### 并发程序

### 一 JMM(java内存模型)

#### 1.1原子性

是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始就不会被别的线程打断。

**public** **class** MutipleLong {

**public** **static** Long *long1*=0l;

**public** MutipleLong() {

// **TODO** Auto-generated constructor stub

}

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

**new** Thread(**new** WriteT(111l)).start();

**new** Thread(**new** WriteT(222l)).start();

**new** Thread(**new** WriteT(333l)).start();

**new** Thread(**new** WriteT(444l)).start();

**new** Thread(**new** ReadeT()).start();

}

}

**class** WriteT **implements** Runnable {

**private** **long** to;

**public** WriteT(**long** longs) {

**this**.to=longs;

}

@Override

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

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

MutipleLong.*long1*=to;

Thread.*yield*();

}

}

}

**class** ReadeT **implements** Runnable{

@Override

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

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

**long** sys=MutipleLong.*long1*;

**if**(sys!=111l&&sys!=222l&&sys!=333l&&sys!=444l){

System.***out***.println(sys);

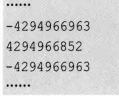
}

}

}

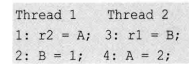
}

如下代码片段所示，如果是32的操作系统你会发现。读线程出现了数据，然而并不是你期待的结果，为什么。因为32位的操作系统每次读写都是以32个byte作为操作单元的，long是64位的数据类型，那么就意味着单个线程要读两次，然而如果无法保证写线程的原子性的话，边写边读就有可能只读到了一半。然而在64位的操作系统中就不会发生这样的问题。



#### 1.2可见性

可见性是指当一个线程修改了变量的值，其他线程是否立马能知道这个修改。对于串行的程序来说是没有这样的问题的。修改以后，再去读取变量的值。自然是变量的新值。在并行程序中就不一定了。如果某个线程修改了一个全局变量，其他线程是否立马能知道这个修改，显然在并行程序中就不一定了。如果CPU1与CPU2同时操作一个类变量t，CPU1把这个变量读取到寄存器或者缓存中。然而CPU2操作这个全局变量t将它修改成其他的值。对于CPU1来说是不可见的,CPU1还在操作着寄存器或者缓存。



比如如上所示的代码，你觉得会不会出现一种可能就是r2=A r1=1。上面的这组指令，看似没有问题，当然顺序执行的话，是不可能发生上面的情况的，不过如果指令重排的话就不一定不会产生这种情况了。

#### 1.3有序性

有序性问题是很相对比较难理解的，可能我们早已经习惯了，代码是从上往下顺序执行的。然而事实在真实的环境下，是存在多线程并发执行的，这个时候程序的执行就未必是从上到下的了。有序性问题的原因是程序在运行的时候可能会进行指令重排，重排后的指令与原指令未必一致。

**public** **class** Demo2 {

**boolean** flag=**false**;

**public** **static** **int** *a*=1;

**public** Demo2() {

}

**public** **void** writer(){

*a*=10;

flag=**true**;

}

**public** **void** reader(){

**if**(flag){

**if**(*a*==1){

System.***out***.println("存在指令重排");

}

}

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Demo2 demo2=**new** Demo2();

ExecutorService executorService=Executors.*newCachedThreadPool*();

**for**(**int** i=0;i<1000000;i++){

executorService.execute(**new** ThreadA(demo2));

}

**for**(**int** j=0;j<1000000;j++){

executorService.execute(**new** ThreadB(demo2));

}

}

}

**class** ThreadA **implements** Runnable{

**private** Demo2 demo2=**null**;

**public** ThreadA(Demo2 demo2) {

**this**.demo2=demo2;

}

@Override

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

demo2.writer();

}

}

**class** ThreadB **implements** Runnable{

**private** Demo2 demo2=**null**;

**public** ThreadB(Demo2 demo2) {

**this**.demo2=demo2;

}

@Override

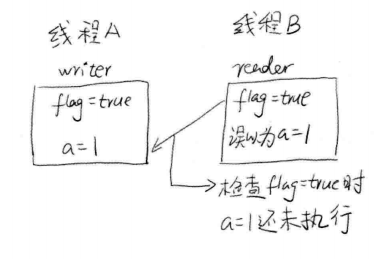
**public** **void** run() {

demo2.reader();

