# 线程基础

## 基本概念

什么是进程：是程序在数据集上运行的一个过程，是系统进行资源分配和调度的基本单位；

什么是线程：是进程中独立运行的子任务，是操作系统进行运算调度的最小单位，它包含在进程中，是进程中实际运作单位

## 线程好处

恰当的使用线程可以降低开发和维护的开销，提高复杂应用的性能（加快响应）；把复杂难以理解的代码转换为直接，简洁的代码这样更容易读写和维护

处理器角度

提示多核服务器的处理器资源的利用率，提高吞吐量；一个线程只能在一个cup资源上运行，在多核处理器的服务器中只有一个cpu资源在运行，其他的就闲置，资源利用率不高

提示响应角度

使用线程将占据时间长的程序中的任务放到后台中去执行，当处理某些复杂的业务逻辑时，可以将数据一致性要求不高的操作交给一个线程来处理，这样响应用户请求的线程就能尽快的处理完成，缩短了响应时间，提升了用户体验

编程模型角度（简化编程）

从程序设计角度

比如：一个应用程序需要从本地文件系统中读取和处理文件的情景，在单线程应用中编写手动处理读取和处理的顺序，那么就必须记录每个文件读取和处理的状态，相反，你可以启动两个线程，每个线程处理一个文件的读取和操作。线程会在等待磁盘读取文件的过程中被阻塞。在等待的时候，其他的线程能够使用CPU去处理已经读取完的文件。其结果就是，磁盘总是在繁忙地读取不同的文件到内存中。这会带来磁盘和CPU利用率的提升。而且每个线程只需要记录一个文件，因此这种方式也很容易编程实现。

## 线程存在的问题

**线程安全性**

在多线程环境下，线程没有经过充分同步，各个操作的顺序和结果是不可预测的，通常问题有竞态条件和数据竞争

竞态条件：当程序运行的正确性取决于相对时间或者调度器所控制的多线程交叉时（多线程环境下程序后续运行行为取决于某个共享变量的值，获取的共享变量可能是过期值，导致程序的运行出现不可预期的结果），就会发生竞态条件

2种典型的竞态条件check-then-act和read-modify-write

Check-then-act：这种情况很有可能取到过时的值来决定下一步的动作（比如判断共享变量是否等于某个值时然后执行对应的代码块）

Read-modify-write：典型的例子就是一个变量的递增操作，变量旧的值被读取，然后修改，然后将修改后的值写入变量

数据竞争：是指2条或者2条以上的线程并发地访问同一个变量，同时至少有一个线程为写线程，这些线程没有进行同步控制访问这个变量。

**线程活跃度问题**

死锁：在多线程环境下2个线程分别持有一个锁，在访问各自临界区时，需要获取彼此当前已经持有的锁，这样2个线程就会一直处于等待获取对方锁的阻塞状态中（死锁是大家都拿不到资源都占用着对方的资源）

解决死锁：

1.尝试定时获取锁,使用lock.tryLock(timeout)

2.避免一个线程同时获取多个锁

3.避免一个线程在锁内同时占有多个资源，尽量保证每个锁只占用一个资源

活锁：任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。另一种情况发生在多线程间的协作，它们为了彼此之间的响应而修改了状态，使得没有一个线程能够继续执行（好比两个人相遇后，为了避开对方都选择其他的路，但是又在另一条路上相遇了）

活锁是拿到资源却又相互释放不执行。当多线程中出现了相互谦让，都主动将资源释放给别的线程使用，这样这个资源在多个线程之间跳动而又得不到执行

解决方式：对于这样重复尝试的操作引入一些随机性

饥饿：线程因无法访问到自己需要的资源而导致无法执行时，这种情况就叫做饥饿；（主要是由于线程的优先级导致线程始终无法获取cpu时间片导致的，尝试使用公平锁来解决饥饿问题，先访问的先执行）

**性能问题**

服务响应时间（可能由于同步造成），

上下文切换：上下文的切换回带来cpu性能的开销

（在单核cpu上执行多个线程，cpu通过给每个线程分配时间片这个机制来实现的；时间片就是cpu分配给线程的执行时间，非常短，cpu通过不停的切换时间片来切换线程之间的执行，让我们感觉就是多线程同时在执行；当前一个任务的时间片执行完成后，或切换到下一个时间片，那么就需要保存切换之前的状态，以便下次执行时恢复上传运行的状态；从任务的保存到再次加载这个过程就叫上下文切换 ）

## 创建线程和运行

### 线程的创建

（线程的创建也会带来性能的消耗的，所有不是线程越多就越好）

1. 继承Thread类
2. 实现Runnable接口
3. 创建里面内部类来实现

这3中方式都要重写run方法来实现线程的任务，线程不能返回结果

1. Callable接口： 这种方式能够返回结果
2. 使用线程池来创建

### 线程的启动

调用Thread类的start方法来启动（start方法内部是调用的一个本地方法start0()，通过操作系统来实现的（通过c语言））

Callable<V>

Callable位于java.util.concurrent包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做call()

这是一个泛型接口，call()函数返回的类型就是传递进来的V类型。

Future

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。Future提供的功能：

判断任务是否完成；

能够中断任务；

能够获取任务执行结果。

FutureTask<V>

FutureTask是Future接口的一个唯一实现类（也就是将Callable的实现交给future的实现类FutureTask，让FutureTask来判断是否执行完成，获取结果）

## 线程的中断

1. 在run方法中添加一个中断标记，当执行到这个标记时结束运行
2. 使用interrupt方法来中断

interrupt();中断线程（中断线程，其实是设置一个线程的中断标识位）

interrupted();（static修饰的静态方法）判断当前线程是否可以中断，返回true表示线程已经调用了interrupt方法，那么可以在后续代码中执行结束线程的代码，这个方法执行后将会将暂停状态标记清除，也即是说再次调用这个方法就会返回false；

isInterrupted();判断线程是否处于可中断，true表示线程已经调用了interrupt方法，那么可以在后续代码中执行结束线程的代码；false表示不能中断

public static boolean interrupted() {

return currentThread().isInterrupted(true);

}

当启动的线程调用interrupt()方法时，线程并不是真的停止线程，而是给当前线程打了一个可以暂停的标记，然后通过调用interrupted方法或者isInterrupted方法来获取暂停标记（true表示可以暂停），那么在线程中执行结束线程代码

如果线程调用了interrupt()方法，那么线程在调用interrupted方法就会返回true，表示线程可以中断；但是当再次调用interrupted方法时，结果就返回false；因为当线程首先调用interrupted方法时已经将线程的中断状态清除了

## 线程优先级

操作系统采用分时的形式调度运行的线程，线程会分配到若干的时间片，当时间片分配完时就会发生调度（上下文切换），并等待下次分配时间片，线程分配时间片的多少决定了线程使用cpu资源的多少，线程优先级就是决定了线程需要分配多或少的处理器资源的属性，也就是决定了线程分配到处理器资源多少的

线程可以划分优先级，优先级越高的线程获得的cpu资源也就越多

在java中优先级分为1-10个等级，如果设置的优先级大于10或者小于1，那么就会抛出异常

线程的优先级具有继承性，比如A线程启动了B线程，那么B线程的优先级和A线程一样

线程通过setProperty()方法可以设置线程的优先级

高优先级的线程总是大部分先执行完，但不代表高优先级的线程全部先执行完；另外，当线程优先级差距很大时，谁先执行完和代码调用的先后顺序无关，线程的优先级具有一定的规则性，也就是cpu尽量将执行资源让给优先级较高的线程

虽然线程优先级较高则优先执行完run方法的任务，但是这个结果不能太肯定，因为线程的优先级还具有随机性，有就是说优先级高的线程不一定每次都先执行完；不要把线程优先级与运行结果的顺序作为衡量的标准，优先级高的并不一定每次都先执行完run()方法中的任务，它们具有不确定性和随机性

## 线程生命周期

线程的状态

NEW:线程创建后，还没有运行时的状态

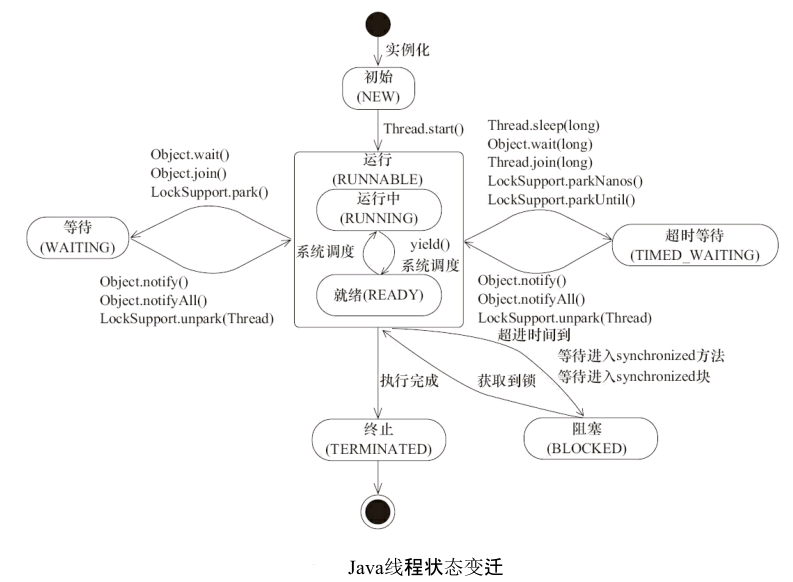
RUNNABLE：线程运行时的状态

BLOCKED：线程阻塞于锁的状态

WAITING：等待状态，表示线程进入等待，进入该状态表示需要其他线程做出一些其他的操作

TIME\_WAITING：超时等待，和waiting状态一样当前线程进入等待，但是不同的是只等待一段时间

TERMINATED：终止状态，表示线程已经执行完毕



## 守护线程

线程分为2种，用户线程和守护线程（也是一种线程，只是在作用点不用）

守护线程是一种支持型线程，它的主要被用作程序中后台调度以及支持型工作。

守护线程是一种特殊的线程，它的特殊性有陪伴的意义，当进程中不存在非守护线程的时候，守护线程会自动销毁；典型的守护线程就是垃圾回收线程，当进程中没有非守护线程了，则垃圾回收线程也就没有存在的必要了，自动销毁。

守护线程可以看作是整个jvm中所有非守护线程的保姆，只要jvm中存在一个非守护线程，那么守护线程就在工作，只有当最后一个非守护线程结束运行时，守护线程才会随着jvm一起结束工作

设置守护线程

Thread的setDaemon(boolean)方法，当参数为true表示设置当前线程为守护线程

设置守护线程需要在线程启动之前设置，不能在启动线程之后设置

# 并发编程

## Synchronized

### 简介

线程安全的主要原因是多个线程同时操作共享资源，导致共享资源和运行过程出现了不可预期的结果

那么就需要一种安全机制来保证，多线程环境下只有一个线程能够访问共享资源，其他线程必须等待这个线程访问结束，才能去获取共享资源

这种方式叫做互斥锁，synchronized关键字可以用在方法或者代码块上，用来保证在同一时刻只有一个线程能够访问它修饰的方法或者代码块，

保证线程对共享变量的排他性和可见性

Synchronized关键字的作用范围

1. 修饰静态方法：对当前类对象加锁，进入同步代码需要获取当前类对象锁（当前类的class对象）
2. 修饰实例方法：对调用当前方法的实例对象加速，进入同步代码块需要获取当前实例对象锁（也就是this）
3. 作用在同步代码块上：指定的对象，对给定对象加锁，进入同步代码块需要获取给定对象锁

### 底层实现

#### Java对象的内存布局

Java对象的内存布局可分为3个区域：对象头，实例数据，对象填充

对象头包含2部分信息：第一部分：存储对象自身运行时的数据，比如hashcode，gc年龄分代，锁状态标志，线程持有的锁，偏向线程id,偏向时间戳，这部分的长度在32位和64位的jvm中分别是32bit和64bit,这部分称为Mark word；第二部分是类型指针，即对象指向它的类元数据的指针，通过这个指针来确定对象是属于哪个类的实例；

对象头里面存储的信息是和对象本身的定义的数据没有关系的额外存储成本，考虑到jvm的空间效率，Mark word被设计为一个可变的数据结构

对象头的Mark word就是实现synchronized关键字锁对象的基础

实例数据：对象实际的数据，包括成员变量

对齐填充：填充对象，保证对象占用的空间是8的整数倍

#### Monitor

当锁状态是重量级锁时，锁标记为10，Mark word中就会存在一个指向互斥量（重量级锁）的指针，这个指针指向的就是monitor对象的起始位置，每个对象都会存在一个与之关联的monitor对象（监视器），monitor可以与对象一起创建销毁或者当线程试图获取对象锁时自动生成；

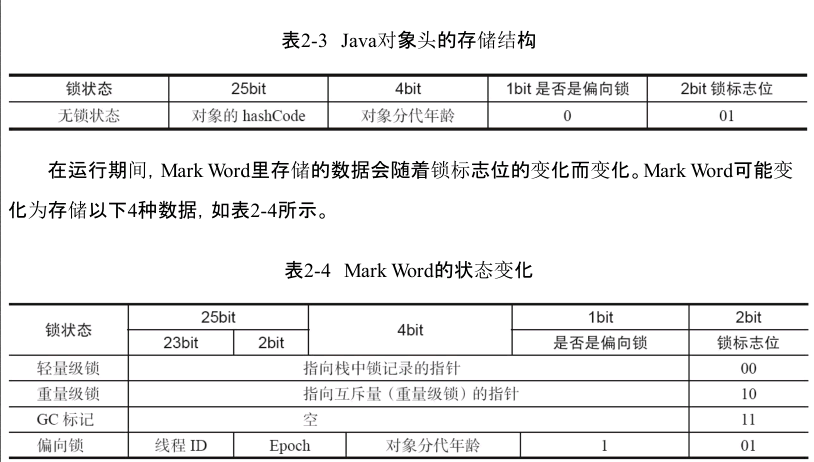
当monitor对象被线程持有时，它变处于锁定状态（实现monitor对象的锁定是jvm内部通过c语言或者c++实现的）；synchronized就是通过这种方式获取锁的

#### Synchronized的实现

当synchronized作用在同步代码块时，java编译器会在同步代码块开始和结束的地方增加2个指令：monitorenter和monitorexit；monitorenter指的是进入同步代码块，monitorexit指的是退出同步代码块；当执行到monitorenter指令时，线程就会去获取锁对象对应的monitor对象，当线程持有对象锁的monitor时，那么就成功获取了锁，锁计数器就为1，那么其他线程获取对象锁的monitor时就会被阻塞；当前获取锁的线程可以再次获取锁，获取后计数器会+1；当执行到指令monitorexit后，锁就会释放，锁释放后计数器就会-1，当计数器值为0是那么当前线程就完全释放了锁；其他线程就能获取锁了；

当synchronized所用在方法上时（实例方法和静态方法），方法的同步不是通过指令完成的，而是通过方法常量池中的方法表结构中的ACC\_SYNCHRONIZED访问标记区分方法是否是同步方法；当调用方法时，调用指令会检查方法的ACC\_SYNCHRONIZED访问标记是否被设置，如果设置了那么就需要当前线程持有monitor对象，然后在执行方法，最后执行完方法后释放monitor；在方法执行期间，执行线程持有了monitor，其他任何线程都无法再获得同一个monitor。如果一个同步方法执行期间抛 出了异常，并且在方法内部无法处理此异常，那这个同步方法所持有的monitor将在异常抛到同步方法之外时自动释放。

### 锁优化



Java早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的Mutex Lock来实现的，而**操作系统实现线程之间的切换时需要从用户态转换到核心态**，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的synchronized效率低的原因；

研究人员发现，大多数对象的加锁和解锁都是在特定的线程中完成。也就是出现线程竞争锁的情况概率比较低Java 6之后，为了减少获得锁和释放锁所带来的性能消耗，引入了轻量级锁和偏向锁

大多数对象的加锁和解锁都是在特定的线程

锁一共有4种状态：无锁状态，偏向锁，轻量级锁，重量级锁

#### 偏向锁

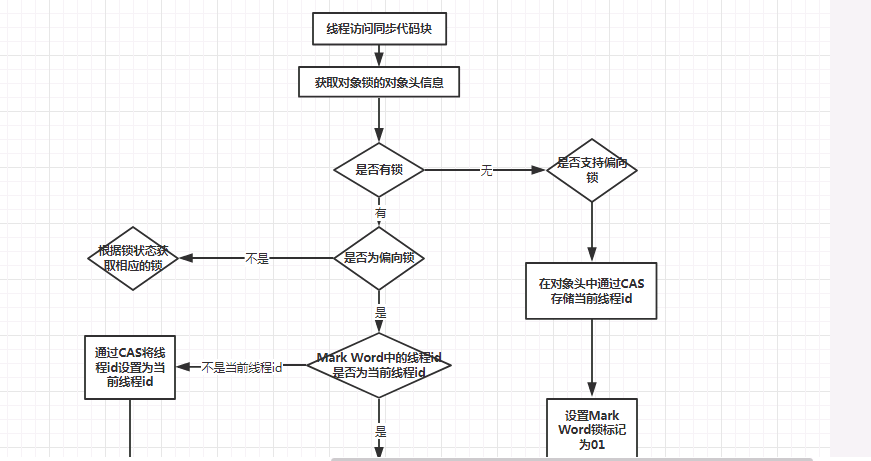
当锁不仅不存在竞争，而且总是由同一线程多次获取锁时，为了减少线程获取锁的开销引入了偏向锁；

