多线程同步

回顾

今天任务

```
1.多线程访问临界资源
        1.1 多线程访问临界资源时的数据安全问题
        1.2 解决临界资源问题
        1.3 锁
        1.4 同步代码块
        1.5 同步方法
        1.6 ReentrantLock类
2. 死锁
3.多线程在单例中的应用
4.线程的通信
        4.1 原理
        4.2 实现
```

教学目标

```
1.了解多线程中临界资源问题产生的原因
2.掌握解决临界资源问题的方案
3.掌握锁的概念
4.掌握同步代码块和同步方法
5.了解ReentrantLock类的使用
6.掌握多线程在单例中的应用
7.了解死锁以及生产者与消费者设计模式
```

1.多线程访问临界资源

1.1 多线程访问临界资源时的数据安全问题

```
        产生原因：有多个线程在同时访问一个资源，如果一个线程在取值的过程中，时间片又被其他线程抢走了，临界资源问题就产生了
```

1.2 解决临界资源问题

```
        解决方案：一个线程在访问临界资源的时候，如果给这个资源“上一把锁”，这个时候如果其他线程也要访问这个资源，就得在“锁”外面等待
```

1.3 锁

```
对象锁：任意的对象都可以被当做锁来使用

        类锁：把一个类当做锁，语法为：类名.class
```

1.4 同步代码块

语法：

```
synchronized(锁) {
        //需要访问临界资源的代码段
}
```

说明：

a.程序走到代码段中，就用锁来锁住了临界资源，这个时候，其他线程不能执行代码段中的代码，只能在锁外边等待

b.执行完代码段中的这段代码，会自动解锁。然后剩下的其他线程开始争抢cpu时间片

c.一定要保证不同的线程看到的是同一把锁，否则同步代码块没有意义

2.2.1 同步代码块和对象锁的使用

```
public class LockUsageDemo01 {
    // 需求：100张
    // 临界资源
    static int count = 100;

    //任何对象都可以充当一个对象锁
    //static Object obj = new Object();
    static String str = new String();

    static Runnable r = new Runnable() {

        @Override
        public void run() {
            while (count > 0) {

                //同步代码块结合对象锁
                synchronized(str) {
                    if(count <= 0) {
                        return;
                    }
                    count--;
                    System.out.println("售票员" + Thread.currentThread().getName() + "售出一张
票，余额为" + count);
                }
            }
        }
    };

    /**
     * 售票的过程中，出现负数的原因：
     * 当其中的一个线程抢到时间片进入到锁中，
     * 这时，其他剩余的三个线程都在while里面，但是同步代码块的外面
     *当最后一张票被售完之后，将自动解锁
     *，但是，其他剩余的三个线程将再次进入同步代码块进行强制售票，出现了负数
     *解决办法：在锁里面进行判断
     */

    public static void main(String[] args) {
        Thread t0 = new Thread(r, "喜羊羊");
        Thread t1 = new Thread(r, "沸羊羊");
        Thread t2 = new Thread(r, "灰太狼");
        Thread t3 = new Thread(r, "小灰灰");

        t0.start();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

2.2.2 同步代码块和类锁的使用

```
public class LockUsageDemo02 {
    // 需求：100张
    // 临界资源
    static int count = 100;

    static Runnable r = new Runnable() {
        @Override
        public void run() {
            while (count > 0) {
                // 同步代码块结合类锁
                //LockUsageDemo03.class
                //任何类，包含自定义的类都可以充当一把类锁
                synchronized (Math.class) {
                    if (count <= 0) {
                        return;
                    }
                    count--;
                    System.out.println("售票员" + Thread.currentThread().getName() + "售出一张
票，余额为" + count);

