# 基础

并发编程的目的是为了程序运行的更快，但是，并不是启动更多的线程就能让程序最大限度地并发执行。如果通过多线程让程序执行任务更快，会面临非常多的困难，比如上下文切换，死锁等问题，以及受限于硬件和软件资源限制问题。

上下文切换：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下一个任务，但是，在切换前会保存上一个任务的状态，以便下次切换回来这个任务时，可以再次加载这个任务的状态，所以任务从保存到再次加载的过程叫做上下文切换。

如何减少上下文切换：无锁并发编程，CAS算法，使用最少线程和使用协程

无锁并发编程：

# Lock接口

与使用synchronized方法或代码块相比，Lock的实现提供了更多扩展的锁操作，lock允许更灵活的结构，可以具有完全不同的属性，并且可以支持多个与之关联的Condition对象。

lock是控制多线程访问共享资源的工具，lock提供对共享资源独有的访问，在一个时刻只能有一个线程获得锁，对共享资源的访问必须先获得的锁，然而一些锁可以允许并发访问共享资源，例如：ReadWriteLock

使用synchronized方法或代码块提供了对与之关联对象的隐式监控器锁的访问，但强迫获取锁和释放锁均要在一个阻塞块中，当多个锁被请求，它们必须以相反的顺序释放，每个锁必须在获取它们的相同的语法范围内释放。

虽然synchronized方法或者代码块的范围机制使得使用监控锁编程更加容易，假如在你工作时刻，你需要以更加灵活的方式使用锁，例如，需要使用“手握手”方式或者“”来并发遍历某中数据结构。

如：手把手进行锁获取和锁释放，先获取锁A，然后再获取锁B，当锁B获得后，释放锁A同时获取锁C，当锁C获取成功后，再释放锁B同时获取锁D，此次类推。

Lock接口的实现比使用synchronized方法或代码块提供了额外的功能，通过提供一个非堵塞的方式尝试获取锁，尝试获取一个可中断的锁或者尝试获取一个超时失效锁。

虽然他缺少了隐式获取释放锁的便捷性，但是却拥有了锁获取和锁释放的可操作性，可中断的方式获取锁以及超时获取锁等多种synchronized关键字所不具有的同步特性。

|  |  |
| --- | --- |
| 特性 | 描述 |
| 尝试非堵塞地获取锁 | 当前线程尝试获取锁，如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁。 |
| 能被中断地获取锁 | 与synchronized不同，获取锁的线程能够被响应中断，当获取到锁的线程被中断时，中断异常将被抛出，同时锁被释放。 |
| 超时获取锁 | 在指定的时间之前获取锁，同时超过指定的时间仍旧无法获取锁，则返回。 |

# AbstractQueuedSynchronizer

**队列同步器：**

是用来构建锁或其他同步组件的基础框架，它使用一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作。

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，在抽象方法的实现中免不了对同步转态进行更改，这是我们就需要使用同步器提供的方法来进行操作（getState()，setState(int)和compareAndSetState(int ,int)），因为它们可以保证状态的改变是安全的。

同步器是实现锁的关键，在锁的实现中聚合同步器，利用同步器实现锁的语义。我们可以这样理解：锁是面向使用者的，它定义了使用者和锁的交互的接口，隐藏了实现细节；同步器是面向锁的实现者，它简化了锁的实现方式，屏蔽了同步状态的管理，线程的排队，等待与唤醒等底层操作。

同步器的设计是基于模板方法模式的，也就是说，使用者需要继承同步器并重写指定的方法，随后将同步器组合在自定义同步组件的实现中，并调用同步器提供的模板方法。而这些模板方法将会调用使用者重写的方法。

## 官方文档说明

为实现依赖于先进先出（FIFO）的等待队列的阻塞锁和同步锁，提供了一个框架，该类被设计为大多数依赖单个原子int值来表示状态的同步器的一个有用的基础。子类必须定义一个受保护的方法，用于改变该状态值，并且定义哪种状态表示对象的获取和释放。考虑这些后，在该类中的其他方法可以实现队列和阻塞机制。子类可以包含其他的状态字段，但是必须使用getState()，setState(int)和compareAndSetState(int ,int)方法原子性的更新int值。

