# Unsafe

## 简介

Java最初被设计为一种安全的受控环境。尽管如此，HotSpot还是包含了一个后门sun.misc.Unsafe，提供了一些可以直接操控内存和线程的底层操作。Unsafe被JDK广泛应用于java.nio和并发包等实现中，这个不安全的类提供了一个观察HotSpot JVM内部结构并且可以对其进行修改，但是不建议在生产环境中使用

## 如何使用

1、Unsafe类是如此地不安全，以至于JDK开发者增加了很多特殊限制来访问它。

1. 私有的构造器
2. 工厂方法getUnsafe()的调用器只能被Bootloader加载，否则抛出SecurityException 异常

2、不过，我们可以通过反射机制轻松获取Unsafe的一个实例

1. **public** **static** Unsafe getUnsafe() {
2. **try** {
3. Field f = Unsafe.**class**.getDeclaredField("theUnsafe");
4. f.setAccessible(**true**);
5. **return** (Unsafe)f.get(**null**);
6. } **catch** (Exception e) {
7. **return** **null**;
8. }
9. }

## 一些有用的方法

### objectFieldOffset

1、objectFieldOffset：能够获取到指定实例变量的在对象内存中的偏移量

1. **public** **native** **long** objectFieldOffset(Field var1);

### compareAndSwapInt

1、compareAndSwapInt：通过比较并替换的机制，修改指定偏移量内存的值

1. **public** **final** **native** **boolean** compareAndSwapInt(Object var1, **long** var2, **int** var4, **int** var5);

### park

1. **public** **native** **void** park(**boolean** var1, **long** var2);

### unpark

1. **public** **native** **void** unpark(Object var1);

### getObjectVolatile

1、getObjectVolatile保证读到最新的对象，即volatile的含义(可视性、有序性)

1. **public** **native** Object getObjectVolatile(Object var1, **long** var2);

# Lock简介

## 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

### 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

### 对象头、monitor

#### 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



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



* 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

#### 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

## 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

### 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

#### 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

### 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

### 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

### 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

### 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)

1. JVM利用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针

* 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)

1. 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)

1. 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



#### 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。

### 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)

1. 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)

1. 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
2. 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



#### 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

### 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

### 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

# AQS框架

## 基本概念

### 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

### 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

### 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

### CLH

1、全称为：Craig，Landin，and Hagersten (CLH)locks

2、优势

1. FIFO
2. lock-free：不加锁(没有唤醒阻塞的系统开销)
3. 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

### LockSupport

#### 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

#### 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

## 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

### 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

1. 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

1. 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

### 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

### 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

}

## 源码详解

### 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

### 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;

### 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



#### 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

#### 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;

}

##### 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**

#### 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2.6*

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

##### 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)

##### 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()

##### 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

#### 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



### 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;

}

#### 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

#### 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！

#### 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

### 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

#### 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**

##### 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

#### 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)

### 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**

#### 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

1. 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

# ConcurrentHashMap(JDK 1.8)

## 常量

### MAXIMUM\_CAPACITY

1. /\*\*
2. \* The largest possible table capacity.  This value must be
3. \* exactly 1<<30 to stay within Java array allocation and indexing
4. \* bounds for power of two table sizes, and is further required
5. \* because the top two bits of 32bit hash fields are used for
6. \* control purposes.
7. \*/
8. **private** **static** **final** **int** MAXIMUM\_CAPACITY = 1 << 30;

1、容器的最大容量，确切等于230。hash字段的最高两位用于存储控制信息

### DEFAULT\_CAPACITY

1. /\*\*
2. \* The default initial table capacity.  Must be a power of 2
3. \* (i.e., at least 1) and at most MAXIMUM\_CAPACITY.
4. \*/
5. **private** **static** **final** **int** DEFAULT\_CAPACITY = 16;

1、默认的table大小，必须是2的幂次，最大为MAXIMUM\_CAPACITY

### MAX\_ARRAY\_SIZE

1. /\*\*
2. \* The largest possible (non-power of two) array size.
3. \* Needed by toArray and related methods.
4. \*/
5. **static** **final** **int** MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8;

1、数组最大容量

### DEFAULT\_CONCURRENCY\_LEVEL

1. /\*\*
2. \* The default concurrency level for this table. Unused but
3. \* defined for compatibility with previous versions of this class.
4. \*/
5. **private** **static** **final** **int** DEFAULT\_CONCURRENCY\_LEVEL = 16;

1、默认的并发级别，虽然定义了，但是没有使用，为了兼容以前的版本

### LOAD\_FACTOR

1. /\*\*
2. \* The load factor for this table. Overrides of this value in
3. \* constructors affect only the initial table capacity.  The
4. \* actual floating point value isn't normally used -- it is
5. \* simpler to use expressions such as {@code n - (n >>> 2)} for
6. \* the associated resizing threshold.
7. \*/
8. **private** **static** **final** **float** LOAD\_FACTOR = 0.75f;

