链滴

Java并发多线程学习笔记

作者：guobing

原文链接：https://ld246.com/article/1470842434905

来源网站：链滴

Java并发多线程学习笔记

标签（）： java 多线程 并发编程 艺术

原子操作的实现原理

#1、术语

比较并交换 compare and swap（一个新值和旧值，比较旧值有没有发生变化，如果没有发生变化则换成新值）

CPU流水线 CPU pipeline --

内存顺序冲突 Memory order violation -- 一般由假共享内存引起，出现冲突时，cpu流水线必须清空

#2、处理器如何实现原子操作

32位IA-32处理器使用缓存加锁或总线加锁的方式来处理多处理器之间的原子操作

一般情况下处理器会自动保证基本内存操作的原子性，但是复杂的内存操作则并不能自动保证原子性比如跨总线宽度，跨多个缓存行和跨页表的访问。

处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

##1.总线锁

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此信号时，其他处理器的求将被阻塞，那该处理器可以独占共享内存。

##2.缓存锁

频繁使用的内存会缓存在处理器的L1，L2，L3高速缓存里

缓存锁定是指内存区域如果被缓存在处理器的缓存行中，并且在lock操作期间被锁定，那么当它执行操作回写到内存时，处理器不在总线上声言LOCK#信号，而是修改内部的内存地址，并允许他的缓存致性机制来保证操作的原子性。因为缓存一致性机制会阻止同时修改两个以上的处理器缓存的内存区数据。

#3 .cas实现原子操作的三大问题

##1. aba问题

解决办法：变量前面追加版本号

##2. 循环时间长开销大

解决办法：pause指令

pause指令的作用

1. 延迟流水线执行指令,使cpu不会消耗过多的执行资源。

2. 可以避免在退出循环的时候因内存顺序冲突而引起的cpu流水线被清空

3. 只能保证一个共享变量的原子操作

解决办法就是把多个共享变量合并成一个共享变量。

另一种办法就是使用锁。只有获得了锁的线程才能操作锁定的内存区域。

线程封闭的方法

- ad-hoc
  - 栈封闭
  - ThreadLocal类.(最优选择)

硬件对并发的支持

- cas.乐观技术.是一种处理器指令
- 非阻塞的计数器.利用cas实现的一个线程安全的计数器.读取旧值,计算出新值,利用cas来设置新值
- JVM对cas的支持.jdk中原子变量类cocurrent.atomic,底层都是cas操作

另一种并发模型Actor模型与Akka框架分析

Akka是一个分布式应用框架,用scala编写,帮助我们降低编写并发、容错、可扩展的应用的门槛,利用色模型提升了抽象等级，并且提供了一个用来构建可扩展的、弹性的以及响应式应用的更好的平台.

- Scala是一种多范式的编程语言，设计初衷是要集成面向对象编程和函数式编程的各种特性。

Actor模型

- 系统中的所有事物都可以扮演一个Actor
- Actor之间完全独立
- 在收到消息时Actor所采取的所有动作都是并行的，在一个方法中的动作没有明确的顺序
- Actor由标识和当前行为描述
- Actor可能被分成原始（primitive）和非原始（non primitive）类别

4. Java多线程编程核心技术笔记

4.1 暂停线程

用suspend暂停线程,用resume恢复线程

缺点: 容易造成公共的同步对象的独占

yield : 放弃当前的cpu资源,让其他任务占用cpu执行时间.但是放弃的时间不确定

4.2 线程优先级

setPriority,优先级较高的线程得到更多的cpu资源

4.3 守护线程

其实就是Daemon线程.当最后一个非守护线程结束时,守护线程也会一起结束工作.典型应用就是GC

4.4 synchronized方法解析

- 非线程安全存在于　实例变量之中，方法内部的私有变量，则是线程安全的。

- 多个线程同时访问同一个对象的共享资源，存在线程安全问题，要同 synchronized关键字。多个程访问不同对象的共享资源，不存在线程安全问题。因为每个对象都有一份，不存在共享的问题
- 以前看起来很简单却一直没理解的一句话：只有共享资源的读写才需要同步化，不是共享资源根本没有同步的必要
- 出现异常，锁自动释放
- 同步不具有继承性

synchronized锁重入问题

synchronized拥有锁功能。锁重入就是当一个线程得到一个对象锁后，再次请求该对象锁是可以得到对象的对象锁的。所以说在一个synchronized内部调用本类的其他synchronized方法或者块时，永是可以得到对象锁的。

自己可以再次获取自己的内部锁

如果不能的话，就会引起死循环。因为当前的锁还没有释放，又不能获取内部锁，只能不断循环获取每次都获取失败。一直处于死锁状态。

synchronized同步语句块

当两个线程并发的访问同一个对象中的synchronized同步块时，一段时间内只能有一个线程被执行。同步方法有区别。同步方法没有一个的限制，只要是同一个对象都可以进入执行。

