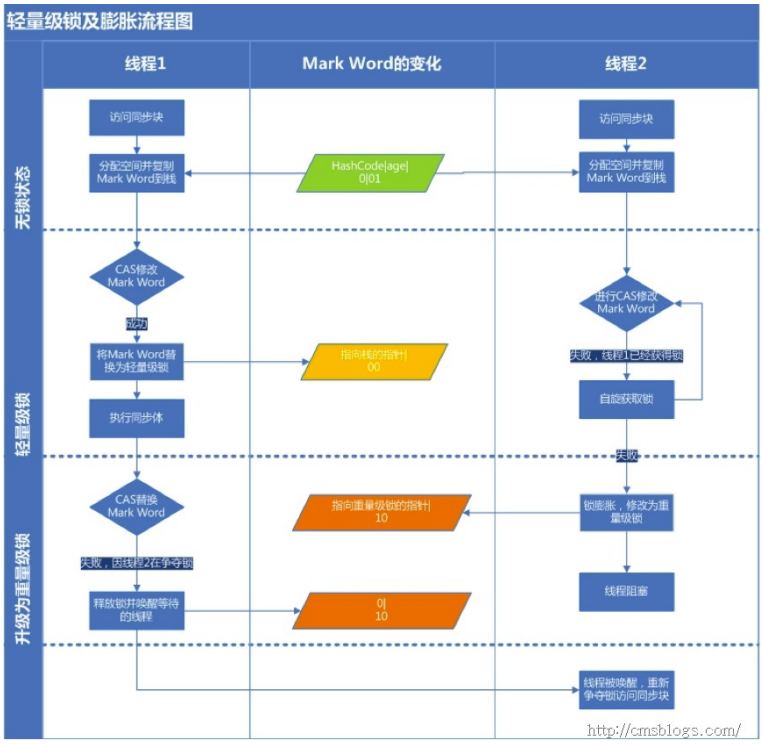
1.2.5. 轻量级锁

1、引入轻量级锁的主要目的是在多没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。当关闭偏向锁功能或者多个线程竞争偏向锁导致偏向锁升级为轻量级锁，则会尝试获取轻量级锁，其步骤如下：

- 获取锁
  1) 判断当前对象是否处于无锁状态(锁标志位 01，偏向锁标志位 0)
     - 若是，则 JVM 首先将在当前线程的栈帧中建立一个名为锁记录(Lock Record)的空间，用于存储锁对象目前的 Mark Word 的拷贝(官方把这份拷贝加了一个 Displaced 前缀，即 Displaced Mark Word)
     - 否则执行步骤(3)
  2) JVM 利用 CAS 操作尝试将对象的 Mark Word 更新为指向 Lock Record 的指正
     - 如果成功表示竞争到锁，则将锁标志位变成 00(表示此对象处于轻量级锁状态)，执行同步操作
     - 如果失败则执行步骤(3)
  3) 判断当前对象的 Mark Word 是否指向当前线程的栈帧
     - 如果是则表示当前线程已经持有当前对象的锁，则直接执行同步代码块
     - 否则只能说明该锁对象已经被其他线程抢占了，这时轻量级锁需要膨胀为重量级锁，锁标志位变成 10，后面等待的线程将会进入阻塞状态
- 释放锁(轻量级锁的释放也是通过 CAS 操作来进行)
  1) 取出在获取轻量级锁保存在 Displaced Mark Word 中的数据
  2) 用 CAS 操作将取出的数据替换当前对象的 Mark Word 中
     - 如果成功，则说明释放锁成功
     - 否则执行(3)
  3) 如果 CAS 操作替换失败，说明有其他线程尝试获取该锁，则需要在释放锁的同时唤醒被挂起的线程

2、对于轻量级锁，其性能提升的依据是"对于绝大部分的锁，在整个生命周期内都是不会存在竞争的"，如果打破这个依据则除了互斥的开销外，还有额外的 CAS 操作，因此在有多线程竞争的情况下，轻量级锁比重量级锁更慢

3、下图是轻量级锁获取和释放过程



1.2.5.1. 另一篇博客中的描述

1、加锁：线程在执行同步块之前，JVM 会先在当前线程的栈桢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，官方称为 Displaced Mark Word。然后线程尝试使用 CAS 将对象头中的 Mark Word 替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，则自旋获取锁，当自旋获取锁仍然失败时，表示存在其他线程竞争锁(两条或两条以上的线程竞争同一个锁)，则轻量级锁会膨胀成重量级锁。

2、解锁：轻量级解锁时，会使用原子的 CAS 操作来将 Displaced Mark Word 替换回到对象头，如果成功，则表示同步过程已完成。如果失败，表示有其他线程尝试过获取该锁，则要在释放锁的同时唤醒被挂起的线程。

1.2.6. 偏向锁

1、引入偏向锁主要目的是：为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径(CAS 原子指令)。

2、那么偏向锁是如何来减少不必要的 CAS 操作呢？我们可以查看 Mark work 的结构就明白了。只需要检查是否为偏向锁、锁标识为以及 ThreadID 即可，处理流程如下：

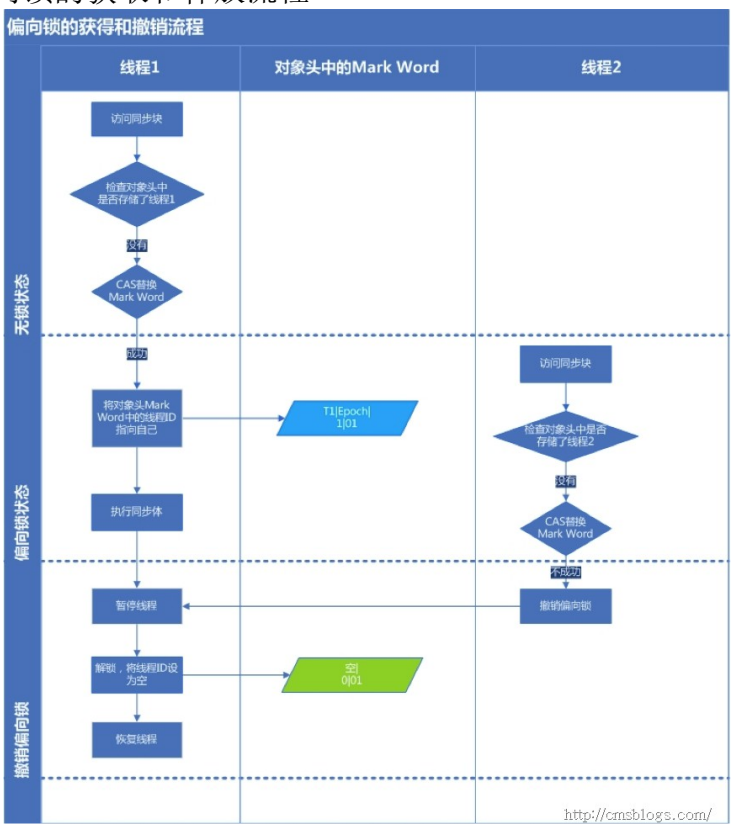
- 获取锁
  1) 检测 Mark Word 是否为可偏向状态(锁标识位 01，偏向锁标志位 1)
  2) 若为可偏向状态，则测试线程 ID 是否为当前线程 ID
     - 如果是，则执行步骤(5)
     - 否则执行步骤(3)
  3) 如果线程 ID 不为当前线程 ID，则通过 CAS 操作竞争锁
     - 竞争成功，将 Mark Word 的线程 ID 替换为当前线程 ID，执行步骤(5)
     - 否则执行步骤(4)