子类必须定义一个非公共的内部帮助类，用于实现封闭类的同步属性，该类并没有实现任何同步接口，而是定义了一些方法，如：acquireInterruptibly(int)方法，在适当的时候可以通过具体的锁和相关的同步器来调用它们，以实现其公共方法。

该及支持独占模式，也支持共享模式。如果是独占模式，则其他线程获取锁一定会失败，如果是共享模式，则其他线程可能会成功获取锁（但不一定）。类并不理解这之间的差异，处理机器的意识到当在共享模式下成功获取某一个锁时，下一个等待线程也必须确定是否可以成功获取该锁。在不同模式下的等待线程，可以共享FIFO队列，通常，子类的是实现仅支持一种模式，但是两者可以发挥作用，如ReadWriteLock类。仅支持独占模式或共享模式的子类并不需要实现自己不支持模式的方法。

该类为内部队列提供了检查，检测和监控的方法，同样也挺condition对象提供了相似的方法。可以使用该类的同步机制，将这些方法导出到自己希望的类中。

该类的序列化操作仅存储了用于维护状态的基础原子整数值，因此反序列化对象有一个空的线程队列。需要序列化的典型子类定义一个readObject方法，该方法在反序列化时将该对象恢复到一个已知的初始状态。

同步器可重写的方法

|  |  |
| --- | --- |
| 方法名称 | 描述 |
| protected boolean tryAcquire(int arg) | 独占式获取同步状态，实现方法需要查询当前状态并判断同步状态是否符合预期，然后进行CAS设置同步转态 |
| protected boolean tryRelease(int arg) | 独占式释放同步状态，等待获取同步状态的线程将有机会获取同步状态 |
| protected boolean tryReleaseShared(int arg) | 共享式释放同步状态 |
| protected int tryAcquireShared(int arg) | 共享式获取同步状态，返回大于等于0的值，表示获取成功，否则，获取失败 |
| protected boolean isHeldExclusively() | 当前同步器是否在独占模式下被线程占用，一般该方法表示是否被当前线程锁独占 |

实现自定义同步组件时，会调用同步器提供的模板方法，这些模板方法与描述如下：

**public final void acquire(int arg)：**独占式获取锁，忽略中断，如果当前线程获取同步状态成功，则有该方法返回，否则，将会进入同步队列等待，该方法将会调用重写的tryAcquire方法。

**public final void acquireInterruptibly(int arg)：**独占式获取锁，但是该方法响应中断，当前线程为获取锁而进入同步队列，如果当前线程被中断，则该方法会抛出异常并返回。

**public final boolean tryAcquireNanos(int arg,long nanosTimeout)：**在acquireInterruptibly基础上增加了超时限制，如果当前线程在指定的时间内没有获取到同步状态，那么会返回false，如果获取到了返回true。

**public final void acquireShared(int arg)：**共享式的获取同步状态，如果当前线程未获取到同步状态，将会进入同步队列等待，与独占式获取的主要区别是在同一时刻可以有多个线程获取到同步状态。

**public final void acquireSharedInterruptibly(int arg)：**与acquireShared相同，该方法响应中断

**public final boolean tryAcquireSharedNanos(int arg,long nanosTimeout)：**添加了超时限制。

**public final boolean release(int arg)：**独占式释放同步状态，该方法会在释放同步状态之后，将同步队列中第一个节点包含的线程唤醒

同步器提供的模板方法基本分为3类：独占式获取与释放同步状态，共享式获取和释放同步状态，查询同步队列中的等待线程情况。

AbstractQueuedSynchronizer内部实现的队列，队列中的元素Node就是保存着线程引用和线程状态的容器，每个线程对同步器的访问，都可以看做是队列中的一个节点：