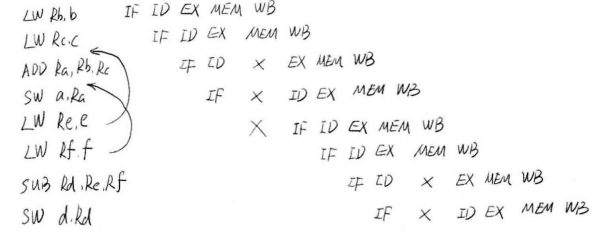
}

}

据说如上代码所示，存在如下图所示的指令重排现象。意思就是说对于线程B来说，线程A中的指令执行，未必是有序的。不过遗憾的是我测试的代码并没有发生指令重排现象。所示这段代码是否会指令重排是不一定的,不过计算机的工作原理当然是存在的。



计算机当中往往把一些



对后续无影响的指令，进行乱序执行，所以才会出现指令重排的现象。

#### 1.4 happen before原则

程序顺序原则: 一个线程内保证语义的串行性。

Volatile:volitile变量的写，发生在读之前，这就保证了volatile变量的可见性。

锁规则:解锁必然发生在随后的加锁之前。

传递:A先与B，B先于C,那么A必然先于C

线程:start方法先于每一个动作

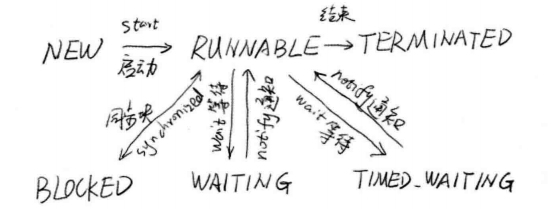
线程操作先于线程终结Thread.join();

线程中断先于被中断的代码

对象的构造函数执行，结束先于finalize();

### 二 线程基础

#### 2.1线程生命周期



如上图所示是线程的生命周期，线程start()是线程的启动状态，等到线程启动以后，这个时候线程处于运行状态，如果线程碰到了synchronied那么线就会处于Blocked状态，如果线程因为某种原因就会出现wait()状态进行永远的等待或者出现tme waitting状态进行限时等待，等待别人的线程唤醒以后又回到了runnable状态。

#### 2.2创建线程的方式

**public** **class** Demo3 {

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

**new** ThreadD().start();

**new** Thread(**new** ThreadC()).start();

}

}

**class** ThreadD **extends** Thread{

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

System.***out***.println("线程ThreadD-------");

}

}

**class** ThreadC **implements** Runnable{

@Override

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

System.***out***.println("线程ThreadC-------");

}

}

#### 2.3停止线程

**public** **class** ReadAndWriteThread {

**private** **static** User *u*=**new** User();

//读线程

**static** **class** ReadThread **extends** Thread{

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

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

**synchronized** (*u*) {

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

**if**(*u*.getAge()!=**null**&&*u*.getName()!=**null**){

**if**(!*u*.getAge().equals(*u*.getName())){

System.***out***.println(*u*.toString());

}

}

}

}

Thread.*yield*();

}

}

}

//写线程

**static** **class** WriteThread **extends** Thread{

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

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

**synchronized** (*u*) {

**long** time=System.*currentTimeMillis*()/1000;

*u*.setName(time+"");

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

*u*.setAge(time+"");

}

Thread.*yield*();

}

}

}

**static** **class** User{

**private** String name;

**private** String age;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getAge() {

**return** age;

}

**public** **void** setAge(String age) {

**this**.age = age;

}

@Override

**public** String toString() {

**return** "User [name=" + name + ", age=" + age + "]";

}

}

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

ReadThread readThread=**new** ReadThread();

readThread.start();

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

WriteThread writeThread = **new** WriteThread();

writeThread.start();

**try** {

Thread.*sleep*(150);