4) 通过 CAS 竞争锁失败，证明当前存在多线程竞争情况，当到达全局安全点，获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码块
5) 执行同步代码块

- 释放锁：偏向锁的释放采用了一种只有竞争才会释放锁的机制，线程是不会主动去释放偏向锁，需要等待其他线程来竞争。偏向锁的撤销需要等待全局安全点(这个时间点是上没有正在执行的代码)。其步骤如下

1) 暂停拥有偏向锁的线程，判断锁对象石是否还处于被锁定状态
2) 撤销偏向锁，恢复到无锁状态(01)或者轻量级锁的状态

2、下图是偏向锁的获取和释放流程



1. 2. 6. 1. 另一篇博客中的描述

1、大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得。偏向锁的目的是在某个线程获得锁之后，消除这个线程锁重入(CAS)的开销，看起来让这个线程得到了偏护。另外，JVM 对那种会有多线程加锁，但不存在锁竞争的情况也做了优化，听起来比较拗口，但在现实应用中确实是可能出现这种情况，因为线程之前除了互斥之外也可能发生同步关系，被同步的两个线程(一前一后)对共享对象锁的竞争很可能是没有冲突的。对这种情况，JVM 用一个 epoch 表示一个偏向锁的时间戳(真实地生成一个时间戳代价还是蛮大的，因此这里应当理解为一种类似时间戳的 identifier)

2、偏向锁的获取：当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程 ID，以后该线程在进入和退出同步块时不需要花费 CAS 操作来加锁和解锁，而只需简单的测试一下对象头的 Mark Word 里是否存储着指向当前线程的偏向锁，如果测试成功，表示线程已经获得了锁，如果测试失败，则需要再测试下 Mark Word 中偏向锁的标识是否设置成 1(表示当前是

偏向锁)，如果没有设置，则使用 CAS 竞争锁，如果设置了，则尝试使用 CAS 将对象头的偏向锁指向当前线程

3、偏向锁的撤销：偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。偏向锁的撤销，需要等待全局安全点(在这个时间点上没有字节码正在执行)，它会首先暂停拥有偏向锁的线程，然后检查持有偏向锁的线程是否活着，如果线程不处于活动状态，则将对象头设置成无锁状态，如果线程仍然活着，拥有偏向锁的栈会被执行，遍历偏向对象的锁记录，栈中的锁记录和对象头的 Mark Word，要么重新偏向于其他线程，要么恢复到无锁或者标记对象不适合作为偏向锁，最后唤醒暂停的线程

4、偏向锁的设置

- 关闭偏向锁：偏向锁在 Java 6 和 Java 7 里是默认启用的，但是它在应用程序启动几秒钟之后才激活，如有必要可以使用 JVM 参数来关闭延迟-XX：BiasedLockingStartupDelay = 0。如果你确定自己应用程序里所有的锁通常情况下处于竞争状态，可以通过 JVM 参数关闭偏向锁-XX:-UseBiasedLocking=false，那么默认会进入轻量级锁状态

1.2.7. 重量级锁

1、重量锁在 JVM 中又叫对象监视器(Monitor)，它很像 C 中的 Mutex，除了具备 Mutex(0|1)互斥的功能，它还负责实现了 Semaphore(信号量)的功能，也就是说它至少包含一个竞争锁的队列，和一个信号阻塞队列(wait 队列)，前者负责做互斥，后一个用于做线程同步

2、重量级锁是使用操作系统互斥量来实现的

1.2.8. 总结

1、偏向锁

- 优点：加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距
- 缺点：如果线程间存在锁竞争，会带来额外的锁撤销的消耗，适用于只有一个线程访问同步块场景

2、轻量级锁

- 竞争的线程不会阻塞，提高了程序的响应速度
- 缺点：如果始终得不到锁竞争的线程使用自旋会消耗 CPU 追求响应时间，锁占用时间很短

3、重量级锁

- 优点：线程竞争不使用自旋，不会消耗 CPU
- 缺点：线程阻塞，响应时间缓慢

2. AQS 框架

2.1. 基本概念

2.1.1. CAS

1、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

2、在 Java 中 CAS 操作是通过 sun.misc.Unsafe 类来实现的，该类采用单例模式，通过 getUnsafe()方法获取唯一实例，但是我们的代码也被禁止调用 getUnsafe()

3、CAS 的优势

1) **不加锁**：现代的 CPU 提供了特殊的指令，可以自动更新共享数据，而且能够检测到其他线程的干扰，而 compareAndSet()就用这些代替了锁定

4、CAS 的劣势

1) **ABA 问题**：因为 CAS 需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是 A，变成了 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，但是实际上却变化了。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，每次变量更新的时候把版本号加一，那么 A-B-A 就会变成 1A-2B-3A
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。如果 JVM 能支持处理器提供的 pause 指令那么效率会有一定的提升，pause 指令有两个作用，第一它可以延迟流水线执行指令(de-pipeline),使 CPU 不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突(memory order violation)而引起 CPU 流水线被清空(CPU pipeline flush)，从而提高 CPU 的执行效率
3) **只能保证一个共享变量的原子操作**：当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。从 Java1.5 开始 JDK 提供了 AtomicReference 类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作

5、锁的另一种分类

1) 独占锁：synchronized 就是一种独占锁，它会导致所有需要此锁的线程挂起，等待锁的释放
2) 乐观锁：每次不加锁去完成操作，如果因为冲突失败就重试，直到成功

2.1.2. 自旋锁

1、自旋锁是指当一个线程尝试获取某个锁时，如果该锁已被其他线程占用，就一直循环检测锁是否被释放，而不是进入线程挂起或睡眠状态

2、自旋锁适用于锁保护的临界区很小的情况，临界区很小的话，锁占用的时间就很短

3、优势

1) lock-free：不加锁(没有唤醒阻塞的系统开销)

4、劣势

1) CPU 开销大

2) 无法响应中断
3) 不支持 FIFO

2.1.3. Ticket Lock

1、优势
1) FIFO
2) lock-free：不加锁(没有唤醒阻塞的系统开销)
2、劣势
1) 无法响应中断
2) CPU 开销大
3) 多个公共线程在共享资源上自旋，开销较大

2.1.4. CLH

1、全称为：Craig，Landin，and Hagersten (CLH)locks
2、优势
1) FIFO
2) lock-free：
3) 加锁(没有唤醒阻塞的系统开销)
4) 仅仅在本地变量上自旋(多个线程不会读写同一个状态资源)
3、劣势
1) 无法响应中断
2) CPU 开销较大

2.1.5. LockSupport

2.1.5.1. suspend/resume、wait/notify

1、suspend()和 resume()方法：
- 这两个方法隶属于 Thread，是 Thread 的非静态方法
- 两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume()被调用，才能使得线程重新进入可执行状态
- 典型地，suspend()和 resume()被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume()使其恢复
- suspend 不能响应中断
- 但 suspend()方法很容易引起死锁问题，已经不推荐使用了
  - 如果一个目标线程 t1 对某一关键系统资源进行了加锁操作，然后在该加锁区块执行 t1.suspend()，那么除非执行 t1.resume()，否则其它线程都将无法访问该系统资源
  - 如果另外一个线程 t2 想要占用资源，那么 t2 必须调用 t1.resume()，如果 t2 调用 t1.resume()之前需要获取该系统资源，那么造成死锁