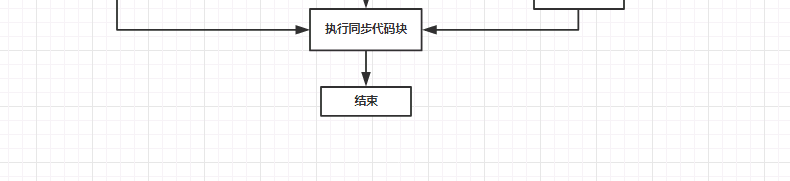
偏向锁的获取

偏向锁状态只能从无锁状态过来

当一个线程访问同步块获取锁时，会在对象头和栈帧中的锁记录中存储锁偏向的线程ID,以后该线程再次进入或者退出同步块时就不需要进行CAS操作来加锁和解锁，只需要判断下对象头的markworld中是否存在指向当前线程的偏向锁，如果有偏向锁id,表示线程已经获取了锁；如果没有偏向锁，就要先检查markworld的锁标记位是否是01（01表示偏向锁，检查是否是偏向锁，可能是因为其他线程执行完成后撤销了偏向锁），如果没有设置，就使用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

线程获取偏向锁流程





偏向锁撤销

偏向锁使用了一种竞争出现才释放锁的机制，所以当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（这个时间点上没有正在执行的字节码）

流程是：首先会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，则将对象头设置为无锁状态；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要么重新偏向于其他的线程，要么恢复到无锁状态或者标记对象不适合作为偏向锁，最后唤醒暂停的线程

总结

1. 暂停拥有偏向锁的线程，检查线程是否存活
2. 处于非活动状态，则设置为无锁状态
3. 存活，则重新偏向于其他线程或者恢复到无锁状态或者标记对象不适合作为偏向锁
4. 唤醒线程

如果一个线程获得了锁，那么锁就进入了偏向模式，此时Mark word的结构也变为偏向锁结构，当这个线程再次请求锁时，无需要在做任何操作（即获取锁的过程），这就省去了大量有关锁申请的操作，从而提供了性能

偏向锁可以提高同步但无竞争的程序性能，但是并不一定总有利，如果程序中的大多数的锁总是被多个不同的线程访问，那么偏向模式就是多余的。

可以使用参数-XX:-UseBiasedLocking=false来禁止偏向锁，那么程序就会默认进入轻量级锁

偏向锁升级为轻量级锁：存在另一个线程进入同步代码块，则升级为轻量级锁

#### 轻量级锁

不是用来代替重量级锁的，本意是在没有多线程竞争的情况下，减少重量级锁使用操作系统互斥量产生的性能消耗

多线程交替进入临界区（多线程交替获取锁情况下）

加锁

当线程执行同步代码块的时候，如果同步对象没有被锁定，jvm会先在当前线程的栈帧中创建用于存储锁记录的空间（就是存放对象的mark word），并将对象头中的mark word复制到锁记录中

然后线程尝试使用CAS将对象头中mark word替换为指向锁记录的指针，如果成功，当前线程获得锁，让锁标志设置为00；如果失败，jvm首先会检查对象的markword的锁记录的指针是否指向当前线程，如果指向当前线程，那么当前线程就已经获取到了锁，如果没有表示其他线程获取到了锁，当前线程便尝试使用自旋来获取锁（不阻塞，通过自旋来等待），当超过自旋次数后就膨胀为重量级锁

当另一个线程与当前线程同时竞争时，锁会升级为重量级锁，为了防止自旋，一旦升级就无法降级（自旋会消耗cpu，为了避免无用的自旋，一旦锁升级成重量级锁，就不会再度恢复到轻量级锁状态）

解锁

使用CAS操作将栈帧中的锁记录替换回对象头，如果成功，整个过程就完成；如果失败说明有其他的线程尝试过获取锁，那么就要在释放锁的同时唤醒被挂起的线程

轻量级锁能够提升程序性能的条件是”对于绝大部分的锁在同一周期能都不存在竞争”；在没有竞争时，轻量级锁使用CAS操作避免了使用互斥量的开销，但是存在竞争时，不仅会在互斥量带来的开销还会发生额外的CAS操作，所以在存在竞争的时候，轻量级锁的性能比较重量级锁更差

轻量级锁升级为重量级锁

有线程获取到了轻量级锁，同时有其他的线程来竞争锁，首先这个线程会尝试自旋来获取锁，当尝试自旋到一定次数（默认是10次）后仍然没有获取到锁，那么就升级为重量级锁

为什么要设计为自旋一定次数后才能升级为重量级锁

一般来说，同步代码块内的代码应该很快就执行结束，这时候线程B 自旋一段时间是很容易拿到锁的，但是如果不巧，没拿到，自旋其实就是死循环，很耗CPU的，因此就直接转成重量级锁咯，这样就不用了线程一直自旋了

#### 重量级锁

其他线程视图获取锁，都会被阻塞，只是持有锁的线程释放之后才会唤醒这些线程

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 使用场景 |
| 偏向锁 | 加锁和解锁不需要额外的性能开销， | 如果存在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的场景 |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的性能 | 如果始终得不到锁竞争的线程，就会使用自旋来等待，会消耗cpu | 追求响应时间，同步块执行的速度非常快 |
| 重量级锁 | 线程竞争锁，不会自旋消耗cpu | 线程阻塞，影响响应 | 追求吞吐量，同步块执行速度较长 |

* 偏向锁：仅有一个线程进入临界区
* 轻量级锁：多个线程交替进入临界区
* 重量级锁：多个线程同时进入临界区

锁的分类：从宏观上分为乐观锁和悲观锁

乐观锁：任务读多写少，遇到并发写的可能性低，每次去拿数据的时候都认为没有被修改，所有不会上锁，但是在更新的时候会判断一下期间是否有别的线程修改了这个数据，采取在写时先读出当前版本号，然后加锁操作，如果失败则重复读-比较-写操作

Java中的乐观锁是基于CAS操作实现的，CAS操作是一种更新的原子操作，比较当前值跟传入的值是否一样，不一样失败，一样就更新

悲观锁：悲观实现，即认为写多，遇到并发写的可能性高，每次去拿数据的时候认为别人会修改，所以每次读写数据的时候都会上锁，这样其他线程来获取数据的时候就会被阻塞知道拿到了锁；

java中的悲观锁就是Synchronized,AQS框架下的锁则是先尝试cas乐观锁去获取锁，获取不到，才会转换为悲观锁，如RetreenLock。

阻塞的代价

如果要阻塞或唤醒一个线程就需要操作系统介入，需要在用户态与核心态之间切换，这种切换会消耗大量的系统资源，因为用户态与内核态都有各自专用的内存空间，专用的寄存器等，用户态切换至内核态需要传递给许多变量、参数给内核，内核也需要保护好用户态在切换时的一些寄存器值、变量等，以便内核态调用结束后切换回用户态继续工作。

如果线程状态切换是一个高频操作时，这将会消耗很多CPU处理时间；

如果对于那些需要同步的简单的代码块，获取锁挂起操作消耗的时间比用户代码执行的时间还要长，这种同步策略显然非常糟糕的。

锁消除

锁消除是Java虚拟机在JIT编译时，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过锁消除，可以节省毫无意义的请求锁时间。

锁粗化就是把很多次锁的请求合并成一个请求，以降低短时间内大量锁请求、同步、释放带来的性能损耗。

## Volatile

### Volatile的作用

1. 保证共享变量的可见性
2. 禁止指令重排

多线程并发编程围绕着三个特性来实现

1. 可见性：一个线程修改了共享变量，另一个线程能够获取这个共享变量修改之后的值
2. 原子性：一个操作或者一组操作要么全部执行，要么全部不执行
3. 有序性：程序执行的顺序按照代码的先后顺序执行

**可见性**

volatile 很适合只有一个线程修改，其他线程读取的情况。volatile 变量被修改之后，对其他线程立即可见。Volatile只能保证共享资源的可见性，但是不能保证操作的原子性，所以在保证操作的原子性时仍然需要加锁；通过lock前缀指令实现的

Volatile底层实现

volatile变量进行写操作的代码在转换为汇编代码时会多出一个Lock前缀指令

这个Lock前缀指令在多核处理器下会引发2件事

1. 将当前处理器缓存行的数据回写到系统内存中，
2. 这个回写操作会使其他cpu中的缓存了该内存地址的数据无效（其他cpu的线程获取该数据时需要重新填充内存）

为了提供访问的速度，通过会将系统内存中的数据读到缓存中然后进行操作，但不知道什么时候将数据回写回内存中

如果对声明为volatile的变量进行写操作，那么jvm就会向操作系统发送一个lock前缀指令，将这个变量所在的缓存行的数据回写到内存中；在多处理器环境下，为了保证各个处理器的缓存是一致的，就会需要实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己的缓存是否是过期的，如果发现自己缓存行对应的内存地址被修改了，就将当前处理的缓存行设置为无效状态，当这个处理器需要操作这个数据的时候，会重新从内存中把数据读到处理器缓存行中

写操作导致缓存中的数据回写到内存中，其他处理器发现自己缓存行对应内存的地址被修改了，那么处理器就将该缓存行设置为无效状态，如果该处理器要操作数据，那么就从内存中把数据重新填充到缓存行中

**指令重排序**

使用volatile的第二个语义就是禁止指令重排序，普通变量不能保证变量的赋值操作顺序和程序代码中的执行顺序一致，因为一个线程方法在执行过程中是无法感知这点的，这也就是java内存模型中描述的所谓的“线程内表现为串行的语义”

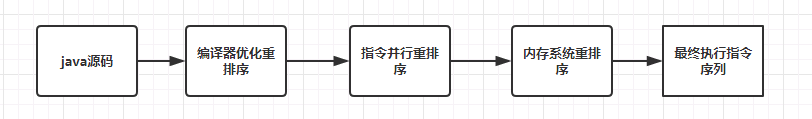
为了提供程序运行的效率和性能，编译器和处理会对指令进行重排序

重排序

编译器和处理器为了优化程序性能而对指令序列进行重排序的一种手段；在重排序时会遵守数据依赖性，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序；这里的数据依赖性仅针对单个处理器中执行的指令序列和单个线程中执行的操作，不同的处理器和不同线程之间的数据依赖性不被编译器和处理器考虑（重排序是在单线程和单处理器环境下进行的）

重排序分为3中

1. 编译器优化重排序：编译器在不改变程序语义的前提下，可以重新安排语句的执行顺序
2. 指令级并行重排序：处理器采用指令级并行技术来将多条指令重叠执行，如果不存在数据的依赖性，处理器可以改变语句对应机器指令的执行顺序
3. 内存系统重排序：由于处理使用缓存和读写缓冲区，这使得加载和存储操作看上去是乱序的



编译器重排序发生在编译器，指令级并行重排序和内存系统重排序时处理器重排序，在多线程环境下，重排序可能会导致内存可见性问题

as-if-serial

as-if-serial语义：不管怎么重排序（编译器和处理器为了提高并行度），（单线程）程序的执行结果不能被改变；为了遵守as-if-serial语义，编译器和处理器不会对存在数据依赖关系的操作进行重排序，这种重排序会改变执行结果

重排序对多线程的影响

（控制依赖：存在if条件判断）

在单线程中对存在控制依赖的操作重排序，不会改变执行结果；但是在多线程环境下，对存在控制依赖的操作重排序，可能会改变执行结果

使用volatile修饰的变量在进行赋值的时候会生成一个内存屏障，来禁止特定类型的重排序，在指令重排序时不能把后面的指令拍到内存屏障之前；

在每个volatile写操作的前面插入storestore屏障，storestore屏障禁止上面的普通写和下面的volatile写重排序

在每个volatile写操作的后面插入storeload屏障，防止上面的volatile写和下面可能有的volatile读/写重排序

在每个volatile读操作的后面插入loadload屏障，禁止下面的普通读操作和上面的volatile读操作重排序，

在每个volatile读操作的后面插入loadstore屏障，禁止下面的普通写操作和上面的volatile读重排序

双重检查锁（double checked locking）

**public** **class** DoubleCheckedLocking

{

**private** **static** **volatile** DoubleCheckedLocking *dcl* = **null**;

**private** DoubleCheckedLocking()

{

}

**public** **static** DoubleCheckedLocking getDoubleCheckedLocking()

{

**if** (*dcl* == **null**)

{

**synchronized** (DoubleCheckedLocking.**class**)

{

**if** (*dcl* == **null**)

*dcl* = **new** DoubleCheckedLocking();

}

}

**return** *dcl*;

}

}

使用synchronized关键字来保证创建对象过程的原子性

创建对象的过程包含三个步骤

1. 在堆内申请一块内存空间
2. 初始化申请好的内存空间
3. 将内存空间的地址赋值给对象

所以创建对象是一个由三步操作组成的复合操作，多线程环境下A 线程执行了第一步、第二步之后发生线程切换，B 线程开始执行第一步、第二步、第三步（因为A 线程singleton 是还没有赋值的），所以为了保障这三个步骤不可中断，可以使用synchronized 在这段代码块上加锁

使用volatile来保证对象的可见性和禁止指令重排

创建对象分为上面的三步，但是可能会将第二步和第三步重排序，所以使用volatile来禁止指令重排序，每个没有禁止，那么执行赋值后，其他线程就能够拿到还没有初始化完成的对象

第二次检查是否为null

A 线程进行判空检查之后开始执行synchronized代码块时发生线程切换(线程切换可能发生在任何时候)，B 线程也进行判空检查，B线程检查 singleton == null 结果为true，也开始执行synchronized代码块，虽然synchronized 会让二个线程串行执行，如果synchronized代码块内部不进行二次判空检查，那么A线程恢复后就会进行二次初始化。

## 原子操作的实现

原子操作：一个或者一系列的不可中断的操作

处理器实现原子操作：处理器提供总线锁和缓存锁两个机制来保证复杂内存操作的原子性

1. 使用总线锁保证操作的原子性

总线锁就是使用处理器提供的一个LOCK#信号，当一个处理器在总线上输出这个信号时，其他处理器会被阻塞，那么该处理器就独占内存；

总线锁把内存和cpu的通信锁住，这个期间其他的处理器不能操作内存地址的数据，所以总线锁的开销很大，

1. 缓存锁保证操作的原子性：

当频繁使用缓存，那么原子操作可以直接在缓存中进行，不需要声明总线锁；在同一时刻只需要保证对某个内存地址的操作是原子性即可；

缓存锁指的是内存区域被缓存到缓存行中，并且在LOCK操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不在总线上声明LOCK#信号，而是修改内存地址，通过缓存一致性机制来保证操作的原子性，因为缓存一致性会阻止同时修改2个以上的处理器缓存的数据，当其他处理器回写已被锁定的缓存行的数据时，会是缓存行设置为无效

缓存一致性机制整体来说，是当某块CPU对缓存中的数据进行操作了之后，就通知其他CPU放弃储存在它们内部的缓存，或者从主内存中重新读取

有2种情况下处理器不会使用缓存锁

1. 当操作的数据不能被处理器缓存时
2. 有些处理器不支持缓存锁定

Java实现操作的原子性：通过锁和CAS

CAS的实现：使用循环CAS实现操作的原子性，jvm的CAS利用了处理器提供的CMPXCHG指令实现的；也即是通过c语言调用cpu底层指令来实现的

CompareAndSwatch：比较和替换，将期望值和内存中的值进行比较，如果相等就用替换值代替内存中的值

根据CAS操作，java提供了java.util.concurrent.atomic包下的原子操作类，实现了用法简单，性能高效，线程安全的更新变量的方式

CAS操作的三大问题

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

从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

循环时间长开销大：自旋CAS如果长时间不成功，会给cpu带来非常大的开销；

只能保证一个变量的原子操作：

原子更新基本类型

AtomicInteger:原子更新Integer

AtomicBoolean：原子更新布尔类型

AtomicLong：原子更新长类型

这3个类提供的方法执行流程几乎一样，以AtomicInteger为例展示相关方法

AtomicInteger常用方法

Int addAndGet(int delta):以原子的方式将输入的值和内存中的实际值相加并返回结果

主要流程循环获取内存中的值，和输入值相加后，调用unsafe的compareAndSwapInt方法和将获取到的值和内存中的实际值进行比较，如果相等，那么就将内存中的实际值替换为相加后的结果，如果不相等，那么就是有线程修改了内存中的值，那么重新获取当前值，再进行以上的步骤，知道成功为止（jdk1.7）

Jdk1.8(直接调用unsafe的getAndAddInt方法)

**public** **final** **int** addAndGet(**int** delta) {

**return** *unsafe*.getAndAddInt(**this**, *valueOffset*, delta) + delta;

}

compareAndSet(int expect,int update):如果输入的期望值expect等于原值，则以原子方式将原值用更新值替换

**public** **final** **boolean** compareAndSet(**int** expect, **int** update) {

**return** *unsafe*.compareAndSwapInt(**this**, *valueOffset*, expect, update);

}

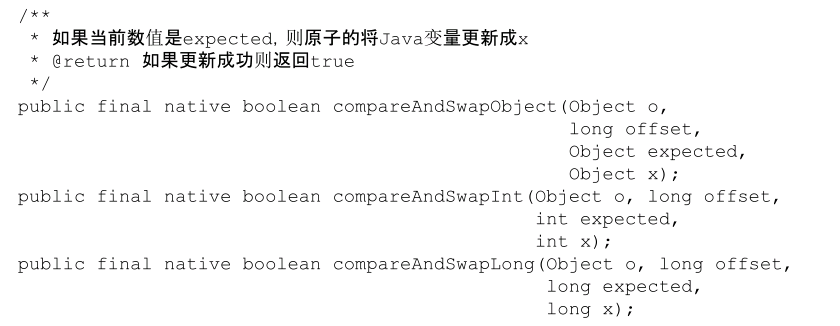
Int getAndIncrement():当前值以原子方式+1，并返回结果；

void lazySet(int v):最终会设置为v,可能导致其他线程在一段时间内获取的是旧值

int getAndSet(int v)以原子的方式设置值，并返货旧值

Atmoic中只提供了3中基本类型的原子更新类；其他的为什么没有

原子更新操作主要是使用了unsafe来实现的，在unsafe中只提供了3类型的原子更新方法，他们是



Unsafe替换只支持3中类型的CAS操作，Integer、long,Object；AtomicBoolean是将boolean类型转换为int类型后在进行转换的（true，false分别对应1,0），在使用int类型的CAS方法替换，所以针对其他基本类型也可以使用这个思路

