javase之多线程

进程线程概念

进程：：操作系统中一个程序的执行周期

线程：进程中的一个任务。一个进程中包含n个线程

1. 进程和线程区别

2.1每个进程拥有拥有自己的一整套变量，是操作系统中资源分配的最小单位，线程依托于进程存在，多个线程共享进程的资源，os中任务调度的基本单位

2.2启动，撤销一个进程的开销要比启动撤销一个线程大得多（线程轻量级）

2.3没有进程就没有线程，进程一旦中止，其内的线程全部撤销

高并发：同一时刻线程的访问量非常非常高

DDos攻击

僵尸设备

1. java多线程实现

2.1继承Thread类实现多线程

Java.lang.Thread是线程操作的核心类JDK1.0提供

新建一个线程最简单的方法就是直接继承Thread类而后覆写类中的run（）方法，（相当于主方法）。run方法是线程的入口

无论哪种方式实现多线程，线程启动一律调用Thread类提供的start方法

Start（）方法解析

首先检查线程状态是否为0，（线程默认状态为0表示未启动）

如果线程已经启动再次调用start方法，会抛出illegalthreadstateexception

一个线程的start()只能调用一次

jvm调用start0方法进行资源分配与系统调度，准备好资源启动线程后回调run()来执行线程的具体任务

2.2Runnable接口实现多线程

1.jiva中多线程的处理就是一个典型的代理模式，其中Thread类完成 资源调度， 系统分配辅助线程业务类，自定义的线程类负责真实实现业务。

(2)使用Runnable接口实现的多线程程序类可以更好地描述资源共享

2.3callable接口实现多线程

Juc：高并发程序包

Java.util.concurrent.Callable<v>;

V call() throws Exception:线程执行后有返回值V

Java.util.Future<v>;

V get() throws interruptedException,

取地Callable接口call方法的返回值

当县城需要返回值时只能采用callable 接口实现多线程。

1. 多线程的常用操作方法

3.1线程的命名与取得

1. 通过构造方法将线程命名

Public Thread(String name)

Public Thread (Runnable target,String name)

1. 设置线程名称

Public final synchronized void setName(String name)

1. 取得线程名称
2. Public final String getName()
3. 取地当前正在执行得线程对象

Public static native Thread currenntThread

主方法是个主线程

java中得main实际上是一个主线程（main）.

3.2线程休眠（sleep）-----单位为毫秒，static native

指得是让线程暂缓执行，等到了预计时间再恢复执行

线程休眠会立即交出cpu,让cpu去执行其他任务，并且线程休眠不会释放对象锁

3.3线程让步（yield）(屈服，让步)static native

暂停当前正在执行得线程对象，并执行其他线程

Yield()会让当前线程交出cpu,但不一定立即交出，yield（）交出cpu后，只能让拥有相同优先级得线程有获取cpu得机会，。Yield（）不会释放对象锁。

从运行状态返回就绪状态。

3.4join方法 final

等待该线程中止，如果在主线程中调用该方法会让主线程修绵，让调用该方法得线程先执行完毕后再恢复执行主线程。

Join（）只是对Object类wait()做了一层包装而已

3.5线程停止

3.5.1手工设置标记位，让线程在满足条件后退出（）推荐

3.5.2使用stop方法强制让线程退出，但是该方法不安全，已经被@Deprecated(过期声明)

3.5.3使用Thread类提供得interrupt()中断线程（就是系统设计了一个标记位，直接使用即可）

interrupt方法只是将线程得状态改为中断状态而已，它不会中断一个正在运行得线程

如果线程调用了wait.()sleep()join()进入阻塞态。调用该线程的interrupt()会抛出interruptedexception,并且将线程interrupt重置为false.