2、wait()和 notify()方法：
- 这两个方法隶属于 Object，是 Object 的非静态方法
- 两个方法配套使用，wait()使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当

对应的 notify()被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify()被调用

- **必须要在 synchronized 块内使用(保证调用这两个方法时，获取该对象的锁)**，但是不用编译器也不会阻止，运行时可能抛出 IllegalMonitorStateException 异常
- wait 可以响应中断

3、初看起来它们与 suspend()和 resume()方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于：suspend()和 resume()方法，阻塞时都不会释放占用的锁(如果占用了的话)；而 wait()和 notify()方法这一对方法则相反

2.1.5.2. park 与 unpark

1、LockSupport 类是 Java6(JSR166-JUC)引入的一个类，提供了基本的线程同步原语。LockSupport 实际上是调用了 Unsafe 类里的函数，归结到 Unsafe 里，只有两个函数

```
public native void unpark(Thread jthread);
public native void park(boolean isAbsolute, long time);
```

2、unpark 函数为线程提供"许可(permit)"，线程调用 park 函数则等待"许可"

- 这个有点像信号量，但是这个"许可"是不能叠加的，"许可"是一次性的
- 比如线程 B 连续调用了三次 unpark 函数，当线程 A 调用 park 函数就使用掉这个"许可"，如果线程 A 再次调用 park，则进入等待状态

3、park 和 unpark 的灵活之处

- unpark 函数可以先于 park 调用，这个正是它们的灵活之处
- 一个线程它有可能在别的线程 unpark 之前，或者之后，或者同时调用了 park，那么因为 park 的特性，它可以不用担心自己的 park 的时序问题，否则，如果 park 必须要在 unpark 之前，那么给编程带来很大的麻烦！！
- 在 Java5 里是用 wait/notify/notifyAll 来同步的。wait/notify 机制有个很蛋疼的地方是，比如线程 B 要用 notify 通知线程 A，那么线程 B 要确保线程 A 已经在 wait 调用上等待了，否则线程 A 可能永远都在等待。编程的时候就会很蛋疼
- park/unpark 模型真正解耦了线程之间的同步，线程之间不再需要一个 Object 或者其它变量来存储状态，不再需要关心对方的状态
- park 可以响应中断，但不是通过抛出 InterruptedException 的方式来中断，中断后中断标志位是 true

4、HotSpot 里 park/unpark 的实现

2.2. AQS 简介

1、谈到并发，不得不谈 ReentrantLock，而谈到 ReentrantLock，不得不谈 **AbstractQueuedSynchronizer(AQS)**

2、类如其名，抽象的队列式的同步器，AQS 定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的 ReentrantLock/Semaphore/CountDownLatch

2.2.1. AQS：CLH 的变体

1、AQS 通过为 Node 增加 predecessor 字段，实现了 CLH 无法处理 timeouts 以及 cancellation，如果一个节点的前继节点的状态为 cancel，那么该节点可以跳过该前继节点继续往前寻找

2、AQS 相对于 CLH 的一个改进是：提供了快速定位后继节点的方法

- 在 CLH 自旋锁中，一个节点状态的改变，在下一次自旋中将被该节点的后继节点侦测到，因此 link 是不需要的
- 在阻塞同步器中，一个节点必须明确地唤醒(unpark)其后继节点
- 注意：对于目前没有可用的技术可以实现**双向链表的 lock-free 原子插入**，因此 next 指针的维护仅仅是简单的赋值，并没有使用 CAS，因此是非并发安全的(可能其他线程依赖某节点的 next 指针时，其尚未赋值，虽然在未来的某时刻，它会被正确地赋值)
- next 字段仅仅提供了一种快速访问后继节点的**尝试：通过 next 无法获取到某节点的后继节点并不代表真的没有后继节点**

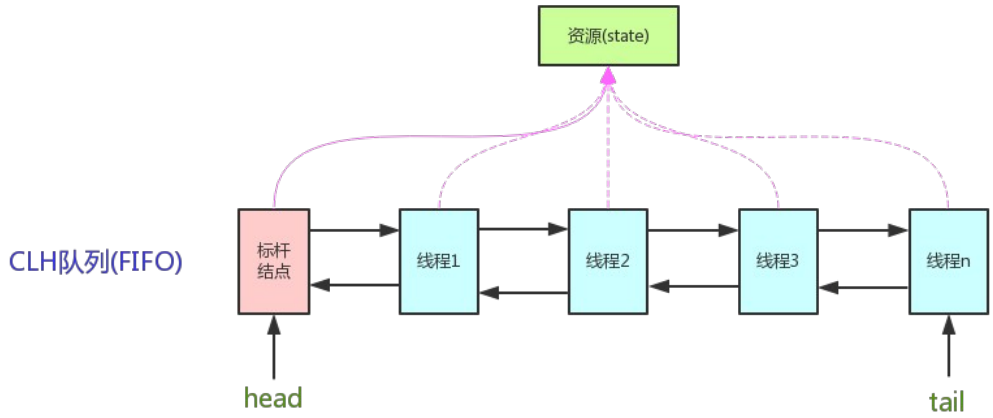
3、AQS 相对于 CLH 的一个改进是：设计节点的状态，用于控制阻塞，而非通过自旋来阻塞当前线程

1) tryAcquire 执行权限的控制
   - 在 AQS 框架中，一个 queued thread 仅能通过执行 tryAcquire 方法且返回 true 来正常返回，tryAcquire 通过其子类来实现
   - 控制的含义在于：确保一个 active 的线程只有在处于队列头部的时候才允许执行 tryAcquire 方法，否则就被阻塞
   - 并不需要为每个节点设置一个状态来表示其是否允许执行 tryAcquire，在 AQS 中仅仅需要判断当前节点的前继是否为 head 来判断该节点是否被允许执行 tryAcquire
2) CANCEL 状态的控制
   - 每个线程的 CANCEL 状态则需要被设计为 Node 节点的字段
3) 阻塞状态的控制
   - 节点的状态用于避免 park 与 unpark 的无效调用，虽然 park 与 unpark 的调用效率与阻塞原语的效率相近
   - 在调用 park 方法之前，线程需要设置"signal me"状态，然后在调用 park 之前再次检查同步和节点状态
   - 释放锁的线程会清除状态
   - 这也避免了要求一个 releasing 的线程取确定其后继直到其后继设定了 signal 状态

4、AQS 与其他 CLH locks 的变体的主要区别是 AQS 依赖于垃圾回收机制来管理节点的存储

1) 依赖于垃圾回收仍然需要 nulling 那些指向这些废弃对象的引用

2.2.2. 框架



1、它维护了一个 volatile int state(代表共享资源)和一个 FIFO 线程等待队列(多线程争用资源被阻塞时会进入此队列)。这里 volatile 是核心关键词。state 的访问方式有三种

1) getState()
2) setState()
3) compareAndSetState()

2、AQS 定义两种资源共享方式：

1) Exclusive(独占，只有一个线程能执行，如 ReentrantLock)
2) Share(共享，多个线程可同时执行，如 Semaphore/CountDownLatch)