Node详解：

Int waitStatus：等待状态，包含如下状态

static final int SIGNAL = -1：后继节点的状态处于等待状态，而当前节点的线程如果释放了同步状态或者被取消，将会通知后继节点，使后继节点的线程得以运行。

static final int CANCELLED = 1：由于在同步队列中等待的线程等待超时或被中断，需要从同步队列中取消等待，节点进入该状态将不会变化。

static final int CONDITION = -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signal()方法后，该节点将会从等待队列中转移到同步队列中，假如到同步状态的获取中。

static final int PROPAGATE = -3：表示下一次共享式同步状态将会无条件地被传播下去

Node prev：前驱节点，当节点假如同步队列时被设置（尾部添加）。

Node next：后继节点

Node nextWaiter：等待队列中的后继节点。如果当前节点是共享的，那么这个字段将是一个SHARED常量，也就是说节点类型和等待队列中的后街节点共用一个子弹。

Thread thread：获取同步状态的线程。

具体模型



# ReentrantLock重入锁

重入锁就是支持重入的锁，它表示该锁能够支持一个线程对资源的重复加锁。此外，该锁还支持获取锁的公平和非公平性选择。

在绝对时间上，先对锁进行获取的请求一定先被满足，那么锁是公平的，反之，是不公平的。

公平锁机制没有非公平锁的效率高，但是，并不是任何情景都可以以TPS作为唯一的指标，公平锁能够减少“饥饿”发生的概率，等待越久的请求越是能够得到优先满足

ReentrantLock：分为公平锁和非公平锁。ReentrantLock构造方法：

//无参构造方法，用于创建非公平锁

public ReentrantLock()

{

sync = new NonfairSync();

}

//根据传入的fair参数来创建公平锁，还是非公平锁

public ReentrantLock(boolean fair)

{

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

}

## 公平锁

表示线程获取锁的顺序是按照线程加锁的顺序来分配的，即先来先得FIFO先进先出顺序

公平性与否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。

### 案例

class Server

{

private ReentrantLock lock ;

private Logger logger = Logger.getLogger(RunFair.class) ;

public Server(boolean isFair)

{

super();

lock = new ReentrantLock(isFair);

}

public void serviceMethod()

{

try

{

lock.lock();

logger.debug("ThreadName:"+Thread.currentThread().getName()+"获得锁");

}finally

{

lock.unlock();

}

}

public class RunFair

{

private static Logger logger = Logger.getLogger(RunFair.class) ;

public static void main(String[] args)

{

Server server = new Server(true) ;

Runnable runnable = new Runnable() {

@Override

public void run()

{

logger.debug("@线程"+Thread.currentThread().getName()+"运行了");

server.serviceMethod();

}

};

Thread[] threadArray = new Thread[10];

for(int i = 0;i<10;i++)

{

threadArray[i] = new Thread(runnable);

}

for(int i = 0;i<10;i++)

{

threadArray[i].start();;

}

}

## 非公平锁

非公平锁就是一个获取锁的抢占机制，是随机获得锁的，和公平锁不一样的就是先来的不一定先得到锁，这种方式可能造成某些线程一直拿不到锁，结果就是不公平的。

## 源码分析

ReentrantLock源码中定义了一个抽象的，静态的内部类Sync（是同步监控的基础，子类实现了公平锁和非公平锁，使用AQS状态代表持有锁的个数），该类继承与AbstractQueuedSynchronizer。并且ReentrantLock实现了Lock接口。

**abstract** **static** **class** Sync **extends** AbstractQueuedSynchronizer {

**private** **static** **final** **long** ***serialVersionUID*** = -5179523762034025860L;

//用于执行Lock接口的lock方法，需要子类实现

**abstract** **void** lock();

//非公平获取锁

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

//获取当前线程

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

//获取状态值

**int** c = getState();

//如果状态值为0，表示锁还没有被占用

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

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

setExclusiveOwnerThread(current);

**return** **true**;

}

}

//判断是否为重入锁，即当前线程与占用锁的线程是否为同一个线程

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

//使同步状态值加1

**int** nextc = c + acquires;

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

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

//更新同步状态值

setState(nextc);

**return** **true**;

}

**return** **false**;

}

//锁的释放

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

//获取状态值

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

//判断当前线程是否和获取锁的线程相同

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

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

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

//如果同步状态值为0，则表示锁被释放

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

free = **true**;

