### JAVA第五阶段—DAY02-JAVA案例

1. 线程的实现

掌握继承Thread、实现Runnable和实现Callable实现线程的区别

/\*\*

\* 自定义线程类

\* @author xray

\*

\*/

public class MyThread extends Thread{

/\*\*

\* 执行线程任务的方法

\*/

public void run(){

//Thread.currentThread()是获得系统当前执行的线程

System.out.println(Thread.currentThread().getName()+"线程执行了！！！");

}

public static void main(String[] args) {

//主线程中执行

System.out.println(Thread.currentThread().getName()+"线程执行了！！！");

//创建线程对象

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

//启动线程

thread1.start();

thread2.start();

}

}

/\*\*

\* 自定义线程类

\* @author xray

\*

\*/

public class MyRunnable implements Runnable{

/\*\*

\* 实现run方法

\*/

@Override

public void run() {

System.out.println("当前执行的线程是："+Thread.currentThread().getName());

}

public static void main(String[] args) {

//创建Thread对象，传入Runnable对象

Thread thread1 = new Thread(new MyRunnable());

//调用start方法

thread1.start();

//使用匿名内部类实现Runnable

Thread thread2 = new Thread(new Runnable(){

@Override

public void run() {

System.out.println("匿名内部类，当前执行的线程是："+Thread.currentThread().getName());

}

});

thread2.start();

//使用Lambda实现Runnable

Thread thread3 = new Thread(()->{

System.out.println("Lambda，当前执行的线程是："+Thread.currentThread().getName());

});

thread3.start();

}

}

1. 银行转账问题

带学生分析线程安全出现的原因，解决方案

/\*\*

\* 银行类

\* @author xray

\*

\*/

public class Bank {

//模拟100个账户的余额

int[] accounts = new int[100];

//初始化每个余额为1000

{

for(int i = 0;i < accounts.length;i++){

accounts[i] = 1000;

}

}

/\*\*

\* 统计总余额

\* @return

\*/

public int getTotalAccounts(){

int sum = 0;

for(int i = 0;i < accounts.length;i++){

sum += accounts[i];

}

return sum;

}

/\*\*

\* 转账

\* @param from 转出账户

\* @param to 转入账户

\* @param money 金额

\*/

public void transfer(int from,int to,int money){

//转出金额

accounts[from] -= money;

System.out.printf("从%d转出%d到%d%n",from,money,to);

//转入金额

accounts[to] += money;

//输出总金额

System.out.println("银行的总金额是："+getTotalAccounts());

}

}

public class TestBank {

public static void main(String[] args) {

Random random = new Random();

Bank bank = new Bank();

//模拟100次转账

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

//每次转账用新线程执行

new Thread(()->{

//产生转出和转入账户的下标

int from = random.nextInt(bank.accounts.length);

int to = random.nextInt(bank.accounts.length);

//随机金额

int money = 100+random.nextInt(500);

//转账

bank.transfer(from, to, money);

}).start();

}

}

}

1. 线程安全解决方案

对比同步代码块、同步方法、同步锁的区别

同步代码块

public void transfer(int from,int to,int money){

synchronized(this){

//转出金额

accounts[from] -= money;

System.out.printf("从%d转出%d到%d%n",from,money,to);

//转入金额

accounts[to] += money;

//输出总金额

System.out.println("银行的总金额是："+getTotalAccount());

}

}

同步方法

public synchronized void transfer(int from,int to,int money){

//转出金额

accounts[from] -= money;

System.out.printf("从%d转出%d到%d%n",from,money,to);

//转入金额

accounts[to] += money;

//输出总金额

System.out.println("银行的总金额是："+getTotalAccount());

}

同步锁

//同步锁对象

private ReentrantLock lock = new ReentrantLock();

public void transfer(int from,int to,int money){

//同步锁上锁

lock.lock();

try{

//转出金额

accounts[from] -= money;

System.out.printf("从%d转出%d到%d%n",from,money,to);

//转入金额

accounts[to] += money;

//输出总金额

System.out.println("银行的总金额是："+getTotalAccount());

}finally{

//释放锁

lock.unlock();

}

}

1. ThreadLocal

演示ThreadLocal是每个线程独立变量的特点

//定义普通的变量

private static int num1 = 0;

//定义线程局部变量

private static ThreadLocal<Integer> local = new ThreadLocal<Integer>(){

//重写initialValue方法，如果不重写初始值为null

protected Integer initialValue() {

//返回初始值

return new Integer(0);

}

};

public static void main(String[] args) {

//创建十个线程

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

new Thread(new Runnable(){

@Override

public void run() {

num1++;//分别修改普通变量

System.out.println("普通变量="+num1);

//修改线程局部变量的值

local.set(local.get() + 1);

System.out.println("线程局部变量="+local.get());

}

}).start();

}

}

1. 原子类

从变量自增问题引出自增的三个步骤，再引出自增的原子操作，总结CAS过程。

下面代码，启动10000个线程中对count执行自增,执行多次后发现，最后count的值出现9999或9997的结果。

static int count = 0;  
public static void main(String[] args) {  
 for(int i = 0;i < 10000;i++){  
 new Thread(()->{count++;}).start();  
 }  
try {  
 Thread.sleep(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
}  
System.out.println("count = " + count);  
}

原因分析： count++的操作分为三步：

1. 读取count的值
2. 计算count+1的值
3. 把新值存入count

java.util.concurrent包中的原子类就应用了CAS机制，以AtomicInteger（原子整数）为例。使用AtomicInteger的自增方法incrementAndGet无论执行多少次，结果都是10000

static AtomicInteger at = new AtomicInteger(0);

public static void main(String[] args) {

for(int i = 0;i < 10000;i++){

new Thread(()->{at.incrementAndGet();}).start();

