Lock锁

2018年7月23日 23:16

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/*
 * JDK5后提供了新的锁对象Lock
 * void lock():获取锁
 * void unlock():释放锁
 * ReentrantLock是Lock接口的实现类
 */
public class SellTicket implements Runnable {
	private int tickets = 100;

	// 定义锁对象
	private Lock lock = new ReentrantLock();

	@Override
	public void run() {
		while (true) {
			// 加锁
			lock.lock();
			try {
				if (tickets > 0) {
					try {
						Thread.sleep(100);
					} catch (InterruptedException e) {
						// TODO Auto-generated catch block
						e.printStackTrace();
					}

					System.out.println(Thread.currentThread().getName() + "正
					在售第" + (tickets--) + "张票");
				}
			} finally {
				// 释放锁
				lock.unlock();
			}
```

```
            }
        }

    }
```

死锁问题

2018年7月27日 11:09

同步的弊端：
效率低
如果出现了同步嵌套，就容易产生死锁问题

死锁问题：
两个或两个以上的线程在争夺资源的过程中，发生的一种相互等代的现象

```
public class Mylock {
    //创建两把所对象
    public static final Object objA = new Object();
    public static final Object objB = new Object();
}

public class DieLock extends Thread {
    private boolean flag;

    public DieLock(boolean flag) {
        this.flag = flag;
    }
    @Override
    public void run() {
        if(flag) {
            synchronized (Mylock.objA) {
                System.out.println("if objA");
                synchronized (Mylock.objB) {
                    System.out.println("if objB");
                }
            }
        }else {
            synchronized (Mylock.objB) {
                System.out.println("else objB");
                synchronized (Mylock.objA) {
                    System.out.println("else objA");
                }
            }
        }
    }
}
```

```
/*
 * 死锁问题
 */
public class DieLockDemo {
    public static void main(String[] args) {
        DieLock d11 = new DieLock(true);
        DieLock d12 = new DieLock(false);

        d11.start();
        d12.start();
    }
}
```

生产者消费者案例

2018年7月27日 11:29

线程间的通信问题：
不同种类的线程间针对同一个资源的操作

等代唤醒机制

```
public class Student {
	String name;
	int age;
	boolean flag;//默认情况是没有数据，如果是true，证明有数据
}


/*
 * 资源类：学生类
 * 设置学生数据：SetThread(生产者)
 * 获取学生数据：GetThread(消费者)
 * 测试类：StudentDemo
 *
 * 在外界把这个数据创建出来，通过构造方法传递给其它的类
 * 同一个数据出现多次:
 *		CPU的一点点时间片就可执行多条语句
 * 姓名和年龄不匹配：
 *		线程安全问题
 *
 * 加锁：不同种类的线程都要加锁，并且是同一把锁
 *
 * 如果消费者先抢到CPU执行权，就会消费数据，但此时数据没有产生，为默认值
 *
 * 等代唤醒机制：
 * Objext类中提供了三个方法：
 *		wait():等代
 *		notify():唤醒单个线程
 *		notifyAll():唤醒所有线程
 *		为什么这些方法不定义在Thread类中?
 *			这写方法的调用必须通过锁对象调用，而锁对象是任意锁对象
 *			因此这些方法必须定义在Object类中
 */
public class StudentDemo {
	public static void main(String[] args) {
		//创建资源
		Student s = new Student();

		SetThread st = new SetThread(s);
		GetThread gt = new GetThread(s);

		Thread t1 = new Thread(st);
		Thread t2 = new Thread(gt);

		t1.start();
		t2.start();
	}
}
```

```
public class SetThread implements Runnable {
	private Student s;
	private int x;

