

1.多线程

1.1 进程和线程

进程：正在运行的程序就是一个进程，单一的顺序控制流，有独立的代码运行空间。

线程：轻量级的进程，单一的顺序控制，有独立的代码运行空间。

进程之间的通信开销比较大，线程之间的通信(进程内部的通信)开销比较小。

1.2 创建线程、多线程

1.2.1 创建线程

1. 实现Runnable接口，重写run方法（业务）
2. 创建线程对象
3. 启动线程 start();

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // TO DO 业务
        System.out.println("子线程: " + Thread.currentThread().getName() + ":" +
new Date());
    }
}
```

```
public class Test {

    public static void main(String[] args) {

        // 获取当前线程的名称
        System.out.println("主线程:" + Thread.currentThread().getName());

        MyRunnable mr = new MyRunnable();
        Thread t = new Thread(mr);
        t.start();

    }

}
```

1.2.2创建多线程

```

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // TO DO 业务
        System.out.println("子线程: " + Thread.currentThread().getName() + ":" +
new Date());
    }
}

```

```

public class Test {

    public static void main(String[] args) {

        // 业务接口
        MyRunnable mr = new MyRunnable();

        // 启动多线程
        Thread t1 = new Thread(mr, "A");
        Thread t2 = new Thread(mr, "B");
        Thread t3 = new Thread(mr, "C");

        t1.start(); // 告诉 JVM 我们需要一个子线程, 由JVM向CPU去申请一个线程
        t2.start();
        t3.start();

        // 现在在执行后会自动销毁
        // t1.start(); // 抛出异常 IllegalStateException
    }

}

```

线程的执行顺序和线程的启动顺序无关。

```

// 启动多线程
Thread t1 = new Thread(mr, "A");
Thread t2 = new Thread(mr, "B");
Thread t3 = new Thread(mr, "C");

t1.start(); // 告诉 JVM 我们需要一个子线程, 由JVM向CPU去申请一个线程

```

子线程: C: Thu Apr 29 10:51:43 CST 2021

子线程: B: Thu Apr 29 10:51:43 CST 2021

子线程: A: Thu Apr 29 10:51:43 CST 2021

线程的执行顺序程序员不能指定

使用Thread类来创建线程: 编写方便, 扩展性没有 Runnable接口号。

```
public class MyThread extends Thread {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }

}
```

```
public class Test {
    public static void main(String[] args) {
        Thread t = new MyThread();
        t.start();
    }
}
```

1.3 线程调度

sleep(long time): 休眠, 让出CPU的资源一定时间后, 重新获取线程资源。

编写一个业务类