}

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("count = " + at.get());

}

1. 生产者和消费者

以卖包子的过程为例，掌握使用线程的等待和通知实现生产者和消费者模式

/\*\*

\* 包子

\*

\*/

public class Baozi {

//包子编号

private int id;

public Baozi(int id){

this.id = id;

}

public String toString(){

return "包子#"+id;

}

}

/\*\*

\* 包子仓库

\* @author xray

\*

\*/

public class BaoziStore {

//最大数量

public static final int MAX\_NUM = 100;

//包子集合

private LinkedList<Baozi> baozis = new LinkedList<>();

//锁对象

private Object lock = new Object();

/\*\*

\* 做包子

\*/

public void makeBaozi(){

//使用同步块

synchronized (lock) {

//判断包子数量是否达到最大值

if(baozis.size() == MAX\_NUM){

System.out.println("仓库满了，生产者等待："+Thread.currentThread().getName());

try {

//让生产者等待

lock.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}else{

//如果仓库还有空间，让生产者继续生产

System.out.println("仓库有空了，生产者继续做包子："+Thread.currentThread().getName());

lock.notifyAll();

}

//创建包子，添加到集合中

Baozi baozi = new Baozi(baozis.size());

baozis.push(baozi);

System.out.println(Thread.currentThread().getName()+"生产者，做了"+baozi);

}

}

/\*\*

\* 吃包子

\*/

public void takeBaozi(){

synchronized (lock) {

//判断如果仓库为空，就让消费者等待

if(baozis.size() == 0){

System.out.println("仓库空了，消费者等一下："+Thread.currentThread().getName());

try {

lock.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}else{

System.out.println("仓库有包子了，消费者快来："+Thread.currentThread().getName());

//如果仓库有包子，就通知消费者来吃

lock.notifyAll();

}

//从栈顶拿包子，然后删除掉

if(baozis.size() > 0){

Baozi baozi = baozis.pop();

System.out.println(Thread.currentThread().getName()+"消费者吃掉了"+baozi);

}

}

}

}

public class TestBaoziStore {

public static void main(String[] args) {

//仓库对象

BaoziStore store = new BaoziStore();

//创建生产者线程，生产100个包子

Thread productor = new Thread(()->{

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

store.makeBaozi();

}

});

//启动生产者

productor.start();

//创建10个消费者，每个吃10个包子

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

Thread consumer = new Thread(()->{

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

store.takeBaozi();

}

});

consumer.start();

}

}

}

1. 阻塞队列

将生产者和消费者模式改为阻塞队列实现，掌握阻塞队列的基本使用

public class TestBaozi {

public static void main(String[] args) {

//创建链表阻塞队列

LinkedBlockingQueue<Baozi> baozis = new LinkedBlockingQueue<>(10);

//创建生产者线程，生产100个包子

Thread productor = new Thread(()->{

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

try {

//创建包子，存入集合

Baozi baozi = new Baozi(i);

baozis.put(baozi);

System.out.println(Thread.currentThread().getName()+"生产了"+baozi);

} catch (Exception e) {

e.printStackTrace();

}

}

});

productor.setName("生产者");

productor.start();

//创建5个消费者，每个吃20个包子

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

Thread consumer = new Thread(()->{

for(int j = 0;j < 20;j++){

try {

//从集合拿包子

Baozi baozi = baozis.take();

System.out.println(Thread.currentThread().getName()+"吃掉了"+baozi);

} catch (Exception e) {

e.printStackTrace();

}

}

});

consumer.setName("消费者"+i);

consumer.start();

}

}

}

1. 线程池

使用Executors类的几种方法和自定义方式实现线程池

/\*\*

\* 测试长度不限的线程池

\*/

public static void testCachedThreadPool(){

//获得长度不限的线程池

ExecutorService pool = Executors.newCachedThreadPool();

//测试执行100个任务

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

final int n = i;

//使用线程池启动线程

pool.execute(()->{

System.out.println(Thread.currentThread().getName()+"执行了！ i --> "+n);

});

try {

Thread.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

//关闭线程池

pool.shutdown();

}

/\*\*

\* 测试长度固定的线程池

\*/

public static void testFixedThreadPool(){

//固定长度线程池

ExecutorService pool = Executors.newFixedThreadPool(20);

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

final int n = i;

//使用线程池启动线程

pool.execute(()->{

System.out.println(Thread.currentThread().getName()+"执行了！ i --> "+n);

});

}

//关闭线程池

pool.shutdown();

}

/\*\*

\* 测试长度单一的线程池

\*/

public static void testSingleThreadPool(){

//单一长度线程池

ExecutorService pool = Executors.newSingleThreadExecutor();

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

final int n = i;

//使用线程池启动线程

pool.execute(()->{

System.out.println(Thread.currentThread().getName()+"执行了！ i --> "+n);

});

}

//关闭线程池

pool.shutdown();

}

/\*\*

\* 测试可以调度的线程池

\*/

public static void testScheduledThreadPool(){

//可调度线程池

ScheduledExecutorService pool = Executors.newScheduledThreadPool(10);

//执行线程,1、线程任务 2、初始的延迟数 3、周期数 4、时间单位

pool.scheduleAtFixedRate(()->{

System.out.println(Thread.currentThread().getName()+"执行了，时间："+new Date());

}, 2, 1, TimeUnit.SECONDS);

}