原子更新数组（更新数组的某个元素）

AtomicIntegerArray ：原子更新整型类型数组的元素

AtomicLongArray ：原子更新长整型类型数组的元素

AtomicReferenceArray：原子更新引用类型数组的元素

AtomicIntegerArray类常用的方法

Int addAndGet(int I,int delta)以原子的方式将输入值与数组中索引i的元素相加

获取索引i位置在数组中的偏移量，更加偏移量获取对应位置的值，相加之后进行原子替换

**public** **final** **int** addAndGet(**int** i, **int** delta) {

**long** offset = checkedByteOffset(i);//获取索引i位置在数组中的偏移量

**while** (**true**) {

**int** current = getRaw(offset);//取出对应位置的值

**int** next = current + delta;

**if** (compareAndSetRaw(offset, current, next))

**return** next;

}

}

使用原子方式替换对应索引位置的值；checkedByteOffset方法获取索引位置的偏移量

**public** **final** **boolean** compareAndSet(**int** i, **int** expect, **int** update) {

**return** compareAndSetRaw(checkedByteOffset(i), expect, update);

}

原子更新引用

AtomicReference ：原子更新引用类型

AtomicReferenceFieldUpdater：原子更新引用类型中的字段

AtomicMarkableReference：原子更新带有标记的引用类型

原子更新属性（字段：更新一个类中的某个字段，使用原子更新属性）

AtomicIntegerFieldUpdater：原子更新整型字段的更新器

AtomicLongFieldUpdater：原子更新长整型字段的更新器

AtomicStampedReference：原子更新带有版本号的引用类型

## Java内存模型

并发编程中需要处理的2个问题：线程之间是如何的通信和线程之间是如何同步的

通信是指线程之间通过某种机制来交换信息，在命令式编程中，线程之间的通信机制有2中：共享内存和消息传递

在共享内存模型中，线程之间共享程序的公共状态，通过读写内存中的公共状态进行隐式通信；

在消息传递的并发模型里，线程之间并没有公共状态，线程之间必须发送消息来进行显示通信；

同步是指程序中用于控制不同线程间操作发送相对顺序的机制（控制各个线程执行的顺序）。在共享内存并发模式里，同步是显示进行的，必须指定某段代码块或者方法在线程之间是互斥执行的。在消息传递模式中，由于消息的发送必须在消息接收前，因此同步是隐式的

Java并发采用的是共享内存模式，java线程之间的通信时隐式的

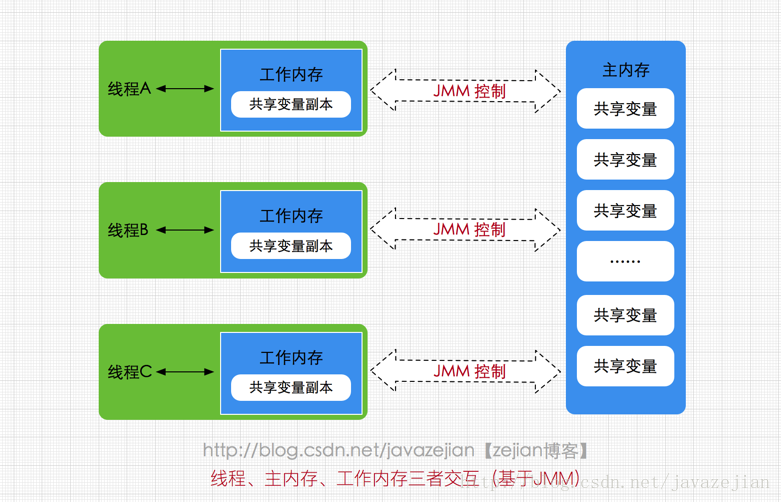
Java内存模型（JMM）的抽象结构

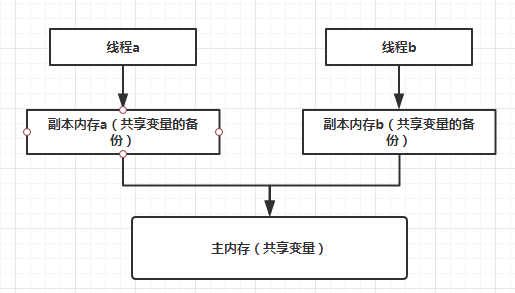
定义：JMM是一个抽象的概念，并不存在的，它描述的是一组规则或规范，通过这些规范定义程序各个变量的访问方式，目的就是屏蔽各种硬件和操作系统的内存访问差异，以实现java程序在各个平台下都能达到一致的内存访问效果

由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个私有工作内存，用于存储该线程用于读写的共享变量，而Java内存模型中规定

所有的变量都存储在主内存中，每个线程还有自己的工作内存，线程的工作内存中保存了该线程使用到的变量（主内存的拷贝），线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，而不能直接读写主内存中的变量。不同线程之间无法直接访问对方工作内存中的变量，线程间变量值的传递均需要在主内存来完成。

主内存、工作内存和jvm的内存区域堆，栈，方法区并不是同一个层次的内存划分，这两者基本上是没有关系的；如果将两者勉强对应，那么从变量，内存，工作内存的定义来看，主内存—》java堆的对象实例数据部分，工作内存对应的就是栈中的部分区域；从更低层次来说主内存就是硬件的内存，工作内存对应的就是寄存器，高速缓存等





线程a,b之间的通信步骤

1. 线程a将副本内存a中的变量刷新到主内存中
2. 线程b从主内存中读取线程a更新后的最新共享变量

从整体来看这个步骤实质上是线程a在向线程b发送消息，而且这个消息必须经过主内存

Jmm定义了8种操作来完成主内存和工作内存之间具体的交互协议（变量从主内存拷贝到工作内存，从工作内存回写到主内存）

作用于主内存

lock：把变量标识为一条线程独占的状态

unlock:把一个处于锁定状态的变量释放出来，释放后的遍历才能被其他的线程锁定

read：把变量的值从主内存传送到线程的工作内存中，以便随后的load动作

作用于工作内存

load：把read操作从主内存中获取到的值放入工作内存的变量副本中

user:把工作内存的变量的值传递给执行引擎，每当虚拟机遇到一个需要使用这个变量值的指令时就会执行这个操作

assign：它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。

store：把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。

write: 它把store操作从工作内存中一个变量的值传送到主内存的变量中。

什么是可见性：一个线程修改了共享变量的值，其他线程也能看到最新修改的值。

可见性问题就是不能及时的获取共享变量更新后的值

可以通过锁来保证操作的原子性，从而来保证共享变量修改后的可见性（synchronized，reentrentLock）,也可以使用volatile，volatile的另一个作用就是禁止指令重排序

JMM的解决方案

JMM定义了一套happens-before原子来保证多线程环境下两个操作间的原子性，可见性以及有序性

**有序性**

**一个线程的所有操作都必须按照程序的顺序来执行**

**所有线程都只能看到一个单一的操作执行顺序，每个操作都必须原子的执行且立刻对所有线程可见**

**Java提供了volatile和synchronized这2个关键字来保证线程之间操作的有序性；volatile关键字本身具有禁止指令重排序的作用，而synchronized则是由“一个变量在同一时刻只能被一个线程进行操作”这个作用决定了持有同一个锁的两个同步只能串行执行**

**happens-before（先行先发原则）**

happens-before的定义:

如果第一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，可见的内容包括内存中共享的变量值，发送的消息，调用的方法。

通过happens-before关系向程序员提供跨越线程的内存可见性保证

Jmm自带的Happens-before规则

**程序顺序规则**：一个线程的中每个操作，happens-before于该线程中的任意后续操作（在一个线程内必须保证语义串行性，也就是说按照代码顺序执行）

**监视器规则**：对一个锁的解锁，happens-before于随后对这个锁的加锁

**Volatile变量规则**：对于一个volatile域（修饰的变量）的写，happens-before与任意后续对于这个volatile域的读

**传递性**：a happen before b,b happen before c,那么a就happens before c

**线程启动规则Start()**:线程的start()方法先于它的每一个动作，如果线程a执行了线程b的start()(启动线程b),那么start（）方法happens-before线程b的任意操作

**线程终止规则Join()**:如果线程a中执行操作线程b的join()方法并成功返回，那么线程b中的任意操作happens-before于线程a从线程b执行join方法成功返回

**线程中断规则：**对线程 interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测线程是否中断。

**对象终结规则**：一个对象的初始化完成先行发生于它的finalize()方法的开始

以上规则java无需使用任何同步手段就能保证先行先生，如果两个操作之间的关系不在上面，并不能从下列规则中推导出来，那么他们就没有顺序性的保证

两个操作之间存在happens-before的关系，并不意味着一个操作一定在另一个操作之前发生，而是要求前一个操作（执行的结果）对后一个操作可见。且前一个操作按顺序排在第二个操作之前执行（第二个操作可以在第一个操作执行过程中执行）

比如

int i=1;

int j=2;

根据程序次序规则，i=1先行发生于j=2，但是j=2的代码完全可能先被执行，这并不影响先行先生的规则，所以时间先后顺序和先行先生原则之间基本没有太大的关系

顺序一致性内存模型

一个多线程程序能正确同步，那么这个程序就是一个没有数据竞争的程序

JMM对正确同步的多线程程序的内存一致性做了如下保证：程序的执行具有顺序一致性

程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同

顺序一致性内存模型的两大特征

一个线程的所有操作必须按照程序的顺序来执行

所有线程都只能看到一个单一的操作执行顺序。在顺序一致性内存模型内，每个操作都必须是原子执行的且对所有线程可见

多线程环境下，在线程执行同步的代码块时，是串行执行的

锁内存语义

锁释放和获取的内存语义

**当线程锁释放，JMM会把该线程对应的副本内存中的共享变量刷新到主内存中，**

**当线程获取锁，JMM会把该线程对应的副本内存置为无效，直接从主内存中获取共享变量**

## AbstractQueuedSynchronizer(AQS)

JUC（java.util.concurrent）

### AQS简介

AbstractQueuedSynchronizer队列同步器是用来构建锁或者其他同步组件的基础框架，这是一个抽象类；

它使用一个int类型的变量表示同步状态，通过内置的FIFO队列来完成线程的排队工作,通过AQS是实现大部分同步需求的基础；

同步器的主要使用方式是继承，子类通过继承同步器并实现它的抽象方法来管理同步状态，子类被推荐定义为自定义同步组件的静态内部类。同步器自身没有实现任何接口，它仅仅是定义了若干同步状态获取和释放的方法来供自定义同步组件使用，同步器既可以支持独占式的获取同步状态，也支持共享式的获取同步状态；JDK中许多并发工具类的内部实现都依赖于AQS，如ReentrantLock, Semaphore, CountDownLatch等等

AQS设计基于模板方法模式，开发者需要继承同步器并且重写指定的方法，将其组合在并发组件的实现中，调用同步器的模板方法，模板方法会调用使用者重写的方法。

### 设计思路

AQS内部维护一个双向队列来管理线程。  
线程会首先尝试获取锁，如果失败，则将当前线程以及等待状态等信息封装成一个Node节点加到同步队列里。  
接着会不断循环尝试获取锁（条件是当前节点为head节点的后继节点才会尝试）,如果失败则会阻塞自己，直至被唤醒；  
而当持有锁的线程释放锁时，会唤醒队列中的后继节点对应的线程。

AQS主要做3件事：1，同步状态管理，2，线程的阻塞和唤醒，3，同步队列的维护

获取修改设置同步状态的方法

getState()：获取同步状态

setState(int newState):设置当前同步状态

compareAndSetState()使用CAS来设置当前同步状态，保证修改状态的线程安全性

### 获取锁和释放锁

下面这几个方法是AQS提供的可重写的方法，在AbstractQueuedSynchronizer类中这些方法都是直接抛出异常的（throw new UnsupportedOperationException();），也就是说需要自来重写来实现具体的判断

**protected** **boolean** tryAcquire(**int** arg)：获取独占锁

**protected** **boolean** tryRelease(**int** arg)：释放独占锁

**protected** **int** tryAcquireShared(**int** arg)：获取共享锁

**protected** **boolean** tryReleaseShared(**int** arg)：释放共享锁

**protected** **boolean** isHeldExclusively()：判断当前线程是否获取了锁

AQS本身将同步状态的管理用模板方法模式都封装好了，模板方法如下

| **方法** | **描述** |
| --- | --- |
| void acquire(int arg) | 获取独占锁。会调用tryAcquire方法，如果未获取成功，则会进入同步队列等待 |
| void acquireInterruptibly(int arg) | 响应中断版本的acquire |
| boolean tryAcquireNanos(int arg,long nanos) | 响应中断+带超时版本的acquire |
| void acquireShared(int arg) | 获取共享锁。会调用tryAcquireShared方法 |
| void acquireSharedInterruptibly(int arg) | 响应中断版本的acquireShared |
| boolean tryAcquireSharedNanos(int arg,long nanos) | 响应中断+带超时版本的acquireShared |
| boolean release(int arg) | 释放独占锁 |
| boolean releaseShared(int arg) | 释放共享锁 |
| Collection getQueuedThreads() | 获取同步队列上的线程集合 |

AQS的内部类Node

Node状态的字段

**static** **final** **int** *CANCELLED* = 1;当前线程因为超时或者中断被取消。节点状态不会被改变；

**static** **final** **int** *SIGNAL* = -1;当前节点的后继节点处于被阻塞或者将要被阻塞的状态，当前节点释放锁或者取消后需要唤醒后继节点。这个状态一般都是后继节点来设置前驱节点的

**static** **final** **int** *CONDITION* = -2;当前节点在condition队列中

**static** **final** **int** *PROPAGATE* = -3; 用于将唤醒后继线程传递下去，这个状态的引入是为了完善和增强共享锁的唤醒机制。在一个节点成为头节点之前，是不会跃迁为此状态的

**volatile** **int** waitStatus;同步状态：默认的初始值为0，表示无状态

获取同步状态的方法

### acquire获取独占锁源码

Acquire方法：获取独占锁，忽略中断

首先通过tryAcquire获取锁，如果获取到了锁，也就是tryAcquire返回true，直接返回

如果没有获取锁，那么首先将当前线程封装在节点中然后插入到队列中，然后调用acquireQueued方法，判断这个节点是否是head节点的后继节点，

是head节点的后继节点的话就尝试去获取锁，如果获取到了就返回；如果获取锁失败，那么就通过LockSupport的park方法阻塞当前线程，直到被唤醒或者被中断，随后通过死循环尝试获取锁，如此直到获取到了锁

不是head节点的后继节点，那么就通过LockSupport的park方法阻塞当前线程，直到被唤醒或者被中断，随后通过死循环尝试获取锁，如此直到获取到了锁

（获取锁的过程就是将线程阻塞，通过忙循环来等待）

**public** **final** **void** acquire(**int** arg) {

**if** (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))

*selfInterrupt*();

}

addWaiter:封装当前线程到Node中，并且添加到队列中

**private** Node addWaiter(Node mode) {

Node node = **new** Node(Thread.*currentThread*(), mode);

// Try the fast path of enq; backup to full enq on failure

Node pred = tail;

Pred不为空表示添加位置不是在head之后

**if** (pred != **null**) {

node.prev = pred;

通过CAS在队列尾部添加节点，添加成功就返回；添加失败那么有其他线程在添加了节点，执行enq方法

**if** (compareAndSetTail(pred, node)) {

pred.next = node;

**return** node;

}

}

首次添加或者上面通过CAS添加失败时执行enq方法来继续添加

enq(node);

**return** node;

}

**private** Node enq(**final** Node node) {

**for** (;;) {

Node t = tail;

**if** (t == **null**) { // Must initialize

当首次添加时，创建一个新节点来作为head节点，这个节点没有线程，只是用来作为初始化的head节点，让tail也指向第一个节点，当释放时获取到这个head，不会调用唤醒阻塞的方法的，

**if** (compareAndSetHead(**new** Node()))

tail = head;

} **else** {

当存在tail节点后，让要添加的节点的前驱节点指向tail,然后使用CAS操作来tail执行向添加的节点，如果操作失败（有其他线程执行了添加节点的操作），继续循环重复以上操作；如果操作成功那么就让原tail节点的next指向当前添加的节点

node.prev = t;

**if** (compareAndSetTail(t, node)) {

t.next = node;

**return** t;

}

}

}

}

acquireQueued方法第一个参数是添加到队列中的节点；这个方法主要的过程就是循环尝试获取这个节点的锁

**final** **boolean** acquireQueued(**final** Node node, **int** arg) {

**boolean** failed = **true**;

**try** {

**boolean** interrupted = **false**;

**for** (;;) {

**final** Node p = node.predecessor();

当这个节点的前驱节点是head节点时，调用tryAcquire方法获取锁，如果成功就将当前节点设置为head节点，然后返回

如果没有获取到那么就去执行下面的代码

**if** (p == head && tryAcquire(arg)) {

setHead(node);

p.next = **null**; // help GC

failed = **false**;

