### JAVA第五阶段—DAY03-JAVA案例

1. volatile修饰变量，改变多线程并发中变量的可见性

* 示例代码

package org.hopu.djp.libDemo.jucTest;

public class Test2 extends Thread {

// boolean flag = false;

volatile boolean flag = false;

int i = 0;

public void run() {

while (!flag) {

i++;

}

}

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

Test2 vt = new Test2();

vt.start();

Thread.sleep(1000);

vt.flag = true;

while(true) {

System.out.println("stop:" + vt.i);

Thread.sleep(1000);

}

}

}

注意：未被 volatile 修饰的变量，主线程中被修改后，不会马上同步到子线程中。而被volatile修饰的变量，主线程中一旦被修改，马上就会同步到子线程中。

1. 原子变量

public class Test3 {

class AtomicDemo implements Runnable{

// private volatile int serialNumber = 0;

private AtomicInteger serialNumber = new AtomicInteger(0);

@Override

public void run() {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

}

System.out.println(getSerialNumber());

}

public int getSerialNumber(){

// return serialNumber++;

return serialNumber.getAndIncrement(); // 原子执行自增1操作

}

}

public void method1() {

AtomicDemo ad = new AtomicDemo();

for (int i = 0; i < 100; i++) {

new Thread(ad).start();

}

}

public static void main(String[] args) {

Test3 t3 = new Test3();

t3.method1();

}

}​

注意：这个例子中，使用普通变量或者volatile变量，都比较容易出现最后累加不到99的情况。这是由于子线程中休眠0.1秒，这期间可能有其他线程读取变量值累加，导致并发结果不对。如果把累加变量声明为原子变量，则可以保证正确运行结果。

1. 并发性容器

* 示例代码

/\*\*

\* CopyOnWriteArrayList/CopyOnWriteArraySet : “写入并复制”

\* 注意：添加操作多时，效率低，因为每次添加时都会进行复制，开销非常的大。并发迭代操作多时可以选择。

\*/

public class Test4 {

class HelloThread implements Runnable{

// private List<String> list = Collections.synchronizedList(new ArrayList<String>());

private CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();

// private List<String> list = new ArrayList<String>();

public HelloThread() {

list.add("AA");

list.add("BB");

list.add("CC");

}

@Override

public void run() {

Iterator<String> it = list.iterator();

while(it.hasNext()){

System.out.println(Thread.currentThread().getName() +":" + it.next());

list.add("AA");

}

}

}

public void go() {

HelloThread ht = new HelloThread();

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

new Thread(ht).start();

}

}

public static void main(String[] args) {

Test4 t4 = new Test4();

t4.go();

}

}

注意：使用传统ArrayList，将会由于并发问题导致异常抛出，而使用CopyOnWriteArrayList，可以正常运行。

* 示例代码

public class Test5 {

private int threadNum = 100;

private CountDownLatch startGate;

private CountDownLatch endGate;

public void go() {

startGate = new CountDownLatch(1);

endGate = new CountDownLatch(threadNum);

long start = System.currentTimeMillis();

//所有阻塞的任务同时开始

// startGate.countDown();

HelloTable ht = new HelloTable();

for(int i=0; i<threadNum; i++) {

new Thread(ht).start();

}

try {

//主线程阻塞,等待其他所有 worker 线程完成后再执行

// endGate.await();

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

long end = System.currentTimeMillis();

System.out.println("总用时: " + (end - start) + "ms");

}

public static void main(String[] args) {

Test5 t = new Test5();

t.go();

}

class HelloTable implements Runnable {

private HashMap<String, Object> table = new HashMap<String, Object>();

// private Hashtable<String, Object> table = new Hashtable<String, Object>();

// private ConcurrentHashMap<String, Object> table = new ConcurrentHashMap<String, Object>();

@Override

public void run() {

// System.out.println("线程"+Thread.currentThread().getName()+"开始运行。");

try {

long l1 = System.currentTimeMillis();

// startGate.await();

Thread.sleep(10);

for(int i=0; i<100000; i++) {

double x = new Random().nextDouble();

// System.out.println(Thread.currentThread().getName() + ":" + String.valueOf(x) + ":");

table.put(String.valueOf(x), x);