3、不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护(如获取资源失败入队/唤醒出队等)，AQS 已经在顶层实现好了。**自定义同步器实现时主要实现以下几种方法**：

1) isHeldExclusively()：该线程是否正在独占资源。只有用到 condition 才需要去实现它
2) tryAcquire(int)：独占方式。尝试获取资源，成功则返回 true，失败则返回 false
3) tryRelease(int)：独占方式。尝试释放资源，成功则返回 true，失败则返回 false
4) tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0 表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源
5) tryReleaseShared(int)：共享方式。尝试释放资源，成功则返回 true，失败则返回 false

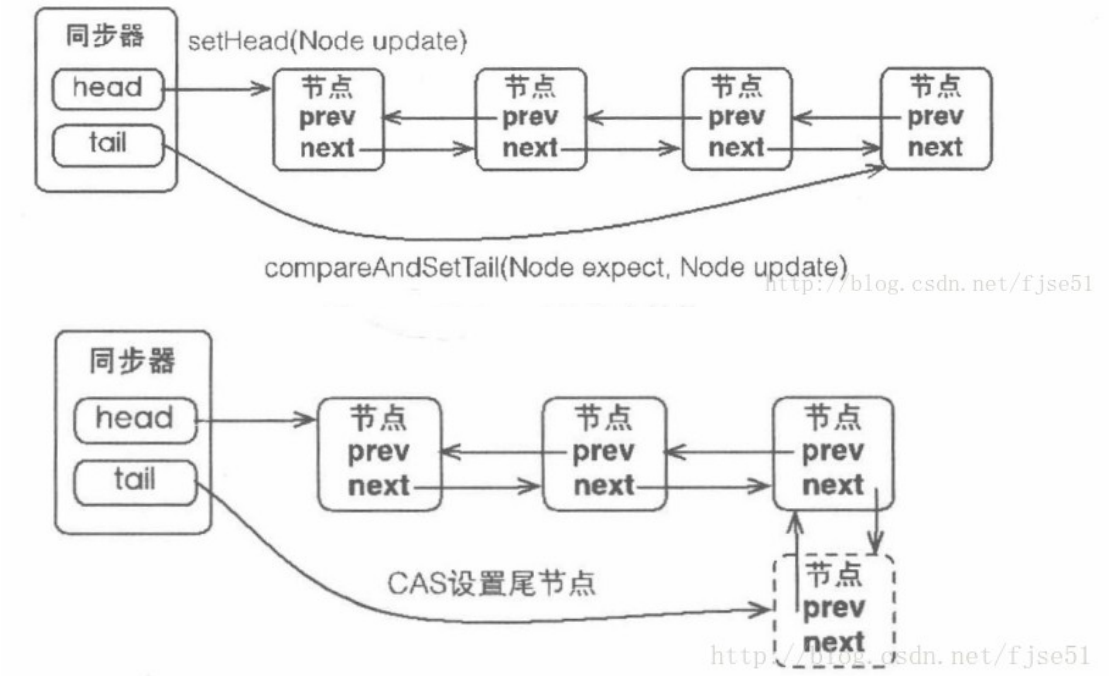
- 以 ReentrantLock 为例，state 初始化为 0，表示未锁定状态。A 线程 lock()时，会调用 tryAcquire()独占该锁并将 state+1。此后，其他线程再 tryAcquire()时就会失败，直到 A 线程 unlock()到 state=0(即释放锁)为止，其它线程才有机会获取该锁。当然，释放锁之前，A 线程自己是可以重复获取此锁的(state 会累加)，这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证 state 是能回到零态的
- 再以 CountDownLatch 以例，任务分为 N 个子线程去执行，state 也初始化

为 N(注意 N 要与线程个数一致)。这 N 个子线程是并行执行的，每个子线程执行完后 countDown()一次，state 会 CAS 减 1。等到所有子线程都执行完后(即 state=0)，会 unpark()主调用线程，然后主调用线程就会从 await()函数返回，继续后余动作

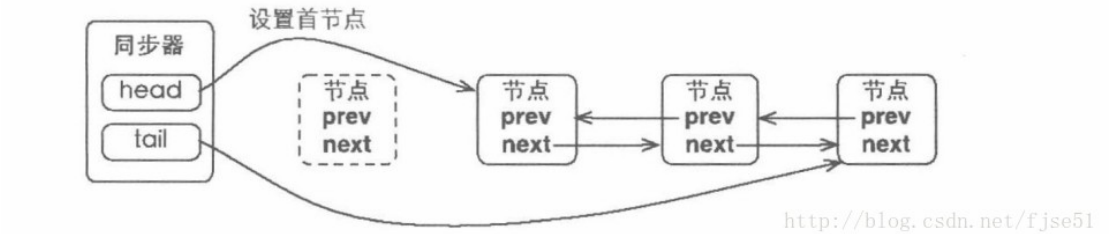
- 一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现 tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared 中的一种即可。但 AQS 也支持自定义同步器同时实现独占和共享两种方式，如 ReentrantReadWriteLock

4、同步队列基本结构

- Node 是构成同步队列的基础，AQS 拥有首节点(head)和尾节点(tail)，没有成功获取资源的线程将会放入到队列的尾部
- 同步器中包含了两个节点类型的引用，一个指向头节点(head)，一个指向尾节点(tail),没有获取到锁的线程，加入到队列的过程必须保证线程安全，因此同步器提供了一个基于 CAS 的设置尾节点的方法 CompareAndSetTail(Node expect,Node update)，它需要传递当前线程认为的尾节点和当前节点，只有设置成功后，当前节点才能正式与之前的尾节点建立关联



5、同步队列遵循 FIFO，首节点是获取锁成功的节点，首节点的线程在释放锁时，将会唤醒后继节点，而后继节点将会在获取到锁时，将自己设置位首节点，过程如下所示



6、设置首节点是由成功获取锁的线程来完成的，由于只有一个线程能够成功获

取锁，因此设置首节点不需要 CAS 操作

2.2.3. 伪代码

```
if (!tryAcquire(arg)){
    node = create andenqueue new node;
    pred = node's effective predecessor;
    while (pred is not headnode || !tryAcquire(arg)) {
        if (pred's signal bitisset)
            park();
        else
            compareAndSet pred's signal bit to true;
        pred = node's effective predecessor;
    }
    head = node;
}

if (tryRelease(arg) &&head node's signal bit is set) {
    compareAndSet head'ssignal bit to false;
    unpark head's successor,if oneexists
}
```

2.3. 源码详解

2.3.1. 静态内部类 Node

1、Node 是 AbstractQueuedSynchronizer 的一个静态内部类

2、字段

表示节点正处在共享模式下等待的标记

```
static final Node SHARED = new Node();
```

表示节点正在以独占模式等待的标记

```
static final Node EXCLUSIVE = null;
```

waitStatus 值，表示线程已取消

```
static final int CANCELLED = 1;
```

waitStatus 值，后继节点将当前节点设为 SIGNAL，意味着当前节点释放后，有义务唤醒后继节点

```
static final int SIGNAL = -1;
```

waitStatus 值，表示线程正在等待条件

```
static final int CONDITION = -2;
```

waitStatus 值，共享模式的头结点可能处于此状态，表示无条件往下传播，引入此状态是为了优化锁竞争，使队列中线程有序地一个一个唤醒

```
static final int PROPAGATE = -3;
```

