java并发总结（一）

一.什么是线程？什么是进程？线程和进程的区别？

1. 线程：

   线程是操作系统能够进行运算调度的最小单位，是CPU调度和分派的基本单位，是进程中实际的运作单位，另外，线程又被称为轻量级进程，程序员可以通过它进行多处理器编程。

2. 进程：

   进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

3. 线程和进程的区别：

   - 进程和线程都是独立的执行路径，但是一个进程可以有多个线程。
   - 每个进程都有自己的内存空间、可执行代码和唯一进程标识符，而每个线程在Java中都有自己的堆栈，但它使用进程主存并与其他线程共享。
   - 线程在操作系统中也称为任务或轻量级进程
   - 来自同一进程的线程可以使用Java中的wait,notify,notifyAll等编程语言结构进行通信，比进程间通信简单得多。
   - Java中进程和线程的另一个区别是线程和进程是如何创建的。与需要复制父进程的进程相比，创建线程很容易。
   - 属于同一进程的所有线程共享系统资源，如文件描述符、堆内存等资源，但每个线程在Java中都有自己的异常处理程序和堆栈。

二.执行start()和run()的区别

如果是调用start方法执行，那么就会创建一个新的线程去执行run()方法。如果是直接调用了run()方法，就是默认当前线程执行了run()方法，不会创建新的线程。

```
package concurrent;

import java.util.concurrent.TimeUnit;

public class StartVSRunDemo {
    public static void main(String[] args) {
        StartVSRun t = new StartVSRun();
        System.out.println(Thread.currentThread().getName() + " Calling start method !
");
        t.start();
        try {
            TimeUnit.SECONDS.sleep(3);
        }catch (InterruptedException e){
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " Calling run method ! ");
        t.run();
```

```
    }
}

class StartVSRun extends Thread{
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is executed run method !
");
    }
}
/**
 * output:
 * main Calling start method !
 * Thread-0 is executed run method !
 * main Calling run method !
 * main is executed run method !
 */
```

从上述代码的执行结果，我们可以看出，调用start()方法，main线程里是创建了一个名为Thread-0的线程，去执行了run()方法，而直接调用了run()方法，main线程不会创建新的线程，而是直接在main线程里执行了run()方法。

三.Runnable和Callable的区别?

- Runnable接口是从JDK 1.0开始的，而Callable是在Java 5.0上添加的。
- Runnable接口使用run()方法定义任务，Callable接口使用call()方法定义任务。
- 方法不返回任何值，它的返回类型为void，而call方法返回值。可调用接口是泛型参数化接口，在创建可调用实现的实例时提供值类型。
- run和call方法的另一个区别是run方法不能抛出checked exception，而call方法在Java中可以抛出checked exception。

四.sleep(),yield(),wait()的区别?

1. sleep()和yield()的相同点
    - sleep()和yield()都是java.lang.Thread的静态方法，用于阻塞当前线程。
    - sleep()和yield()是用于从当前线程中释放CPU资源，但是不会释放当前线程所持有的锁。
2. sleep()和yield()的不同点
    - sleep()有两个重载的方法sleep(long milliseond)和sleep(long millis, int nanos)。
    - sleep()比yield()更加稳定，因为sleep()方法不会导致正在执行的线程被阻塞时去放弃任何监视器。
    - 如果中断了正在sleep的线程，sleep()会抛出InterruptedExcepiton异常，但是yield不会。
3. sleep()和wait()的不同点。

    wait()?

    wait()是Object类中定义的，所有的对象都可以使用，并且与notify,notifyAll方法一起使用，实现线程之间的通信。

    wait()通过判断条件，可以使线程处于等待状态，比如在生产者消费者模式中，生产者在队满的时候应该等待，消费者应该在队空的时候等待。

    notify()方法用于唤醒正在等待的线程，生产者在队空的队列里放入了数据，就可通知消费者来消费了。

- wait()方法必须被声明在synchronized或者block 代码块里，如果实现同步，就会抛出IllegalMonitorStateException 异常，但是sleep就不需要同步。
- wait()方法操作的是object对象，sleep操作的是当前线程。
- wait()方法被调用了，会释放锁，但是sleep()只是释放cpu资源，阻塞当前线程，但不会释放当前线程所持有的所有锁。
- wait()方法一般都是放在循环里调用的，只要队空，消费者就一直等待。但是sleep最好不要放在循环里。
- wait()方法是Object类的本地方法，但是sleep是Thread的静态方法。