```
public class MyRunnable implements Runnable {

    @Override
    public void run() {

        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName());
        }

    }

}
```

测试类

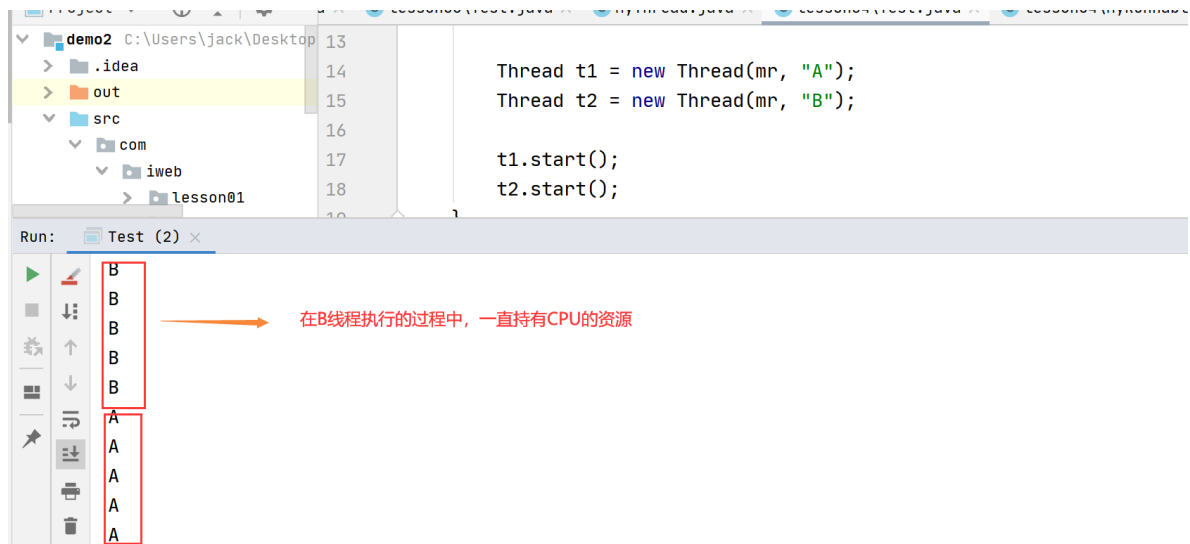
```
public class Test {

    public static void main(String[] args) {
        // 启动2个线程来执行任务
        MyRunnable mr = new MyRunnable();

        Thread t1 = new Thread(mr, "A");
        Thread t2 = new Thread(mr, "B");

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

}
```



代码修改

```

for (int i = 0; i < 5; i++) {
    System.out.println(Thread.currentThread().getName());
    try {
        Thread.sleep(1); // 当前线程休眠1ms，这个1ms期间其他线程有机会获取CPU的资源
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

A
B
B
A
B
A
A
B
B
A

线程之前由切换

yield()：让步，让出CPU的资源一个计算频率。重新获取线程资源。



join(): 线程的加入, 一个线程强行加入另外一个线程, 直到加入的线程执行完毕后才释放线程资源。

```
public class MyRunnable1 implements Runnable {  
  
    private Thread thread;  
  
    public void setThread(Thread thread) {  
        this.thread = thread;  
    }  
  
    @Override  
    public void run() {  
        // 当执行 i = 3 的时候 加入一个线程  
        for (int i = 0; i < 5; i++) {  
            if (i == 3) {  
                try {  
                    // 加入线程  
                    thread.start();  
                    thread.join();  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
            System.out.println(Thread.currentThread().getName());  
        }  
    }  
}
```

```
public class MyRunnable2 implements Runnable {  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 5; i++) {  
            System.out.println(Thread.currentThread().getName());  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```

    }
}
}
}

```

```

public class Test {

    public static void main(String[] args) {

        MyRunnable1 m1 = new MyRunnable1();
        MyRunnable2 m2 = new MyRunnable2();

        Thread t1 = new Thread(m1, "A");
        Thread t2 = new Thread(m2, "B");

        m1.setThread(t2);

        t1.start();

    }

}

```


1.4 线程优先级&守护线程&线程组（了解）

1.4.1 线程优先级

```

public final static int MIN_PRIORITY = 1;

```

 最小优先级

```

/**
 * The default priority that is assigned to a thread.
 */

```

```

public final static int NORM_PRIORITY = 5;

```

```

/**
 * The maximum priority that a thread can have.
 */

```

```

public final static int MAX_PRIORITY = 10;

```

 最大优先级

```

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

        MyRunnable mr = new MyRunnable();

        Thread t1 = new Thread(mr, "A");
        Thread t2 = new Thread(mr, "B");
        Thread t3 = new Thread(mr, "C");

        // 线程默认的优先级：通知 JVM 希望这个线程优先执行，但是不一定优先
        int priority = t1.getPriority();
        System.out.println(priority); // 5
    }
}

```

```

        t1.setPriority(1); // 设定为最低优先级
        t2.setPriority(10); // 设定为最高优先级
        t3.setPriority(10);

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

    }
}

```

1.4.2 守护线程

守护线程是运行在程序后台的一个线程，当程序中的线程全部是守护线程的时候守护线程才退出。

Java中的 GC 就是一个守护线程。

```

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

public class Test {
    public static void main(String[] args) throws Exception {
        MyRunnable mr = new MyRunnable();
        Thread t = new Thread(mr, "守护线程");

        t.setDaemon(true); // 设定当前线程为守护线程

        t.start();

        Thread.sleep(3000);

        System.out.println("main over");

    }
}

```

1.4.3 线程组

1. 当一个线程被创建的时候默认的线程组是 main 线程组
2. 可以自定一个线程组，这个自定义的线程组默认是 main 线程组的子线程组
3. 可以指定一个线程的线程组。

```

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println(
            Thread.currentThread().getName()
                + "父线程组:"
                + Thread.currentThread().getThreadGroup().getName()
                + "祖父线程组:" +
            Thread.currentThread().getThreadGroup().getParent().getName()
        );
    }
}