1、负载因子，实际过程中用的是n - (n >>> 2)来表示n\* LOAD\_FACTOR

### TREEIFY\_THRESHOLD

1. /\*\*
2. \* The bin count threshold for using a tree rather than list for a
3. \* bin.  Bins are converted to trees when adding an element to a
4. \* bin with at least this many nodes. The value must be greater
5. \* than 2, and should be at least 8 to mesh with assumptions in
6. \* tree removal about conversion back to plain bins upon
7. \* shrinkage.
8. \*/
9. **static** **final** **int** TREEIFY\_THRESHOLD = 8;

1、当由于插入操作导致某个槽位中Node数量超过8时，将会用红黑树来组织这些Node

### UNTREEIFY\_THRESHOLD

1. /\*\*
2. \* The bin count threshold for untreeifying a (split) bin during a
3. \* resize operation. Should be less than TREEIFY\_THRESHOLD, and at
4. \* most 6 to mesh with shrinkage detection under removal.
5. \*/
6. **static** **final** **int** UNTREEIFY\_THRESHOLD = 6;

1、当由于删除操作导致某个槽位中Node数量低于6时，将会用链表来组织这些Node

2、为什么UNTREEIFY\_THRESHOLD和TREEIFY\_THRESHOLD设置为不同的值？我觉得这是为了避免插入删除交替执行时会造成Node组织结构在链表和红黑树之间不断地切换

### MIN\_TREEIFY\_CAPACITY

1. /\*\*
2. \* The smallest table capacity for which bins may be treeified.
3. \* (Otherwise the table is resized if too many nodes in a bin.)
4. \* The value should be at least 4 \* TREEIFY\_THRESHOLD to avoid
5. \* conflicts between resizing and treeification thresholds.
6. \*/
7. **static** **final** **int** MIN\_TREEIFY\_CAPACITY = 64;

1、bin中节点被树化所对应的容器的最小容量，否则容器应该扩容，因为一个槽位容纳了太多的节点

### MIN\_TRANSFER\_STRIDE

1. /\*\*
2. \* Minimum number of rebinnings per transfer step. Ranges are
3. \* subdivided to allow multiple resizer threads.  This value
4. \* serves as a lower bound to avoid resizers encountering
5. \* excessive memory contention.  The value should be at least
6. \* DEFAULT\_CAPACITY.
7. \*/
8. **private** **static** **final** **int** MIN\_TRANSFER\_STRIDE = 16;

1、<未完成>：不懂啥意思

### RESIZE\_STAMP\_BITS

1. /\*\*
2. \* The number of bits used for generation stamp in sizeCtl.
3. \* Must be at least 6 for 32bit arrays.
4. \*/
5. **private** **static** **int** RESIZE\_STAMP\_BITS = 16;

1、<未完成>：不懂啥意思

### MAX\_RESIZERS

1. /\*\*
2. \* The maximum number of threads that can help resize.
3. \* Must fit in 32 - RESIZE\_STAMP\_BITS bits.
4. \*/
5. **private** **static** **final** **int** MAX\_RESIZERS = (1 << (32 - RESIZE\_STAMP\_BITS)) - 1;

1、表扩张或收缩时，最多可以用多少个线程来并发执行

### RESIZE\_STAMP\_SHIFT

1. /\*\*
2. \* The bit shift for recording size stamp in sizeCtl.
3. \*/
4. **private** **static** **final** **int** RESIZE\_STAMP\_SHIFT = 32 - RESIZE\_STAMP\_BITS;

1、<未完成>：不懂啥意思

### Node#控制位状态解释

1. /\*
2. \* Encodings for Node hash fields. See above for explanation.
3. \*/
4. **static** **final** **int** MOVED     = -1; // hash for forwarding nodes
5. **static** **final** **int** TREEBIN   = -2; // hash for roots of trees
6. **static** **final** **int** RESERVED  = -3; // hash for transient reservations
7. **static** **final** **int** HASH\_BITS = 0x7fffffff; // usable bits of normal node hash

1、<未完成>：不懂啥意思

### NCPU

1. /\*\* Number of CPUS, to place bounds on some sizings \*/
2. **static** **final** **int** NCPU = Runtime.getRuntime().availableProcessors();

1、获取CPU数量

## 字段

### table

1. /\*\*
2. \* The array of bins. Lazily initialized upon first insertion.
3. \* Size is always a power of two. Accessed directly by iterators.
4. \*/
5. **transient** **volatile** Node<K,V>[] table;

1、bins的数组，延迟加载(直到第一次插入操作)，大小一定是2的幂次，可被迭代器直接访问

### nextTable

1. /\*\*
2. \* The next table to use; non-null only while resizing.
3. \*/
4. **private** **transient** **volatile** Node<K,V>[] nextTable;