}

long l2 = System.currentTimeMillis();

System.out.println("线程"+Thread.currentThread().getName()+"运行时间：" + (l2 - l1));

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

// endGate.countDown();

}

}

}

}

注意：此案例中，使用HashMap线程不安全容器，会导致并发异常出现，使用ConcurrentHashMap这种线程安全容器可以正常运行。

1. CountDownLatch闭锁

* 示例代码

public class Test6 {

class latchDemo implements Runnable {

private CountDownLatch latch;

private int x = 0;

public latchDemo(CountDownLatch latch) {

this.latch = latch;

}

@Override

public void run() {

try {

long start = System.currentTimeMillis();

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

x += 2;

}

String name = Thread.currentThread().getName();

System.out.println(name + ":x=" + x);

long end = System.currentTimeMillis();

System.out.println(name + "耗费时间为：" + (end - start) + "ms");

} finally {

latch.countDown(); // 必须的操作，表示结束

}

}

}

public void go() {

int latchSize = 10;

CountDownLatch latch = new CountDownLatch(latchSize); // 5表示有5个线程

latchDemo l = new latchDemo(latch);

long start = System.currentTimeMillis();

for(int i=0; i<5; i++) {

new Thread(l).start();

}

try {

latch.await(); // 主进程等待，直到计算达到latchSize数量的子进程执行latch.countDown

} catch (InterruptedException e) {

e.printStackTrace();

}

long end = System.currentTimeMillis();

System.out.println("总耗费时间为：" + (end - start) + "ms");

}

public static void main(String[] args) {

Test6 t = new Test6();

t.go();

}

}

注意：此案例中，使用HashMap线程不安全容器，会导致并发异常出现，使用ConcurrentHashMap这种线程安全容器可以正常运行。

1. CountDownLatch闭锁

* 示例代码

public class Test6 {

class latchDemo implements Runnable {

private CountDownLatch latch;

private int x = 0;

public latchDemo(CountDownLatch latch) {

this.latch = latch;

}

@Override

public void run() {

try {

long start = System.currentTimeMillis();

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

x += 2;

}

String name = Thread.currentThread().getName();

System.out.println(name + ":x=" + x);

long end = System.currentTimeMillis();

System.out.println(name + "耗费时间为：" + (end - start) + "ms");

} finally {

latch.countDown(); // 必须的操作，表示结束

}

}

}

public void go() {

int latchSize = 10;

CountDownLatch latch = new CountDownLatch(latchSize); // 5表示有5个线程

latchDemo l = new latchDemo(latch);

long start = System.currentTimeMillis();

for(int i=0; i<5; i++) {

new Thread(l).start();

}

try {

latch.await(); // 主进程等待，直到计算达到latchSize数量的子进程执行latch.countDown

} catch (InterruptedException e) {

e.printStackTrace();

}

long end = System.currentTimeMillis();

System.out.println("总耗费时间为：" + (end - start) + "ms");

}

public static void main(String[] args) {

Test6 t = new Test6();

t.go();

}

}

注意：当latchSize比实际启动的子线程多时，主进程会等待countDown达标，但是实际不可能达到，所以主进程会一直等待下去。

当latchSize比实际启动的子线程少时，主进程会在countDown达标时，直接停止等待，唤起执行，而不是等待所有子线程结束。

1. Callable 接口

* 示例代码

public class Test7 {

/\*\*

\* 一、创建执行线程的方式三：实现Callable接口。相较于实现Runnable接口的方式，方法可以有返回值，并且可以抛出异常

\* 二、执行Callable方式，需要FutureTask实现类的支持，用于接受运算结果。FutureTask是Future接口的实现类

\*/

class ThreadDemo implements Callable<Integer> {

private int sum = 0;

@Override

public Integer call() throws Exception {

System.out.println(Thread.currentThread().getName() + "线程开始");

for (int i = 0; i <= 100; i++) {

sum += i;

}

System.out.println(Thread.currentThread().getName() + "线程结束");

return sum;

}

}

public void go() {

ThreadDemo td = new ThreadDemo();

List<FutureTask<Integer>> list1 = new ArrayList<FutureTask<Integer>>();