状态字段，仅接受值：

1) *SIGNAL:值为-1，后继节点的线程处于等待状态，而当前节点的线程如果释放了资源或者被取消，将会通知后继节点，使后继节点的线程得以运行*
2) *CANCELLED:值为1，由于在同步队列中等待的线程等待超时或者被中*

断，需要从同步队列中取消等待，节点进入该状态将不会变化

3) *CONDITION: 值为-2，节点在等待队列中，节点线程等待在 Condition 上，当其他线程对 Condition 调用了 singal 方法后，该节点将会从等待队列中转移到同步队列中，加入到对资源的获取中*
4) *PROPAGATE: 值为-3，表示下一次共享模式资源获取将会无条件地传播下去???*
5) *INITIAL: 初始状态值为0，在此状态下，独占模式不会释放后继节点*

volatile int waitStatus;

链接到前驱节点，当前节点/线程依赖它来检查 waitStatus。在入同步队列时被设置，并且仅在移除同步队列时才归零(为了 GC 的目的)。被取消的线程永远不会成功获取，并且线程只取消自身，而不是任何其他节点

volatile Node prev;

链接到后续节点，当前节点/线程释放时释放。在入同步队列期间分配，在绕过取消的前驱节点时调整，并在出同步队列时取消(为了 GC 的目的)。enq 操作不会分配前驱节点的 next 字段，直到附加之后，因此看到一个为 null 的 next 字段不一定意味着该节点在队列的末尾。但是，如果 next 字段显示为 null,我们可以从尾部扫描 prev，仔细检查。被取消的节点的 next 字段被设置为指向节点本身而不是 null，以使 isOnSyncQueue 更方便操作。调用 isOnSyncQueue 时，如果节点(始终是放置在条件队列上的节点)正等待在同步队列上重新获取，则返回 true

volatile Node next;

volatile Thread thread;

Node nextWaiter;

- 状态为 0 代表刚初始化，其后面没有需要唤醒的节点，或者为已经释放资源的节点

2.3.2. 重要字段

1、首尾节点

private transient volatile Node head;// head 所指的标杆结点，就是当前获取到资源的那个结点或 null(null 即代表队列尚未初始化)

private transient volatile Node tail;//当队列初始化完毕后，该 tail 为队列中最后一个节点(并非尾后节点，是有效节点)，若 tail==null 则队列为空

2、状态

private volatile int state;

2.3.3. acquire(int)

1、此方法是独占模式下线程获取共享资源的顶层入口。如果获取到资源，线程直接返回，否则进入等待队列，直到获取到资源为止，且整个过程忽略中断的影响。这也正是 lock()的语义，当然不仅仅只限于 lock()。获取到资源后，线程就可以去执行其临界区代码了。下面是 acquire()的源码

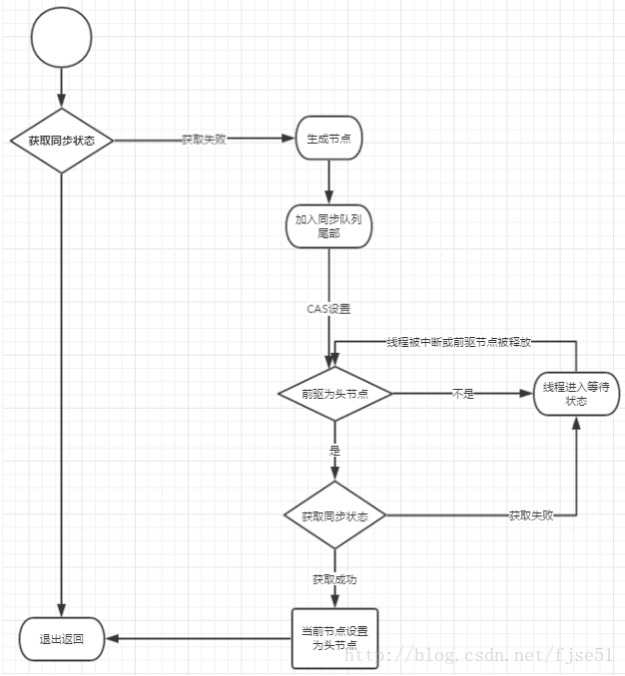
```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
```

```
            selfInterrupt();
        }
```

2、函数流程如下：

1) tryAcquire()尝试直接去获取资源，如果成功则直接返回
2) addWaiter()将该线程加入等待队列的尾部，并标记为独占模式
3) acquireQueued()使线程在等待队列中以"死循环"(适量的自旋)的方式获取资源，**一直获取到资源后才返回，**如果在整个等待过程中被中断过，则返回 true，否则返回 false
4) 如果线程在等待过程中被中断过，它是不响应的(意思就是被吞了)。只是获取资源后才再进行自我中断 selfInterrupt()，将中断补上

3、独占式资源获取流程如下



2.3.3.1. tryAcquire(int)

1、此方法尝试去获取独占资源。如果获取成功，则直接返回 true，否则直接返回 false。这也正是 tryLock()的语义，还是那句话，当然不仅仅只限于 tryLock()。如下是 tryAcquire()的源码：

```
    protected boolean tryAcquire(int arg) {
        throw new UnsupportedOperationException();
    }
```

2、AQS 只是一个框架，具体资源的获取/释放方式交由自定义同步器去实现。这里之所以没有定义成 abstract，是因为独占模式下只用实现 tryAcquire-tryRelease，而共享模式下只用实现 tryAcquireShared-tryReleaseShared。如果都定义成 abstract，那么每个模式也要去实现另一模式下的接口

2.3.3.2. addWaiter(Node)

1、此方法用于将当前线程加入到等待队列的队尾，并返回当前线程所在的结点，如下是源码

```
    private Node addWaiter(Node mode) {
```

```
        以给定模式构造结点。mode 有 EXCLUSIVE(独占)和 SHARED(共享)
        当前线程会被绑定到这个 Node 节点中去
        Node node = new Node(Thread.currentThread(), mode);
        尝试快速方式直接放到队尾，以下情况会失败：
          1)  当 tail 为 null，此时可能该队列尚未初始化
          2)  当调用 compareAndSetTail 执行 CAS 时失败，即当代表当前线程
              的 Node 节点插入失败(有代表其他线程插入成功)
        Node pred = tail;
        if (pred != null) {
            此时 node 还不知道是否可以入队(下一句 CAS 操作不知道是否能
            返回 true)，但是修改 node 节点的 prev 指针是无害的!!!
            node.prev = pred;
            if (compareAndSetTail(pred, node)) {
                这里修改了队列中原有节点的指针，如果在 CAS 之后，该句
                之前有其他线程访问队列，如果依赖节点的 next 指针，可能
                会出现问题
                pred.next = node;
                return node;
            }
        }
        上一步失败则通过 enq 入队
        enq(node);
        return node;
    }
```

2.3.3.2.1. enq(Node)

