day17_多线程高级

一 内容回顾（列举前一天重点难点内容）

1.1 教学重点:

```
1.掌握线程的基本概念
2.掌握线程的生命周期
3.掌握常见线程的方法
4.掌握线程同步的实现
5.掌握synchronized的使用
```

1.2 教学难点:

```
1.多线程的原理的理解
2.对单例实现线程同步
这里主要是咱们的课程中还没有讲解单例设计模式的内容,不过在单例中使用线程同步是他都一个重要应用,大家要掌握.
```

二 教学目标

```
1.掌握单生产者单消费者模式
2.掌握Lock的基本使用
3.了解使用Lock实现多生产者多消费者
4.了解使用synchronized实现多生产者多消费者
5.了解线程池的理解
```

三 教学导读

3.1. 多线程高级

在学习多线程的过程中,高级阶段属于对多线程的深入理解分析应用,对于我们来说,首先应该扎实的学好多线程基础,在这个基础上深入学习多线程高级.在面试以及实际生产应用中,多线程高级的内容使用较多.

四 教学内容

4.1. 线程通信(会)

4.1.1. 线程通信-打印机打印实例

4.1.1.1. 线程通信基本实现

- 实例:打印机打印
- 实现功能:不断输入不断输出
- 总结:需要给输入任务和输出任务同时加一把锁,保证两个任务之间是同步的给两个任务加一把锁:可以是desc或者Object.class

不建议使用Object.class:由于Object的使用范围太大,可能造成不必要的错误.desc合适,因为他只被当前的两个任务共享.

注意:对于当前的情况只给一个线程加锁,无法实现两个线程的同步.

- 示例代码

```
/*
分析:
两个线程:输入线程和输出线程
两个任务区:输入任务,输出任务
一份数据

*/
```

```java
public class Demo2 {
    public static void main(String[] args) {
        //数据对象
        Desc desc = new Desc();
        //创建两个任务
        Input input = new Input(desc);
        Output output = new Output(desc);
        //创建线程
        Thread in = new Thread(input);
        Thread out = new Thread(output);
        //开启线程
        in.start();
        out.start();
    }
}

//数据类
class  Desc {
    String name;
    String sex;
}

//输入任务
class  Input implements  Runnable{
    Desc desc;

    public Input(Desc desc) {
        this.desc = desc;
    }

    int i = 0;

    public void run() {
```

```
        while (true){
            synchronized (desc) {
                if (i == 0) {
                    desc.name = "特没谱";
                    desc.sex = "男";
                } else {
                    desc.name = "安倍小三";
                    desc.sex = "女";
                }
                i = (i + 1) % 2;
            }
        }
    }
}
//输出任务
class  Output implements  Runnable{
   Desc desc;

    public Output(Desc desc) {
            this.desc = desc;
    }

    public void run() {
        while (true){
            synchronized (desc) {
                System.out.println(desc.name + "       "
+ desc.sex);
            }
        }
    }
}
```