1、扩容时会用到，正常情况下为null

### baseCount

1. /\*\*
2. \* Base counter value, used mainly when there is no contention,
3. \* but also as a fallback during table initialization
4. \* races. Updated via CAS.
5. \*/
6. **private** **transient** **volatile** **long** baseCount;

1、计数值，通常在无竞争的情况下被使用，通过CAS操作来更新

### sizeCtl

1. /\*\*
2. \* Table initialization and resizing control.  When negative, the
3. \* table is being initialized or resized: -1 for initialization,
4. \* else -(1 + the number of active resizing threads).  Otherwise,
5. \* when table is null, holds the initial table size to use upon
6. \* creation, or 0 for default. After initialization, holds the
7. \* next element count value upon which to resize the table.
8. \*/
9. **private** **transient** **volatile** **int** sizeCtl;

1、table初始化或者调整大小的控制信息

1. -1：table正在进行初始化
2. -N(N>1)：正在resizing，N-1就是参与resizing的线程数量
3. 其他情况：

* 当table未初始化，表示table需要初始化的大小
* 当table初始化完成，表示table的容量，默认是table大小的0.75倍

### transferIndex

1. /\*\*
2. \* The next table index (plus one) to split while resizing.
3. \*/
4. **private** **transient** **volatile** **int** transferIndex;

1、<未完成>：不懂啥意思

### cellsBusy

1. /\*\*
2. \* Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
3. \*/
4. **private** **transient** **volatile** **int** cellsBusy;

1、<未完成>：不懂啥意思

### counterCells

1. /\*\*
2. \* Table of counter cells. When non-null, size is a power of 2.
3. \*/
4. **private** **transient** **volatile** CounterCell[] counterCells;

1、<未完成>：不懂啥意思

### views

1. // views
2. **private** **transient** KeySetView<K,V> keySet;
3. **private** **transient** ValuesView<K,V> values;
4. **private** **transient** EntrySetView<K,V> entrySet;

1、<未完成>：不懂啥意思

### unsafe字段

1. **private** **static** **final** sun.misc.Unsafe U;
2. //以下常量用于计算偏移量，因为Unsafe的CAS操作是基于字段偏移量的
3. **private** **static** **final** **long** SIZECTL;  //对应于sizeCtl
4. **private** **static** **final** **long** TRANSFERINDEX;  //对应于transferIndex
5. **private** **static** **final** **long** BASECOUNT;  //对应于baseCount
6. **private** **static** **final** **long** CELLSBUSY;  //对应于cellsBusy
7. **private** **static** **final** **long** CELLVALUE;  //对应于CounterCell#value
8. **private** **static** **final** **long** ABASE;
9. **private** **static** **final** **int** ASHIFT;
11. **static** {
12. **try** {
13. U = sun.misc.Unsafe.getUnsafe();
14. Class<?> k = ConcurrentHashMap.**class**;
15. SIZECTL = U.objectFieldOffset
16. (k.getDeclaredField("sizeCtl"));
17. TRANSFERINDEX = U.objectFieldOffset
18. (k.getDeclaredField("transferIndex"));
19. BASECOUNT = U.objectFieldOffset
20. (k.getDeclaredField("baseCount"));
21. CELLSBUSY = U.objectFieldOffset
22. (k.getDeclaredField("cellsBusy"));
23. Class<?> ck = CounterCell.**class**;
24. CELLVALUE = U.objectFieldOffset
25. (ck.getDeclaredField("value"));
26. Class<?> ak = Node[].**class**;
27. ABASE = U.arrayBaseOffset(ak);
28. **int** scale = U.arrayIndexScale(ak);
29. **if** ((scale & (scale - 1)) != 0)
30. **throw** **new** Error("data type scale not a power of two");
31. ASHIFT = 31 - Integer.numberOfLeadingZeros(scale);
32. } **catch** (Exception e) {
33. **throw** **new** Error(e);
34. }
35. }

