**一、锁**

**锁是用来解决线程同步问题的一种方式。**

**1.可重入锁**

如果锁具备可重入性，则称作为可重入锁。重入性是锁的分配机制，基于线程的分配，而不是基于方法调用的分配。

代码实例：

private void printHello() {

lock.lock();

System.out.print("Hello,");

printWorld();

lock.unlock();

}

private void printWorld() {

lock.lock();

System.out.print("world.");

lock.unlock();

}

/\*\*

\* private Lock lock = new UnReentrantLock();

\* 不可重入锁，执行结果：

\* Hello,

\*

\* private Lock lock = new ReentrantLock();

\* 可重入锁，执行结果：

\* Hello,world.

\* Process finished with exit code 0

\*/

public static void main(String[] args) {

ReentrantLockTest test = new ReentrantLockTest();

test.printHello();

}

**2.读写锁**

读写锁将对一个资源（比如文件）的访问分成了2个锁，一个读锁和一个写锁。

正因为有了读写锁，才使得多个线程之间的读操作不会发生冲突。

ReadWriteLock就是读写锁，它是一个接口，ReentrantReadWriteLock实现了这个接口。

可以通过readLock()获取读锁，通过writeLock()获取写锁。

读读共享，读写互斥，写读互斥，写写互斥。

代码实例：

/\*\*

\* 执行读操作

\*/

private void read(Thread thread) {

try {

lock.lock();

int i = 0;

while (i ++ < 5) {

Thread.sleep(10);

System.out.println(thread.getName() + "正在执行读操作");

}

System.out.println(thread.getName()+"读操作完毕");

} catch (Exception e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

/\*\*

\* private Lock lock = new ReentrantLock();

\* 非读写锁，执行结果：

\* Thread-0正在执行读操作

\* Thread-0正在执行读操作

\* Thread-0正在执行读操作

\* Thread-0正在执行读操作

\* Thread-0正在执行读操作

\* Thread-0读操作完毕

\* Thread-1正在执行读操作

\* Thread-1正在执行读操作

\* Thread-1正在执行读操作

\* Thread-1正在执行读操作

\* Thread-1正在执行读操作

\* Thread-1读操作完毕

\*

\* Process finished with exit code 0

\*

\* private Lock lock = new ReentrantReadWriteLock().readLock();

\* 读写锁，执行结果：

\* Thread-0正在执行读操作

\* Thread-0正在执行读操作

\* Thread-1正在执行读操作

\* Thread-0正在执行读操作

\* Thread-1正在执行读操作

\* Thread-0正在执行读操作

\* Thread-1正在执行读操作

\* Thread-0正在执行读操作

\* Thread-0读操作完毕

\* Thread-1正在执行读操作

\* Thread-1正在执行读操作

\* Thread-1读操作完毕

\*

\* Process finished with exit code 0

\*

\*/

public static void main(String[] args) {

final ReentrantReadWriteLockTest test = new ReentrantReadWriteLockTest();

new Thread() {

@Override

public void run() {

test.read(Thread.currentThread());

}

}.start();

new Thread() {

@Override

public void run() {

test.read(Thread.currentThread());

}

}.start();

}

**3.公平锁**

公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。

非公平锁即无法保证锁的获取是按照请求锁的顺序进行的。这样就可能导致某个或者一些线程永远获取不到锁。

而对于ReentrantLock和ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。

public ReentrantReadWriteLock(boolean fair)

**4.可中断锁**

可中断锁：顾名思义，就是可以相应中断的锁。

如果某一线程A正在执行锁中的代码，另一线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了，想先处理其他事情，可以让它中断自己或者在别的线程中中断它，这种就是可中断锁。可通过lock.lockInterruptibly()获取可中断锁。

**二、闭锁 CountDownLatch**

一家人一起吃个饭，通知大家下班去饭店集合。假设：3个人在不同的地方上班，必须等到3个人到场才能吃饭，用程序如何实现呢？

public class Test1

{

/\*\*

\* 模拟爸爸去饭店

\*/

public static void fatherToRes()

{

System.out.println("爸爸步行去饭店需要3小时。");

}

/\*\*

\* 模拟我去饭店

\*/

public static void motherToRes()

{

System.out.println("妈妈挤公交去饭店需要2小时。");

}

/\*\*

\* 模拟妈妈去饭店

\*/

public static void meToRes()

{

System.out.println("我乘地铁去饭店需要1小时。");