四.notify和notifyAll的区别?

notify()和notifyAll()方法的主要区别是，如果Java中有多个线程在等待任何锁，notify方法只向一个等待的线程发送通知，但是不能保证通知哪个线程，而notifyAll通知所有等待该锁的线程。

```
package concurrent;

import java.util.logging.Level;
import java.util.logging.Logger;

public class NotifyWaitDemo {

    private volatile boolean go = false;

    public static void main(String args[]) throws InterruptedException {
        final NotifyWaitDemo test = new NotifyWaitDemo();

        Runnable waitTask = new Runnable(){

            @Override
            public void run(){
                try {
                    test.shouldGo();
                } catch (InterruptedException ex) {
                    Logger.getLogger(NotifyWaitDemo.class.getName()).
                            log(Level.SEVERE, null, ex);
                }
                System.out.println(Thread.currentThread() + " finished Execution");
            }
        };

        Runnable notifyTask = new Runnable(){

            @Override
            public void run(){
                test.go();
                System.out.println(Thread.currentThread() + " finished Execution");
            }
        };

        Thread t1 = new Thread(waitTask, "WT1"); //will wait
        Thread t2 = new Thread(waitTask, "WT2"); //will wait
        Thread t3 = new Thread(waitTask, "WT3"); //will wait
        Thread t4 = new Thread(notifyTask,"NT1"); //will notify
```

```
        //starting all waiting thread
        t1.start();
        t2.start();
        t3.start();

        //pause to ensure all waiting thread started successfully
        Thread.sleep(200);

        //starting notifying thread
        t4.start();

    }
    private synchronized void shouldGo() throws InterruptedException {
        while(go != true){
            System.out.println(Thread.currentThread()
                    + " is going to wait on this object");
            wait();
            System.out.println(Thread.currentThread() + " is woken up");
        }
        go = false;
    }


    private synchronized void go() {
        while (go == false){
            System.out.println(Thread.currentThread()
                    + " is going to notify all or one thread waiting on this object");

            go = true;
            //notify();
            notifyAll();
        }

    }
}
```

五.wait()和join()的区别

- wait()和join()方法之间最重要的区别是，wait()在Object中声明和定义，而join()在Thread类中定义。
- wait()必须在同步块中，否则它将抛出IllegalMonitorStateException，但在Java中调用join()方法没有这样的要求。
- 线程调用wait()方法，它释放wait()对象持有的任何锁，wait()在对象上被调用，但是调用join()方法不会释放任何监视器或锁，join()被线程调用。
- 它们之间的另一个区别是wait()用于线程间通信，join()用于在多个线程之间添加排序，例如，一个线程在第一个线程执行完后开始执行。
- 可以通过使用notify()和notifyAll()方法调用对象类的wait()方法来唤醒正在等待的线程，但是不能不间断地中断join施加的等待，或者除非调用join的线程已经执行完毕。

```
public class JoinThread extends Thread{
```

```
    JoinThread(String name){
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println("join - " + getName() + " " + i);
        }
    }

    public static void main(String[] args) throws InterruptedException{
        for (int i = 0; i < 100; i++) {
            if(i == 20){
                JoinThread jt = new JoinThread(" 被join的线程 ");
                jt.start();
                //main线程调用了jt线程的join方法，main线程必须等jt执行完了，才能继续执行
                jt.join();
            }
            System.out.println(Thread.currentThread().getName()+ " " + i);
        }
    }
}
```