**return** interrupted;

}

不是head节点或者没有获取锁，根据前驱节点判断是否要当前节点的线程阻塞

**if** (*shouldParkAfterFailedAcquire*(p, node) &&

parkAndCheckInterrupt())

interrupted = **true**;

}

} **finally** {

**if** (failed)当获取锁过程中出现异常时取消获取锁

cancelAcquire(node);

}

}

*shouldParkAfterFailedAcquire：*这个方法根据前驱节点的状态判断是否要阻塞当前节点的线程

**private** **static** **boolean** shouldParkAfterFailedAcquire(Node pred, Node node) {

**int** ws = pred.waitStatus;取出前驱节点的状态

**if** (ws == Node.*SIGNAL*)如果状态为signal，那么就表示前驱节点释放锁的时候会唤醒后继节点，所以后继节点（当前节点）可以阻塞

**return** **true**;

**if** (ws > 0) {状态大于就是cancelled状态，那么这个节点就不能被阻塞，也不需要存在队列中，向前遍历，更新当前节点的前驱为往前第一个非取消节点。 返回false， 当前线程会之后会再次回到循环并尝试获取锁。

**do** {

node.prev = pred = pred.prev;

} **while** (pred.waitStatus > 0);

pred.next = node;

} **else** {

当状态为0或者-3时，将节点状态设置为SIGNAL,在之后会回到循环再次尝试获取锁，因为状态改成了signal,所以这个方法就返回true，表示可以阻塞线程

*compareAndSetWaitStatus*(pred, ws, Node.*SIGNAL*);

}

**return** **false**;

}

中断线程并且返回中断状态

**private** **final** **boolean** parkAndCheckInterrupt() {

LockSupport.*park*(**this**);

**return** Thread.*interrupted*();

}

实现节点node取消获取锁

**private** **void** cancelAcquire(Node node) {

传入的节点为空直接返回

**if** (node == **null**)

**return**;

将节点的线程设置为null

node.thread = **null**;

找到当前节点的前驱节点的第一个非取消状态的前驱节点，pred最后就指向node节点的第一个非取消状态的前驱节点

Node pred = node.prev;

**while** (pred.waitStatus > 0)

node.prev = pred = pred.prev;

pred的next的状态为取消状态

Node predNext = pred.next;

直接把当前节点的等待状态置为取消,后继节点即便也在cancel可以跨越node节点

node.waitStatus = Node.*CANCELLED*;

**if** (node == tail && compareAndSetTail(node, pred)) {

如果当前节点是tail，那么就将prev节点设置为tail节点，并且将prev节点的next节点设置为null

*compareAndSetNext*(pred, predNext, **null**);

} **else** {

这里主要是来设置prev节点的next节点

**int** ws;

**if** (pred != head &&

((ws = pred.waitStatus) == Node.*SIGNAL* ||

(ws <= 0 && *compareAndSetWaitStatus*(pred, ws, Node.*SIGNAL*))) &&

pred.thread != **null**) {

当前节点不为tail，也不为head,取出node的next节点，当next节点不是取消状态时，将prev的next设置为node的next节点

Node next = node.next;

**if** (next != **null** && next.waitStatus <= 0)

*compareAndSetNext*(pred, predNext, next);

} **else** {

/\* \* 这时说明pred == head或者pred状态取消或者pred.thread == null

\* 在这些情况下为了保证队列的活跃性，需要去唤醒一次后继线程。

\* 举例来说pred == head完全有可能实际上目前已经没有线程持有锁了，

\* 自然就不会有释放锁唤醒后继的动作。如果不唤醒后继，队列就挂掉了。

\* \* 这种情况下看似由于没有更新pred的next的操作，队列中可能会留有一大把的取消节点。

\* 实际上不要紧，因为后继线程唤醒之后会走一次试获取锁的过程，

\* 失败的话会走到shouldParkAfterFailedAcquire的逻辑。

\* 那里面的if中有处理前驱节点如果为取消则维护pred/next,踢掉这些取消节点的逻辑。

\*/

unparkSuccessor(node);

}

node.next = node; // help GC

}

}

找到node节点的非取消的后继节点，如果存在，那么就通过通过LockSupport的*unpark方法*唤醒这个节点对应的线程

如果这就节点不存在或者是取消的，那么就从为节点依次向前取出列node最近的非取消节点，如果这个节点存在那么就唤醒线程

唤醒node之后距离node最近的非取消节点

**private** **void** unparkSuccessor(Node node) {

**int** ws = node.waitStatus;

**if** (ws < 0)

*compareAndSetWaitStatus*(node, ws, 0);

Node s = node.next;

**if** (s == **null** || s.waitStatus > 0) {

s = **null**;

**for** (Node t = tail; t != **null** && t != node; t = t.prev)

**if** (t.waitStatus <= 0)

s = t;

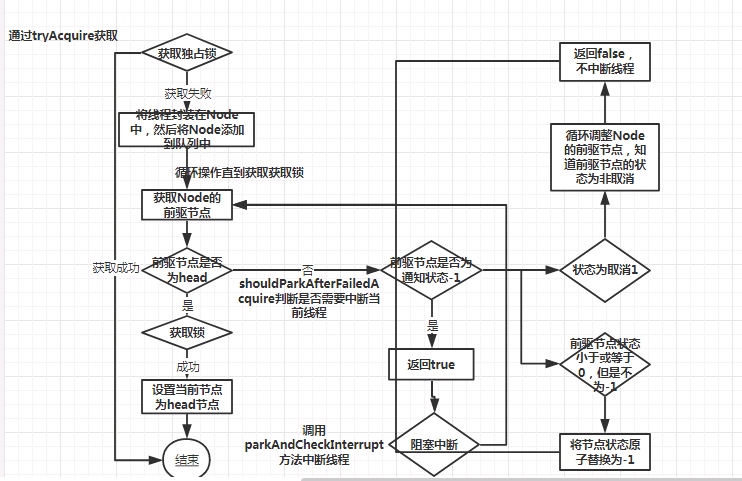
}

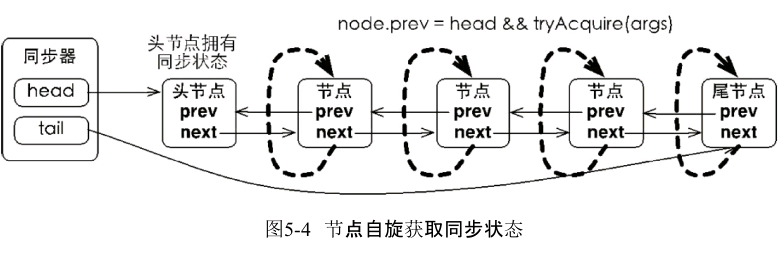
**if** (s != **null**)

LockSupport.*unpark*(s.thread);

}

获取独占锁流程图





### release释放独占锁源码

释放锁首先调用tryRelease释放掉锁状态，当锁状态完全释放后，后面的线程就可以获取锁；是否同步状态，唤醒后继节点

当通过tryRelease释放掉锁后，获取的head节点存在3中情况

1. head节点为null，就是只有一个线程获取锁，没有其他线程竞争，那么就不会生成队列，head就为null
2. 当前线程所在节点是在acquireQueued中new出来的，通过setHead设置的，获取的head就是当前线程对应的（也就是后面的节点还没有来得及执行setHead方法）
3. 线程释放锁后，有新的线程在acquireQueued方法中通过setHead设置了head节点

第三种情况可以分为2种情况

情况1

1. 线程A通过acquireQueued方法获取的锁，通过setHead设置为了head节点
2. 线程B通过tryAcquire获取锁失败后，在acquireQueued中循环
3. 当线程A释放掉锁后，线程B通过tryAcquire获取锁，然后将线程B设置为head，此时的head节点就是线程B
4. A释放时获取到的head就是线程B的节点

情况2

1. 线程A通过tryAcquire获取锁，此时的head为null
2. 线程B就只能进入acquireQueued方法，生成等待队列，队列的head为一个傀儡node
3. 当a释放，b还没有获取锁时，获取的节点就是这个傀儡node

当获取的head节点的状态是唤醒状态时，调用unpartSuccessor方法，唤醒后继非取消的节点

**public** **final** **boolean** release(**int** arg) {

**if** (tryRelease(arg)) {

Node h = head;

判断head是否为空和判断waitStatus是否！=0，就是判断当前head是否为傀儡节点（这个节点没有线程，只是用来作为初始化的head节点）；当head是线程节点，并且是可以被唤醒的状态时执行唤醒线程的方法

**if** (h != **null** && h.waitStatus != 0)

unparkSuccessor(h);

**return** **true**;

}

**return** **false**;

}

首先判断状态是否小于0，如果是就将状态修改为0

然后找到距离当前节点最近的可以被唤醒的节点（也就是状态小于0 ，这也是要先将当前节点状态修改为0的原因，防止找到的被唤醒节点是自己）

**private** **void** unparkSuccessor(Node node) {

**int** ws = node.waitStatus;

**if** (ws < 0)

*compareAndSetWaitStatus*(node, ws, 0);

Node s = node.next;

**if** (s == **null** || s.waitStatus > 0) {

s = **null**;

**for** (Node t = tail; t != **null** && t != node; t = t.prev)

**if** (t.waitStatus <= 0)

s = t;

}

**if** (s != **null**)

LockSupport.*unpark*(s.thread);

}

Release做的事情就是

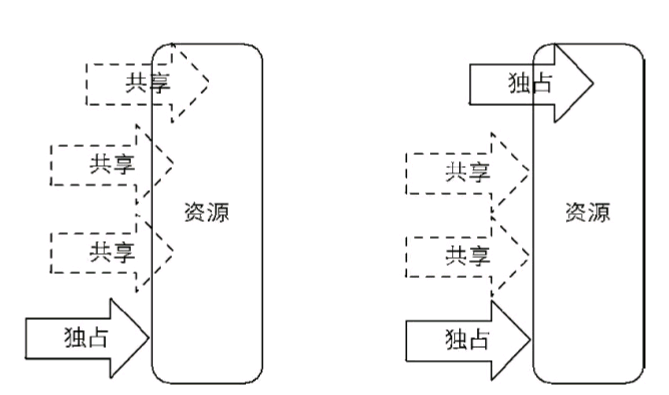
调用tryRelease

如果tryRelease方法返回true也就是锁已经完全释放完了，那么就唤醒后继节点（唤醒离当前节点最近的可被唤醒的节点）

### acquireShared获取共享锁

共享式锁和独占锁的区别就是在同一时刻独占锁只能有一个线程能够获取锁，而共享锁在同一时刻可以有多个线程同时获取锁；

以文件的读写为例，如果一个程序在对文件进行读操作，那么这一时刻对应该文件的写操作都会被阻塞，而读操作能够同时进行；写操作要求对资源的独占式方法，而读操作可以是共享式访问



读锁和写锁是不能共存的，当前获取到读锁时，其他线程的读锁也能够获取到同步状态，但是写锁不能

当前线程获取到写锁时，其他的写锁和读锁都不能获取到同步状态

acquireShared方法获取共享式锁

通过tryAcquireShared方法获取共享式锁，tryAcquireShared方法返回的是int类型的值，如果返回的值大于0，表示获取到了同步锁

小于0时，那么就调用doAcquireShared,通过自旋来获取共享锁

**public** **final** **void** acquireShared(**int** arg) {

**if** (tryAcquireShared(arg) < 0)

doAcquireShared(arg);

}

**private** **void** doAcquireShared(**int** arg) {

**final** Node node = addWaiter(Node.*SHARED*);

**boolean** failed = **true**;

**try** {

**boolean** interrupted = **false**;

**for** (;;) {

在循环中，不断获取当前节点的前驱节点，如果前驱节点是head节点，就通过tryAcquireShared方法获取锁，

如果返回的值大于0表示获取到了锁，获取锁后调用setHeadAndPropagate方法将当前节点设置为head节点 ，并且判断后继节点的类型，决定是否释放锁锁来唤醒后继节点（如果是共享锁，那么就释放）

**final** Node p = node.predecessor();

**if** (p == head) {

**int** r = tryAcquireShared(arg);

**if** (r >= 0) {

setHeadAndPropagate(node, r);

p.next = **null**; // help GC

**if** (interrupted)

*selfInterrupt*();

failed = **false**;

**return**;

}

}

**if** (*shouldParkAfterFailedAcquire*(p, node) &&

parkAndCheckInterrupt())

interrupted = **true**;

}

} **finally** {

**if** (failed)

cancelAcquire(node);

}

}

setHeadAndPropagate方法主要工作是将node设置新的head节点，判断是否唤醒后继节点

**private** **void** setHeadAndPropagate(Node node, **int** propagate) {

Node h = head; 取出原head节点

setHead(node);//设置node为head节点

propagate是tryAcquireShared的返回值

/\*

\* propagate是tryAcquireShared的返回值，这是决定是否传播唤醒的依据之一。

\* h.waitStatus为SIGNAL或者PROPAGATE时也根据node的下一个节点共享来决定是否传播唤醒，

\* 这里为什么不能只用propagate > 0来决定是否可以传播在本文下面的思考问题中有相关讲述。

\*/

Propagate大于0表示可以获取共享锁

h.waitStatus小于0

**if** (propagate > 0 || h == **null** || h.waitStatus < 0) {

Node s = node.next;

**if** (s == **null** || s.isShared())

doReleaseShared();

}

}

/\*\*

\* 这是共享锁中的核心唤醒函数，主要做的事情就是唤醒下一个线程或者设置传播状态。

\* 后继线程被唤醒后，会尝试获取共享锁，如果成功之后，则又会调用setHeadAndPropagate,将唤醒传播下去。

\* 这个函数的作用是保障在acquire和release存在竞争的情况下，保证队列中处于等待状态的节点能够有办法被唤醒。

\*/

**private** **void** doReleaseShared() {

/\*

\* 以下的循环做的事情就是，在队列存在后继线程的情况下，唤醒后继线程；

\* 或者由于多线程同时释放共享锁由于处在中间过程，读到head节点等待状态为0的情况下，

\* 虽然不能unparkSuccessor，但为了保证唤醒能够正确稳固传递下去，设置节点状态为PROPAGATE。

\* 这样的话获取锁的线程在执行setHeadAndPropagate时可以读到PROPAGATE，从而由获取锁的线程去释放后继等待线程。

\*/

**for** (;;) {

Node h = head;

**if** (h != **null** && h != tail) {

**int** ws = h.waitStatus;

**if** (ws == Node.*SIGNAL*) {

**if** (!*compareAndSetWaitStatus*(h, Node.*SIGNAL*, 0))

**continue**; // loop to recheck cases

unparkSuccessor(h);

}

**else** **if** (ws == 0 &&

!*compareAndSetWaitStatus*(h, 0, Node.*PROPAGATE*))

**continue**; // loop on failed CAS

}

**if** (h == head) // loop if head changed

**break**;

}

}

### releaseShared释放共享锁

releaseShared释放共享锁

释放的方法主要是tryReleaseShared，如果共享锁完全释放，那么就返回true，然后唤醒后继线程

**public** **final** **boolean** releaseShared(**int** arg) {

**if** (tryReleaseShared(arg)) {

doReleaseShared();

**return** **true**;

}

**return** **false**;

}

### tryAcquireNanos定时获取独占锁

在指定时间内尝试获取独占锁，获取到锁就返回true，如果没有获取到锁，线程在指定时间内获取过程阻塞，如果超时，那么就返回false；这是synchronized关键字不具备的特性

当线程是中断状态时，抛出InterruptedException异常

调用tryAcquire方法来获取锁（也就是获取同步状态），获取到就表示已经拿到了锁直接返回true

否则就是当前已经有其他线程获取，就执行doAcquireNanos方法进入阻塞队列中

**public** **final** **boolean** tryAcquireNanos(**int** arg, **long** nanosTimeout)

**throws** InterruptedException {

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

**return** tryAcquire(arg) ||

doAcquireNanos(arg, nanosTimeout);

}

doAcquireNanos方法的整体逻辑和获取独占锁的类似，只是在循环判断的过程中加入了判断获取锁花费的时间逻辑

**private** **boolean** doAcquireNanos(**int** arg, **long** nanosTimeout)

**throws** InterruptedException {

**if** (nanosTimeout <= 0L)

**return** **false**;

得到获取锁的截止时刻

**final** **long** deadline = System.*nanoTime*() + nanosTimeout;

创建当前线程对应的node节点

**final** Node node = addWaiter(Node.*EXCLUSIVE*);

**boolean** failed = **true**;

**try** {

**for** (;;) {

循环判断节点的前置节点是否是head节点，

如果是就会尝试去获取锁

如果不是就是判断是否到达截止时间，到了结束循环；没有到就将当前线程阻塞（阻塞方式就是调用LockSupport.*parkNanos方法来将线程阻塞，阻塞时间就是获取锁过程剩余的时间*）

**final** Node p = node.predecessor();

**if** (p == head && tryAcquire(arg)) {

setHead(node);

p.next = **null**; // help GC

failed = **false**;

**return** **true**;

}

判断是否到达了截止时刻，如果到了就返回不再循环

nanosTimeout = deadline - System.*nanoTime*();

**if** (nanosTimeout <= 0L)

**return** **false**;

**if** (*shouldParkAfterFailedAcquire*(p, node) &&

nanosTimeout > *spinForTimeoutThreshold*)