                    //在锁中进行sleep，  不会释放锁标记，其他线程仍然访问不了
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            }
        }
    };

    public static void main(String[] args) {
        Thread t0 = new Thread(r, "喜羊羊");
        Thread t1 = new Thread(r, "沸羊羊");
        Thread t2 = new Thread(r, "灰太狼");
        Thread t3 = new Thread(r, "小灰灰");

        t0.start();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

1.5 同步方法

2.3.1 同步非静态方法

```
public class LockUsageDemo03 {
    // 需求：100张
    // 临界资源
    static int count = 100;

    static Runnable r = new Runnable() {
        @Override
        public void run() {
            while (count > 0) {
                sellTickets();
            }
        }

        //同步方法，作用和同步代码块一样
        public synchronized void sellTickets() {
            if (count <= 0) {
                return;
            }
            count--;
            System.out.println("售票员" + Thread.currentThread().getName() + "售出一张票，余额为"
+ count);
        }
    };

    public static void main(String[] args) {
        Thread t0 = new Thread(r, "喜羊羊");
        Thread t1 = new Thread(r, "沸羊羊");
        Thread t2 = new Thread(r, "灰太狼");
        Thread t3 = new Thread(r, "小灰灰");

        t0.start();
        t1.start();
        t2.start();
        t3.start();
    }
}
```


2.3.2 同步静态方法

```
public class LockUsageDemo04 {
    // 需求：100张
    // 临界资源
    static int count = 100;

    static Runnable r = new Runnable() {
        @Override
        public void run() {
            while (count > 0) {
                sellTickets();
            }
        }

        //同步方法，作用和同步代码块一样
        public static synchronized void sellTickets() {
            if (count <= 0) {
                return;
            }
            count--;
            System.out.println("售票员" + Thread.currentThread().getName() + "售出一张票，余额为"
+ count);
        }
    };

    public static void main(String[] args) {
        Thread t0 = new Thread(r, "喜羊羊");
        Thread t1 = new Thread(r, "沸羊羊");
        Thread t2 = new Thread(r, "灰太狼");
        Thread t3 = new Thread(r, "小灰灰");

        t0.start();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

1.6 ReentrantLock类

通过显式定义同步锁对象来实现同步,同步锁提供了比synchronized代码块更广泛的锁定操作

注意：最好将 unlock的操作放到finally块中

通过使用ReentrantLock这个类来进行锁的操作,它实现了Lock接口，使用ReentrantLock可以显式地加锁、释放锁

> 案例一：模拟售票

```java
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantLockDemo01 {
    // 需求：100张
    // 临界资源
    static int count = 100;

    // 定义一个ReentrantLock类的对象
    static ReentrantLock lock = new ReentrantLock();

    static Runnable r = new Runnable() {

        @Override
        public void run() {
            while (count > 0) {

                //加锁
                // lock()
                lock.lock();

                if (count <= 0) {
                    return;
                }
                count--;
                System.out.println("售票员" + Thread.currentThread().getName() + "售出一张票，余额为" + count);

                //解锁
                // unlock()
                lock.unlock();

                //注意：lock（）和unlock（）都是成对出现的
            }
        }
    };

    public static void main(String[] args) {
        Thread t0 = new Thread(r, "喜羊羊");
        Thread t1 = new Thread(r, "沸羊羊");
        Thread t2 = new Thread(r, "灰太狼");
        Thread t3 = new Thread(r, "小灰灰");

        t0.start();
        t1.start();
        t2.start();
        t3.start();

        System.out.println("hello");
    }
}
```

案例二：模拟银行卡存取钱

```java
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantLockDemo02 {
    // 需求：有一张银行卡，两个线程向其中存钱，两个线程取钱
    /*
     * 临界资源： 银行卡
     *
     * 注意：要保证临界资源的同步性
     */

    //存钱线程的target
    static Runnable r0 = new Runnable() {

        @Override
        public void run() {
            while(true) {
                //获取单例对象
                Card card = Card.currentInstance();
                //调用storeMoney方法
                card.storeMoney(500);
            }
        }
    };

    //取钱线程的target
    static Runnable r1 = new Runnable() {

        @Override
        public void run() {
            while(true) {
                //获取单例对象
                Card card = Card.currentInstance();
                //调用getMoney方法
                card.getMoney(1000);
            }
        }
    };
    public static void main(String[] args) {
        //存钱线程
        Thread t0 = new Thread(r0, "王大妈");
        Thread t1 = new Thread(r0, "李老头");