4.1.1.2. 线程通信功能进阶

- 实例:打印机打印
- 功能实现:一次输入一次输出
- 总结:在上面线程通信基本实现的基础上改进代码,通过分别给输入和输出线程设置wait和notify状态,实现一次输入一次输出
- 示例代码

```
 /*
  * 两个线程:输入线程和输出线程
  * 两个任务区:输入任务,输出任务
  * 一份数据
  */


public class Demo3 {
    public static void main(String[] args) {
        //数据对象
        Desc1 desc = new Desc1();
        //创建两个任务
        Input1 input = new Input1(desc);
        Output1 output = new Output1(desc);
        //创建线程
        Thread in = new Thread(input);
        Thread out = new Thread(output);
        //开启线程
        in.start();
        out.start();
    }
}

//数据类
class  Desc1 {
```

```
26      String name;
27      String sex;
28
29      boolean flag = false;//用于执行唤醒等待的切换
30  }
31
32  //输入任务
33  class  Input1 implements  Runnable{
34      Desc1 desc;
35
36      public Input1(Desc1 desc) {
37          this.desc = desc;
38      }
39
40      int i = 0;
41
42      public void run() {
43          while (true){
44              synchronized (desc) {
45                  if (desc.flag == true){
46                      //让当前的线程等待
47                      //wait方法要在同步下使用,因为要使用同步
    锁
48                      try {
49                          desc.wait();//当执行这行代码的时
    候,这里对应的是哪个线程,就操作的是哪个线程
50                      } catch (InterruptedException e) {
51                          e.printStackTrace();
52                      }
53                  }
54                  if (i == 0) {
55                      desc.name = "特没谱";
56                      desc.sex = "男";
```

```
                } else {
                    desc.name = "安倍小三";
                    desc.sex = "女";
                }
                i = (i + 1) % 2;

                //状态切换
                desc.flag = !desc.flag;

                //唤醒输出线程
                //唤醒的是同一把锁下的线程,因为现在只有一个输入线程,一个输出线程.所以这里唤醒的是输出线程
                //当线程池中没有被当前的锁标记的线程可唤醒时,我们称为空唤醒,空唤醒不影响程序的执行.
                desc.notify();
            }
        }
    }
}
//输出任务
class  Output1 implements  Runnable{
    Desc1 desc;

    public Output1(Desc1 desc) {
        this.desc = desc;
    }

    public void run() {
        while (true){
            synchronized (desc) {
                if (desc.flag == false){
                    try {
                        desc.wait();
```

```
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println(desc.name + "
" + desc.sex);

                desc.flag = !desc.flag;

                desc.notify();
            }
        }
    }
}
```

4.1.1.3. 线程通信功能优化

- 实例:打印机打印
- 功能:对一次输入一次输出代码的改进
- 总结:进行了代码优化

  面向对象的精髓:谁的活儿谁干,不是你的活儿不要干

  将数据准备的活儿从输入任务输出任务提出来,放入数据类Desc2

```
package com.qf.test;

public class Demo4 {
    public static void main(String[] args) {
        //创建了一份数据
        Desc2 desc = new Desc2();
        //创建了输入任务和输出任务对象
        Input2 input = new Input2(desc);
        Output2 output = new Output2(desc);
```

```
10          //创建输入线程和输出线程
11          Thread in = new Thread(input);
12          Thread out = new Thread(output);
13          //开启线程
14          in.start();
15          out.start();
16      }
17  }
18
19  //数据类
20  class  Desc2{
21      String name;
22      String sex;
23
24      boolean flag = false;//用于执行唤醒等待的切换
25
26      //负责输入
27      public void setData(String name,String sex) {
28          if (flag == true) {//当flag值为true,就让当前的线
    程处于等待状态
29              try {
30                  wait();
31              } catch (InterruptedException e) {
32                  // TODO Auto-generated catch block
33                  e.printStackTrace();
34              }//当执行这行代码的时候,这里对应的是哪个线程,就操
    作的是哪个线程
35          }
36
37          this.name = name;
38          this.sex = sex;
39
40          flag = !flag;
```

```
41
42          notify();//唤醒的是通一把锁下的线程,因为现在只有一个输入线程,一个输出线程.所以这里唤醒的是输出线程
43          //当线程池中没有被当前的锁标记的线程可唤醒时,我们成为空唤醒,空唤醒不影响程序的执行.
44      }
45      //负责输出
46      public void getData() {
47          if (flag == false) {//让输出线程等待
48              try {
49                  wait();
50              } catch (InterruptedException e) {
51                  // TODO Auto-generated catch block
52                  e.printStackTrace();
53              }
54          }
55          System.out.println("姓名:"+name+"    性别:"+sex);
56
57          flag = ! flag;
58
59          notify();//唤醒的是输入线程
60      }
61  }
62
63  //输入任务
64  class Input2 implements  Runnable{
65      Desc2 desc;
66      public Input2(Desc2 desc) {
67          this.desc = desc;
68      }
69
70      @Override
```

```
    public void run() {
        int i=0;
        while (true) {
            synchronized (desc) {
                if (i == 0) {
                    desc.setData("超超", "男");
                }else {
                    desc.setData("欣欣", "女");
                }
                i=(i+1)%2;
            }
        }
    }
}

//输出任务
class  Output2 implements  Runnable{
    Desc2 desc;
    public Output2(Desc2 desc) {
        this.desc = desc;
    }
    @Override
    public void run() {
        while (true){
            synchronized (desc) {
                desc.getData();
            }
        }
    }
}