LockSupport.*parkNanos*(**this**, nanosTimeout);

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

}

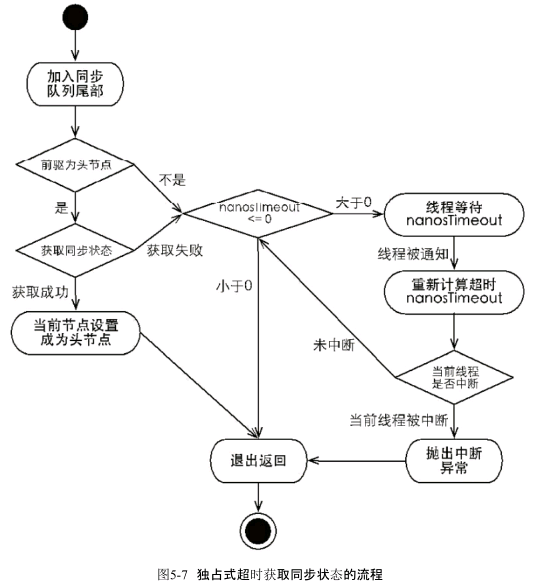
} **finally** {

**if** (failed)

cancelAcquire(node);

}

}



### tryAcquireSharedNanos定时获取共享锁

tryAcquireSharedNanos方法获取超时共享锁锁，如果超过指定时间还没有或得锁，那么就放弃获取锁

首先通过tryAcquireShared尝试获取锁，如果获取了就返回，如果获取释放就调用doAcquireSharedNanos自旋获取锁，如果自旋时间超过了设置的时间还没有获取到锁，那么就返回不在去尝试获取锁了

**public** **final** **boolean** tryAcquireSharedNanos(**int** arg, **long** nanosTimeout)

**throws** InterruptedException {

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

**return** tryAcquireShared(arg) >= 0 ||

doAcquireSharedNanos(arg, nanosTimeout);

}

AbstractQueuedSynchronizer类定义了doAcquireNanos(int arg,long nanosTimeout)方法可以在指定时间内获取同步状态，获取到了同步状态，那么就返回true，没有获取就返回false；这个方法提供了java关键字synchronized不具备的特性

一个线程获取不到锁而被阻塞在synchronized之外时，对该线程进行中断操作，此时该线程的中断标记会被修改，但是线程依然会被synchronized阻塞，等待获取锁

doAcqireNanos方法的实现

方法开始时获取当前时间System.*nanoTime*()，然后进入死循环，在指定时间内获取同步状态，如果超时还没有获取同步状态，那么就抛出InterruptedException异常

在循环内

获取当前节点的前驱节点，判断是否是head节点，如果是head那么就去获取同步状态，成功获取状态后就是将当前节点设置为head节点，然后返回，

如果前驱节点不是head节点或者获取同步状态失败，那么就判断当前是否超期

此时的时间减去最开始进入方法的时间在减去循环消耗的时间（nanosTimeout）就是当期的等待时间，如果结果小于等于0，那么就超时了返回false；如果大于等于0

等待时间在设置的时间返回内存，调用*shouldParkAfterFailedAcquire* 方法判断是否要将当前线程中断，这里多了一个等待时间大于*spinForTimeoutThreshold* （这个的值初始化为1000）的判断，因为是纳秒级别的判断，在非常短的时间内是做到精确判断，因此在非常短的情况下同步器会进入无条件的快速自旋

**private** **boolean** doAcquireNanos(**int** arg, **long** nanosTimeout)

**throws** InterruptedException {

**long** lastTime = System.*nanoTime*();

**final** Node node = addWaiter(Node.*EXCLUSIVE*);

**boolean** failed = **true**;

**try** {

**for** (;;) {

**final** Node p = node.predecessor();

**if** (p == head && tryAcquire(arg)) {

setHead(node);

p.next = **null**; // help GC

failed = **false**;

**return** **true**;

}

**if** (nanosTimeout <= 0)

**return** **false**;

**if** (*shouldParkAfterFailedAcquire*(p, node) &&

nanosTimeout > *spinForTimeoutThreshold*)

LockSupport.*parkNanos*(**this**, nanosTimeout);

**long** now = System.*nanoTime*();

nanosTimeout -= now - lastTime;

lastTime = now;

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

}

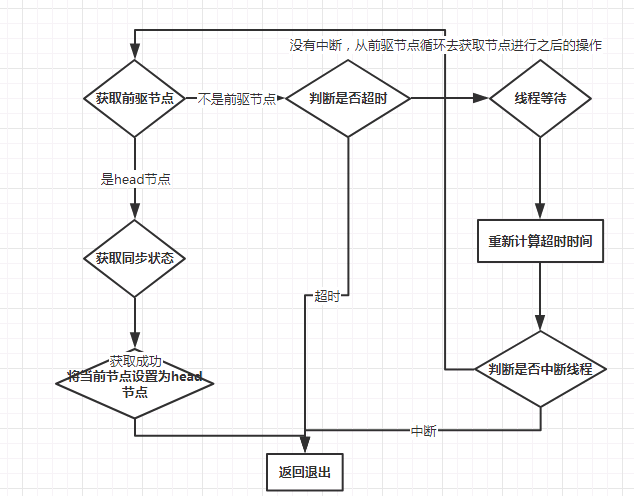
} **finally** {

**if** (failed)

cancelAcquire(node);

}

}



释放定时锁是使用释放独占锁的方式

一个简单的自定义锁

/\*\*

\*

\* 自定义锁:独占锁

\*

\*/

**public** **class** MyLock

{

**private** Sync sync = **new** Sync();

**private** **final** **static** **int** *num* = 1;

// 1.获取锁

**public** **void** lock()

{

sync.acquire(*num*);

}

// 2.释放锁

**public** **void** unlock()

{

sync.release(*num*);

}

**private** **static** **class** Sync **extends** AbstractQueuedSynchronizer

{

**private** **static** **final** **long** *serialVersionUID* = -6775343868729816577L;

// 返回true，表示获取到了锁

**public** **boolean** tryAcquire(**int** num)

{

// 状态初始值为0

**int** state = getState();

**if** (compareAndSetState(state, state + num))

{

// 成功替换，那么就获取到了锁

**return** **true**;

}

**return** **false**;

}

**public** **boolean** tryRelease(**int** num)

{

**int** state = getState();

**if** (compareAndSetState(state, state - num) && getState() == 0)

{

**return** **true**;

}

**return** **false**;

}

}

}

## LOCK

### Lock接口

锁是用来控制多个线程访问共享资源的方式，一个锁能够防止多个线程同时访问同一个共享资源

在Lock接口出现之前，java中使用synchronized来实现锁功能，java se5之后就新增lock接口来实现锁功能，Lock提供了synchronized的类似的功能，只是需要显示的实现获取锁和释放锁，虽然缺少了隐式获取和释放锁的便捷，但确拥有获取锁和释放锁的可操作性，可中断的获取锁，以及超时获取锁等synchronized不具备的功能

使用synchronized隐式的获取锁，但是他将获取锁和释放固化，也就是先获取在释放，这种方式简化了锁的管理，但是可扩展性没有显示的锁获取和释放好；

Lock提供了synchronized关键字不具备的特性

尝试非阻塞的获取锁：当前线程获取锁，如果没有被其他线程得到，则成功获取并持有锁

能被中断的获取锁：获取到锁的线程能够响应中断，当获取到锁的线程被中断时，中断异常会抛出，同时释放锁

超时获取锁：超过指定时间仍没有获取到锁，那么就返回

Lock接口提供的API

**public** **interface** Lock {

**void** lock();//获取锁，调用该方法当前线程会获取锁，当获取锁后，从该方法返回

**void** lockInterruptibly() **throws** InterruptedException;可中断的获取锁，在锁的获取过程中可以中断当前线程

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

使用如下，能够获取锁，那么就执行指定的代码段，不能的话就执行其他的代码

Lock lock = ...;

if (lock.tryLock()) {如果获取了锁

try {

\* // manipulate protected state

\*} finally {

\* lock.unlock();

\*}

\*} else {没有获取锁

\* // perform alternative actions

}

**boolean** tryLock(**long** time, TimeUnit unit) **throws** InterruptedException;指定时间内获取锁，该方法返回情况有3种：

1. 当前线程获取了锁
2. 在指定时间内没有获取到锁，返回false
3. 在指定时间内被中断返回

**void** unlock();释放锁

Condition newCondition();返回等待通知组件，该组件和当前线程绑定，只有获取到了锁才能够调用组件的wait方法

### 重入锁

ReentrantLock：支持重新进入的锁，该锁支持一个线程对资源的重复加锁，该锁还支持获取锁时的公平和非公平选择

Synchronized关键字隐式的支持锁重进入，在调用lock方法时，已经获取到锁的线程，能够再次调用lock()方法获取锁而不会阻塞

锁获取的公平性，如果在绝对时间上，先对锁进行获取的请求一定先被满足，那么这个锁是公平的，反之，是不公平的。公平的获取锁，也就是等待时间最长的线程最优先获取锁。ReentrantLock提供一个构造函数，能够控制锁是否是公平的；

公平锁机制往往没有非公平锁的效率高，但是公平锁能够减少饥饿发送的概率，等待越久的线程越能够得到满足

重进入：指任意线程获取到锁后能够在该线程中再次获取该锁而不会被锁阻塞

非公平锁的重入实现

当线程首次获取锁后，将当前线程变量保存下来；如果有线程再次调用lock方法获取锁，会判断当前线程是否和已经获取锁的线程进行比较，如果相等，那么该线程已经获取过了锁，则将同步状态值进行递增并且返回true；

ReentrantLock的实现

内部结构

内部类Sync：继承AQS，提供同步控制锁的基础

FairSync：继承Sync，提供公平锁

NonfairSync：继承Sync，提供非公平锁

无参构造函数，使用非公平锁

**public** ReentrantLock() {

sync = **new** NonfairSync();

}

**public** ReentrantLock(**boolean** fair) {

sync = fair ? **new** FairSync() : **new** NonfairSync();

}

非公平锁的加锁过程

Lock

java.util.concurrent.locks.ReentrantLock.NonfairSync.lock()

首先通过CAS判断当前同步状态是否为0，为0，那么就替换为1，操作能够成本，那么就表示当前线程获取到了锁，就将当前线程赋值给exclusiveOwnerThread这个引用，

如果操作不能成功，那么就调用AQS的acquire方法来获取锁

**final** **void** lock() {

**if** (compareAndSetState(0, 1))

setExclusiveOwnerThread(Thread.*currentThread*());

**else**

acquire(1);

}

NonfairSync重写AQS的方法tryAcquire，在方法中执行父类sync的nonfairTryAcquire方法，这个就是获取非公平锁的方法

获取非公平锁过程

取出当前同步状态当状态为0时，使用CAS设置指定的同步状态，成功那么就表示获取到了锁

状态不为0 ，或者替换失败就表示有其他线程获取到了锁，判断当前线程和获取到锁的线程是否是同一个线程，如果是同一个线程那么就表示重入，设置新的状态值，然后返回true

如果不是同一个线程，那么就返回false，然后执行acquireQueued方法

**final** **boolean** nonfairTryAcquire(**int** acquires) {

**final** Thread current = Thread.*currentThread*();

**int** c = getState();

**if** (c == 0) {

**if** (compareAndSetState(0, acquires)) {

状态为0时首次获取锁，获取后将线程保存下来

setExclusiveOwnerThread(current);

**return** **true**;

}

}

和当前线程进行比较

**else** **if** (current == getExclusiveOwnerThread()) {

**int** nextc = c + acquires;

**if** (nextc < 0) // overflow

**throw** **new** Error("Maximum lock count exceeded");

setState(nextc);

**return** **true**;

}

**return** **false**;

}

释放锁过程

调用tryRelease方法释放锁，让当前状态减去释放状态，如果相减的结果为0，那么锁就完全释放了，就释放独占锁返回true，否则就是锁还没有释放完，就返回false

当tryRelease返回true后，获取head节点，如果head节点不为空并且状态不是无状态，那么就通过unparkSuccessor方法来唤醒后继待获取锁的线程

unlock—》release

**public** **final** **boolean** release(**int** arg) {

**if** (tryRelease(arg)) {

Node h = head;

**if** (h != **null** && h.waitStatus != 0)

unparkSuccessor(h);

**return** **true**;

}

**return** **false**;

}

**protected** **final** **boolean** tryRelease(**int** releases) {

**int** c = getState() - releases;

**if** (Thread.*currentThread*() != getExclusiveOwnerThread())

**throw** **new** IllegalMonitorStateException();

**boolean** free = **false**;

**if** (c == 0) {

free = **true**;

setExclusiveOwnerThread(**null**);将独占锁设置为null

}

setState(c);设置释放后的状态

**return** free;

}

公平锁和非公平锁获取的区别

公平性是针对获取锁而言的，如果一个锁是公平的，那么锁的获取顺序就应该符合请求的绝对时间顺序，也就是FIFO;

这是获取公平锁的方法，和获取非公平锁相比只多了一个方法hasQueuedPredecessors的判断，判断加入当前同步队列的节点是否有前驱节点，如果返回true则表示有线程比当前线程更早得到请求获取锁，因此需要等待前驱节点获取并释放锁后才能获取锁

**protected** **final** **boolean** tryAcquire(**int** acquires) {

**final** Thread current = Thread.*currentThread*();

**int** c = getState();

**if** (c == 0) {

**if** (!hasQueuedPredecessors() &&

compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

**return** **true**;

}

}

**else** **if** (current == getExclusiveOwnerThread()) {

**int** nextc = c + acquires;

**if** (nextc < 0)

**throw** **new** Error("Maximum lock count exceeded");

setState(nextc);

**return** **true**;

}

**return** **false**;

}

**public** **final** **boolean** hasQueuedPredecessors() {

Node t = tail; // Read fields in reverse initialization order

Node h = head;

Node s;

**return** h != t &&

((s = h.next) == **null** || s.thread != Thread.*currentThread*());

}

公平锁每次都是从同步队列中取第一个节点获取到锁，而非公平锁出现了一个线程连续获取锁的情况；

非公平锁可能使线程处于饥饿状态，但是仍然让其作为重入锁的默认实现，是因为非公平锁的开销比公平锁小，公平锁保证了锁的获取按照FIFO原则，而代价就是进行大量的线程切换，非公平锁可能造成线程饥饿，但是极少的线程切换保证了较大的吞吐量

非公平锁获取

一个线程获取公平锁的地方有2处

1. 通过tryAcquire方法获取
2. 通过tryAcquire方法没有获取，记入自旋队列中获取

当一个线程释放了锁后，会通知后继节点来获取锁，同时也存在一个新的线程通过1方式来获取锁，当新线程获取锁后队列中的线程就会继续等待，这个就造成了获取锁的非公平性

公平获取锁

获取锁的方式也是2中，和非公平的是一样的，只是在通过1获取锁的时候，如果已经有等待队列了，那么就只能将线程放到等待队列中，按队列顺序来获取锁

Condition等待、通知

ReentrantLock r = **new** ReentrantLock();

ConditionObject condition = (ConditionObject) r.newCondition();

condition.await();

// 将当前线程封装到node实例中，形成node链表，然后阻塞线程LockSupport.park(this);

condition.signal();// 唤醒first节点 LockSupport.unpark(node.thread);

condition.signalAll();// 唤醒所有的节点 遍历链表的节点LockSupport.unpark(node.thread);

Synchronized和ReentrantLock

 ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似轮询锁、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。

*reentrant* 锁意味着什么呢？简单来说，它有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加1，然后锁需要被释放两次才能获得真正释放。这模仿了 synchronized 的语义；如果线程进入由线程已经拥有的监控器保护的 synchronized 块，就允许线程继续进行，当线程退出第二个（或者后续） synchronized 块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个 synchronized 块时，才释放锁。

 Lock 和 synchronized 有一点明显的区别 —— lock 必须在 finally 块中释放。否则，如果受保护的代码将抛出异常，锁就有可能永远得不到释放

高并发环境下Lock的在cpu调度花费的时间较少，更有效的利用cpu

Lock提供了公平锁和非公平锁2中选择

Synchronized只有非公平锁

既然如此，我们什么时候才应该使用 ReentrantLock 呢？答案非常简单 —— 在确实需要一些 synchronized 所没有的特性的时候，比如时间锁等候、可中断锁等候、无块结构锁、多个条件变量或者轮询锁。 ReentrantLock 还具有可伸缩性的好处，应当在高度争用的情况下使用它，但是请记住，大多数 synchronized 块几乎从来没有出现过争用，所以可以把高度争用放在一边。我建议用 synchronized 开发，直到确实证明 synchronized 不合适，而不要仅仅是假设如果使用 ReentrantLock “性能会更好”。请记住，这些是供高级用户使用的高级工具

### 读写锁

ReentrantLock是排他锁，但是这个锁只允许在同一时刻只会有一个线程持有锁，其他线程必须处于等待状态；

读锁允许多个读线程访问，但是在写线程访问的时候，所有的读线程和写线程均被阻塞；

读写锁是一对锁：读锁和写锁，通过分离读锁和写锁，使得性能相比其他的排他锁有了很大的提升

写锁是独占锁，读锁是共享锁，在同一个线程中读写锁是可以相互重入的

Java并发包中提供的读写锁：ReentrantReadWriteLock

特性

公平性：支持非公平(默认)和公平的获取锁方式，性能上还是非公平的优于公平

重进入：支持重进入，读锁在获取读锁后能够再次获取读锁，写锁获取后能够再次获取写锁，同时也能获取读锁

锁降级：遵循获取写锁，获取读锁再次释放写锁的次序，写锁能够降级成读锁

读写锁的一个使用场景，在使用hashmap时，创建读锁和写锁，在调用HashMap的get方法是使用读锁来保证线程安全，在调用set方法时调用写锁来保证线程安全

ReadWriteLock接口：这个接口只定义了获取读锁和写锁的两个方法

**public** **interface** ReadWriteLock {

Lock readLock();