List<Thread> list2 = new ArrayList<Thread>();

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

// 1.执行Callable方式，需要FutureTask实现类的支持，用于接受运算结果

FutureTask<Integer> result = new FutureTask<>(td);

Thread t1 = new Thread(result);

t1.start();

list1.add(result);

list2.add(t1);

}

for(int i=0; i<list1.size(); i++) {

// 2.接收线程运算后的结果

try {

Integer sum = list1.get(i).get(); // FutureTask可用于闭锁

System.out.println(list2.get(i).getName() + ".result=" + sum);

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

}

}

public static void main(String[] args) {

Test7 t7 = new Test7();

t7.go();

}

}

注意：Callable 需要依赖FutureTask，并可以提供返回值，在FutureTask调用get方法获取返回值时，默认线程会被阻塞，直到接受到返回值。也可以指定超时时间，若超过时限仍未返回，就直接返回null。

1. Lock 同步锁

* 示例代码

public class Test8 {

class Ticket implements Runnable{

private Integer tick = 100;

private Lock lock = new ReentrantLock();

@Override

public void run() {

while(true){

lock.lock(); //上锁

synchronized(tick) {

try {

boolean b = false;

if (tick > 0) {

try {

Thread.sleep(200);

} catch (InterruptedException e) {

}

System.out.println(Thread.currentThread().getName() + " 完成售票，余票为：" + --tick);

} else {

break;

}

} finally {

lock.unlock(); //释放锁

}

}

}

}

}

public void go() {

Ticket ticket = new Ticket();

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

new Thread(ticket, i+"号窗口").start();

}

}

public static void main(String[] args) {

Test8 t8 = new Test8();

t8.go();

}

}

注意：lock上锁以后，其他线程的加锁请求会被阻塞，直到unlock，其他线程才能开始竞争锁资源。

1. Condition 控制线程通信

* 示例代码

public class Test9 {

public static ReentrantLock lock = new ReentrantLock();

public static Condition condition = lock.newCondition();

class CondationDemo implements Runnable {

@Override

public void run() {

try {

lock.lock();

System.out.println("Thread is await");

condition.await();

System.out.println("Thread is going on");

} catch (Exception e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

}

public void go() throws InterruptedException {

CondationDemo c1 = new CondationDemo();

Thread t1 = new Thread(c1);

t1.start();

Thread.sleep(2000);

// 通知线程t1继续执行

lock.lock();

condition.signal();

lock.unlock();

System.out.println("go on.");

}

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

Test9 t9 = new Test9();

t9.go();

}

}

注意：在t1执行的过程中，调用了await()，进入等待并释放锁，然后主线程中调用了signal()，让t1继续执行。

1. Condition 控制线程交替执行

* 示例代码

public class Test10 {

class AlternateDemo {

private int number = 1; // 当前正在执行的线程标记

private Lock lock = new ReentrantLock();

private Condition condition1 = lock.newCondition();

private Condition condition2 = lock.newCondition();

private Condition condition3 = lock.newCondition();

public void loopA() {

lock.lock();

try {

// 1.判断

if (number != 1) {

condition1.await();

}

// 2.打印

System.out.print(Thread.currentThread().getName());

// 3.唤醒

number = 2;

condition2.signal();

} catch (Exception e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

public void loopB() {

lock.lock();

try {

if (number != 2) {

try {

condition2.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.print(Thread.currentThread().getName());

number = 3;

condition3.signal();

} finally {

lock.unlock();

}

}

public void loopC() {

lock.lock();

try {

if (number != 3) {

try {

condition3.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.print(Thread.currentThread().getName());

number = 1;

condition1.signal();

} finally {

lock.unlock();

}

}

}

public void go() {

AlternateDemo ad = new AlternateDemo();

new Thread(new Runnable() {

@Override

public void run() {

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

ad.loopA();

}

}

}, "A").start();

new Thread(new Runnable() {

@Override

public void run() {

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

ad.loopB();

}

}

}, "B").start();

new Thread(new Runnable() {

@Override

public void run() {

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

ad.loopC();

}

}

}, "C").start();

}

public static void main(String[] agrs) {

Test10 t10 = new Test10();

t10.go();