        //取钱线程
        Thread t2 = new Thread(r1, "李老大");
        Thread t3 = new Thread(r1, "李老二");

        t0.start();
        t1.start();
        t2.start();
        t3.start();

    }
```

```
}

// 银行卡类---使用单例
class Card {
    // 实例化一个私有的静态的当前类的实例
    private static Card instance = new Card();

    // 余额
    private int rest = 1000;

    // 实例化一个ReentrantLock的对象
    ReentrantLock lock = new ReentrantLock();

    // 将构造方法私有化
    private Card() {
    }

    // 提供一个对外的开放的方法，将实例返回
    public static Card currentInstance() {
        return instance;
    }

    // rest的get/set方法
    public void setRest(int rest) {
        this.rest = rest;
    }

    public int getRest() {
        return rest;
    }

    // 存钱
    public void storeMoney(int num) {
        // 加锁
        lock.lock();

        this.rest += num;
        System.out.println(Thread.currentThread().getName() + "存了" + num + ",余额为: " +
rest);
        // 解锁
        lock.unlock();
    }

    // 取钱
    public void getMoney(int num) {

        // 加锁
        lock.lock();

        if (num > rest) {

            num = rest;
```

```
            }

            this.rest -= num;
            System.out.println(Thread.currentThread().getName() + "取了" + num + ",余额为: " +
rest);
            // 解锁
            lock.unlock();
        }
    }
```

2.死锁

> 每个人都拥有其他人需要的资源，同时又等待其他人拥有的资源，并且每个人在获得所有需要的资源之前都不会放弃已经拥有的资源
>
> 当多个线程完成功能需要同时获取多个共享资源的时候可能会导致死锁

```
public class Test {
    public static void main(String[] args) {

        DeadLock d1 = new DeadLock();
        d1.setName("线程1");
        DeadLock d2 = new DeadLock();
        d2.setName("线程2");

        d1.start();
        d2.start();
    }
}

class DeadLock extends Thread {

    //声明两个对象，作为两个锁对象
    static Object o1 = new Object();
    static Object o2 = new Object();

    //设计一个布尔变量
    static boolean boo = true;

    @Override
    public void run() {
        if(boo) {
            synchronized(o1) {
                System.out.println(currentThread().getName()
                        +"获取了第一个锁对象o1，等待第二个锁对象o2");
                boo = !boo;
                try {
                    sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                synchronized(o2) {
                    System.out.println(currentThread().getName()
                            +"获取了两个锁对象o1和o2，即将结束并释放两个锁对象");
                }
            }
        }else {
            synchronized(o2) {
                System.out.println(currentThread().getName()
                        +"获取了第一个锁对象o2，等待第二个锁对象o1");
                boo = !boo;
                try {
                    sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block

                    e.printStackTrace();
```

```
                }
                synchronized(o1) {
                    System.out.println(currentThread().getName()
                            +"获取了两个锁对象o2和o1，即将结束并释放两个锁对象");
                }
            }
        }
    }
}
```

3.多线程在单例中的应用

单例的实现方式：懒汉式和饿汉式

其中，懒汉式是线程不安全的，当有多条线程同时访问单例对象时，则会出现多线程临界资源问题

3.1 多线程访问单例-饿汉式

```
public class SigletonDemo01 {
    static HashSet<King> hs = new HashSet<>();
    static Runnable r = new Runnable() {

        @Override
        public void run() {
            //获取单例对象
            King king = King.currentInstance();

            //将获取到的单例对象添加到集合中
            hs.add(king);
        }
    };

    public static void main(String[] args) {
        //需求：多条线程同时去获取单例对象，然后将获取到的单例对象添加到HashSet中

        //创建多个线程对象，同时访问 单例对象
        for (int i = 0; i < 10000; i++) {
            Thread thread = new Thread(r);
            thread.start();
        }

        System.out.println(hs);
    }
}

class King {
    private static King instance = new King();

    private King(){}

    public static King currentInstance() {
        return instance;
    }
}
```


3.2 多线程访问单例-懒汉式