## 内部类

### Node

1. /\*\*
2. \* Key-value entry.  This class is never exported out as a
3. \* user-mutable Map.Entry (i.e., one supporting setValue; see
4. \* MapEntry below), but can be used for read-only traversals used
5. \* in bulk tasks.  Subclasses of Node with a negative hash field
6. \* are special, and contain null keys and values (but are never
7. \* exported).  Otherwise, keys and vals are never null.
8. \*/
9. **static** **class** Node<K,V> **implements** Map.Entry<K,V> {
10. **final** **int** hash;
11. **final** K key;
12. **volatile** V val;
13. **volatile** Node<K,V> next;
15. Node(**int** hash, K key, V val, Node<K,V> next) {
16. **this**.hash = hash;
17. **this**.key = key;
18. **this**.val = val;
19. **this**.next = next;
20. }
22. **public** **final** K getKey()       { **return** key; }
23. **public** **final** V getValue()     { **return** val; }
24. **public** **final** **int** hashCode()   { **return** key.hashCode() ^ val.hashCode(); }
25. **public** **final** String toString(){ **return** key + "=" + val; }
26. **public** **final** V setValue(V value) {
27. **throw** **new** UnsupportedOperationException();
28. }
30. **public** **final** **boolean** equals(Object o) {
31. Object k, v, u; Map.Entry<?,?> e;
32. **return** ((o **instanceof** Map.Entry) &&
33. (k = (e = (Map.Entry<?,?>)o).getKey()) != **null** &&
34. (v = e.getValue()) != **null** &&
35. (k == key || k.equals(key)) &&
36. (v == (u = val) || v.equals(u)));
37. }
39. /\*\*
40. \* Virtualized support for map.get(); overridden in subclasses.
41. \*/
42. Node<K,V> find(**int** h, Object k) {
43. Node<K,V> e = **this**;
44. **if** (k != **null**) {
45. **do** {
46. K ek;
47. **if** (e.hash == h &&
48. ((ek = e.key) == k || (ek != **null** && k.equals(ek))))
49. **return** e;
50. } **while** ((e = e.next) != **null**);
51. }
52. **return** **null**;
53. }
54. }

### TreeNode

1. **static** **final** **class** TreeNode<K,V> **extends** Node<K,V> {
2. TreeNode<K,V> parent;  // red-black tree links
3. TreeNode<K,V> left;
4. TreeNode<K,V> right;
5. TreeNode<K,V> prev;    // needed to unlink next upon deletion
6. **boolean** red;
8. TreeNode(**int** hash, K key, V val, Node<K,V> next,
9. TreeNode<K,V> parent) {
10. **super**(hash, key, val, next);
11. **this**.parent = parent;
12. }
14. Node<K,V> find(**int** h, Object k) {
15. **return** findTreeNode(h, k, **null**);
16. }
18. /\*\*
19. \* Returns the TreeNode (or null if not found) for the given key
20. \* starting at given root.
21. \*/
22. **final** TreeNode<K,V> findTreeNode(**int** h, Object k, Class<?> kc) {
23. **if** (k != **null**) {
24. TreeNode<K,V> p = **this**;
25. **do**  {
26. **int** ph, dir; K pk; TreeNode<K,V> q;
27. TreeNode<K,V> pl = p.left, pr = p.right;
28. **if** ((ph = p.hash) > h)
29. p = pl;
30. **else** **if** (ph < h)
31. p = pr;
32. **else** **if** ((pk = p.key) == k || (pk != **null** && k.equals(pk)))
33. **return** p;
34. **else** **if** (pl == **null**)
35. p = pr;
36. **else** **if** (pr == **null**)
37. p = pl;
38. **else** **if** ((kc != **null** ||
39. (kc = comparableClassFor(k)) != **null**) &&
40. (dir = compareComparables(kc, k, pk)) != 0)
41. p = (dir < 0) ? pl : pr;
42. **else** **if** ((q = pr.findTreeNode(h, k, kc)) != **null**)
43. **return** q;
44. **else**
45. p = pl;
46. } **while** (p != **null**);
47. }
48. **return** **null**;
49. }
50. }

### TreeBin

### ForwardingNode

1. **static** **final** **class** ForwardingNode<K,V> **extends** Node<K,V> {
2. **final** Node<K,V>[] nextTable;
3. ForwardingNode(Node<K,V>[] tab) {
4. **super**(MOVED, **null**, **null**, **null**);
5. **this**.nextTable = tab;
6. }
8. Node<K,V> find(**int** h, Object k) {
9. // loop to avoid arbitrarily deep recursion on forwarding nodes
10. outer: **for** (Node<K,V>[] tab = nextTable;;) {
11. Node<K,V> e; **int** n;
12. **if** (k == **null** || tab == **null** || (n = tab.length) == 0 ||
13. (e = tabAt(tab, (n - 1) & h)) == **null**)
14. **return** **null**;
15. **for** (;;) {
16. **int** eh; K ek;
17. **if** ((eh = e.hash) == h &&
18. ((ek = e.key) == k || (ek != **null** && k.equals(ek))))
19. **return** e;
20. **if** (eh < 0) {
21. **if** (e **instanceof** ForwardingNode) {
22. tab = ((ForwardingNode<K,V>)e).nextTable;
23. **continue** outer;
24. }
25. **else**
26. **return** e.find(h, k);
27. }
28. **if** ((e = e.next) == **null**)
29. **return** **null**;
30. }
31. }
32. }
33. }