3.6线程优先级

线程的优先级是指，优先级越高，越有可能先执行而已，仅仅是有可能而已

设置优先级

Public final void setPriority(int new Priority)

取得优先级

Publlc final int getPriority()

Thread.MAX\_PRIORITY=10

Thread.NORM\_PRIORITY=5

Thread.MIN\_PRIORITY=1

主线程得优先级是5，中优先级

线程得继承性：优先级可以继承

在A线程中启动B线程，则A和B得优先级一样

3.7守护线程（后台线程）

守护线程是一种特殊得线程，属于陪伴线程

java中两种线程：用户线程，守护线程，isDaemon()

典型得守护线程：垃圾回收线程

只要当前JVM进程中存在任何一个用户线程没有结束，守护线程就一直在工作，只有最后一个用户线程停止后，守护线程会随着JVM进程一同停止

setDaemon()将当前线程设置为守护线程，必须在线程启动之前设置。

Thread1.setDaemon(true);

Thread.currentThread .getName()+””

java中启动得线程默认为用户线程，包括main线程

线程的同步

线程同步：每一个线程对象轮番抢占共享资源带来的问题（eg：厕所）

4.1同步处理

黄牛A卖出了0票

黄牛B卖出了-1票

4.1.1使用synchronizd关键字来处理同步问题。

synchronizd处理同步有两种模式：同步代码块。同步方法

解决这个问题，加锁

1.同步代码块：要使用同步代码块必须要设置一个锁定的对象，一般可以锁当前对象this,当多个线程共享一个对象时锁才有用，如果有多个对象，会各锁各的没有竞争了

Synchronizd（this）{

}

1. 同步方法

在方法上添加Synchronized关键字，表示此方法只有一个线程能进入，隐式锁对像this,

Class MyThread implements Runnable{

Private int ticket=1000;

Public void run(){

For(int i=0;i<1000;i++){

Sale();

}

}

Public Synchronizd void sale(){

If(ticket>0){

System.out.println(Thread.currentThread().getName+”还剩下”+ticket--+”票”)；

}

}

Public class Test{

Public static void main(){

MyThread mt=new MyThread();

Thread thread1=new Thread(mt,name:”黄牛A”);

Thread thread2=new Thread(mt,name:”黄牛B”)；

thread1.start();

thread2.start();

}}

synchronizd对象锁，锁的是对象不是代码，

4.1.2 Synchronizd底层实现

同步代码块：

锁类的实例对象

Synchronizd（this）{ }

锁类对象（class对象）Synchronizd（类名称.class）{}---全局锁

任意实例对象

String lock=””;

Synchronizd(lock){ }

同步方法

普通方法+Synchronizd：锁的是当前对象

静态方法+Synchronizd：

锁的是类---全局锁，效果等同于同步代码块的锁类对像

4.1.3对象锁（monitor）机制---JDK6之前的Synchronized（重量级锁）

同步代码块：

执行同步代码块后首先要执行monitorenter指令，退出时执行monitorexit指令，

使用内建锁（Synchronizd）进行同步，关键在于要获取指定锁对象monitor对象，当线程获取monitor后才能继续下执行，否则只能等待，这个过程是互斥的，即同一时刻只有一个线程能够获取到对象monitor

通常一个monitor指令会包含若干个monitorexit指令，原因在于在JVM需要确保锁在正常执行路径，以及异常执行路径上都能够正确的解锁。

同步方法：

当使用Synchronizd标记方法时，编译后字节码中方法的访问标记多了一个ACC\_SYNCHRONIZED该标记表示，进入该方法时，JVM需要进行monitorenter操作，退出该方法时，无论是否正常返回，JVM均需要进行monitorexit操作。

当执行monitorenter时，如果目标锁对象monitor计数器为0，表示此对象没有被任何其他对象所持有，此时JVM会将该锁对象的持有线程设置为当前线程，并将计数器加1，如果目标锁对象monitor计数器不为0，判断锁对象的持有线程是否是当前线程，如果是，再次将计数器加1，（锁的可重入性），如果锁对象的持有线程不是当前线程，当前线程需要等待，直至持有线程释放锁。