```

4.1.2. 生产者消费者模式

生产者消费者问题是研究多线程程序经典问题之一，它描述是有一块缓冲区作为仓库，生产者可以将产品放入仓库，消费者则可以从仓库中取走产品。在Java中一共有四种方法支持同步，其中前三个是同步方法，一个是管道方法。

（1）**Object**的wait() / notify()方法 （2）**Lock**和**Condition**的await() / signal()方法 （3）**BlockingQueue**阻塞队列方法
（4）**PipedInputStream / PipedOutputStream**

这里只对第一种第二种做讲解,其他的有兴趣的小伙伴可以自己进阶学习

咱们前面通过---打印机打印实例--的深入理解,最终的代码实现的就是生产者消费者模式,对应的是单生产者单消费者.下面我们就使用标准模型代码理解一下.

4.1.2.1. 单生产者消费者(会)

```
package com.qf.test;
 /*
 * 单生产者单消费者
 * 需要的线程:两个---一个生产线程一个消费线程
 * 需要的任务:两个---一个生产任务一个消费任务
 * 需要数据:一份---产品
 */
public class Demo5 {
    public static void main(String[] args) {
        //准备数据
        Product product = new Product();
        //准备任务
        Producer producer = new Producer(product);
        Consumer consumer = new Consumer(product);
```

```
        //准备生产线程消费线程
        Thread pro = new Thread(producer);
        Thread con = new Thread(consumer);
        //开启线程
        pro.start();
        con.start();
    }
}

//创建数据类--产品
class Product {
    String name;//名字
    double price;//价格
    int number;//数量

    //标识--控制唤醒等待
    boolean flag = false;

    //准备生产
    public synchronized void  setProduce(String
name,double price){
        if (flag == true){
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.name = name;
        this.price = price;
```

```
44
   System.out.println(Thread.currentThread().getName()+"
   生产了:"+this.name+"   价格:"+this.price+"  数
   量:"+this.number);
45
46          number++;
47
48          flag = !flag;
49          notify();
50      }
51      //准备消费
52      public synchronized void  getConsume(){
53          if (flag == false){
54              try {
55                  wait();
56              } catch (InterruptedException e) {
57                  e.printStackTrace();
58              }
59          }
60
61
   System.out.println(Thread.currentThread().getName()+"
   消费了:"+this.name+"   价格:"+this.price);
62
63          flag = !flag;
64          notify();
65      }
66  }
67
68  //创建生产任务
69  class Producer implements Runnable{
70      Product product;
71
```

```
72      public Producer(Product product) {
73          this.product = product;
74      }
75
76      @Override
77      public void run() {
78          while (true) {
79              product.setProduce("bingbing", 10);
80          }
81      }
82  }
83  //创建消费任务
84  class Consumer implements  Runnable{
85      Product product;
86
87      public Consumer(Product product) {
88          this.product = product;
89      }
90
91      @Override
92      public void run() {
93          while (true) {
94              product.getConsume();
95          }
96      }
97  }
```

4.1.2.1. 多生产者多消费者(了解)

总结:将单生产者单消费者代码不做更改,直接再添加一个生产线程,一个消费线程,就形成了多生产者多消费者多消费者.