1、此方法用于将 node 加入队尾。源码如下

```
    private Node enq(final Node node) {
        不断循环，直到成功加入队尾
        for (;;) {
            Node t = tail;
            if (t == null) {
                此时队列是空的，通过 CPU 完成 CAS 执行队列初始化动作，
                此时 head 和 tail 指向同一个标记节点，队列为空(已初始化完
                毕，但是没有有效节点)
                记这个 new Node() 节点为 DummyNode(非静态字段都是 0 或
                null，该节点无关联线程，自然也不会主动释放)，这个
                DummyNode 节点如何唤醒其后继节点???
                if (compareAndSetHead(new Node()))
                    compareAndSetHead 对 head 的期望值为 null
                    tail = head;
            } else {
                node.prev = t;
                利用 CPU 的 CAS 指令尝试放入尾部，该 CAS 的线程安全性保
                证同一时刻只有一个节点插入到尾部，也就是 node.prev 一定
```

```
                    是正确的
                    if (compareAndSetTail(t, node)) {
                    如果此处其他线程正在执行 unparkSuccessor，查找后继有效节点，此时是不能依赖于 next 指针的，因为此时尚未赋值，会导致从前往后无法遍历整个队列，因此必须从后往前遍历，因为 prev 是保证已经连上的
                        t.next = node;
                        return t;
                    }
                }
            }
        }
```

2、通过 compareAndSetTail(Node expect,Node update)方法来确保节点能够被线程安全的添加到同步队列的尾部。在 enq(final Node node)方法中，同步器通过"死循环"来保证节点的正确添加，在死循环中，只有通过 CAS 将节点设置为尾节点后，当前线程才能从该方法返回，否则，当前线程不断得尝试设置。**enq(final Node node)方法将并发添加节点的请求通过 CAS 变得串行化了**

2.3.3.3. acquireQueued(Node, int)

1、通过 tryAcquire()和 addWaiter()，该线程获取资源失败，已经被放入等待队列尾部了。该线程下一步：进入等待状态休息，直到其他线程彻底释放资源后唤醒自己，自己再拿到资源，然后就可以去干自己想干的事了

2、节点进入到同步队列后，进入了一个自旋的过程，每个节点都在自省的观察，当条件满足，获取到了资源时，就可以从这个自旋中退出，否则继续自旋并且阻塞节点的线程

3、源码如下

```
    final boolean acquireQueued(final Node node, int arg) {
        标记是否成功拿到资源
        boolean failed = true;
        try {
            标记是否被中断过，因为如果产生中断会被吞，因此要将该参数传出，以供外层函数重新恢复中断
            boolean interrupted = false;
            for (;;) {
                final Node p = node.predecessor();
                如果该节点的前继节点 p 是 head，有以下三种情况
                1) 可能是 p 节点释放资源唤醒 node 节点
                2) 可能是 node 节点被中断了
                3) p 为 DummyNode 节点，该节点没有设置 thread 字段，此时执行 tryAcquire(arg)仍然有可能失败，原因如下：
                    ● 为什么当前线程会执行到这里的原因就是因为当前线程执行 tryAcquire()失败，即代表了有个线程 t_out 正在占用资源，该线程 t_out 从未进入过队列。如果此时还未释放资源，因此 node 节点再次尝试获取
```

```
                    会失败
                    ● 在这种情况下，DummyNode 通知其后继节点是通过 t_out 执行 release 来触发的，详见 release(int)
                if (p == head && tryAcquire(arg)) {
                    ➢ 如果获取成功，那么将 head 指向该节点，因此 head 指向的节点就是当前获取到资源的节点或者 null
                    ➢ setHead 函数做三件事：
                      1) 将 head 置为 node
                      2) 置空 node.thread 字段
                      3) 置空 node.prev 字段
                    ➢ 这段代码只有获取了锁的线程才能够执行，因此不会引发线程安全问题
                    setHead(node);
                    p.next = null;
                    标记已经成功拿到资源
                    failed = false;
                    return interrupted;
                }
                如果 node 的前继节点并非 head，或者 node 抢占资源失败，那么进入睡眠
                if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                如果失败了，就将该节点的状态变为 cancel
                cancelAcquire(node);
        }
    }
```

4、只有当线程的前驱节点是头节点才能继续获取资源，原因如下

1) 头节点是成功获取到资源的节点，而头节点的线程释放了资源后，将会唤醒后继节点，后继节点的线程被唤醒后需要检查自己的前驱节点是否是头节点
2) 维护同步队列的 FIFO 原则

2.3.3.3.1. shouldParkAfterFailedAcquire(Node, Node)

1、此方法主要用于检查状态，看看自己是否真的可以去休息了，因为在队列中的节点并非全部有效节点，那些放弃执行的节点仍然会处于队列中，等待这种放弃的节点自然是不可行的，并且这些放弃的节点是由该方法来清除的

2、源码如下

```
    private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;
        if (ws == Node.SIGNAL)
```

- *如果已经告诉前驱拿完号后通知自己一下，那就可以安心休息了*
- *一种极端情况：此时 head 正在执行，且状态为 SIGNAL，在 node 节点 park 之前，head 唤醒了 node 节点，即当前获取锁的线程先 unpark 后继线程，然后后继线程再 park，这样也是可以的，这正是 park/unpark 的灵活之处*

return true;

if (ws > 0) {

- ***此时 ws 的状态只能是 CANCEL***
- *如果前驱放弃了，那就一直往前找，直到找到第一个处于正常等待状态(SIGNAL 或 0)的节点，并排在它的后边*
- *注意，这里并没有重置这些异常节点的 next 与 prev 字段，因为可以保证的是，node 节点，与第一个找到的正常节点并没有指向这些异常节点，这些异常节点经过 GC 的可达性分析将会被判为不可达(废弃节点相互引用，也会被 GC 回收)*

do {

node.prev = pred = pred.prev;//这里修改 node 的 prev 字段

这里 while 不用去判断 pred 是否为空，因为至少 head 是正在执行的线程不会>0

} while (pred.waitStatus > 0);

pred.next = node;

- *以上修改 node 节点的 prev 指针和 pred 的 next 指针的语句是并发安全的，因为其他线程对于队列的修改并不在这之中*

} else {

- *此时 ws 的状态可以是 0,-2(CONDITION),-3(PROPAGATE)*
- *如果前驱正常，那就把前驱的状态设置成 SIGNAL，告诉它拿完号后通知自己一下*
- *队列初始化插入的 DummyNode 节点，其 waitState 为 0，是通过这里将其变为 SIGNAL 的*
- *waitStatus must be 0 or PROPAGATE ???*
- *这里尝试将 pred 的状态改为 SINGNAL(可能此时 pred 节点状态会改为 CANCELL，因此可能失败)*

compareAndSetWaitStatus(pred, ws, Node.SIGNAL);

}

但是不立即 park，返回 false 后通过 acquiredQueue 再次尝试获取锁，为什么需要返回 false

1) *前继节点状态为 CANCEL，向前找到有效节点，删除那些无效节点*
2) *前继节点状态正常，但不为 SIGNAL，设置前继节点状态 SIGNAL 可能会失败*

return false;

}

3、整个流程中

1) 如果前驱节点状态是 SIGNAL，那么去休息(在休息之前，前节点可能会执

行 release，此时先 unpark 再 park)

2) 如果前驱节点状态为 CANCEL，找到有效的前继节点，并再次尝试拿号(适量自旋)

3) 如果前驱结点的状态不是 SIGNAL，尝试(此时该节点可能被 CANCELL，或者被释放变为状态 0)将其状态改为 SIGNAL，同时可以再尝试下看有没有机会轮到自己拿号(适量自旋)

2.3.3.3.2. parkAndCheckInterrupt()