writeThread.~~stop~~();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

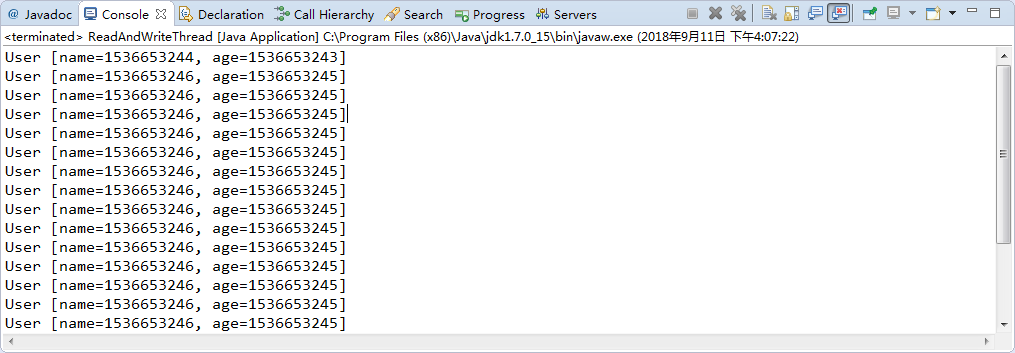
}

}

}

}

如上代码所示，其实到jdk高版本以后，大家可以看到线程的stop方法是已经被禁止了的。这其实是一种不对的终止线程的方式。



如上代码所示终止线程，容易出现数据不一致的问题。那我们要如何去停止一个线程呢。

//写线程

**static** **class** WriteThread **extends** Thread{

**boolean** isStopme=**false**;

**public** **void** stopMe(**boolean** stopMe){

**this**.isStopme=stopMe;

}

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

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

**if**(isStopme){

**break**;

}

**synchronized** (*u*) {

**long** time=System.*currentTimeMillis*()/1000;

*u*.setName(time+"");

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

*u*.setAge(time+"");

}

Thread.*yield*();

}

}

}

如上代码所示，我们设置标志位,优雅的结束掉这个线程。

Thread.*currentThread*().isInterrupted() 判断当前线程是否被中断

Thread.*interrupted*(); 判断当前线程是否被中断，并清除当前中断状态

threadInterrupted.interrupt(); 中断当前线程

**public** **class** ThreadInterrupted **extends** Thread{

@Override

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

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

System.***out***.println(System.*currentTimeMillis*());

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

Thread.*interrupted*();

}

**if**(Thread.*currentThread*().isInterrupted()){

**break**;

}

}

}

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

ThreadInterrupted threadInterrupted=**new** ThreadInterrupted();

threadInterrupted.start();

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

threadInterrupted.interrupt();

}

}

#### 2.4 wait()与notify()

**public** **class** ThreadInfo{

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

ExecutorService executorService=Executors.*newCachedThreadPool*();

Object object=**new** Object();

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

executorService.execute(**new** WaitThread(object));

}

**try** {

Thread.*sleep*(1000);

notifyThread notifyThread=**new** notifyThread(object);

notifyThread.start();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**class** WaitThread **extends** Thread{

**private** Object obj=**new** Object();

**public** WaitThread(Object ob) {

**this**.obj=ob;

}

@Override

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

**synchronized** (obj) {

System.***out***.println("等待-------------"+Thread.*currentThread*().getName());

**try** {

obj.wait();//等待以后会释放锁

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("唤醒--------------"+Thread.*currentThread*().getName());

}

}

}

**class** notifyThread **extends** Thread{

**private** Object obj=**new** Object();

**public** notifyThread(Object ob) {

**this**.obj=ob;

}

@Override

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

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

**synchronized** (obj) {

System.***out***.println("一次唤醒一个线程");

obj.notify();

**try** {

Thread.*sleep*(3000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

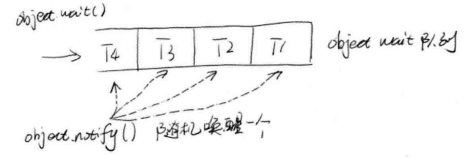
}

}

}



上图为如上代码所示的执行结果，可以看到wait()方法实际上在它等待的时候是释放锁的。也就是说其他的线程进来以后依然会被锁住。然而对于notify方法来说它一次却只能唤醒一个随机的线程。Wait(),notify()方法都是对象的方法。



如上图所示，wait与notify方法满足，如上等待队列所示的等待与唤醒机制。

#### 2.5 wait()与notifyAll()

**public** **class** ThreadInfo{

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

ExecutorService executorService=Executors.*newCachedThreadPool*();

Object object=**new** Object();

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

executorService.execute(**new** WaitThread(object));

}

NotifyAllThread notifyAllThread=**new** NotifyAllThread(object);

notifyAllThread.start();

}

}

**class** WaitThread **extends** Thread{

**private** Object obj=**new** Object();

**public** WaitThread(Object ob) {

**this**.obj=ob;

}

@Override

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

**synchronized** (obj) {

System.***out***.println("等待-------------"+Thread.*currentThread*().getName());

**try** {

obj.wait();//等待以后会释放锁

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("唤醒--------------"+Thread.*currentThread*().getName());

}

}

}

**class** NotifyAllThread **extends** Thread{

**private** Object obj=**new** Object();

**public** NotifyAllThread(Object ob) {

**this**.obj=ob;

}

@Override

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

**synchronized** (obj) {

System.***out***.println("一次唤醒所有线程");