	public SetThread(Student s) {
		this.s = s;
	}

	@Override
	public void run() {
		while (true) {
			synchronized (s) {
				if (s.flag) {
					try {
						s.wait();
					} catch (InterruptedException e) {
						// TODO Auto-generated catch block
						e.printStackTrace();
					}
				}
				if (x % 2 == 0) {
					s.name = "椰子皮";// 刚执行到这里就被别人抢到了执行权
					s.age = 22;
				} else {
					s.name = "屠龙";// 刚执行到这里就被别人抢到了执行权
					s.age = 23;
				}
				x++;
				s.flag = true;
				s.notify();// 唤醒t2，唤醒并不表示可以立即执行，还要抢CPU执行权
			}
		}
	}
}
```

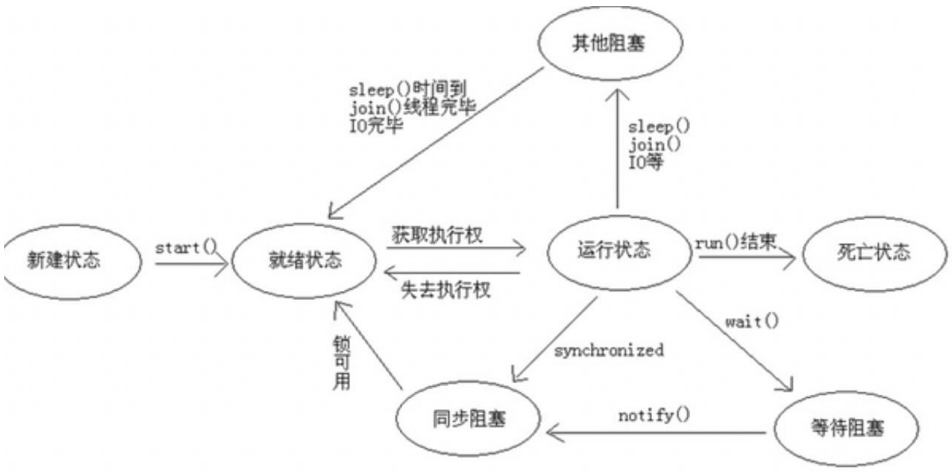
```
public class GetThread implements Runnable {
	private Student s;

	public GetThread(Student s) {
		this.s = s;
	}

	@Override
	public void run() {
		while (true) {
			synchronized (s) {
				if(!s.flag) {
					try {
						s.wait();//t2等待，立即释放锁，醒过来时从这里继续
					} catch (InterruptedException e) {
						// TODO Auto-generated catch block
						e.printStackTrace();
					}
				}
				System.out.println(s.name + ":" + s.age);
				s.flag=false;
				s.notify();
			}
		}
	}
}
```

线程状态转换图

2018年7月27日 14:53



常见的情况：

A:新建--就绪--运行--死亡

B:新建--就绪--运行--就绪--运行--死亡

C:新建--就绪--运行--其他阻塞--就绪--运行--死亡

D:新建--就绪--运行--同步阻塞--就绪--运行--死亡

E:新建--就绪--运行--等待阻塞--同步阻塞
--就绪--运行--死亡

线程组

2018年7月27日 14:57

ThreadGroup

```
/*
 * 线程组：把多个线程组合在一起
 * 它可以对一批线程进行分类管理，java允许程序直接对线程组进行控制
 */
public class ThreadGroupDemo {
	public static void main(String[] args) {
		// method1();
		method2();
	}

	private static void method2() {
		// 如何修改线程组
		ThreadGroup tg = new ThreadGroup("新组");

		MyRunnable my = new MyRunnable();
		Thread t1 = new Thread(tg, my, "王若潇");
		Thread t2 = new Thread(tg, my, "椰子皮");

		ThreadGroup tg1 = t1.getThreadGroup();
		ThreadGroup tg2 = t2.getThreadGroup();
		System.out.println(tg1.getName());// main
		System.out.println(tg2.getName());
	}

	private static void method1() {
		MyRunnable my = new MyRunnable();
		Thread t1 = new Thread(my, "王若潇");
		Thread t2 = new Thread(my, "椰子皮");

		// 线程类里面的方法
		ThreadGroup tg1 = t1.getThreadGroup();
		ThreadGroup tg2 = t2.getThreadGroup();

		// 线程组里的方法
```

```
            System.out.println(tg1.getName());// main
            System.out.println(tg2.getName());// main
            System.out.println(Thread.currentThread().getThreadGroup().getName());//
            main
            // 线程默认属于main线程组
        }
}
```

线程池

2018年7月27日 15:17