1、ForwardingNode：一个特殊的Node节点，hash值为-1，其中存储nextTable的引用

2、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

## Utils方法

### spread

1. /\*\*
2. \* Spreads (XORs) higher bits of hash to lower and also forces top
3. \* bit to 0. Because the table uses power-of-two masking, sets of
4. \* hashes that vary only in bits above the current mask will
5. \* always collide. (Among known examples are sets of Float keys
6. \* holding consecutive whole numbers in small tables.)  So we
7. \* apply a transform that spreads the impact of higher bits
8. \* downward. There is a tradeoff between speed, utility, and
9. \* quality of bit-spreading. Because many common sets of hashes
10. \* are already reasonably distributed (so don't benefit from
11. \* spreading), and because we use trees to handle large sets of
12. \* collisions in bins, we just XOR some shifted bits in the
13. \* cheapest possible way to reduce systematic lossage, as well as
14. \* to incorporate impact of the highest bits that would otherwise
15. \* never be used in index calculations because of table bounds.
16. \*/
17. **static** **final** **int** spread(**int** h) {
18. **return** (h ^ (h >>> 16)) & HASH\_BITS;
19. }

1、该方法用于对hashCode()做一些调整，这样调整的原因有

* 使最高位(符号位为0)
* 在一个小表中，一系列浮动的键将会保持连续性
* 由于表的大小一般不会太大，因此高位对于槽位计算的影响将会微乎其微，因此将高位的影响向低位扩展，这样h ^ (h >>> 16)得到的值将由高位和低位共同决定，但是只用了16位的位

### tableSizeFor

1. /\*\*
2. \* Returns a power of two table size for the given desired capacity.
3. \* See Hackers Delight, sec 3.2
4. \*/
5. **private** **static** **final** **int** tableSizeFor(**int** c) {
6. **int** n = c - 1;
7. n |= n >>> 1;
8. n |= n >>> 2;
9. n |= n >>> 4;
10. n |= n >>> 8;
11. n |= n >>> 16;
12. **return** (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;
13. }

1、将c调整为大于c的2的幂次方，真尼玛神奇???

### comparableClassFor

1. /\*\*
2. \* Returns x's Class if it is of the form "class C implements
3. \* Comparable<C>", else null.
4. \*/
5. **static** Class<?> comparableClassFor(Object x) {
6. **if** (x **instanceof** Comparable) {
7. Class<?> c; Type[] ts, as; Type t; ParameterizedType p;
8. **if** ((c = x.getClass()) == String.**class**) // bypass checks
9. **return** c;
10. **if** ((ts = c.getGenericInterfaces()) != **null**) {
11. **for** (**int** i = 0; i < ts.length; ++i) {
12. **if** (((t = ts[i]) **instanceof** ParameterizedType) &&
13. ((p = (ParameterizedType)t).getRawType() ==
14. Comparable.**class**) &&
15. (as = p.getActualTypeArguments()) != **null** &&
16. as.length == 1 && as[0] == c) // type arg is c
17. **return** c;
18. }
19. }
20. }
21. **return** **null**;
22. }

### compareComparables

1. /\*\*
2. \* Returns k.compareTo(x) if x matches kc (k's screened comparable
3. \* class), else 0.
4. \*/
5. @SuppressWarnings({"rawtypes","unchecked"}) // for cast to Comparable
6. **static** **int** compareComparables(Class<?> kc, Object k, Object x) {
7. **return** (x == **null** || x.getClass() != kc ? 0 :
8. ((Comparable)k).compareTo(x));
9. }

### table access methods

1. /\*
2. \* Volatile access methods are used for table elements as well as
3. \* elements of in-progress next table while resizing.  All uses of
4. \* the tab arguments must be null checked by callers.  All callers
5. \* also paranoically precheck that tab's length is not zero (or an
6. \* equivalent check), thus ensuring that any index argument taking
7. \* the form of a hash value anded with (length - 1) is a valid
8. \* index.  Note that, to be correct wrt arbitrary concurrency
9. \* errors by users, these checks must operate on local variables,
10. \* which accounts for some odd-looking inline assignments below.
11. \* Note that calls to setTabAt always occur within locked regions,
12. \* and so in principle require only release ordering, not
13. \* full volatile semantics, but are currently coded as volatile
14. \* writes to be conservative.
15. \*/
17. @SuppressWarnings("unchecked")
18. **static** **final** <K,V> Node<K,V> tabAt(Node<K,V>[] tab, **int** i) {
19. **return** (Node<K,V>)U.getObjectVolatile(tab, ((**long**)i << ASHIFT) + ABASE);
20. }
22. **static** **final** <K,V> **boolean** casTabAt(Node<K,V>[] tab, **int** i,
23. Node<K,V> c, Node<K,V> v) {
24. **return** U.compareAndSwapObject(tab, ((**long**)i << ASHIFT) + ABASE, c, v);
25. }
27. **static** **final** <K,V> **void** setTabAt(Node<K,V>[] tab, **int** i, Node<K,V> v) {
28. U.putObjectVolatile(tab, ((**long**)i << ASHIFT) + ABASE, v);
29. }