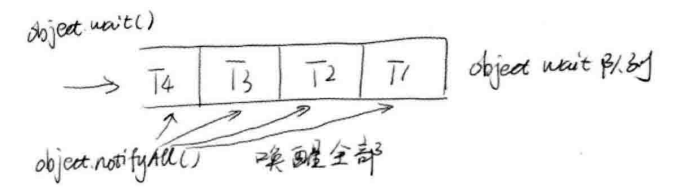
obj.notifyAll();

}

}

}

如上代码所示是线程通信的第二种方式，唤醒所有的线程。



如上图所示描述了这个等待与唤醒的过程。在这里要强调一下的是wait()与sleep()都是等待的方法，然而wait()是可以被唤醒的，并且唤醒以后还能获取到锁。但是sleep()是不能被唤醒的。

#### 2.6 join()与yield()

Join() 线程等待

Join(long xx) 线程等待，指定时间

如上图所示两个方法，一个是让当前线程一直阻塞等待，直到目标线程执行完。另一个是让目标线程等待指定的秒数，目标线程再执行。

**public** **class** Thread\_Join\_demo7 **extends** Thread{

**private** **static** AtomicInteger *ato*=**new** AtomicInteger();

@Override

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

**while**(*ato*.get()<1000000){

*ato*.incrementAndGet();

}

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Thread\_Join\_demo7 threads=**new** Thread\_Join\_demo7();

threads.start();

//threads.join();

System.***out***.println(*ato*.get());

}

}

如上代码所示，原本我们期待的值是1000000但是由于在主线程中，程序是正常执行的，不会等待，所以这里可能输出的值是一个小于1000000的任意值，如果使用join()的话，就会让当前主线程等待其他线程执行完，以后再输出结果。就是1000000。如果我们打开注释以后，就能获取到我们预想的那个值。

Yield()线程让步

这是一个静态方法，一旦执行，它会使线程让出CPU.让出CPU以后，并不代表不在争夺CPU资源。是否能够再分配就不一定了。如果一个线程不是很重要，优先级相对比较低。又担心它占用大量CPU资源，就可以调用yield()方法来进行适当的线程让步。

#### 2.7线程组

**public** **class** ThreadG {

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

ThreadGroup threadGroup=**new** ThreadGroup("ftpServer");

Thread th=**new** Thread(threadGroup,**new** Thread1(),"线程一");

Thread th1=**new** Thread(threadGroup,**new** Thread2(),"线程二");

th.start();

th1.start();

System.***out***.println("xxx==="+threadGroup.activeCount());

threadGroup.list();

}

}

**class** Thread1 **implements** Runnable{

@Override

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

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

}

}

**class** Thread2 **implements** Runnable{

@Override

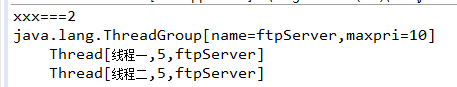
**public** **void** run() {

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

}

}

如上所示是线程组的概念，如上所示



执行以后得到如上图所示的结果。这个是在告诉我们通过线程组可以

#### 2.8线程优先级

**public** **class** ThreadPropriorty {

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

HighThread highThread=**new** HighThread();

MidieThread midieThread=**new** MidieThread();

LowThread lowThread=**new** LowThread();

lowThread.setPriority(Thread.***MAX\_PRIORITY***);

midieThread.setPriority(Thread.***NORM\_PRIORITY***);

highThread.setPriority(Thread.***MIN\_PRIORITY***);

lowThread.start();

midieThread.start();

highThread.start();

}

}

**class** HighThread **extends** Thread{

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

**for**(**int** i=0;i<1000;i++){

System.***out***.println("HighThread"+Thread.*currentThread*().getName());

}

}

}

**class** MidieThread **extends** Thread{

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

**for**(**int** i=0;i<1000;i++){

System.***out***.println("MidieThread"+Thread.*currentThread*().getName());

}

}

}

**class** LowThread **extends** Thread{

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

**for**(**int** i=0;i<1000;i++){

System.***out***.println("LowThread"+Thread.*currentThread*().getName());

}

}

}

如上面的代码所示，会产生这样的现象，线程优先级低的线程总是在最后执行完成的。这也证实了一点，线程的优先级决定了这个线程拿到CPU的概率。虽然线程命名的时候已经命名好了高低优先级，然而最终还是由设置的优先级来决定线程的情况。