}

/\*\*

\* 模拟一家人到齐了

\*/

public static void togetherToEat()

{

System.out.println("一家人到齐了，开始吃饭");

}

public static void main(String[] args)

{

new Thread()

{

public void run()

{

fatherToRes();

};

}.start();

new Thread()

{

public void run()

{

motherToRes();

};

}.start();

new Thread()

{

public void run()

{

meToRes();

};

}.start();

togetherToEat();

}

}

结果：

一家人到齐了，开始吃饭

我乘地铁去饭店需要1小时。

妈妈挤公交去饭店需要2小时。

爸爸步行去饭店需要3小时。

一个人都没到，就开始吃饭了。。。。。。

用CountLatchDown来实现：

private static CountDownLatch latch = new CountDownLatch(3);

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

{

new Thread()

{

public void run()

{

fatherToRes();

latch.countDown();

};

}.start();

new Thread()

{

public void run()

{

motherToRes();

latch.countDown();

};

}.start();

new Thread()

{

public void run()

{

meToRes();

latch.countDown();

};

}.start();

latch.await();

togetherToEat();

}

结果：

我乘地铁去饭店需要1小时。

妈妈挤公交去饭店需要2小时。

爸爸步行去饭店需要3小时。

一家人到齐了，开始吃饭

使用了CountDowmLatch 实现了我们的需求。

Latch闭锁的意思，是一种同步的工具类。类似于一扇门：在闭锁到达结束状态之前，这扇门一直是关闭着的，不允许任何线程通过，当到达结束状态时，这扇门会打开并允许所有的线程通过。且当门打开了，就永远保持打开状态。

作用：可以用来确保某些活动直到其他活动都完成后才继续执行。

使用场景：

1、例如我们上例中所有人都到达饭店然后吃饭；

2、某个操作需要的资源初始化完毕

3、某个服务依赖的线程全部开启等等...

CountDowmLatch是一种灵活的闭锁实现，包含一个计数器，该计算器初始化为一个正数，表示需要等待事件的数量。countDown方法递减计数器，表示有一个事件发生，而await方法等待计数器到达0，表示所有需要等待的事情都已经完成。

**三、关卡 CyclicBarrier**

CyclicBarrier是一个同步辅助类，它允许一组线程相互等待直到所有线程都到达一个公共的屏障点。

在程序中有固定数量的线程，这些线程有时候必须等待彼此，这种情况下，使用CyclicBarrier很有帮助。

这个屏障之所以用循环修饰，是因为在所有的线程释放彼此之后，这个屏障是可以重新使用的。

假若有若干个线程都要进行写数据操作，并且只有所有线程都完成写数据操作之后，这些线程才能继续做后面的事情，此时就可以利用CyclicBarrier了：

public class Test {

public static void main(String[] args) {

int N = 4;

CyclicBarrier barrier = new CyclicBarrier(N);

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

new Writer(barrier).start();

}

static class Writer extends Thread{

private CyclicBarrier cyclicBarrier;

public Writer(CyclicBarrier cyclicBarrier) {

this.cyclicBarrier = cyclicBarrier;

}

@Override

public void run() {

System.out.println("线程"+Thread.currentThread().getName()+"正在写入数据...");

try {

Thread.sleep(5000); //以睡眠来模拟写入数据操作

System.out.println("线程"+Thread.currentThread().getName()+"写入数据完毕，等待其他线程写入完毕");

cyclicBarrier.await();

} catch (InterruptedException e) {

e.printStackTrace();

}catch(BrokenBarrierException e){

e.printStackTrace();

}

System.out.println("所有线程写入完毕，继续处理其他任务...");

}

}

}

执行结果：

线程Thread-0正在写入数据...

线程Thread-3正在写入数据...

线程Thread-2正在写入数据...

线程Thread-1正在写入数据...

线程Thread-2写入数据完毕，等待其他线程写入完毕

线程Thread-0写入数据完毕，等待其他线程写入完毕

线程Thread-3写入数据完毕，等待其他线程写入完毕

线程Thread-1写入数据完毕，等待其他线程写入完毕

所有线程写入完毕，继续处理其他任务...

所有线程写入完毕，继续处理其他任务...

所有线程写入完毕，继续处理其他任务...

所有线程写入完毕，继续处理其他任务...