```
public class SigletonDemo02 {
    static HashSet<Queue> hs = new HashSet<>();
    static Runnable r = new Runnable() {

        @Override
        public void run() {
            // 获取单例对象
            Queue king = Queue.currentInstance();

            // 将获取到的单例对象添加到集合中
            hs.add(king);
        }
    };

    public static void main(String[] args) {

        // 创建多个线程对象，同时访问 单例对象
        for (int i = 0; i < 1000; i++) {
            Thread thread = new Thread(r);
            thread.start();
        }

        //[com.qianfeng.day19.Queue@ca4a1b4, com.qianfeng.day19.Queue@1733c6a5]
        System.out.println(hs);
    }
}

class Queue {
    private static Queue instance;

    private Queue() {
    }

    //使用同步代码块，同步方法，以及同步锁都可以解决
    public synchronized static Queue currentInstance() {
        if (instance == null) {
            instance = new Queue();
        }

        return instance;
    }![生产者与消费者设计模式](C:\Users\Administrator\Desktop\大纲细化\第21天-多线程2\生产者与消费者设计模式.png)
}
```

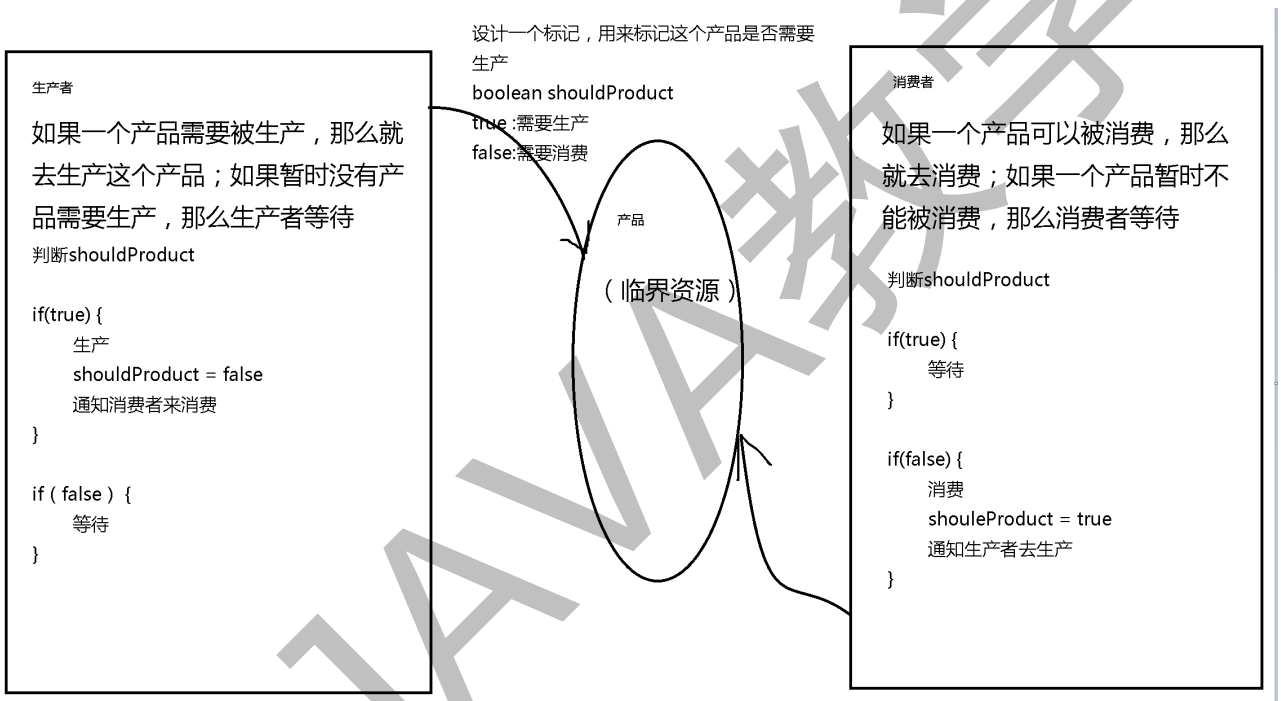
4.线程的通信【生产者与消费者设计模式】

4.1 原理

它描述的是有一块缓冲区作为仓库，生产者可以将产品放入仓库，消费者可以从仓库中取走产品，解决生产者/消费者问题，我们需要采用某种机制保护生产者和消费者之间的同步

同步问题核心在于：如何保证同一资源被多个线程并发访问时的完整性，常用的方法就是加锁，保证资源在任意时刻只被一个线程访问

画图分析：



4.2 实现

> 方式一：采用wait()、notify()和notifyAll()方法

`wait()`：当缓冲区已满或空时，生产者/消费者线程停止自己的执行，放弃锁，使自己处于等待状态，让其他线程执行

- 是Object的方法
- 调用方式：对象.wait();
- 表示释放 对象 这个锁标记，然后在锁外边等待（对比sleep(),sleep是抱着锁休眠的）
- 等待，必须放到同步代码段中执行

`notify()`：当生产者/消费者向缓冲区放入/取出一个产品时，向其他等待的线程发出可执行的通知，同时放弃锁，使自己处于等待状态

- 是Object的方法
- 调用方式：对象.notify();
- 表示唤醒 对象 所标记外边在等待的一个线程

notifyAll():全部唤醒

·是Object的方法
·调用方式：对象.notifyAll()
·表示唤醒　对象 所标记外边等待的所有线程