}

}

注意：通过为每个线程分配不同的Condition，可以实现指定线程的启动和终止。

1. Semaphore信号量

* 示例代码

public class Test11 {

static Semaphore semaphore = new Semaphore(2);

public class semaphoreDemo implements Runnable {

@Override

public void run() {

Thread thread = Thread.currentThread();

try {

semaphore.acquire();

System.out.println(System.currentTimeMillis() + "," + thread.getName() + ",获取许可!");

TimeUnit.SECONDS.sleep(3);

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

semaphore.release();

System.out.println(System.currentTimeMillis() + "," + thread.getName() + ",释放许可!");

}

}

}

public void go() {

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

new Thread(new semaphoreDemo()).start();

}

}

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

Test11 t11 = new Test11();

t11.go();

}

}

注意：此案例中，限定许可只有2个，10个线程启动后，只能有2个线程获取到许可后执行，其他线程处于阻塞状态。线程执行完成后释放许可，随后其他线程竞争许可。

1. ReadWrite读写锁

* 示例代码

public class Test12 {

class ReadWriteLockDemo implements Runnable {

private int number = 0;

public ReadWriteLockDemo(int number) {

this.number = number;

}

private ReadWriteLock lock = new ReentrantReadWriteLock();

public void get() {

lock.readLock().lock();//加锁

try {

System.out.println(Thread.currentThread().getName() + " : " + number);

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.readLock().unlock();//解锁

}

}

public void set(int number) {

lock.writeLock().lock();//加锁

try {

System.out.println(Thread.currentThread().getName());

this.number = number;

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.writeLock().unlock();//解锁

}

}

@Override

public void run() {

this.get();

// this.set((int)(Math.random() \* 101));

}

}

public void go() {

ReadWriteLockDemo rw = new ReadWriteLockDemo(12);

for (int i = 0; i < 100; i++) {

new Thread(rw).start();

}

}

public static void main(String[] args) {

Test12 t12 = new Test12();

t12.go();

}

}

注意：readLock的加锁，不会影响到其他线程加读锁的操作，而writeLock加锁，会导致其他线程加写锁操作被阻塞。

1. synchronized同步锁

* 示例代码

public class Test13 implements Runnable {

public static synchronized void method1() {

String name = Thread.currentThread().getName();

System.out.println(name + ".method1 begin.");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(name + ".method1 end.");

}

public static synchronized void method2() {

String name = Thread.currentThread().getName();

System.out.println(name + ".method2 begin.");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(name + ".method2 end.");

}

@Override

public void run() {

String name = Thread.currentThread().getName();

int i = Integer.valueOf(name.substring(name.length() - 1));

if (i % 2 == 0) {

method1();

} else {

method2();

}

}

public static void main(String[] args) {

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

Test13 t13 = new Test13();

new Thread(t13).start();

}

}

}

注意：所有的非静态同步方法用的都是同一把锁——实例对象本身，也就是说如果一个实例对象的非静态同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获取锁的方法释放锁后才能获取锁，可是别的实例对象的非静态同步方法因为跟该实例对象的非静态同步方法用的是不同的锁，所以毋须等待该实例对象已获取锁的非静态同步方法释放锁就可以获取他们自己的锁。

所有的静态同步方法用的也是同一把锁——类对象本身，这两把锁是两个不同的对象，所以静态同步方法与非静态同步方法之间是不会有竞态条件的。但是一旦一个静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取锁，而不管是同一个实例对象的静态同步方法之间，还是不同的实例对象的静态同步方法之间，只要它们同一个类的实例对象！

1. 线程池与线程调度

* 示例代码

public class Test15 {

class taskDemo implements Runnable {

private int taskNum;

public taskDemo(int taskNum) {

System.out.println(Thread.currentThread().getName() + " : taskNum=" + taskNum + " init.");

this.taskNum = taskNum;

}

@Override

public void run() {

System.out.println(Thread.currentThread().getName() + " : taskNum=" + taskNum + " begin.");

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " : taskNum=" + taskNum + " end.");

}

}

// 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

public void go1() throws ExecutionException, InterruptedException {

int poolSize = 5;