**四、信号量 Semaphore**

Semaphore翻译成字面意思为 信号量，Semaphore可以控同时访问的线程个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。

acquire()用来获取一个许可，若无许可能够获得，则会一直等待，直到获得许可。

release()用来释放许可。

假若一个工厂有5台机器，但是有8个工人，一台机器同时只能被一个工人使用，只有使用完了，其他工人才能继续使用。那么我们就可以通过Semaphore来实现：

public class Test {

public static void main(String[] args) {

int N = 8; //工人数

Semaphore semaphore = new Semaphore(5); //机器数目

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

new Worker(i,semaphore).start();

}

static class Worker extends Thread{

private int num;

private Semaphore semaphore;

public Worker(int num,Semaphore semaphore){

this.num = num;

this.semaphore = semaphore;

}

@Override

public void run() {

try {

semaphore.acquire();

System.out.println("工人"+this.num+"占用一个机器在生产...");

Thread.sleep(2000);

System.out.println("工人"+this.num+"释放出机器");

semaphore.release();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

执行结果：

工人0占用一个机器在生产...

工人1占用一个机器在生产...

工人2占用一个机器在生产...

工人4占用一个机器在生产...

工人5占用一个机器在生产...

工人0释放出机器

工人2释放出机器

工人3占用一个机器在生产...

工人7占用一个机器在生产...

工人4释放出机器

工人5释放出机器

工人1释放出机器

工人6占用一个机器在生产...

工人3释放出机器

工人7释放出机器

工人6释放出机器

下面对上面说的三个辅助类进行一个总结：

1）CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：

CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；

而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；

另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

2）Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限。

**五、线程安全**

线程安全：经常用来描绘一段代码。指在并发的情况之下，该代码经过多线程使用，线程的调度顺序不影响任何结果。这个时候使用多线程，我们只需要关注系统的内存，cpu是不是够用即可。反过来，线程不安全就意味着线程的调度顺序会影响最终结果，如不加事务的转账代码：

void transferMoney(User from, User to, float amount){

to.setMoney(to.getBalance() + amount);

from.setMoney(from.getBalance() - amount);

}

同步：Java中的同步指的是通过人为的控制和调度，保证共享资源的多线程访问成为线程安全，来保证结果的准确。如上面的代码简单加入synchronized关键字。

**可见性：**

可见性是值一个线程对共享变量的修改，对于另一个线程来说是否是可以看到的。线程操作的是自己的工作内存，而不会直接操作主内存。如果线程对变量的操作没有刷写会主内存的话，仅仅改变了自己的工作内存的变量的副本，那么对于其他线程来说是不可见的。而如果另一个变量没有读取主内存中的新的值，而是使用旧的值的话，同样的也可以列为不可见。

**原子性：**

原子性是指操作是不可分的。其表现在于对于共享变量的某些操作，应该是不可分的，必须连续完成。例如a++，对于共享变量a的操作，实际上会执行三个步骤，1.读取变量a的值  2.a的值+1  3.将值赋予变量a 。 这三个操作中任何一个操作过程中，a的值被人篡改，那么都会出现我们不希望出现的结果。所以我们必须保证这是原子性的。Java中的Atomic类和锁的机制解决了原子性的问题。

**有序性：**

有序性是指程序在执行的时候，程序的代码执行顺序和语句的顺序是一致的。

在Java内存模型中，允许编译器和处理器对指令进行重排序，但重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

举个例子：

线程A:

context = loadContext();

inited = true;

线程B:

while(!inited ){

sleep

}

doSomethingwithconfig(context);

如果线程A发生了重排序：

inited = true;

context = loadContext();

那么线程B就会拿到一个未初始化的content去配置，从而引起错误。

因为这个重排序对于线程A来说是不会影响线程A的正确性的，而如果loadContext()方法被阻塞了，为了增加Cpu的利用率，这个重排序是可能的。

一系列操作如果无法保证happens-before原则，就说明这段操作无法保证有序性。

happens-before原则：

1）程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作。

2）锁定规则：一个unLock操作先行发生于后面对同一个锁的Lock()操作。

3）volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作。

4）传递规则：如果操作A先行发生与操作B,而操作B先行发生于操作C，则操作A先行发生于操作C。

5）线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作。

6）线程终端规则：对线程interrupt()方法的调用先行发生与被中断线程的代码检测到中断事件的发生（只有执行了interrupt()方法才可以检测到中断事件的发生）。

7）线程终结规则：线程中所有操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束，Thread.isAlive()的返回值手段检测到线程已经终止执行。

8）对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始。

1.原子性：Atomic类、synchronized、Lock

2.可见性：synchronized、volatile

3.有序性：happens-before原则