当执行monitorexit时，JVM会将锁对象的计数器-1，当计数器值减为0，代表该锁对象已经被释放。

写代码证明锁的可重入与互斥

Class MyThread implements Runnable{

}

Jdk1.6之前的机制,之后用Lock

ReentrantLock

CAS：compare And Swap(比较交换机制)---乐观锁

悲观锁：假设每一次执行同步代码块均会产生冲突，所以当线程获取锁成功，会阻塞其他尝试获取该锁的线程。（JDK1.6之前的内建锁）

乐观锁：假设所有线程访问共享资源时不会出现冲突，既然不会出现冲突自然就不会阻塞其他线程，线程不会出现阻塞状态。（没有锁的概念）

CAS（无锁操作）

使用CAS 叫做比较交换来判断是否出现冲突，出现冲突就重试当前操作，直到不冲突为止。

1.1CAS 操作过程

一般来说，CAS （V O N）

V:内存中地址存放的实际值

O；预期值（旧值）

N：更新后的值

当执行CAS 后，如果V==O,即内存中旧值与内存中实际值相等，表示上次修改改制后没有任何进程再次修改改制，因此可以将N替换到内存中

V！=O，表示该内存中的值已经被其他线程做了修改，所以无法将N替换，返回最新的值V

当多个线程使用CAS 操作同一个变量时，只有一个线程会成功，并成功更新变量值，其余线程均会失败，失败线程会重新尝试或将线程挂起。（阻塞）

元老级的内建锁（synchronized）最主要的问题：当存在线程竞争的情况下，会出现线程阻塞，以及唤醒带来的性能问题，对应互斥同步（阻塞同步）效率很低

而CAS 并不是武断的将线程挂起，会尝试若干次CAS 操作，并非进行耗时的挂起与唤醒操作，因此非阻塞式同步。

1.2CAS 问题

1.2.1 ABA问题

解决思路：沿用数据库的乐观锁机制，添加版本号1A--2B--3A（）

JDK1.5 提供atomic 包下的AtomicStampedReference

类来解决CAS 的ABA问题

1.2.2自旋会浪费大量的处理器资源（CPU）

与线程阻塞相比，自旋会浪费大量的CPU资源，因为此时线程仍处于运行状态，只不过跑的是无用指令，，期望在无用指令时，锁能够释放出来。

解决思路：自适应自旋

根据以往自旋时等待时能否获取倒锁，来动态调整自旋的时间（循环尝试数量）

如果在上一次自旋时获取倒锁，则此次自选时间稍微长一点，如果在上一次自旋 结束时还没有获取倒锁，此次自选时间稍微短一点。

1.2.3公平性

处于阻塞状态的线程无法立刻竞争被释放的锁，而处于自旋状态的线程很可能先获取到锁

内建锁无法实现公平性

java对象头

JDK1.6之后，对内建锁做了优化，（新增偏向锁，轻量级锁）

锁状态在对象头mark word中

无锁状态0 01

偏量锁1 01

轻量级锁00

重量级锁10

这四种状态随着竞争情况逐渐升级，锁可以升级不能降级，为了提高获得锁与释放锁的效率

2.1偏向锁

最乐观的锁，从始至终只有一个线程请求一把锁

偏向锁获取

当一个线程访问同步代码块并获取锁时，会在对象头和栈帧中的锁记录中记录存储偏向锁的线程ID,以后该线程再次进入同步块时不再需要CAS加锁或解锁，只需简单测试下对象头的mark word中偏向所线程ID是否是当前线程ID，如果成功，表示线程已经获取到锁直接进入代码块运行。

如果测试失败，检查当前偏向锁字段是否为0，如果为0，采用CAS操作将偏向锁字段设置为1，并且更新自己的线程ID到mark word字段中。

如果为1，表示此时偏向所已经被别的线程获取，则此线程不断尝试使用CAS获取偏向锁或者将偏向锁撤销，升级为轻量级锁（升级概率较大）

关于偏向所的撤销

偏向锁使用一种等待竞争出现才释放锁的机制，当有其他线程尝试竞争偏向锁时，持有偏向所的线程才会释放偏向锁

小tip:偏向锁的撤销开销较大，需要等待线程进入全局安全点safepoint（当前线程在CPU上没有执行任何有用字节码）

偏向锁从JDK6之后默认开启，但是它在应用程序启动几秒后才激活，-XX:biasedLockingStartupDelay=0,将延迟关闭，JVM一启动就激活怕偏向所