ExecutorService pool = Executors.newFixedThreadPool(poolSize);

for (int i = 0; i < poolSize\*3; i++) {

taskDemo task = new taskDemo(i);

pool.submit(task);

}

pool.shutdown();

}

// 可安排在给定的延迟后运行或定期执行的命令。

public void go2() {

int poolSize = 5;

ScheduledExecutorService pool = Executors.newScheduledThreadPool(poolSize);

for (int i = 0; i < poolSize\*3; i++) {

taskDemo task = new taskDemo(i);

// 任务将在延迟2秒后启动

pool.schedule(task, 2, TimeUnit.SECONDS);

}

pool.shutdown();

}

// 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

public void go3() {

int poolSize = 8;

ExecutorService pool = Executors.newCachedThreadPool();

for(int j=0; j<3; j++) {

for (int i = 0; i < poolSize; i++) {

taskDemo task = new taskDemo(i);

//

pool.submit(task);

}

try {

System.out.println("主线程休眠");

Thread.sleep(10 \* 1000);

System.out.println("主线程唤醒");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

pool.shutdown();

}

// 单任务线程池可以确保任务按照提交的顺序被执行

public void go4() {

ExecutorService pool = Executors.newSingleThreadExecutor();

int index = 1;

for(int j=0; j<3; j++) {

for (int i = 0; i < 2; i++) {

taskDemo task = new taskDemo(index++);

pool.submit(task);

}

}

pool.shutdown();

}

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

Test15 t15 = new Test15();

t15.go4();

}

}

注意：

* Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回收）
* Executors.newFixedThreadPool(int)（固定大小线程池）
* Executors.newSingleThreadExecutor()（单个后台线程）
* Executors.newScheduledThreadPool(int) （可以指定大小，并按指定的延迟时间与周期定时执行）

1. ForkJoinPool 分支/合并框架

* 示例代码

public class Test16 {

class ForkJoinSumCalculate extends RecursiveTask<Long> {

/\*\*

\*

\*/

private static final long serialVersionUID = -259195479995561737L;

private long start;

private long end;

private static final long THURSHOLD = 10000L; //临界值

public ForkJoinSumCalculate(long start, long end) {

this.start = start;

this.end = end;

}

@Override

protected Long compute() {

// System.out.println("doTask for " + start +" to " + end);

long length = end - start;

if(length <= THURSHOLD){

long sum = 0L;

for (long i = start; i <= end; i++) {

sum += i;

}

return sum;

}else{

long middle = (start + end) / 2;

ForkJoinSumCalculate left = new ForkJoinSumCalculate(start, middle);

left.fork(); //进行拆分，同时压入线程队列

ForkJoinSumCalculate right = new ForkJoinSumCalculate(middle+1, end);

right.fork(); //

return left.join() + right.join();

}

}

}

public void test1(){

Instant start = Instant.now();

long sum = 0L;

for (long i = 0L; i <= 50000000000L; i++) {

sum += i;

}

System.out.println(sum);

Instant end = Instant.now();

System.out.println("耗费时间为：" + Duration.between(start, end).toMillis());//35-3142-15704

}

public void test2() {

Instant start = Instant.now();

ForkJoinPool pool = new ForkJoinPool();

ForkJoinTask<Long> task = new ForkJoinSumCalculate(0L, 50000000000L);

Long sum = pool.invoke(task);

System.out.println(sum);

Instant end = Instant.now();

System.out.println("耗费时间为：" + Duration.between(start, end).toMillis());//166-1996-10590

}

public static void main(String[] args) {

Test16 t16 = new Test16();

t16.test2();

}

}

注意：当执行新的任务时它可以将其拆分分成更小的任务执行，并将小任务加到线程队列中，然后再从一个随机线程的队列中偷一个并把它放在自己的队列中。

相对于一般的线程池实现，fork/join框架的优势体现在对其中包含的任务的处理方式上.在一般的线程池中，如果一个线程正在执行的任务由于某些原因无法继续运行，那么该线程会处于等待状态。而在fork/join框架实现中，如果某个子问题由于等待另外一个子问题的完成而无法继续运行。那么处理该子问题的线程会主动寻找其他尚未运行的子问题来执行.这种方式减少了线程的等待时间，提高了性能。