Lock writeLock();

}

ReentrantReadWriteLock实现了ReadWriteLock接口

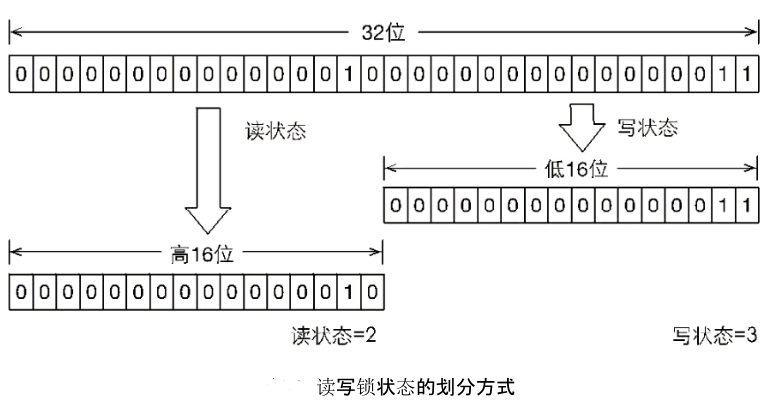
读写锁的设计和实现

读写状态的设计

读写锁同一依赖于自定义同步器（AbstractQueuedSynchronizer AQS）来实现同步功能,而且读写状态就是其同步器状态

读写锁的自定义同步器需要在一个同步状态（一个整型变量）上维护多个读锁和一个写锁

在一个整型变量上维护多种状态，就一定要按位切割使用这个变量，读写锁将变量切分成两个部分，高16位是读，低16位是写



**读写锁判断各自的规则**：

当前状态为s

获取写状态：将s和0X0000FFFF（高16为0，低16位为1）进行与运算，获取的结果就是写状态；

写状态加+1：就是状态值直接+1

获取读状态：将s无符号向右位移16位（s>>>16）结果就是读状态

读状态加+1：就是将1向左位移16位后结果和s相加

读锁的获取和释放

读锁是一个支持重进入的共享锁，它能够被多个线程同时获取，在没有被写锁访问的时候，任何一个获取读锁的线程都会成功的获取读锁，写锁已经被其他线程获取，则进入等待状态；

读锁的每次释放(每次都是线程安全的，可能有多个线程同时释放读锁)都是减少读状态，减少的值是（1<<16）

实现减少读状态的方式是：通过死循环不断的将当前状态和内存中的状态进行比较替换操作

ReentrantReadWriteLock的构造函数

**public** ReentrantReadWriteLock() {

默认创建非公平锁

**this**(**false**);

}

**public** ReentrantReadWriteLock(**boolean** fair) {

sync = fair ? **new** FairSync() : **new** NonfairSync();

readerLock = **new** ReadLock(**this**);

writerLock = **new** WriteLock(**this**);

}

内部类

Sync继承AQS

FairSync：公平锁，继承Sync

NonfairSync：非公平锁，继承Sync

ReadLock：读锁

WriteLock：写锁

非公平读锁

在sync中重写tryAcquireShared方法来获取锁

java.util.concurrent.locks.ReentrantReadWriteLock.Sync.tryAcquireShared(int)

**public** **void** lock() {

sync.acquireShared(1);

}

首先调用tryAcquireShared尝试获取锁，获取到锁也就是tryAcquireShared返回值大于0，方法直接返回

如果没有获取到锁，也就是有其他线程获取了锁，那么就调用doAcquireShared方法来获取共享锁

**public** **final** **void** acquireShared(**int** arg) {

**if** (tryAcquireShared(arg) < 0)

doAcquireShared(arg);

}

**protected** **final** **int** tryAcquireShared(**int** unused) {

Thread current = Thread.*currentThread*();

**int** c = getState();

exclusiveCount方法获取写锁的数量，判断是否有写锁，

如果是写锁，判断当前线程是否是已经获取的锁的线程，如果没有直接返回

**if** (*exclusiveCount*(c) != 0 &&

getExclusiveOwnerThread() != current)

**return** -1;

readerShouldBlock方法获取读锁的数量，

**int** r = *sharedCount*(c);

读线程不会阻塞，当前读锁的数量小于最大值，使用CAS常规将状态修改

**if** (!readerShouldBlock() &&

r < *MAX\_COUNT* &&

compareAndSetState(c, c + *SHARED\_UNIT*)) {

if条件都满足时，次数已经获取到了读锁，

下面这些就是用来记录获取读锁次数的操作

**if** (r == 0) {

firstReader = current;

firstReaderHoldCount = 1;

} **else** **if** (firstReader == current) {

firstReaderHoldCount++;

} **else** {

HoldCounter rh = cachedHoldCounter;

**if** (rh == **null** || rh.tid != *getThreadId*(current))

cachedHoldCounter = rh = readHolds.get();

**else** **if** (rh.count == 0)

readHolds.set(rh);

rh.count++;

}

**return** 1;

}

如果之前没有获取到读锁，那么就调用fullTryAcquireShared方法来不断循环尝试获取

**return** fullTryAcquireShared(current);

}

读锁的释放

调用releaseShared方法，首先释放当前线程的状态，然后唤醒后继线程

**public** **final** **boolean** releaseShared(**int** arg) {

**if** (tryReleaseShared(arg)) {

doReleaseShared();

**return** **true**;

}

**return** **false**;

}

释放状态：首先修改当前线程获取到的读锁次数的记录，然后修改锁状态

**protected** **final** **boolean** tryReleaseShared(**int** unused) {

Thread current = Thread.*currentThread*();

**if** (firstReader == current) {

// assert firstReaderHoldCount > 0;

**if** (firstReaderHoldCount == 1)

firstReader = **null**;

**else**

firstReaderHoldCount--;

} **else** {

HoldCounter rh = cachedHoldCounter;

**if** (rh == **null** || rh.tid != *getThreadId*(current))

rh = readHolds.get();

**int** count = rh.count;

**if** (count <= 1) {

readHolds.remove();

**if** (count <= 0)

**throw** unmatchedUnlockException();

}

--rh.count;

}

上面的是修改获取读锁的次数

下面就是修改同步状态

**for** (;;) {

**int** c = getState();

**int** nextc = c - *SHARED\_UNIT*;

**if** (compareAndSetState(c, nextc))

**return** nextc == 0;//当读锁释放完后返回true

}

}

读锁的释放均减少读状态，减少的值为1<<16(0000 0000 0000 0001 0000 0000 0000 0000)

写锁的获取和释放

写锁是一个支持重进的排他锁，如果当前线程已经获取了写锁，就增加写状态

如果当前线程在获取写锁时，读锁已经获取了或者该线程不是已经获取写锁的线程，则当前进入等待状态（也就是在获取读锁时，不能够获取写锁，必须等待读锁释放）

如果存在读锁就不能获取写锁，因为读写锁要保证写锁的操作对读锁可见，只有等读锁全部释放了，写锁才能被当前线程获取，而写锁一旦被获取，则其他读写锁就不能在后续的访问中均被阻塞

ReentrantReadWriteLock

**protected** **final** **boolean** tryAcquire(**int** acquires) {

Thread current = Thread.*currentThread*();

**int** c = getState();

**int** w = *exclusiveCount*(c);

**if** (c != 0) {

// (Note: if c != 0 and w == 0 then shared count != 0)

同步状态不为0，

如果1：写锁状态为0，那么就是存在读锁，那么就不能获取写锁

如果1：写锁状态不为0，但是但前线程不是已经获取写锁的线程（也就是有其他线程获取了写锁），也不能获取写锁

**if** (w == 0 || current != getExclusiveOwnerThread())

**return** **false**;

写锁的数量超过了最大，就抛出异常

**if** (w + *exclusiveCount*(acquires) > *MAX\_COUNT*)

**throw** **new** Error("Maximum lock count exceeded");

// Reentrant acquire

setState(c + acquires);//设置锁状态

**return** **true**;

}

**if** (writerShouldBlock() ||

!compareAndSetState(c, c + acquires))

**return** **false**;

setExclusiveOwnerThread(current);

**return** **true**;

}

exclusiveCount方法返回写状态的方法（当前状态和65535进行与运算）如果结果为0，表还没有获取锁状态

static int exclusiveCount(int c) {

return c & EXCLUSIVE\_MASK;

}

写锁的释放和ReentrantLock的释放过程基本类似，每次释放均减少写状态，当写状态为0时，表示写已被释放，从而等待的线程就能够继续访问读写锁，

锁降级

锁降级是指写锁降级为读锁；如果一个写锁释放后重新去获取读锁这个过程不是降级过程；

锁降级是指持有的写锁时，去获取读锁，随后释放写锁的过程(也就是持有写锁的同时，获取读锁，然后将写锁释放的过程)

当前线程持有写锁，在同步代码块中要获取读锁，然后释放写锁，这样写锁就降价为读锁了

锁降级过程中读锁的获取是否必要：是必要的，主要是为了保证数据的可见性，如果当前线程不获取读锁而直接释放写锁，而有另外一个线程T获取了写锁，并对数据进行了修改，那么当前线程就无法感知线程T的数据更新；如果当前线程遵循锁降级的步骤，则线程T就会被阻塞，直到当前线程使用的数据并释放读锁之后，线程T才能获取写锁进行数据修改

ReentrantReadWriteLock不支持锁升级，目的是保证数据的可见性；保证各个线程看到的数据都是一致的

**public** **class** CachedData

{

Object data;

**volatile** **boolean** cacheValid;

**final** ReentrantReadWriteLock rwl = **new** ReentrantReadWriteLock();

**void** processCachedData()

{

rwl.readLock().lock();

**if** (!cacheValid)

{

// Must release read lock before acquiring write lock

//必须要释放之前的读锁

rwl.readLock().unlock();

// 锁降级从获取到写锁开始

rwl.writeLock().lock();

**try**

{

// Recheck state because another thread might have

// acquired write lock and changed state before we did.

**if** (!cacheValid)

{

data = "hello";

cacheValid = **true**;

}

// Downgrade by acquiring read lock before releasing write lock

// 重入获取读锁

rwl.readLock().lock();

} **finally**

{

// 释放写锁，释放完成后就完成了锁降级，写锁降级为读锁，因为上面修改了的数据在写锁释放之后还要使用（use方法），所以使用读锁

// 来阻塞其他线程修改数据，因为此时其他线程修改了的数据当前是无法感知的，需要用读锁来阻塞其他写线程

rwl.writeLock().unlock(); // Unlock write, still hold read

}

}

**try**

{

use(data);

} **finally**

{

rwl.readLock().unlock();

}

}

**private** **void** use(Object data2)

{

System.*out*.println("-----" + data2);

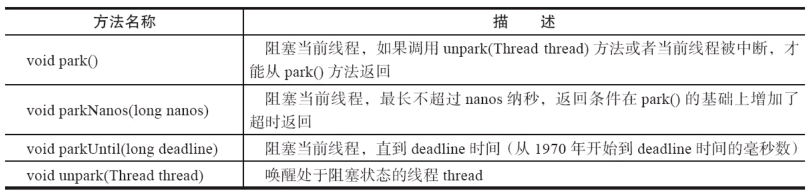
}

}

LockSupport

LockSupport定义了一组公共的静态方法，提供了最基本的线程阻塞和唤醒功能；

定义了一组以park开头的方法用来阻塞当前线程，以unpark开头的方法来唤醒一个被阻塞的线程。



## 并发工具

### CountDownLatch

功能：让主线程等待所有子线程执行完成后才继续执行主线程，最简单的方法就是使用join

当前线程等待join线程执行结束，其原理就是不停的检查join线程是否存活，如果存活那么就阻塞当前线程，如果join线程执行完成后，就让当前线程执行；join方法也可以指定主线程阻塞时间，如果join线程的执行超过了指定时间，会唤醒当前线程

主线程等待所有子线程执行完成后才能进行执行，最简单的做法是使用join方法，

Join方法用于让当前线程等待join线程执行结束，其实现原理是不停检查join线程是否存活，如果线程存活则让当前线程永远等待；

Join方法源码

在join方法中执行join(millis)方法（重载方法），参数为0，表示要一直等待

当参数为0是，循环检查当前线程是否存活，如果存活调用wait方法一致等待

当指定了等待时间，那么就在判断线程存活的同时判断当前时间是否是超时，如果线程存活且超时那么就返回

**public** **final** **void** join() **throws** InterruptedException {

join(0);

}

**public** **final** **synchronized** **void** join(**long** millis)

**throws** InterruptedException {

**long** base = System.*currentTimeMillis*();

**long** now = 0;

**if** (millis < 0) {

**throw** **new** IllegalArgumentException("timeout value is negative");

}

**if** (millis == 0) {

循环判断当前线程是否存活，存活的话就让主线程等待

**while** (isAlive()) {

wait(0);暂停时间为0

}

} **else** {

循环判断当前线程是否存储，如果存活，并且等待的时间超过理论指定时间，那么就结束循环判断直接返回，没有超过就继续等待

**while** (isAlive()) {

**long** delay = millis - now;

**if** (delay <= 0) {

**break**;

}

wait(delay);

now = System.*currentTimeMillis*() - base;

}

}

}

CountDownLatch提供了join的功能，并且功能更多

CountDownLatch的构造函数，构造函数带有一个int类型的参数来作为计数器，比如需要等待N个线程，那么就传入N;

构造函数（构造函数参数必须大于等于0，如果设置的线程数量超过了实际的线程数量（并且没有设置等待时间），那么主线程就会一直等待）

**public** CountDownLatch(**int** count) {

**if** (count < 0) **throw** **new** IllegalArgumentException("count < 0");

**this**.sync = **new** Sync(count);

}

调用await方法来阻塞线程，首先判断同步状态是否为0，不为0，那么就阻塞当前线程，调用countDown方法，将状态值-1

当值为0 时就唤醒所有被阻塞的线程

当调用await方法时会阻塞当前线程，当调用countDown方法时，计数器就减1，当数量减为0时，当前线程就会被唤醒；

计数器的值必须大于0，只是等于0时，调用await方法是不会阻塞当前线程的，

await方法还有一个带有时间的重载方法await(long time,TimeUtil unit),这个方法在等待指定的时间后就不会再阻塞线程

内部定义Sync内部类来实现主线程的等待

Sync类：继承AbstractQueuedSynchronizer类

构造函数初始化时，设置同步状态，值就是CountDownLatch构造函数设置的等待线程数量

锁是共享锁，获取共享锁的判断就是当前共享状态是否为0，如果那么就不需要获取锁了（等待的线程数量为0）（这里的锁主要是锁住主线程）

在Sync中重写tryAcquireShared方法，来实线锁的获取

重写tryReleaseShared原子方式实现状态的-1，直到同步状态变为0

**private** **static** **final** **class** Sync **extends** AbstractQueuedSynchronizer {

**private** **static** **final** **long** *serialVersionUID* = 4982264981922014374L;

Sync(**int** count) {

setState(count);

}

**int** getCount() {

**return** getState();

}

同步状态等于0，表示获取到了锁，所以返回一个大于0的值，默认为1，同步状态不等于0，返回一个小于0的值，默认-1，表示没有获取到锁进入阻塞中

**protected** **int** tryAcquireShared(**int** acquires) {

**return** (getState() == 0) ? 1 : -1;

}

**protected** **boolean** tryReleaseShared(**int** releases) {

// Decrement count; signal when transition to zero

**for** (;;) {

**int** c = getState();

**if** (c == 0)

**return** **false**;

**int** nextc = c-1;

**if** (compareAndSetState(c, nextc))

**return** nextc == 0;

}

}

}

countDownLatch的await方法会阻塞线程，直到N变为0才会唤醒主线程

awai方法有2个，

无参表示一直阻塞，直到线程都执行完成

**public** **void** await() **throws** InterruptedException {

sync.acquireSharedInterruptibly(1);

}

参数是时间长度和时间单位，表示阻塞指定时间

**public** **boolean** await(**long** timeout, TimeUnit unit)

**throws** InterruptedException {

**return** sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));

}

调用countDown方法，N就会-1，用到多线程时，只需要把这个countdownLatch的引用传递到线程里即可

**public** **void** countDown() {

sync.releaseShared(1);