六.countDownLatch()和CyclicBarrier()有什么区别?

```
import java.util.concurrent.CountDownLatch;

public class CountDownLatchDemo {
    public static void main(String args[]) throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(4);
        Worker first = new Worker(1000, latch, "WORKER-1");
        Worker second = new Worker(2000, latch, "WORKER-2");
        Worker third = new Worker(3000, latch, "WORKER-3");
        Worker fourth = new Worker(4000, latch, "WORKER-4");

        // Main thread will wait until all thread finished
        //latch.await();
        System.out.println(latch.getCount());
        System.out.println(Thread.currentThread().getName() + " has finished");
        fourth.start();
        third.start();
        second.start();
        first.start();
    }
}

class Worker extends Thread {
    private int delay;
    private CountDownLatch latch;

    public Worker(int delay, CountDownLatch latch, String name) {
```

```
        super(name);
        this.delay = delay;
        this.latch = latch;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(delay);
            latch.countDown();
            System.out.println(latch.getCount());
            System.out.println(Thread.currentThread().getName() + " has finished");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

/**
 * 4
 * main has finished
 * 3
 * WORKER-1 has finished
 * 2
 * WORKER-2 has finished
 * 1
 * WORKER-3 has finished
 * 0
 * WORKER-4 has finished
 */

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.logging.Level;
import java.util.logging.Logger;

public class CyclicBarrierDemo {
    //Runnable task for each thread
    private static class Task implements Runnable {

        private CyclicBarrier barrier;

        public Task(CyclicBarrier barrier) {
            this.barrier = barrier;
        }

        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + " is waiting on
barrier");
                barrier.await();
```

```
                System.out.println(Thread.currentThread().getName() + " has crossed the
barrier");
            } catch (InterruptedException ex) {
                Logger.getLogger(CyclicBarrierDemo.class.getName()).log(Level.SEVERE,
null, ex);
            } catch (BrokenBarrierException ex) {
                Logger.getLogger(CyclicBarrierDemo.class.getName()).log(Level.SEVERE,
null, ex);
            }
        }
    }

    public static void main(String args[]) {

        //creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()
        final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){
            @Override
            public void run(){
                //This task will be executed once all thread reaches barrier
                System.out.println("All parties are arrived at barrier, lets play");
            }
        });

        //starting each of thread
        Thread t1 = new Thread(new Task(cb), "Thread 1");
        Thread t2 = new Thread(new Task(cb), "Thread 2");
        Thread t3 = new Thread(new Task(cb), "Thread 3");

        t1.start();
        t2.start();
        t3.start();



    }

}
```

CyclicBarrier 和 CountDownLatch 都可以用来让一组线程等待其它线程。与 CyclicBarrier 不同的是，CountdownLatch 不能重新使用.

七.interrupted 和 isInterruptedd方法的区别?

interrupted() 和 *isInterrupted()*的主要区别是前者会将中断状态清除而后者不会。Java多线程的中断机制是用内部标识来实现的，调用*Thread.interrupt()*来中断一个线程就会设置中断标识为true。当中断线程调用静态方法*Thread.interrupted()*来检查中断状态时，中断状态会被清零。而非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有有可能被其它线程调用中断来改变。

八.synchronized方法和代码块有什么区别?

- synchronized方法与代码块的一个显著区别是，synchronized块通常会缩小锁的范围。由于锁的作用域与性能成反比，所以最好只锁定代码的关键部分。使用synchronized块的一个最好的例子是在单例模式中进行双重检查锁定。
- 同步块提供了对锁的粒度控制，因为您可以使用任意的锁来提供临界段代码的互斥。另一方面，同步方法总是锁定由这个关键字表示的当前对象或类级锁(如果它的静态同步方法)。
- 在同步方法的情况下，锁在进入方法时被线程获取，离开方法时被线程释放，或者正常释放，或者抛出异常释放。另一方面，对于同步块，线程在进入同步块时获取锁，离开同步块时释放锁。

九.synchronized和ReentrantLock的区别

synchronized是java的内建同步机制，它提供了互斥的语义和可见性，当一个线程已经获取到锁时，其他线程在尝试获取锁的时候只能等待或者阻塞。

ReentrantLock是java5提供的锁实现，是可重入锁。它的语义和synchronized基本相同，可以通过调用lock()方法，可以获取锁，但是一定要最后释放锁资源，调用unlock()方法即可，这样是代码更加灵活，并且提供了很多实用的方法，这些synchronized未必能够做到，比如，ReentrantLock可以控制公平性，但是synchronized就不能，ReentrantLock也可以利用条件控制线程之间的通信。

synchronized和ReentrantLock的性能不能一概而论，synchronized在java5之后得到了很大的性能优化。在低竞争资源的情况下，性能甚至会优于ReentrantLock。

ReentrantLock相比于synchronized，可以像普通对象一样使用，所以可以利用其提供的各种遍历方法，进行精细的同步操作，甚至是实现synchronized难以表达的用例。

如：

- 带超时的获取锁尝试
- 可以判断是否有线程，或者是某个特定的线程，在排队等待获取锁。
- 可以响应中断请求。
- 可以灵活运用Condition