1、如果线程找好安全休息点后，那就可以安心去休息了。**此方法就是让线程去休息，真正进入等待状态**

```
private final boolean parkAndCheckInterrupt() {
    调用 park()使线程进入 waiting 状态
    LockSupport.park(this);
    如果被唤醒，查看自己是不是被中断的，并且重置中断标志位
    return Thread.interrupted();
}
```

2、park()会让当前线程进入 waiting 状态。在此状态下，有两种途径可以唤醒该线程：

1) 被 unpark()

2) 被 interrupt()

2.3.3.3.3. 小结

1、看了 shouldParkAfterFailedAcquire()和 parkAndCheckInterrupt()，现在让我们再回到 acquireQueued()，总结下该函数的具体流程：

1) 结点进入队尾后，检查状态，找到安全休息点

2) 调用 park()进入 waiting 状态，等待 unpark()或 interrupt()唤醒自己

3) 被唤醒后，看自己是不是有资格能拿到号。如果拿到，head 指向当前结点，并返回从入队到拿到号的整个过程中是否被中断过；如果没拿到，继续流程 1

2.3.3.4. 小结

1、再来总结下 acquire 的流程

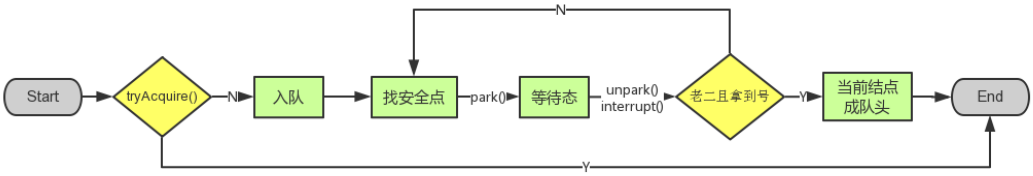
1) 调用自定义同步器的 tryAcquire()尝试直接去获取资源，如果成功则直接返回

2) 没成功，则 addWaiter()将该线程加入等待队列的尾部，并标记为独占模式

3) acquireQueued()使线程在等待队列中休息，有机会时(轮到自己，会被 unpark())会去尝试获取资源。获取到资源后才返回。如果在整个等待过程中被中断过，则返回 true，否则返回 false

4) 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断 selfInterrupt()，将中断补上。

2、再用流程图总结一下



2.3.4. release(int)

1、此方法是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了(即 state=0)，它会唤醒等待队列里的其他线程来获取资源。这也正是 unlock()的语义，当然不仅仅只限于 unlock()

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        h 节点有以下两种情况
          1)  h 节点此时关联的线程此时正在释放资源
          2)  h 节点为 DummyNode(此时调用 release()的线程压根没有入队过)
        Node h = head;
        如果 h 为空或者 h 的状态为 0，则不唤醒后继节点
        if (h != null && h.waitStatus != 0)
            无论 h 节点是当前线程关联的节点还是 DummyNode，都会唤醒 h 之后的有效节点
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

2.3.4.1. tryRelease(int)

1、此方法尝试去释放指定量的资源。下面是 tryRelease()的源码

```
protected boolean tryRelease(int arg) {
    throw new UnsupportedOperationException();
}
```

2、跟 tryAcquire()一样，这个方法是需要独占模式的自定义同步器去实现的。正常来说，tryRelease()都会成功的，因为这是独占模式，该线程来释放资源，那么它肯定已经拿到独占资源了，直接减掉相应量的资源即可(state-=arg)，也不需要考虑线程安全的问题。但要注意它的返回值，上面已经提到了，release()是根据 tryRelease()的返回值来判断该线程是否已经完成释放掉资源了！所以自义定同步器在实现时，如果已经彻底释放资源(state=0)，要返回 true，否则返回 false

2.3.4.2. unparkSuccessor(Node)

1、此方法用于唤醒等待队列中下一个线程。下面是源码

```
private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
```

```
        if (ws < 0)
            由于该方法会唤醒其有效后继，因此该节点的SIGNAL状态可以被清空了(SIGNAL就是后继节点让该节点释放时取唤醒它)
            可能失败原因???
            compareAndSetWaitStatus(node, ws, 0);
        s代表node的下一个有效节点
        这里进行了一次尝试：直接定位当前节点的后继节点，但不一定成功，因为next指针是不可靠的
          1) Node的next指针的赋值并没有使用CAS，可能在赋值前就会有访问
          2) 如果某节点的next指针非空，那么它一定指向的是正确的节点
          3) next的不可依赖性仅仅指使用时机，而非指向
        Node s = node.next;
        if (s == null || s.waitStatus > 0) {
            s = null;
            ➢ s为空可能是由于node的next节点尚未赋值，并非后继无节点了
            ➢ 这里必须从后往前找，因为next指针是不可靠的，prev指针是可靠的
            for (Node t = tail; t != null && t != node; t = t.prev)
                if (t.waitStatus <= 0)
                    s = t;
        }
        if (s != null)
            唤醒下一个有效节点
            LockSupport.unpark(s.thread);
    }
```

2、这个函数并不复杂。一句话概括：**用 unpark()唤醒等待队列中最前边的那个未放弃线程，这里我们也用 s 来表示吧**。此时，再和 acquireQueued()联系起来，s 被唤醒后，进入 if (p == head && tryAcquire(arg))的判断(即使 p!=head 也没关系，它会再进入 shouldParkAfterFailedAcquire()寻找一个安全点。这里既然 s 已经是等待队列中最前边的那个未放弃线程了，那么通过 shouldParkAfterFailedAcquire()的调整，s 也必然会跑到 head 的 next 结点，下一次自旋 p==head 就成立啦)，然后 s 把自己设置成 head 标杆结点，表示自己已经获取到资源了，acquire()也返回了！

2.3.4.3. 小结

1、release()是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了(即 state=0)，它会唤醒等待队列里的其他线程来获取资源

2.3.5. acquireShared(int)

1、此方法是共享模式下线程获取共享资源的顶层入口。它会获取指定量的资源，获取成功则直接返回，获取失败则进入等待队列，直到获取到资源为止，整个过程忽略中断。下面是 acquireShared()的源码

```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
```

2、这里 tryAcquireShared()依然需要自定义同步器去实现。但是 AQS 已经把其返回值的语义定义好了：

1) 负值代表获取失败
2) 0 代表获取成功，但没有剩余资源
3) 正数表示获取成功，还有剩余资源，其他线程还可以去获取

3、所以这里 acquireShared()的流程就是：

1) tryAcquireShared()尝试获取资源，成功则直接返回
2) 失败则通过 doAcquireShared()进入等待队列，直到获取到资源为止才返回

4、共享式获取与独占式获取的最主要区别在于同一时刻能否有多个线程同时获取到资源。通过调用 acquireShared(int arg)方法可以共享式得获取资源

2.3.5.1. doAcquireShared(int)