-XX:-UseBiasedLocking=false,关闭偏向所，程序默认进入轻量级所。

2.2轻量级锁，不停CAS

轻量级锁：多个线程在不同时间段请求同一把锁，也就是基本不存在锁竞争，针对此种状况，JVM采用轻量级锁来避免线程的阻塞以及唤醒。

加锁：线程在执行同步代码块之前，JVM先在当前线程的栈帧中创建用于存储锁记录的空间，并将对象头的mark word 字段直接复制到此空间中，然后线程尝试使用CAS将对象头的mark word替换为指向锁记录的指针（指向当前线程）如果成功表示获取到轻量级所，如果失败,表示其他线程竞争轻量级所，当前线程使用自旋来不断尝试

锁的释放：

解锁时，会使用CAS将复制的mark word替换回对象头，如果成功，表示没有竞争发生，正常解锁，如果失败，表示当前锁存在竞争,进一步膨胀为重量级锁

重量级锁会阻塞。唤醒请求枷锁的线程，针对的是多个线程同一时刻竞争同一把锁的情况，JVM采用自适应自旋，来避免线程在面对非常小的同步快时，仍会被阻塞以及唤醒

轻量级所采用CAS操作，将锁对象的标记字段替换为指向线程的指针，存储着锁对象原本的标记字段，针对的时多个线程在不同时段申请同一把锁的情况

偏向所只会在第一次请求时采用CAS操作，在锁对象的mark word字段中记录下当前线程id，此后运行中持有偏向锁的线程不再有加锁过程。针对的是锁仅会被同一线程持有。（自己独有一个）

1. 生产者与消费者模型

模板：模板类--核心算法--抽象方法延迟到子类实现，--AQS

工厂:

代理：代理类-Thread

单例:

1. 生产-消费者模型
   1. wait（）方法

使得当前线程立刻停止运行，处于等待状态，（WAIT）并将当前线程置入锁对象的等待队列中，直到被通知（notify（））或被终断为止

使用条件，只能在同步方法或同步代码块中实现，必须是内建锁

Wait()调用后立即释放对象锁

Public final void wait() throws InterruptedException

死等，知道被唤醒或中断

Public final native void wait(Long timeout) throws InterruptedException

超时等待，若在规定时间内未被唤醒，线程退出。

单位：毫秒

Public final native void wait(Long timeout，int nanos) throws InterruptedException

在2的基础上增加了纳秒控制

* 1. notify（）

语义：唤醒处于等待状态的线程

使用条件：notify（）也必须在同步方法或同步代码块中调用，用来唤醒等待该对象的其他线程，如果有多个线程等待，随机挑选一个唤醒。

notify（）方法调用后，当前线程不会立马释放对象锁，要等到当前线程执行完毕后再释放对象锁。

notifyall（）

唤醒该对象所有处于等待状态的线程

3.4线程由运行太到阻塞态（wait）

1. 调用sleep(),立即交出CPU，不释放锁
2. 线程调用阻塞式IO（BIO方法）
3. 线程获取锁失败进入阻塞状态
4. 线程调用wait()
5. 线程调用suspend(),将线程挂起

每个锁对象都有两个队列，一个称为同步队列，存储获取锁失败的队列，另有一个称为等待队列，存储调用wait()等待的线程，将县城唤醒实际上是从等待队列调入同步队列。

单个生产者消费者

Class Goods{

Private goodsName;

Privaye int count;

Public Synchronized void setgoods(String goodsname){

//当前还有商品，需要消费者消费

if（count>0）

Wait()

This.goodsname=goodsname;

This.count++;

System,out.println(“生产”+toString())；

notify（）//唤醒消费者；

}

Public Synchronized void getgoods(String goodsname){

//当前没有商品了，等一下

if（count==0）{

Wait()；}

This.count--;

System,out.println(“消费”+toString())；

Notify();//唤醒生产者}

}