1、((long)i << ASHIFT) + ABASE：不懂是啥意思

## 方法

### put

1. **public** V put(K key, V value) {
2. **return** putVal(key, value, **false**);
3. }

#### putVal

1. /\*\* Implementation for put and putIfAbsent \*/
2. **final** V putVal(K key, V value, **boolean** onlyIfAbsent) {
3. **if** (key == **null** || value == **null**) **throw** **new** NullPointerException();
4. //对hashCode()的返回值进行改造，作为Node的hash值
5. **int** hash = spread(key.hashCode());
6. **int** binCount = 0;
7. **for** (Node<K,V>[] tab = table;;) {
8. Node<K,V> f; **int** n, i, fh;
9. //当表未初始化时，初始化表
10. **if** (tab == **null** || (n = tab.length) == 0)
11. tab = initTable();
12. //i是table中定位索引位置，n是table的大小
13. //有个小问题：如果添加元素时，表进行了一次扩容，那么下面的语句不会将元素添加到原来的table中去吗???
14. **else** **if** ((f = tabAt(tab, i = (n - 1) & hash)) == **null**) {
15. **if** (casTabAt(tab, i, **null**,
16. **new** Node<K,V>(hash, key, value, **null**)))
17. **break**;                   // no lock when adding to empty bin
18. }
19. //如果f的hash值为-1，说明当前f是ForwardingNode节点，意味着有其他线程正在扩容，则一起进行扩容操作
20. **else** **if** ((fh = f.hash) == MOVED)
21. tab = helpTransfer(tab, f);
22. **else** {
23. //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实现并发
24. V oldVal = **null**;
25. **synchronized** (f) {
26. //再次进行判断，防止被其他线程修改
27. **if** (tabAt(tab, i) == f) {
28. //f节点是一个链表节点
29. **if** (fh >= 0) {
30. binCount = 1;
31. //将新值更新到链表中(若存在键相同的节点，则更新值，否则插入到链表尾部)
32. **for** (Node<K,V> e = f;; ++binCount) {
33. K ek;
34. **if** (e.hash == hash &&
35. ((ek = e.key) == key ||
36. (ek != **null** && key.equals(ek)))) {
37. oldVal = e.val;
38. //onlyIfAbsent为true表示不允许键已存在
39. **if** (!onlyIfAbsent)
40. e.val = value;
41. **break**;
42. }
43. Node<K,V> pred = e;
44. **if** ((e = e.next) == **null**) {
45. pred.next = **new** Node<K,V>(hash, key,
46. value, **null**);
47. **break**;
48. }
49. //可能会有else吗
50. }
51. }
52. //f节点是TreeBin节点(一颗红黑树)，则通过红黑树插入
53. **else** **if** (f **instanceof** TreeBin) {
54. Node<K,V> p;
55. //手动置为安全数值，避免下面进行的将链表转换成红黑树的操作
56. binCount = 2;
57. //putTreeVal方法查找或添加一个节点，当添加节点时返回null，若节点已存在，则返回该节点
58. **if** ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
59. value)) != **null**) {
60. oldVal = p.val;
61. **if** (!onlyIfAbsent)
62. p.val = value;
63. }
64. }
65. }
66. }
67. **if** (binCount != 0) {
68. //当节点数量大于阈值，需要将链表转为红黑树
69. **if** (binCount >= TREEIFY\_THRESHOLD)
70. treeifyBin(tab, i);
71. **if** (oldVal != **null**)
72. **return** oldVal;
73. **break**;
74. }
75. }
76. }
77. addCount(1L, binCount);
78. **return** **null**;
79. }

#### initTable

1. **private** **final** Node<K,V>[] initTable() {
2. Node<K,V>[] tab; **int** sc;
3. //在调用本方法前，都会进行类似"(tab = table) == null || tab.length == 0"这样的判断，为什么这里还需要再次判断：因为可能有多个线程同时触发了该方法，而只有一个线程能够执行初始化操作
4. **while** ((tab = table) == **null** || tab.length == 0) {
5. //已经有其他线程正在执行初始化操作，因此当前线程让出CPU时间，进行自旋，直至初始化结束(退出while循环)
6. **if** ((sc = sizeCtl) < 0)
7. Thread.yield(); // lost initialization race; just spin
8. //尝试用CAS操作更改sizeCtl，将其置为-1，初始化只能有一个线程。若成功，那么执行初始化操作的就是当前线程，否则会自旋直至其他线程初始化table完毕
9. **else** **if** (U.compareAndSwapInt(**this**, SIZECTL, sc, -1)) {
10. **try** {
11. //这里为什么需要再次进行判断???
12. **if** ((tab = table) == **null** || tab.length == 0) {
13. **int** n = (sc > 0) ? sc : DEFAULT\_CAPACITY;
14. @SuppressWarnings("unchecked")
15. Node<K,V>[] nt = (Node<K,V>[])**new** Node<?,?>[n];
16. table = tab = nt;
17. //等价于0.75\*n，用位运算效率较高。但是为什么变小了???
18. sc = n - (n >>> 2);
19. }
20. } **finally** {
21. sizeCtl = sc;
22. }
23. **break**;
24. }
25. }
26. **return** tab;
27. }

1、sizeCtl默认为0，当调用带有初始化大小的构造方法时，会将该指定大小通过方法tableSizeFor调整为2的幂次，为什么需要2的幂次???