}

在CountDownLatch中await方法就是用来获取锁，当计数器值为0的时候才能够获取到锁，countDown方法就是用来释放锁的

### CyclicBarrier

让一组线程到达一个屏障时被阻塞，直到最后的一个线程到达屏障时，屏障才会开放，所有被拦截在屏障的线程才会继续执行

构造函数

带整数类型参数的构造函数，参数表示屏障拦截的线程数

**public** CyclicBarrier(**int** parties) {

**this**(parties, **null**);

}

带有整数类型和Runnable类型的参数，除了表示屏障拦截的线程数之外，第二个参数表示优先执行的线程

**public** CyclicBarrier(**int** parties, Runnable barrierAction) {

**if** (parties <= 0) **throw** **new** IllegalArgumentException();

**this**.parties = parties;

**this**.count = parties;

**this**.barrierCommand = barrierAction;

}

设置屏障方法（阻塞线程），调用一次await()方法，设置的线程数量不为0，那么就进入等待，并且线程数量的值-1，当这个值减为0时就会唤醒所有等待的线程

内部字段属性

同步锁

**private** **final** ReentrantLock lock = **new** ReentrantLock();

线程拦截器，用来阻塞线程（通过重入锁的条件队列来阻塞线程）

**private** **final** Condition trip = lock.newCondition();

拦截的线程数

**private** **final** **int** parties;

优先执行的线程

**private** **final** Runnable barrierCommand;

表示屏障的当前状态，内置一个Boolean类型的参数，表示当前屏障是否可用

**private** Generation generation = **new** Generation();

计数器，初始值和parties相同，当调用await方法后值减1，直到为0时唤醒所有的线程

**private** **int** count;

await内存调用dowait方法

**public** **int** await() **throws** InterruptedException, BrokenBarrierException {

**try** {

**return** dowait(**false**, 0L);

} **catch** (TimeoutException toe) {

**throw** **new** Error(toe); // cannot happen;

}

}

dowait方法

这个方法通过重入锁ReentrantLock来保证多线程操作的安全性

1.取出当前屏障状态，检测状态，如果状态为true，表示屏障被破坏，抛出异常

2.检测当前线程是否已经被中断了，被中断，那么就破坏当前屏障，恢复所有的阻塞线程，抛出异常

3.将计数器-1，如果减一后的结果为0，那么表示所有的线程都到达了屏障，如果存在有限执行的线程，就先执行这个线程，然后就唤醒所有的阻塞线程.当成功唤醒所有线程后，调用nextGeneration方法重置当前屏障，让其可以重新使用

4.如果计数器不为0，那么就阻塞当前线程

通过ReentrantLock和它的等待通知组件condition来实现的，同count来表示阻塞的线程数（在构造时传入值），调用await方法，在内部首先让

Count-1，如果为0 ，那么就表示已经到达屏障，就唤醒阻塞的线程（trip.signalAll();唤醒所有等待的线程），如果没有就同condition的await方法来阻塞当前线程，当值为0时就调用

SingleAll方法唤醒所有的线程

当最后一个线程到达屏障时，计数器-1，此时计数器只为0，判读是否有优先执行的任务，如果有就执行这个任务；然后调用nextGeneration方法唤醒其他线程

当线程没有全部到达屏障时，计数器-1，计数器大于0，进入死循环中，使用ReentrantLock的等待通知组件阻塞线程（trip.await();），等待最后一个线程到达屏障后，唤醒线程，结束循环

**private** **int** dowait(**boolean** timed, **long** nanos)

**throws** InterruptedException, BrokenBarrierException,

TimeoutException {

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**final** Generation g = generation;

**if** (g.broken)

**throw** **new** BrokenBarrierException();

**if** (Thread.*interrupted*()) {

breakBarrier();

**throw** **new** InterruptedException();

}

**int** index = --count;

**if** (index == 0) { // tripped

**boolean** ranAction = **false**;

**try** {

**final** Runnable command = barrierCommand;

**if** (command != **null**)

command.run();

ranAction = **true**;

nextGeneration();

**return** 0;

} **finally** {

**if** (!ranAction)

breakBarrier();

}

}

// loop until tripped, broken, interrupted, or timed out

**for** (;;) {

**try** {

**if** (!timed)

trip.await();

**else** **if** (nanos > 0L)

nanos = trip.awaitNanos(nanos);

} **catch** (InterruptedException ie) {

**if** (g == generation && ! g.broken) {

breakBarrier();

**throw** ie;

} **else** {

// We're about to finish waiting even if we had not

// been interrupted, so this interrupt is deemed to

// "belong" to subsequent execution.

Thread.*currentThread*().interrupt();

}

}

**if** (g.broken)

**throw** **new** BrokenBarrierException();

当最后一个线程到达屏障时，调用nextGeneration方法，就生成了一个新的generation对象，所以g！= generation，也就是这个时候结束循环

**if** (g != generation)

**return** index;

**if** (timed && nanos <= 0L) {

breakBarrier();

**throw** **new** TimeoutException();

}

}

} **finally** {

lock.unlock();

}

}

breakBarrier方法破壳当前屏障

主要操作是重置generation状态

重置计数器

唤醒所有的线程

**private** **void** breakBarrier() {

generation.broken = **true**;

count = parties;

trip.signalAll();

}

让屏障可以再次使用，唤醒等待的线程，重置计数器，设置可用标记

**private** **void** nextGeneration() {

trip.signalAll();

count = parties;

generation = **new** Generation();

}

我们设想一下，如果初始化时，指定了线程 parties = 4，前面有 3 个线程调用了 await 等待，在第 4 个线程调用 await 之前，我们调用 reset 方法，那么会发生什么？

首先，打破栅栏，那意味着所有等待的线程（3个等待的线程）会唤醒，await 方法会通过抛出 BrokenBarrierException 异常返回。然后开启新的一代，重置了 count 和 generation，相当于一切归零了。

当线程运行完成后可以调用reset方法，让屏障数据状态重置，那么就可以重新使用了

**public** **void** reset() {

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

breakBarrier(); // break the current generation

nextGeneration(); // start a new generation

} **finally** {

lock.unlock();

}

}

应用场景：用于多线程计算数据，最后合并计算结果的场景

CyclicBarrier和CountDownLatch的区别

CountDownLatch计数器只能使用一次，而CyclicBarrier可以使用reset()方法重置，所以CyclicBarrier能够处理更为复杂的业务场景；比如计算错误，可重置计数器，并且重新执行

### Semaphore

信号量：用来控制访问特定资源的线程数量，它通过协调各个线程，以及保证合理的使用公共资源

比如当有几十个线程在读取文件，读取导数据后需要将数据保存到数据库中，但是连接池中只有10个连接，这是就必须控制只有10线程能够获取到数据库连接保存的数据，否则会报无法获取连接，这个时候就可以用Semaphore来控制流量

Semaphore控制并发执行过程

在构造函数中指定并行运行线程数量，

用法也简单

使用acquire获取一个许可，线程获取许可后执行完成线程，调用release归还许可

### Exchanger

Exchanger使用于线程间协作的工具类，用于进行线程间的数据交换，它提供了一个同步点，两个线程可以交换彼此的数据；两个线程通过exchange方法进行数据交换，如果第一个线程先执行exchange()方法，它会一直等待第二个线程也执行exchange方法，当两个线程都到达同步点时，这两个线程可以进行数据交换，将本线程生成的数据传递给对方

## 线程池

线程池好处

1. 降低资源消耗：重复利用创建的线程降低线程创建和销毁带来的性能消耗
2. 提高响应速度：当任务到达时，任务不需要等待创建线程就能被执行
3. 提高线程的可管理性：线程是稀缺资源，如果无限制的创建线程不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配，调优，监控

线程上下文切换：单核cpu通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到另外一个任务的时间片；切换之前会保存上一个任务的状态，以便下一次切换回这个任务时，可以在加载这个任务的状态；从任务保存到再加载任务的过程称为一次上下文切换；

通过线程来并发执行任务并不一定会比单线程运行效率高，因为线程的创建和上下文切换都会带来性能的消耗。

如何减少上下文：无锁并发编程，CAS算法，尽量少的使用线程和使用协程

无锁并发编程：多线程竞争锁时，会引起上下文切换

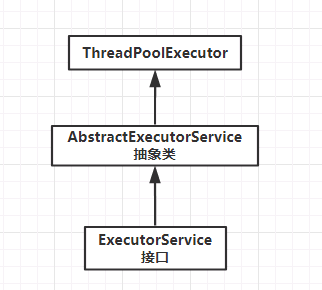
CAS算法：使用CAS算法来更新数据，不需要加锁

使用最少线程：避免创建不必要的线程

协程：在单线程里面实现多任务的调度，并在单线程中维持多个任务的切换

ThreadPoolExecutor

类继承体系



构造函数参数

**int corePoolSize**,核心线程数量，当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，当执行任务数量大于核心线程数量时，就不用在创建；当线程池调用prestartAllCoreThreads（）方法时，线程会提前创建并开启所有的核心线程

**int** **maximumPoolSize**,工作线程，非核心线程数，线程池中最大允许的线程数量 ，线程会定期被回收（当任务队列满了的时候，继续添加任务，这时就创建工作线程来执行新来的任务）

**long keepAliveTime**,线程空闲时等待的时间，如果等待超过这个时间线程就会被终止

**TimeUnit unit,**时间单位

**BlockingQueue<Runnable> workQueue**：任务队列（队列是阻塞的，能够保证取出和添加的线程安全性）

常用队列

ArrayBlockingQueue:

LinkedBlockingQueue:

SynchronousQueue:

PriorityBlocingQueue:

**ThreadFactory threadFactory,**线程工厂，创建工作线程；DefaultThreadFactory默认线程工厂

**RejectedExecutionHandler handler**：拒绝策略（多余的任务处理器）（默认是抛出异常）

其他策略

DiscardOldestPolicy丢弃队列最近的一个任务，并执行当前队列

DiscardPolicy不处理，直接丢弃

如果线程池中的线程数量大于 corePoolSize 时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；

如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过 keepAliveTime，线程也会被终止。

Int类型有32为，使用前3为来表示线程状态，后29为来表示线程容量

Ctl用来记录线程池状态和线程数量（总共32位，前3位表示线程状态，后29位表示线程数量）

**private** **final** AtomicInteger ctl = **new** AtomicInteger(*ctlOf*(*RUNNING*, 0));

位数，值为29，Integer.size的值为32

**private** **static** **final** **int** *COUNT\_BITS* = Integer.*SIZE* - 3;

容量：工作线程容量，

1向左位移29位减一

0000 0000 0000 0000 0000 0000 0000 0001

0010 0000 0000 0000 0000 0000 0000 0000 向左位移29为

0001 1111 1111 1111 1111 1111 1111 1111 位移后减一

**private** **static** **final** **int** *CAPACITY* = (1 << *COUNT\_BITS*) - 1;

1111 1111 1111 1111 1111 1111 1111 1111 -1的二进制

1110 0000 0000 0000 0000 0000 0000 0000 向左移动29位的结果，前3为表示状态

**int** RUNNING = -1 << COUNT\_BITS;可以接收新任务，执行队列任务，为负数

0000 0000 0000 0000 0000 0000 0000 0000

**int** *SHUTDOWN* = 0 << *COUNT\_BITS*; 不可用接收新任务，但是可以执行队列中还没有执行的任务

0000 0000 0000 0000 0000 0000 0000 0001 1的二进制

0010 0000 0000 0000 0000 0000 0000 0000 向左移动29位

**int** STOP = 1 << COUNT\_BITS;不可用接收新任务，也不可执行队列中还没有完成的任务

0000 0000 0000 0000 0000 0000 0000 0010 2的二进制

0100 0000 0000 0000 0000 0000 0000 0000 向左移动29位

**int** TIDYING = 2 << COUNT\_BITS;所有线程执行结束，工作线程数量为0；调用terminated()方法，方法执行过程中就是TIDYING状态

0000 0000 0000 0000 0000 0000 0000 0011 3的二进制

0110 0000 0000 0000 0000 0000 0000 0000 向左移动29位

**int** TERMINATED = 3 << COUNT\_BITS; terminated()方法执行完成后就是当前状态

线程之间状态的转换

RUNNING🡪SHUTDOWN：调用shutdown()方法,或者隐式的调用了finalize方法

RUNNING OR SHUTDOWN 🡪STOP：调用shutdownNow（）方法

SHUTDOWN 🡪TIDYING ：当线程池和任务都为空的时候

STOP🡪TIDYING：当线程池为空的时候

TIDYING🡪TERMINATED 当terminated()完成后

RUNNING -> SHUTDOWN

On invocation of shutdown(), perhaps implicitly in finalize()

(RUNNING or SHUTDOWN) -> STOP

On invocation of shutdownNow()

SHUTDOWN -> TIDYING

When both queue and pool are empty(任务队列和线程池都为空)

STOP -> TIDYING

When pool is empty(线程池为空)

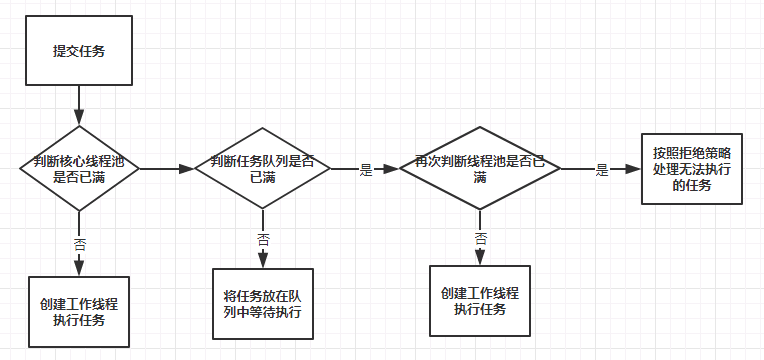
TIDYING -> TERMINATED

When the terminated() hook method has completed

创建线程池后，调用execute来执行任务

execute方法，方法的参数是实现Runnable接口的实例对象

流程图



判断线程池中工作线程数量是否达到了maximumpoolsize的值

这个方法的流程分为3种情况

1. 当少于核心线程池（corePoolSize）数量的线程池在运行时，开启一个新的线程让传入的command作为第一个任务运行；实现过程在addWorker方法中:首先获取运行状态，如果当前不是可运行任务状态，那么就返回false，也即是当前不能添加新任务；
2. 核心线程满了之后，当前是运行状态，将任务放入任务队列中，放到队列中后再次校验当前是否是运行状态；如果当前线程池不是RUNNING状态，那么就从队列中删除任务，并且执行拒绝策略；如果当前线程池中工作线程为空，那么就新增一个线程来执行任务
3. 当任务队列满了之后，不能往队列中放任务的时，就尝试新增一个工作线程，如果新增失败，就执行拒绝策略

**public** **void** execute(Runnable command) {

**if** (command == **null**)

**throw** **new** NullPointerException();

**int** c = ctl.get();取出线程池运行状态和线程个数（前3位表示线程状态，后29位表示线程数量）

当前工作线程数量小于核心线程池数量（*workerCountOf方法计算线程的数量*），调用addWorker()方法创建新线程运行任务，传来的任务作为第一个任务来运行，如果addWorker返回false，则直接返回

**if** (*workerCountOf*(c) < corePoolSize) {

**if** (addWorker(command, **true**))

**return**;

c = ctl.get();

}

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

**int** recheck = ctl.get();

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

reject(command);

**else** **if** (*workerCountOf*(recheck) == 0)

addWorker(**null**, **false**);

}

**else** **if** (!addWorker(command, **false**))

reject(command);

}

流程详细步骤

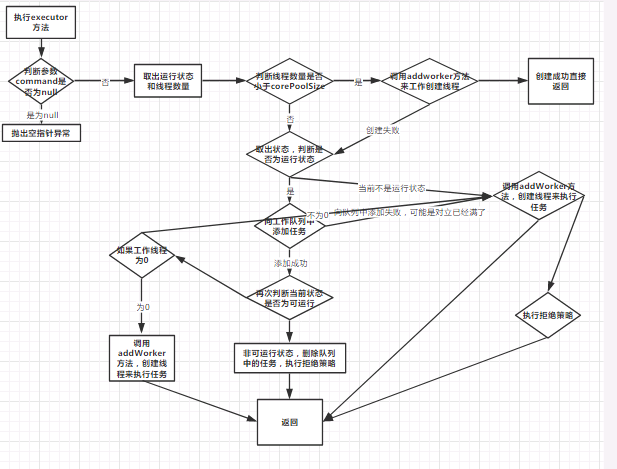
1. 调用executor方法，传入Runnable对象
2. 判断Runnable对象是否为空，如果为空就抛出空指针异常，不为空向下执行
3. 取出线程池状态和线程数量

当线程数量小于核心线程池的数量时，创建工作线程并将任务交给新的工作线程执行，如果添加工作线程成功，那么就返回

如果失败就再次获取线程池状态和线程数量，交给后续步骤操作

1. 当前线程数量已经等于或者超过了核心线程池中的数量，那么就想工作队列中添加任务，如果添加成功，就再次校验运行状态，如果运行状态不是RUNNING，那么就删除任务，然后执行拒绝策略；否则如果当前工作线程为0，那么就创建新的工作线程执行任务
2. 当前状态不是运行状态，就尝试能否创建新的线程来执行任务，如果不能就执行拒绝策略

Executor方法的核心流程就是：首先判断核心线程池是否已经满了，如果满了就将新的task放到任务队列中等待执行，如果任务队列也满了，那么就尝试创建工作线程来执行任务，如果工作线程也到达最大数量，那么就执行拒绝策略



addworker方法：第一个参数表示创建线程将要执行的第一个任务，第二个参数布尔类型，用来决定当前线程数量是和核心线程池数量比较还是和最大线程数进行比较

方法分为2部分：第一部分是循环判断当前状态并且增加线程池数量（使用原子方法+1），第二部分就是新增工作线程

第一部分的循环包括了2个循环：

外层循环用来校验线程池的状态：当状态为SHUTDOWN，STOP，TIDYING,TERMINATED中的一种并且！（当前状态==SHUTDOWN,传入的任务为null，队列不为空），条件成立那么就返回false（当前状态为STOP,TIDYING,TERMINATED；当前状态SHUTDOWN，并且已经有第一个任务；当前状态SHUTDOWN,并且任务队列为空）

内层循环用来增加线程数量：获取线程数量，首先个初始化的容量进行比较，如果已经超过了初始线程容量，那么就返回false；没有超过最大初始容量那么就根据第二个参数来判断当前数量是和核心池数量比较还是和最大线程池数量比较，当大于核心池数量或者最大线程池数量，那么就会返回false；当线程数量满足要求时，那么就通过CAS操作将线程池梳理+1，如果添加成功那么就结束循环执行后续操作，添加不成功（可能有其他地方在新增线程）那么继续循环判断