```

```

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

        MyRunnable mr = new MyRunnable();
        Thread t = new Thread(mr, "A");
        t.start();

        ThreadGroup tg = new ThreadGroup("线程组A");
        Thread t1 = new Thread(tg, mr, "B");

        t1.start();
    }
}

```

1.5数据共享

多个线程执行的代码来自于同一个run方法, 操作的是同一个对象的数据。

```

public class Good {

    private Integer count = 2; // count>=0

    public void setCount(Integer count) {
        this.count = count;
    }

    public Integer getCount() {
        return count;
    }

    public void change() {
        if (count >= 1) {
            count--;
        }
    }
}

```



```

public class GoodRunnable implements Runnable {

    private Good good;

    public GoodRunnable(Good good) {
        this.good = good;
    }

    @Override
    public void run() {
        good.change();
    }

}

```

```

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

        // 使用3个线程同时对Good 中的 count 进行消费
        Good good = new Good();
        GoodRunnable gr = new GoodRunnable(good);

        Thread t1 = new Thread(gr);
        Thread t2 = new Thread(gr);
        Thread t3 = new Thread(gr);

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

        Thread.sleep(100);

        System.out.println("剩余商品数量:" + good.getCount());

    }
}

```

1.6 线程同步与锁

1.6.1 数据共享的问题

对上面的代码 Good 中修改代码，发现商品在消费后可能出现 - 1 的情况。这种情况对于业务是不能发生的。

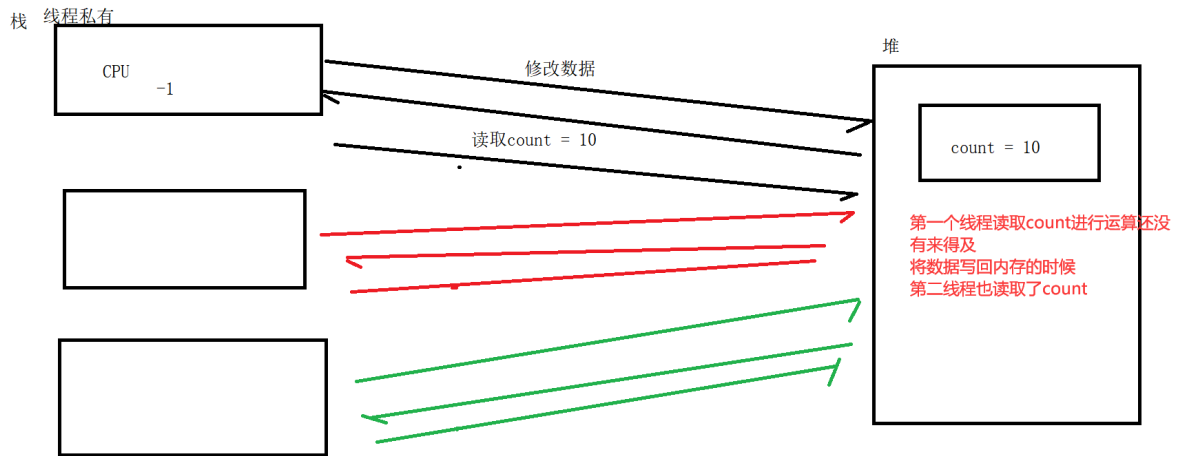
```

public void change() {
    if (count >= 1) {
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        count--;
    }
}

```

加上这段代码

产生以上问题的原因



要解决上面的问题，需要对修改数据部分的代码进行同步

同步就是顺序执行，异步是相对于同步而言(多线程是实现异步的一种方案)。

同步方法

```
public synchronized void change() {
    if (count >= 1) {
        System.out.println("1");
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        count--;
    }
}
```

同步方法

同步代码块：持有同一对象的锁。

```
public void change() {
    // 一些业务：不需要同步
    synchronized (this) { // 局部同步
        if (count >= 1) {
            System.out.println("1");
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count--;
        }
    }
    // 一些业务：不需要同步
}
```

持有同一对象的锁，而不是指定对象的锁

this也可以是其他的同一对象

```

public void change() {
    // 一些业务：不需要同步
    synchronized ("lock") { // 局部同步
        if (count >= 1) {
            System.out.println("1");
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count--;
        }
    }
    // 一些业务：不需要同步
}