```
public class ProductorAndConsumerDemo01 {
    // 定义一个标记，用来表示产品是否需要被生产
    static boolean shouldProduct = true;

    // 静态成员内部类
    // 生产者线程的target
    static class Productor implements Runnable {
        //需要生产的产品
        private Product p;

        public Productor(Product p) {
            this.p = p;
        }

        @Override
        public void run() {
            while(true) {
                synchronized("") {
                    if(shouldProduct) {
                        //生产
                        this.p.setName("老婆饼");
                        System.out.println("生产者" + Thread.currentThread().getName() + "生产
了一件产品" + this.p);

                        //修改状态
                        shouldProduct = false;

                        //通知消费者来进行消费
                        "".notifyAll();
                    } else {
                        //等待
                        try {
                            "".wait();
                        } catch (InterruptedException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                        }
                    }
                }
            }
        }
    }

    //消费者线程的target
    static class Consumer implements Runnable {
        //需要被消费的产品
        private Product p;

        public Consumer(Product p) {

            this.p = p;
```

```
        }

        @Override
        public void run() {
            while(true) {
                synchronized("") {
                    if(shouldProduct) {
                        try {
                            "".wait();
                        } catch (InterruptedException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                        }
                    } else {
                        //消费
                        System.out.println("消费者" + Thread.currentThread().getName() + "消费
了一件产品" + this.p);

                        //修改状态
                        shouldProduct = true;

                        //通知生产者去生产
                        "".notifyAll();
                    }
                }
            }
        }
    }

    public static void main(String[] args) {
        //说明：如果锁外面只有一个生产者或者一个消费者等待的话，则使用notify唤醒，
        //但是，如果锁外面有多个生产者或者多个消费者等待的话，则使用notifyAll唤醒
        //实例化一个产品
        Product p = new Product();

        //生产者
        Productor productor = new Productor(p);
        Thread t0 = new Thread(productor,"老王头");
        Thread t1 = new Thread(productor,"老李头");
        Thread t2 = new Thread(productor,"老赵头");
        Thread t3 = new Thread(productor,"老刘头");
        t0.start();
        t1.start();
        t2.start();
        t3.start();

        //消费者
        Consumer consumer = new Consumer(p);
        Thread thread0 = new Thread(consumer,"小凳子");
        Thread thread1 = new Thread(consumer,"小桌子");
        Thread thread2 = new Thread(consumer,"小李子");
```