# ThreadLocal

# BlockingQueue

# ReentrantLock

# ThreadPool

## 重要数据结构

### ctl

1、这个变量是整个类的核心，AtomicInteger保证了对这个变量的操作是原子的，通过巧妙的操作，ThreadPoolExecutor用这一个变量保存了两个内容：

1. 所有有效线程的数量
2. 各个线程的状态(runState)

2、低29位存线程数，高3位存runState,这样runState有5个值

1. RUNNING:-536870912
2. SHUTDOWN:0
3. STOP:536870912
4. TIDYING:1073741824
5. TERMINATED:1610612736

3、状态含义解释

1. RUNNING状态：线程池正常运行，可以接受新的任务并处理队列中的任务；
2. SHUTDOWN状态：不再接受新的任务，但是会执行队列中的任务；
3. STOP状态：不再接受新任务，不处理队列中的任务

4、围绕ctl变量有一些操作，了解这些方法是看懂后面一些晦涩代码的基础

private static int runStateOf(int c) {

return c & ~CAPACITY;

}

//这个方法用于取出runState的值 因为CAPACITY值为：00011111111111111111111111111111，~为按位取反操作，则~CAPACITY值为：11100000000000000000000000000000，再同参数做&操作，就将低29位置0了，而高3位还是保持原先的值，也就是runState的值

private static int workerCountOf(int c) {

return c & CAPACITY;

}

//这个方法用于取出workerCount的值，因为CAPACITY值为：00011111111111111111111111111111，所以&操作将参数的高3位置0了，保留参数的低29位，也就是workerCount的值

private static int ctlOf(int rs, int wc) {

return rs | wc;

}

//将runState和workerCount存到同一个int中，就是"|"运算的意思是，假设rs的值是101000，wc的值是000111，则他们位或运算的值为101111

// 只有RUNNING状态会小于0

private static boolean isRunning(int c) {

return c < SHUTDOWN;

}

### corePoolSize

1、核心线程池大小，活动线程小于corePoolSize则直接创建，大于等于则先加到workQueue中，队列满了才创建新的线程

private volatile int corePoolSize;

### keepAliveTime

1、线程从队列中获取任务的超时时间，也就是说如果线程空闲超过这个时间就会终止

private volatile long keepAliveTime;

### workQueue

1、阻塞队列

private final BlockingQueue<Runnable> workQueue;

### mainLock

1、重入锁

private final ReentrantLock mainLock = new ReentrantLock();

### workers

1、任务集合???

private final HashSet<Worker> workers = new HashSet<Worker>();

### termination

private final Condition termination = mainLock.newCondition();

### Worker

1、继承自AQS框架，这个Worker是一个封装了Thread的对象，维护了了一个同步阻塞队列，感觉有点乱???为什么要继承AQS???

2、源码如下

private final class Worker

extends AbstractQueuedSynchronizer

implements Runnable {

/\*\*

\* This class will never be serialized, but we provide a

\* serialVersionUID to suppress a javac warning.

\*/

private static final long serialVersionUID = 6138294804551838833L;

/\*\* Thread this worker is running in. Null if factory fails. \*/

final Thread thread;

/\*\* Initial task to run. Possibly null. \*/

Runnable firstTask;

/\*\* Per-thread task counter \*/

volatile long completedTasks;

/\*\*

\* Creates with given first task and thread from ThreadFactory.

\* @param firstTask the first task (null if none)

\*/

Worker(Runnable firstTask) {

setState(-1); // inhibit interrupts until runWorker

this.firstTask = firstTask;

this.thread = getThreadFactory().newThread(this);//???

}

/\*\* Delegates main run loop to outer runWorker \*/

public void run() {

runWorker(this);

}

// Lock methods

//

// The value 0 represents the unlocked state.