```

同一对象

注意：

1. 锁是对象的
2. 线程是CPU的
3. 当线程进入同步方法或者同步代码块的时候会自动获取对象的锁，执行完同步方法或者同步代码块后自动释放锁。

1.6.2线程交互

线程交互的前提是 同步方法或者同步代码块，在同步方法或者同步代码块中，我们可能需要让其他线程加入到业务中。

wait()：释放锁，进入等待队列。

notify()：唤醒等待队列中的一个线程。

notifyAll()：唤醒等待队列中的所有线程

以上三个方法是Object的。

```

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "进入了线程");
        synchronized ("lockA") {
            System.out.println(Thread.currentThread().getName() + "拿到了锁");
            try {
                Thread.sleep(2000); // 释放的是CPU
                System.out.println("2S 过去了 ");
                "lockA".wait(); // 释放锁
                Thread.sleep(2000); // 释放的是CPU
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "释放锁");
        }
    }
}

```

```

public class Test {

```

```

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

    MyRunnable mr = new MyRunnable();

    Thread t1 = new Thread(mr, "A");
    Thread t2 = new Thread(mr, "B");

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

    Thread.sleep(5000);

    synchronized ("lockA") {
        // System.out.println("唤醒其中一个线程");
        // "lockA".notify();
        System.out.println("唤醒所有线程");
        "lockA".notifyAll();
    }

}
}

```

1.6.3 死锁（重要，代码要会手写）

```

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

        String lockA = "lockA";
        String lockB = "lockB";

        // 线程A
        Thread a = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (lockA) {
                    System.out.println(Thread.currentThread().getName() + "获取"
+ lockA);

                    synchronized (lockB) {
                        System.out.println(Thread.currentThread().getName() +
"获取" + lockB);
                    }
                }
            }
        }, "A");

        // 线程B
        Thread b = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (lockB) {

```

```

        System.out.println(Thread.currentThread().getName() + "获取"
+ lockB);

        synchronized (lockA) {
            System.out.println(Thread.currentThread().getName() +
"获取" + lockA);
        }
    }
    }, "B");

    // 启动线程
    a.start();
    b.start();

}
}

```

线程死锁的解决方案：

1. 注意加锁的顺序
2. 使用java提供的工具查看死锁 jvisualvm.exe
3. 使用JDK提供的Lock接口中 tryLock() 来解决死锁

练习

作业1：

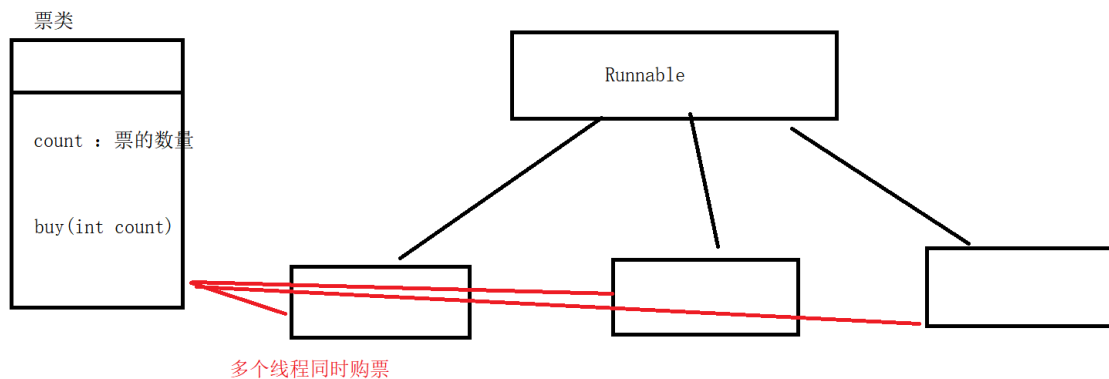
通过多线程实现交替打印A B A B 各10次 后程序退出

分析：

1. 线程A 和线程B 的打印代码需要同步代码块中
2. 线程A 打印后 notify() wait() 和线程B 之前交替执行
3. 第一次打印和最后一次打印的 wait() notify() 的执行

作业2：

设计一个购票系统：可以多个线程同时进行购票，确保购票的业务正确。



数据共享来实现, 需要同步方法或者同步代码块。