//

setExclusiveOwnerThread(**null**);

}

//更新状态值，表示其他线程可以获取锁

setState(c);

**return** free;

}

//判断当前线程是否获取锁

**protected** **final** **boolean** isHeldExclusively() {

**return** getExclusiveOwnerThread() == Thread.*currentThread*();

}

**final** ConditionObject newCondition() {

**return** **new** ConditionObject();

}

**final** Thread getOwner() {

**return** getState() == 0 ? **null** : getExclusiveOwnerThread();

}

**final** **int** getHoldCount() {

**return** isHeldExclusively() ? getState() : 0;

}

**final** **boolean** isLocked() {

**return** getState() != 0;

}

**private** **void** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException {

s.defaultReadObject();

setState(0); // reset to unlocked state

}

}

# 原子类

原子本意是“不能被进一步分割的最小粒子”，而原子操作是“不可被中断的一个或一系列操作”。

参考：java并发编程的艺术。

## 原子操作

在Java中可以通过锁和循环CAS的方式来实现原子操作。

**package** code.lsh.atomic;

**import** java.util.concurrent.atomic.\*；

**import** java.util.\*;

**public** **class** Counter

{

**private** AtomicInteger atomicI = **new** AtomicInteger();

**private** **int** i = 0;

**private** **void** safeCount()

{

**for**(;;)

{

**int** i = atomicI.get();

**boolean** suc = atomicI.compareAndSet(i, ++i);

**if**(suc) **break**;

}

}

**private** **void** count() {

i++;

}

**public** **static** **void** main(String[] args)

{

**final** Counter counter = **new** Counter();

List<Thread> ts = **new** ArrayList<>();

**for**(**int** i = 0;i<100;i++)

{

Thread t = **new** Thread(()->{

**for**(**int** ii = 0;ii<1000;ii++)

{

counter.count();

counter.safeCount();

}

});

ts.add(t);

}

ts.parallelStream().forEach(Thread::start);

ts.stream().forEach(t -> {

**try** {

t.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

});

System.***out***.println(counter.i);

System.***out***.println(counter.atomicI.get());

}

}

Counter类定义一个非线程安全计数器和使用CAS实现线程安全计数器。非线程安全方法操作变量i，而线程安全计数器操作AtomicInteger变量，我们启动了100个线程，每一个线程对这个两个变量进行1000次的自增操作。

运行结果：



我们知道，对于基本类型的变量的读取和赋值是原子性的，要么都成功，要么都失败，这些操作是不可被中断的。然而对于++i是非原子的，是线程不安全的。

## 深入理解CAS

CAS（CompareAndSwap）：

在JDK1.5中，JDK的并发包里提供了一些类来支持原子操作，如AtomicInteger，AtomicBoolean等，而这些原子类的实现原理就是CAS，在java为我们提供了CAS的实现类主要是Unsafe类。

Unsafe类使用了单例模式，需要通过一个静态方法getUnsafe()来获取。但Unsafe类做了限制，如果是普通的调用的话，它会抛出一个SecurityException异常；只有由根类加载器加载的类才能调用这个方法，这里我是用了反射来获取Unsafe实例。

@CallerSensitive

public static Unsafe getUnsafe() {

Class<?> caller = Reflection.getCallerClass();

if (!VM.isSystemDomainLoader(caller.getClassLoader()))

throw new SecurityException("Unsafe");

return theUnsafe;

}

### CAS原子操作问题

在Java并发中有一些并发框架也使用了自旋CAS方式来实现原子操作，但是CAS仍然存在三大问题：ABA问题，循环时间长开销大，以及只能保证一个共享变量的原子操作。

1. ABA问题。因为CAS需要在操作值的时候，检查值有没有发生变化，如果没有发生变化则更新，假如现在有两个线程threadA，threadB和threadC，在threadA想将变量A转为B，然而线程threadB将变量转为B了，threadC有将变量B转为A，在threadA执行时发现A并没有发生变化，threadA并不知道变量A已经被threadB和threadC操作过了。