```
/*
 * 线程池中的每一个代码结束后并不会死亡，而是再次回到线程池中成为空闲状态，等代下一个对象使用
 *
 * 如何实现线程池：
 * 1.创建一个线程池对象，并控制靠创建几个线程对象
 * 2.线程池的线程可以执行：Runnalbe或Callable对象的线程
 * 3.调用方法 submit
 */
public class ExecutorsDemo {
	public static void main(String[] args) {
		ExecutorService pool =Executors.newFixedThreadPool(2);

		pool.submit(new MyRunnable());
		pool.submit(new MyRunnable());

		//结束线程池
		pool.shutdown();
	}
}


public class MyRunnable implements Runnable{
	@Override
	public void run() {
		for(int x=0;x<100;x++) {
			System.out.println(Thread.currentThread().getName()+":"+x);
		}
	}
}
```

Callable

2018年7月27日 16:09

```
/*
 * 多线程实现的方式3
 */
public class CallableDemo {
	public static void main(String[] args) {
		ExecutorService pool =Executors.newFixedThreadPool(2);

		pool.submit(new MyCallable());
		pool.submit(new MyCallable());

		pool.shutdown();
	}
}
```

```
/*
 * Callable:是带泛型的接口
 * 这里的泛型是call()方法的返回值
 */
public class MyCallable implements Callable {

	@Override
	public Object call() throws Exception {
		for(int x=0;x<100;x++) {
			System.out.println(Thread.currentThread().getName()+":"+x);
		}
		return null;
	}

}
```

线程池求和

2018年7月27日 16:18

```
public class MyCallable implements Callable<Integer>{
    private int number;

    public MyCallable(int number) {
        this.number = number;
    }

    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for(int x=0;x<number;x++) {
            sum+=x;
        }
        return sum;
    }
}
```

```
public class CallableDemo {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService pool = Executors.newFixedThreadPool(2);

        Future<Integer> f1 = pool.submit(new MyCallable(100));
        Future<Integer> f2 = pool.submit(new MyCallable(200));

        Integer i1 = f1.get();
        Integer i2 = f2.get();

        System.out.println(i1);
        System.out.println(i2);

        pool.shutdown();
    }
}
```

匿名内部类改进多线程

2018年7月27日 16:28

```
/*
 * 匿名内部类改进多线程
 */
public class ThreadDemo {
	public static void main(String[] args) {
		// 继承Thread类来实现多线程
		new Thread() {
			@Override
			public void run() {
				for (int x = 0; x < 100; x++) {
					System.out.println(Thread.currentThread().getName() + ":"
					+ x);
				}
			}
		}.start();

		// 实现Runnable来实现多线程
		new Thread(new Runnable() {

			@Override
			public void run() {
				for (int x = 0; x < 100; x++) {
					System.out.println(Thread.currentThread().getName() + ":"
					+ x);
				}
			}
		}) {
		}.start();

		// 更有难度的
		new Thread(new Runnable() {

			@Override
			public void run() {
				for (int x = 0; x < 100; x++) {
					System.out.println("hello" + ":" + x);
```

```
                }
            }
        }) {
            public void run() {
                for (int x = 0; x < 100; x++) {
                    System.out.println("world" + ":" + x);
                }
            }
        }.start();//输出world
    }
}
```

定时器

2018年7月27日 17:20

```
import java.util.Timer;
import java.util.TimerTask;

/*
 * 定时器：可以让我们在指定时间做某件事，也可以重复做某件事
 * 依赖Timer和TimerTask两个类
 * Timer：定时
 * TimerTask：任务
 */
public class TimerDemo {
	public static void main(String[] args) {
		// 创建定时器对象
		Timer t = new Timer();

		//三秒后执行任务
		t.schedule(new MyTask(t), 3000);
	}
}

class MyTask extends TimerTask {
	private Timer t;
	public MyTask() {}

	public MyTask(Timer t) {
		this.t=t;
	}

	@Override
	public void run() {
		System.out.println("hello");
		t.cancel();
	}
}
```

```
t.schedule(new MyTask2(), 3000,2000);
```

设计模式

2018年7月28日 23:22

设计模式：经验的总结

1.创建型：创建对象
2.结构型：对象的组成
3.行为型：对象的功能

简单工厂模式
工程方法模式