```
            Thread thread3 = new Thread(consumer,"小姨子");
            thread0.start();
            thread1.start();
            thread2.start();
            thread3.start();
        }
    }
    class Product {
        private String name;
        int id;

        public void setName(String name) {
            this.name = name;
            id++;
        }
        @Override
        public String toString() {
            return "Product [name=" + name + ", id=" + id + "]";
        }
    }
```

> 方式二：采用ReentrantLock类中的newCondition()方法结合Condition类中的await()、signal()和signalAll()方法

```
public class ProductorAndConsumerDemo02 {
    // 定义一个标记，用来表示产品是否需要被生产
    static boolean shouldProduct = true;

    static ReentrantLock lock = new ReentrantLock();

    // 监视器：生产者
    static Condition c1 = lock.newCondition();
    // 监视器：消费者
    static Condition c2 = lock.newCondition();

    // 静态成员内部类
    // 生产者线程的target
    static class Productor implements Runnable {
        // 需要生产的产品
        private Product p;

        public Productor(Product p) {
            this.p = p;
        }

        @Override
        public void run() {
            while (true) {
                // 加锁
                lock.lock();

                if (shouldProduct) {
                    // 生产
                    this.p.setName("老婆饼");
                    System.out.println("生产者" + Thread.currentThread().getName() + "生产了一
件产品" + this.p);

                    // 修改状态
                    shouldProduct = false;

                    // 通知消费者来进行消费
                    c2.signalAll();
                } else {
                    // 等待
                    try {
                        c1.await();
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }

                // 解锁
                lock.unlock();
            }
```

```
        }
    }

    // 消费者线程的target
    static class Consumer implements Runnable {
        // 需要被消费的产品
        private Product p;

        public Consumer(Product p) {
            this.p = p;
        }

        @Override
        public void run() {
            while (true) {
                //加锁
                lock.lock();

                if (shouldProduct) {
                    try {
                        c2.await();
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                } else {
                    // 消费
                    System.out.println("消费者" + Thread.currentThread().getName() + "消费了一
件产品" + this.p);

                    // 修改状态
                    shouldProduct = true;

                    // 通知生产者去生产
                    c1.signalAll();
                }
                //解锁
                lock.unlock();
            }
        }
    }

    public static void main(String[] args) {
        // 说明：如果锁外面只有一个生产者或者一个消费者等待的话，则使用notify唤醒，
        // 但是，如果锁外面有多个生产者或者多个消费者等待的话，则使用notifyAll唤醒
        // 实例化一个产品
        Product p = new Product();

        // 生产者
        Productor productor = new Productor(p);

        Thread t0 = new Thread(productor, "老王头");
```

```
        Thread t1 = new Thread(productor, "老李头");
        Thread t2 = new Thread(productor, "老赵头");
        Thread t3 = new Thread(productor, "老刘头");
        t0.start();
        t1.start();
        t2.start();
        t3.start();

        // 消费者
        Consumer consumer = new Consumer(p);
        Thread thread0 = new Thread(consumer, "小凳子");
        Thread thread1 = new Thread(consumer, "小桌子");
        Thread thread2 = new Thread(consumer, "小李子");
        Thread thread3 = new Thread(consumer, "小姨子");
        thread0.start();
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
class Product {
    private String name;
    int id;

    public void setName(String name) {
        this.name = name;
        id++;
    }
    @Override
    public String toString() {
        return "Product [name=" + name + ", id=" + id + "]";
    }
}
```

总结

课前默写

1.分别使用继承Thread类和实现Runnable接口的方式创建线程

2.设计程序，演示join方法的使用

作业

1.春运到了，某个火车站四个售票员出售某个车次最后100张车票的情形。
分析：
火车票： 临界资源
四个售票员：四个线程实例 ， 出售车票：卖一张，车票就少一张

2.使用线程安全的懒汉式实现之前的阿里巴巴董事长的题目

3.完善昨天的作业，模拟多个人通过一个山洞的模拟。这个山洞每次只能通过一个人，每个人通过山洞的时间为5秒，随机生成10个人，同时准备过此山洞，显示一下每次通过山洞人的姓名

面试题

1.多线程临界资源问题是如何产生的，该怎么解决

2.简述生产者与消费者设计模式的实现原理