第二部分新增线程，创建worker对象，worker在构造过程中通过对象工厂将worker对象装换为一个线程对象，并保存在worker内的变量thread中；worker对象创建后，使用重入锁进行同步阻塞其他线程新增线程；将worker对象添加到workers集合中；worker对象创建并添加成功后就启动线程执行任务 最后返回true

如果创建添加过程中出现异常，那么就删除集合中的worker对象并且返回false

**private** **boolean** addWorker(Runnable firstTask, **boolean** core) {

retry:

**for** (;;) {

**int** c = ctl.get();

**int** rs = *runStateOf*(c);

当前状态为STOP,TIDYING,TERMINATED

当前状态SHUTDOWN，并且已经有第一个任务

当前状态SHUTDOWN,并且任务队列为空

**if** (rs >= *SHUTDOWN* &&

! (rs == *SHUTDOWN* &&

firstTask == **null** &&

! workQueue.isEmpty()))

**return** **false**;

**for** (;;) {

**int** wc = *workerCountOf*(c);

**if** (wc >= *CAPACITY* ||

wc >= (core ? corePoolSize : maximumPoolSize))

**return** **false**;

**if** (compareAndIncrementWorkerCount(c))

**break** retry;

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

**if** (*runStateOf*(c) != rs)

**continue** retry;

// else CAS failed due to workerCount change; retry inner loop

}

}

**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);

**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) {

t.start();

workerStarted = **true**;

}

}

} **finally** {

**if** (! workerStarted)

addWorkerFailed(w);

}

**return** workerStarted;

}

执行任务：

1. 当线程数量没有到达核心线程数量的时候

创建内部内对象worker实例worker类实现了Runnable接口，添加一个任务的时候，将任务实例中并且创建内部线程，调用start方法启动执行任务，执行就是不断循环判断内部变量task是否为空或者能不能从任务队列中获取到任务（getTask方法返回不为空），

线程初始化时就已经传入了task，task不为null，就直接执行run方法；再次判断时就是从队列中取任务，所以当没有到达核心线程时，任务都是线程直接执行了，并没有保存到队列中

1. 当线程数量 到达核心线程数量的时候

将任务保存到任务队列中，那么核心线程就能够从队列中拿到任务，然后就执行

1. 当添加的任务量超过了队列的容量，就创建新的工作线程来执行后面添加的任务，工作线程数量满了之后就执行拒绝策略

ThreadPoolExecutor的worker，继承AbstractQueuedSynchronizer类，实现Runnable接口；当创建工作完成后调用start方法启动线程，就执行run方法，方法内执行runWorker方法

**public** **void** run() {

runWorker(**this**);

}

启动线程后首先执行创建时传入的第一个任务，然后通过getTask从任务列表中去取任务来执行，如果线程池状态为STOP或者比状态值比stop大，那么就终止当前线程，并且线程数量-1（CAS实现）；第一个任务执行完成后调用getTask方法从队列中取出任务来执行；如果getTask方法返回不为空，那么线程就循环执行任务，如果取出的任务为空那么就线程就执行结束

**final** **void** runWorker(Worker w) {

Thread wt = Thread.*currentThread*();

Runnable task = w.firstTask;

w.firstTask = **null**;

w.unlock(); // allow interrupts

**boolean** completedAbruptly = **true**;

**try** {

**while** (task != **null** || (task = getTask()) != **null**) {

w.lock();

**if** ((*runStateAtLeast*(ctl.get(), *STOP*) ||

(Thread.*interrupted*() &&

*runStateAtLeast*(ctl.get(), *STOP*))) &&

!wt.isInterrupted())

wt.interrupt();

**try** {

beforeExecute(wt, task);

Throwable thrown = **null**;

**try** {

task.run();

} **catch** (RuntimeException x) {

thrown = x; **throw** x;

} **catch** (Error x) {

thrown = x; **throw** x;

} **catch** (Throwable x) {

thrown = x; **throw** **new** Error(x);

} **finally** {

afterExecute(task, thrown);

}

} **finally** {

task = **null**;

w.completedTasks++;

w.unlock();

}

}

completedAbruptly = **false**;

} **finally** {

processWorkerExit(w, completedAbruptly);

}

}

**private** Runnable getTask() {

**boolean** timedOut = **false**; // Did the last poll() time out?

retry:

**for** (;;) {

**int** c = ctl.get();

**int** rs = *runStateOf*(c);

当前状态为SHUTDOWN并且任务队列为空时返回null

当前状态为STOP以及更大的状态时，也返回null

工作线程数量-1

**if** (rs >= *SHUTDOWN* && (rs >= *STOP* || workQueue.isEmpty())) {

decrementWorkerCount();

**return** **null**;

}

**boolean** timed; // Are workers subject to culling?

**for** (;;) {

**int** wc = *workerCountOf*(c);

timed = allowCoreThreadTimeOut || wc > corePoolSize;

**if** (wc <= maximumPoolSize && ! (timedOut && timed))

**break**;

**if** (compareAndDecrementWorkerCount(c))

**return** **null**;

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

**if** (*runStateOf*(c) != rs)

**continue** retry;

// else CAS failed due to workerCount change; retry inner loop

}

**try** {

Runnable r = timed ?

workQueue.poll(keepAliveTime, TimeUnit.*NANOSECONDS*) :

workQueue.take();

**if** (r != **null**)

**return** r;

timedOut = **true**;

} **catch** (InterruptedException retry) {

timedOut = **false**;

}

}

}

默认决绝策略

直接抛出异常RejectedExecutionException

**final** **void** reject(Runnable command) {

handler.rejectedExecution(command, **this**);

}

**public** **static** **class** AbortPolicy **implements** RejectedExecutionHandler {

**public** AbortPolicy() { }

**public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {

**throw** **new** RejectedExecutionException("Task " + r.toString() +

" rejected from " +

e.toString());

}

}

关闭线程池

shutdown()方法：调用线程池将不再接受新的任务 ，然后将先前放在队列中的任务执行完成

将线程状态设置为SHUTDOWN，然后让每个线程执行interrupt方法来中断线程

**public** **void** shutdown() {

**final** ReentrantLock mainLock = **this**.mainLock;

mainLock.lock();

**try** {

checkShutdownAccess();

advanceRunState(*SHUTDOWN*);

interruptIdleWorkers();

onShutdown(); // hook for ScheduledThreadPoolExecutor

} **finally** {

mainLock.unlock();

}

tryTerminate();

}

ShutdownNow()方法：立即停止所有的执行任务，并将队列中的任务返回

将线程的状态设置为STOP,然后让每个线程执行interrupt方法来中断线程

**public** List<Runnable> shutdownNow() {

List<Runnable> tasks;

**final** ReentrantLock mainLock = **this**.mainLock;

mainLock.lock();

**try** {

checkShutdownAccess();

advanceRunState(*STOP*);

interruptWorkers();

tasks = drainQueue();清除对类中的任务

} **finally** {

mainLock.unlock();

}

tryTerminate();

**return** tasks;

}

**提交任务**

submit方法在是在父类AbstractExecutorService中实现的

submit方法有3个重载方法，将任务添加到线程池中执行（使用execute方法来执行任务），如果需要获取返回执行，调用Future的get方法会阻塞主线程直到获取到返回结果

参数为Runnable

**public** Future<?> submit(Runnable task) {

**if** (task == **null**) **throw** **new** NullPointerException();

RunnableFuture<Void> ftask = newTaskFor(task, **null**);

execute(ftask);

**return** ftask;

}

参数除了Runnable，执行任务的的返回值result

**public** <T> Future<T> submit(Runnable task, T result) {

**if** (task == **null**) **throw** **new** NullPointerException();

RunnableFuture<T> ftask = newTaskFor(task, result);

execute(ftask);

**return** ftask;

}

参数是Callable

**public** <T> Future<T> submit(Callable<T> task) {

**if** (task == **null**) **throw** **new** NullPointerException();

RunnableFuture<T> ftask = newTaskFor(task);

execute(ftask);

**return** ftask;

}

方法流程

调用submit，传入Runnable或者Callable

判断输入的参数是否为空，如果为空，那么就抛出异常，如果不为空，那么就执行后续流程

将传入的对象通过newTaskFor方法转换为RunnableFuture对象，将RunnableFuture对象交给execute方法执行，返回RunnableFuture对象

shutdown和shutdownNow区别

shutdown和shutdownNow这两个方法的作用都是关闭线程池，流程大致相同，只有几个步骤不同，如下

1. 加锁
2. 检查关闭权限
3. CAS改变线程池状态
4. **设置中断标志(线程池不在接收任务，队列任务会完成)/中断当前执行的线程**
5. **调用onShutdown方法（给子类提供的方法）/获取队列中的任务**
6. 解锁
7. 尝试将线程池状态变成终止状态TERMINATED
8. **结束/返回队列中的任务**

总结

线程池减少了线程的开销，提供了线程的复用，ThreadPoolExecutor提供了一个为实现的方法，比如beforeExecutor，afterExecutor等方法，可以通过这些方法对象进行进一步管理和统计

创建线程池

构造函数的参数

corePoolSize：核心线程数量，当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使空闲的线程能够执行新的任务也会创建线程，当创建的线程数量等于核心线程数时，就不在创建

maximumPoolSize：最大线程数：当线程数量到达核心线程数量后，不在创建线程，而是将传入的任务存放到阻塞队列中，当阻塞队列满了之后，有创建线程来 执行新增的任务，但是创建的线程数量不能超过这个最大线程数

keepAliveTime:线程存活时间：

timeUnit:线程存活时间的时间单位

ThreadFactory：用来创建工作线程的线程工厂，默认是DefaultThreadFactory

BlockQueue：任务阻塞队列：可以选择的队列有

ArrayBlockingQueue：一个基于数组的有界的阻塞队列，次队列按FIFO的原则对元素进行排序

LinkedBlockingQueue：一个基于链表的阻塞队列，按FIFO的原则进行排序，吞吐量比ArrayBlockingQueue高；Executors.newFixedThreadPool()来创建ThreadPoolExecutor

，因为是无界的所以不推荐使用这种方式来创建线程池，因为如果一直往队列中添加任务，那么就容易出现oom异常

RejectedExecutionHandler:拒绝策略，当线程池和队列都满了的时候，不能在创建线程或者往任务队列中添加任务，那么就采取策略来处理新提交的任务，默认是AbortPolicy，表示无法处理新任务并抛出异常

Jdk提供的其他的拒绝策略

CallerRunsPolicy d;// 只用调用者所在的线程来运行任务，通过主线程来执行

**public** **static** **class** CallerRunsPolicy **implements** RejectedExecutionHandler {

**public** CallerRunsPolicy() { }

**public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {

**if** (!e.isShutdown()) {

r.run();

}

}

}

DiscardOldestPolicy s;// 丢弃队列中的最近一个任务，并执行当前任务

**public** **static** **class** DiscardOldestPolicy **implements** RejectedExecutionHandler {

**public** DiscardOldestPolicy() { }

**public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {

**if** (!e.isShutdown()) {

e.getQueue().poll();丢弃最近的任务

e.execute(r);

}

}

}

DiscardPolicy p;// 不处理，不丢弃

**public** **static** **class** DiscardPolicy **implements** RejectedExecutionHandler {

**public** DiscardPolicy() { }

**public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {

拒绝方法为空方法

}

}

分配线程池

要合理的分配线程池，就需要分析任务的特性

任务的性质：cpu密集型任务和IO密集型任务，混合型任务

任务的优先级：高、中、低

任务的执行时间：长、中、短

任务的依赖性：是否依赖系统其他资源

Cpu密集型任务：就是需要进行复杂运算或者运算量比较多的任务，对cpu消耗较多

Io密集型任务：就涉及网络io,磁盘io的任务都是io密集型任务，这里任务的特点是消耗cpu资源较少，任务的大部分时间都是在等待IO操作，对应io密集型任务，任务越大，cpu效率越高，但是也是有限度的

性质不同的任务可以用不同的规模的线程池分开处理：cpu密集型任务应尽量分配小的线程，比如cpu数量+1,个线程的线程池，由于IO密集型的任务并不是一直在执行任务，应尽量分配多的线程，比如2\*cpu数量

混合型任务：包括了cpu密集型和io密集型任务，那么就将任务分为2种，将其拆分为cpu密集型任务和io密集型任务；如果两个任务拆分的时间相差不太大，那么分解后的吞吐量高于串行的吞吐量。如果两个任务执行的时间相差太大，则没有必要进行分解

优先级不同的任务，可以使用PriorityBlockingQueue来处理，可以让优先级高的任务先执行；执行时间不同的任务可以交给不同规模的线程池来处理，或者可以使用优先级队列让执行时间短的任务先执行

获取线程池状态

taskCount：线程池需要执行的任务数量

completedTaskCount:线程池运行过程中以完成的任务数量

largestPoolSize：线程池中创建过的最大工作线程数量

getPoolSize():线程池的线程数量

getActiveCount():获取活动的线程数

建议使用有界的队列存放任务，如果使用无界队列，当阻塞的任务越来越多时就可能撑爆内存

## Executors框架

Executors框架

使用Executors创建不同类型的线程池

**Executors.*newFixedThreadPool***

Executors.*newFixedThreadPool*(2);创建固定工作线程数量的ThreadPoolExecutor线程池，就是执行ThreadPoolExecutor的构造函数

CorePoolSize,maximumPoolSize的数量是一样的，

Keepalivetime：空闲线程等待新任务的时间，超过这个时间，线程就会被终止，设置为0，多余的空闲线程会立刻被终止

任务队列是LinkedBlockingQueue，无界队列；

无界工作队列的影响

1. 当线程数量达到corePoolSize后，新任务就会添加到无界队列中，线程池中的线程数量不会超过corePoolSize
2. maximumPoolSize是一个无效的参数，因为任务队列不会满
3. keepAliveTime是一个无效的参数
4. 使用了无界队列，运行中的拒绝策略不会发送

**public** **static** ExecutorService newFixedThreadPool(**int** nThreads) {

**return** **new** ThreadPoolExecutor(nThreads, nThreads,

0L, TimeUnit.*MILLISECONDS*,

**new** LinkedBlockingQueue<Runnable>());

}

**Executors.*newCachedThreadPool***

Executors.*newCachedThreadPool*();大小无界的线程池，可以进行线程的自动回收。创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程；

CorePoolSize为0，

maximumPoolSize值为Integer的最大值，表示可以创建Integer.MAX\_VALUE个线程

keepAliveTime:线程空闲等待时间为60秒，超过60秒线程就会终止，时间单位为秒

存放任务的队列为SynchronousQueue队列，SynchronousQueue是一个只能存放一个元素的阻塞队列，插入操作必须等待另线程的对应的异常操作

SynchronousQueue没有容量，那么就意味着提交的任务都是交给工作线程来执行的，如果提交任务的速度大于了线程执行任务的速度，CachedThreadPool就会不断的创建线程，在极端情况下会因为创建过多的线程而耗尽cpu和内存资源

工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

**public** **static** ExecutorService newCachedThreadPool() {

**return** **new** ThreadPoolExecutor(0, Integer.*MAX\_VALUE*,

60L, TimeUnit.*SECONDS*,

**new** SynchronousQueue<Runnable>());

}

**Executors.*newSingleThreadExecutor***

Executors.*newSingleThreadExecutor* ();创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

corePoolSize，maximumPoolSize的值都为1，那么就只会创建一个线程来执行任务，也就是说任务是按顺序一个一个的执行

使用LinkedBlockingQueue队列来保存任务

**public** **static** ExecutorService newSingleThreadExecutor() {

**return** **new** FinalizableDelegatedExecutorService

(**new** ThreadPoolExecutor(1, 1,

0L, TimeUnit.*MILLISECONDS*,

**new** LinkedBlockingQueue<Runnable>()));

}

FinalizableDelegatedExecutorService线程池，创建ThreadPoolExecutor对象来作为构造函数的参数，使用装饰器模式，内部执行任务还是交给ThreadPoolexecutor来完成

**Executors.newScheduledThreadPool**

Executors.newScheduledThreadPool(corePoolSize) ，创建一个定长线程池，支持定时及周期性任务执行，适用于后台线程执行周期任务，同时为了满足后台资源管理的需求而限制后台线程的数量的应用场景

ScheduledThreadPoolExecutor继承ThreadPoolExecutor，主要是用来在给定的延迟之后运行任务，或者执行定时任务；ScheduledThreadPoolExecutor功能和Timer类型，但是比Timer更加强大，灵活，Timer对应的是单个后台任务，ScheduledThreadPoolExecutor可以指定多个对应的后台线程数据

**public** ScheduledThreadPoolExecutor(**int** corePoolSize) {

**super**(corePoolSize, Integer.*MAX\_VALUE*, 0, TimeUnit.*NANOSECONDS*,

**new** DelayedWorkQueue());

}

DelayedWorkQueue：无界队列，那么maximumPoolSize这个参数没有意义

Executors.newSingleThreadScheduledExecutor();适用于单个线程后台执行周期性任务

DelegatedScheduledExecutorService通过ScheduledThreadPoolExecutor来实现的，ScheduledThreadPoolExecutor作为其构造函数的参数，使用装饰器模式，具体的实现是由ScheduledThreadPoolExecutor来完成的

## ThreadLocal

提供线程局部变量

## 并发容器

### ConcurrentHashMap

### ConcurrentLinkedQueued