// The value 1 represents the locked state.

protected boolean isHeldExclusively() {

return getState() != 0;

}

protected boolean tryAcquire(int unused) {

if (compareAndSetState(0, 1)) {

//独占锁，且不可重入

setExclusiveOwnerThread(Thread.currentThread());

return true;

}

return false;

}

protected boolean tryRelease(int unused) {

setExclusiveOwnerThread(null);

setState(0);

return true;

}

public void lock() { acquire(1); }

public boolean tryLock() { return tryAcquire(1); }

public void unlock() { release(1); }

public boolean isLocked() { return isHeldExclusively(); }

void interruptIfStarted() {

Thread t;

if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {

try {

t.interrupt();

} catch (SecurityException ignore) {

}

}

}

}

3、设计成不可重入的原因是为了避免任务执行的代码中修改线程池的变量

## 重要方法

### execute

1、execute方法主要三个步骤：

1. 活动线程小于corePoolSize的时候创建新的线程
2. 活动线程大于corePoolSize时都是先加入到任务队列当中
3. 任务队列满了再去启动新的线程，如果线程数达到最大值就拒绝任务

2、源码

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

/\*

\* Proceed in 3 steps:

\*

\* 1. If fewer than corePoolSize threads are running, try to

\* start a new thread with the given command as its first

\* task. The call to addWorker atomically checks runState and

\* workerCount, and so prevents false alarms that would add

\* threads when it shouldn't, by returning false.

\*

\* 2. If a task can be successfully queued, then we still need

\* to double-check whether we should have added a thread

\* (because existing ones died since last checking) or that

\* the pool shut down since entry into this method. So we

\* recheck state and if necessary roll back the enqueuing if

\* stopped, or start a new thread if there are none.

\*

\* 3. If we cannot queue task, then we try to add a new

\* thread. If it fails, we know we are shut down or saturated

\* and so reject the task.

\*/

int c = ctl.get();

//当活动线程数量小于核心线程数量

if (workerCountOf(c) < corePoolSize) {

//直接启动新的线程。第二个参数true:addWorker中会重新检查workerCount是否小于corePoolSize

if (addWorker(command, true))

return;

c = ctl.get();

}

//当活动线程数量大于核心线程数量，添加到队列中

if (isRunning(c) && workQueue.offer(command)) {

int recheck = ctl.get();

//处于非RUNNING状态

if (! isRunning(recheck) && remove(command))

reject(command); //采用线程池指定的策略拒绝任务

//线程池处于RUNNING状态 || 线程池处于非RUNNING状态但是任务移除失败

else if (workerCountOf(recheck) == 0)

//这行代码是为了SHUTDOWN状态下没有活动线程了，但是队列里还有任务没执行这种特殊情况

//添加一个null任务是因为SHUTDOWN状态下，线程池不再接受新任务

addWorker(null, false);

}

//两种情况：

1. 非RUNNING状态拒绝新的任务
2. 队列满了启动新的线程失败(workCount > maximumPoolSize)

else if (!addWorker(command, false))

reject(command);

}

### addWork

1、源码如下

private boolean addWorker(Runnable firstTask, boolean core) {

retry:

for (;;) {

int c = ctl.get();

int rs = runStateOf(c);

//条件成立的情况

1. rs>0
2. rs=SHUTDOWN且firstTask!=null
3. rs=SHUTDOWN且firstTask == null且workQueue为空

//条件不成立的情况

1. rs=RUNNING
2. rs=SHUTDOWN且firstTask == null且workQueue不为空

if (rs >= SHUTDOWN &&

! (rs == SHUTDOWN &&

firstTask == null &&

! workQueue.isEmpty()))

return false;

for (;;) {

int wc = workerCountOf(c);

//当wc比容量大或者wc比核心或者最大线程数量多时，返回false，core为true时，与核心线程数量比，反之与最大线程数量比

if (wc >= CAPACITY ||

wc >= (core ? corePoolSize : maximumPoolSize))

return false;

//通过CAS操作添加Work，保证同一时刻只有一个线程往ThreadPool内加入Work

if (compareAndIncrementWorkerCount(c))

break retry;

c = ctl.get(); // Re-read ctl

//当状态没有变化时，继续尝试添加Work，否则回到大循环

if (runStateOf(c) != rs)

continue retry;

}

}

boolean workerStarted = false;

boolean workerAdded = false;

Worker w = null;

try {

final ReentrantLock mainLock = this.mainLock;

w = new Worker(firstTask);

final Thread t = w.thread;

if (t != null) {

mainLock.lock();

try {

// Recheck while holding lock.

// Back out on ThreadFactory failure or if

// shut down before lock acquired.

int c = ctl.get();

int rs = runStateOf(c);

//条件成立的情况

1. RUNNING状态
2. SHUTDOWN且firstTask == null

if (rs < SHUTDOWN ||

(rs == SHUTDOWN && firstTask == null)) {

if (t.isAlive()) // precheck that t is startable

throw new IllegalThreadStateException();

workers.add(w);

int s = workers.size();

if (s > largestPoolSize)

largestPoolSize = s;

workerAdded = true;

}

} finally {

mainLock.unlock();

}

if (workerAdded) {

//???此方法到底什么意思

t.start();

workerStarted = true;

}

}

} finally {

if (! workerStarted)

addWorkerFailed(w);

}

return workerStarted;

}

concurrent结构