- 不在 synchronized块中时异步执行，在synchronized中时同步执行
- synchronized真的持有当前调用对象的锁
- 多个线程调用同一个对象中不同名称的同步方法或者同步块，是按顺序执行的。即同步的，阻的。
- 锁非this对象。
- 静态同步方法和 synchronized(class)其实是一样的。

4.5 线程通信

等待、通知机制

JOIN

使得调用者x正常的执行run方法，而当前线程阻塞，线程x销毁后再执行后面的方法。join具有使线排队运行的作用。有类似同步的运行效果。和synchronized相似。但是join内部是用wait来等待的。而ynchronized使用对象监视器来同步

- join内部是用wait是实现的，当执行wait后，当前线程的锁被释放.其他线程可以调本对象中的非un方法。
- join只是说本对象的run执行完成前，当前线程是阻塞的，其他对象的run必须等到本对象的run行完成之后才能执行。
- 但是本对象的非run方法还是可以被调用的。
- sleep没有释放本对象的锁。wait释放了本对象的锁

Notify

用来通知可能等待该对象的对象锁的其他线程。

ReentrantLock的使用

ReentrantLock和synchronized类似，可以实现线程之间的同步互斥。并且更强大。多了多路分支通知能。

- 多路分支通知的实现。在一个lock对象中创建多个condition实例，线程注册到指定的conditio中。从而有选择的进行通知。更加灵活。
- notify/notifyAll通知时是由jvm随机选择的。但是 ReentrantLock可以实现选择通知哦。这点较强大。
- synchronized相当于整个lock中只有一个单一的condition，所有的线程都注册在这个对象上。
- synchronized中的wait、notify、notifyAll分别相当于ReentrantLock中的await、signal、signlAll

公平锁与非公平锁

1. 公平锁

线程获取锁的顺序是按照线程枷锁的顺序来分配的。即按照先进先出的FIFO顺序。

2. 非公平锁

获取锁的抢占机制。是随机获取锁的。

ReentrantReadWriteLock

reentrantLock完全互斥排他。效率低下。

ReentrantReadWriteLock表示有两个锁。读锁，也称为共享锁。写锁，也称为排它锁。多个读锁之不互斥，读写锁之间互斥，写写互斥。

yield方法

放弃当前的cpu资源，让给其它的任务占用cpu时间。但是放弃的时间不确定。

countDownLatch

和join方法类似。但是比join的功能更多。

jdk1.5之后提供了这个功能。

```
package com.study.thread;

import java.util.concurrent.CountDownLatch;

/**
 * 接受int类型的参数作为计数器。当计数器为0时不再阻塞当前线程。
 * Created by guobing on 2016/8/3.
 */
public class JoinCountDownLatch {

    static CountDownLatch cdl = new CountDownLatch(2);

    public static void main(String [] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(1);
                cdl.countDown();
                System.out.println(2);
```

```
                cdl.countDown();
            }
        }).start();

        try {
            cdl.await();
            System.out.println(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

同步屏障CyclicBarrier

意思是：让一组达到一个屏障时阻塞，直到最后一个线程到达屏障时，屏障才会开门。线程才继续运。

构造方法，CyclicBarrier(int parties),参数表示要拦截的线程数量。每个线程调用await方法告诉CyclicBarrier到达屏障了。然后当前线程被阻塞。

```
package com.study.thread;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

/**
 * 当改成3的时候，线程一直运行。因为没达到3.不会停止。
 * Created by guobing on 2016/8/3.
 */
public class CyclicBarrierTest {

    static CyclicBarrier cb = new CyclicBarrier(3);

    public static void main(String [] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    // await告诉CyclicBarrier已经达到内存屏障
                    cb.await();
                    System.out.println(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
            }
        }).start();

        try {
            cb.await();
            System.out.println(2);
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

还有更高级的用法

```
// 所有线程到达屏障时，优先执行`barrierAction`
public CyclicBarrier(int parties, Runnable barrierAction) {
        if (parties <= 0) throw new IllegalArgumentException();
        this.parties = parties;
        this.count = parties;
        this.barrierCommand = barrierAction;
}
```

用处是：

多线程计算数据，最后合并计算结果的场景。比如计算用户银行一年的流水。

比较：

和CountDownLatch相比，CyclicBarrier能处理的场景更复杂。

控制并发线程数的semaphore

用来控制访问特定资源的线程数量。

```
package com.study.thread;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * 控制资源的线程并发访问数量
 * Created by guobing on 2016/8/3.
 */
public class SemaphoreTest {
    private static final int count = 30;
    private static ExecutorService threadPool = Executors.newFixedThreadPool(count);

    private static Semaphore s = new Semaphore(10);

    public static void main(String [] args) {
        for(int i = 0; i < count; i++) {
            threadPool.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        // 获取许可证
                        s.acquire();
                        System.out.println("save data");
                        // 释放许可证
                        s.release();
```

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

    threadPool.shutdown();
  }
}
```