1、此方法用于将当前线程加入等待队列尾部休息，直到其他线程释放资源唤醒自己，自己成功拿到相应量的资源后才返回。下面是 doAcquireShared()的源码

```
private void doAcquireShared(int arg) {
    加入队列尾部
    final Node node = addWaiter(Node.SHARED);
    标志是否成功
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            如果 node 是 head 的后继节点，因为 head 是拿到资源的线程(或 DummyNode)，此时 node 被唤醒，很可能是 head(或者 t_out)用完资源来唤醒自己的
            if (p == head) {
                尝试获取资源
                int r = tryAcquireShared(arg);
                获取成功
                if (r >= 0) {
                    将 head 指向自己，还有剩余资源可以再唤醒之后的线程，如果 node 节点的后继节点依然成功获取资源，那么 node 节点将被直接移出队列
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    如果等待过程中被打断过，此时将中断补上
                    if (interrupted)
                        selfInterrupt();
                    failed = false;
                    return;
```

```
                    }
                }
                if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

2、跟独占模式比，还有一点需要注意的是，这里只有线程是 head.next 时("老二")，才会去尝试获取资源，有剩余的话还会唤醒之后的队友。那么问题就来了，假如老大用完后释放了 5 个资源，而老二需要 6 个，老三需要 1 个，老四需要 2 个。因为老大先唤醒老二，老二一看资源不够自己用继续 park()，也更不会去唤醒老三和老四了。独占模式，同一时刻只有一个线程去执行，这样做未尝不可；但共享模式下，多个线程是可以同时执行的，现在因为老二的资源需求量大，而把后面量小的老三和老四也都卡住

3、**只要进了队列，只要前一个不被唤醒，后续的节点就不会被唤醒**

2.3.5.1.1. setHeadAndPropagate(Node, int)

1、源码如下

```
    private void setHeadAndPropagate(Node node, int propagate) {
        Node h = head; // Record old head for check below
        setHead(node);
        如果还有剩余量，继续唤醒下一个邻居线程
        if (propagate > 0 || h == null || h.waitStatus < 0) {
            Node s = node.next;
            if (s == null || s.isShared())
                doReleaseShared();
        }
    }
```

2、此方法在 setHead()的基础上多了一步，就是自己苏醒的同时，如果条件符合(比如还有剩余资源)，还会去唤醒后继结点，毕竟是共享模式

2.3.5.2. 小结

1、acquireShared()的流程

1) tryAcquireShared()尝试获取资源，成功则直接返回
2) 失败则通过 doAcquireShared()进入等待队列 park()，直到被 unpark()/interrupt()并成功获取到资源才返回。整个等待过程也是忽略中断的。

2、其实跟 acquire()的流程大同小异，只不过多了个自己拿到资源后，还会去唤醒后继队友的操作(共享)

2.3.6. releaseShared()

1、此方法是共享模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了(即 state=0)，它会唤醒等待队列里的其他线程来获取资源。下面是 releaseShared()的源码

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

2、此方法的流程也比较简单，一句话：释放掉资源后，唤醒后继。

3、跟独占模式下的 release()相似，但有一点稍微需要注意：

1) 独占模式下的 tryRelease()在完全释放掉资源(state=0)后，才会返回 true 去唤醒其他线程，这主要是基于可重入的考量

2) 共享模式下的 releaseShared()则没有这种要求，一是共享的实质--多线程可并发执行；二是共享模式基本也不会重入吧(至少我还没见过)，所以自定义同步器可以根据需要决定返回值

4、**该方法与独占式主要区别在于 tryReleaseShared(int arg)方法必须确保资源线程安全释放。一般是通过循环和 CAS 来保证的。因为释放资源的操作会同时来自多个线程**

2.3.6.1. doReleaseShared()

1、源码如下

```
private void doReleaseShared() {
    /*该函数必须保证能够有效传递，即便有其他 in-progress 线程正在执行 acquireShared 或者 releaseShared
    如果当前状态为0，必须保证能够有效传递，因此设为PROPAGATE???*/
    for (;;) {
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            if (ws == Node.SIGNAL) {
                /*可能失败的原因：有多个活动线(只有一个是队列头，其他在队列之外)程正在 release，为了避免多次唤醒后继节点，于是采用 CAS 操作
                为什么要在这里执行将节点状态从 SIGNAL 到0 的转变，因为 unparkSuccessor 中会将节点从负状态转换为 0 状态，如果多个线程同时执行 unparkSuccessor，将会导致多发 park 信号量，因此必须保证只有一个线程可以唤醒其后继节点*/
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue;// loop to recheck cases
                /*无论活动线程是否在队列中，只能唤醒头节点的后继
```

```
                    unparkSuccessor(h);
                    如果唤醒成功，那么该后继将会取代当前头结点成为队列的头结点，从而会导致该函数最后判断返回 false，继续循环传播
                    如果唤醒失败，那么后继又会重新将该头结点设为 SIGNAL 状态，并且头状态没有变化退出循环，此次传播以失败告终
                }
                只有这一处会将节点设为 PROPAGATE 状态，PROPAGATE 是一个非持久状态
                如果头结点的状态为 0，可能情况如下：
                  1) 上一次循环已经将头结点状态置为 0，因此需要将其置为 PROPAGATE，以表示正处于传播状态，避免再次进入该 if 语句执行 CAS 操作，节省开销
                  2) 后面不可能没有节点，因为 head==tail 进不到这里
                为什么要将其从 0 变为 PROPAGATE：假设上一次循环 t1 将 head 从 SIGNAL，变为 0，并唤醒了后继。此时如果紧跟着 t2 释放资源，也会执行该函数，会尝试将 0 变为 PROPAGATE，如果成功，t2 完成任务。此时如果 t3 释放资源，看到 head 状态时 PROPAGATE，就不会执行任何 CAS 操作
                else if (ws == 0 &&
                !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                    continue; // loop on failed CAS
            }
            如果头结点没有变化，说明唤醒的线程获取不了锁(需要的资源大于现有的资源)
            if (h == head) // loop if head changed
                break;
        }
    }
```

2、这个方法就一个目的，就是把当前结点设置为 SIGNAL 或者 PROPAGATE，如果当前结点不为空且不是尾结点，先判断当前结点的状态位是否为 SIGNAL，如果是就设置为 0，**因为共享模式下更多使用 PROPAGATE 来传播**，SIGNAL 会被经过两步改为 PROPAGATE:

1) compareAndSetWaitStatus(h, Node.SIGNAL, 0)
2) compareAndSetWaitStatus(h, 0, Node.PROPAGATE)

3、为什么要经过两步呢

1) 博客上给出的原因：原因在 unparkSuccessor 方法：

```
private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);
    ...
}
```

- 如果直接从 SIGNAL 到 PROPAGATE,那么到 unparkSuccessor 方法里面又被设置为 0：SIGNAL--PROPAGATE---0----PROPAGATE
- 对头结点相当于多做了一次 compareAndSet 操作，其实性能也殊途同归啦

2) 我的理解

- 如果头结点为 SIGNAL 状态，那么将其设为 0，并唤醒后继节点，执行 doAcquireShared 方法，如果获取失败，又会将其有效前继设为 SIGNAL 然后 park
- 如果头结点为 0 状态，那么将其设为 PROPAGATE，此时无需再次唤醒其后继节点，因为之前已经做过尝试了，避免无效的 park/unpark 开销

4、在 acquireShared(int arg)方法中，同步器调用 tryAcquireShared(int arg)方法尝试获取资源，其返回值为 int 类型，当返回值大于 0 时，表示能够获取资源。因此，在共享式获取的自旋过程中，成功获取资源并且退出自旋的条件就是 tryAcquireShared(int arg)方法返回值大于等于 0。共享式释放资源状态是通过调用 releaseShared(int arg)方法

3. ThreadLocal

4. BlockingQueue

5. ThreadPool

concurrent 结构