- 出现的错误1

    - 错误描述:当有两个生产线程,两个消费线程同时存在的时候,有可能出现生产一次,消费多次或者生产多次消费一次的情况.
    - 原因:当线程被重新唤醒之后,没有判断标记,直接执行了下面的代码
    - 解决办法:将标记处的if改成while
- 出现的错误2
    - 问题描述:继续运行程序,会出现死锁的情况(4个线程同时处于等待状态)
    - 原因:唤醒的是本方的线程,最后导致所有的线程都处于等待状态.
    - 解决办法:将notify改成notifyAll.保证将对方的线程唤醒

```
package com.qf.test;
/*
 * 多生产者多消费者
 * 需要的线程:四个---两个生产线程两个消费线程
 * 需要的任务:两个---一个生产任务一个消费任务
 * 需要数据:一份---产品
 *
 * 生产任务与消费任务共用一个数据--产品类

 * 要求:最终也要实现一次生产一次消费
 */
public class Demo6 {
    public static void main(String[] args) {
        //准备数据
        Product1 product = new Product1();
        //准备任务
        Producer1 producer = new Producer1(product);
        Consumer1 consumer = new Consumer1(product);
        //准备生产线程消费线程
        Thread pro1 = new Thread(producer);
        Thread pro2 = new Thread(producer);
        Thread con1 = new Thread(consumer);
```

```
        Thread con2 = new Thread(consumer);
        //开启线程
        pro1.start();
        con1.start();
        pro2.start();
        con2.start();
    }
}

//创建数据类--产品
class Product1 {
    String name;//名字
    double price;//价格
    int number;//数量

    //标识--控制唤醒等待
    boolean flag = false;

    //准备生产
    public synchronized void  setProduce(String name,double price){
        while (flag == true){
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.name = name;
        this.price = price;
```

```
52
System.out.println(Thread.currentThread().getName()+"生产了:"+this.name+"   价格:"+this.price+"  数量:"+this.number);
53
54          number++;
55
56          flag = !flag;
57          //notify();
58          notifyAll();
59      }
60      //准备消费
61      public synchronized void  getConsume(){
62          while (flag == false){
63              try {
64                  wait();
65              } catch (InterruptedException e) {
66                  e.printStackTrace();
67              }
68          }
69
70
System.out.println(Thread.currentThread().getName()+"消费了:"+this.name+"   价格:"+this.price);
71
72          flag = !flag;
73          //notify();
74          notifyAll();
75      }
76  }
77
78  //创建生产任务
79  class Producer1 implements Runnable{
```

```
    Product1 product;

    public Producer1(Product1 product) {
        this.product = product;
    }

    @Override
    public void run() {
        while (true) {
            product.setProduce("bingbing", 10);
        }
    }
}

//创建消费任务
class Consumer1 implements  Runnable{
    Product1 product;

    public Consumer1(Product1 product) {
        this.product = product;
    }

    @Override
    public void run() {
        while (true) {
            product.getConsume();
        }
    }
}
```

4.1.3. Lock锁

- 为什么使用Lock锁?

  在我们使用synchronized进行同步的时候,锁对象是Object类的对象,使用的wait,notify方法都来自Object类,但是咱们知道并不是所有的对象都会用到同步,所以这样用法不太合理,而且锁相关的功能很多,Lock就是将锁面向对象的结果.不光是将锁面向对象了,同时将wait,notify等方法也做了面相对象处理.形成了Condition接口.当我们想实现多生产者多消费者模式时,可以使用Lock实现同步,同时配合Condition接口实现唤醒等待.

```
    //创建锁对象
    Lock lock = new ReentrantLock();
    //用于生产任务的Condition
    Condition proCon = lock.newCondition();
    //用于消费任务的Condition
    Condition conCon = lock.newCondition();
```

- 比较synchronized和Lock

  1.synchronized:从jdk1.0就开始使用的同步方法-称为隐式同步

  synchronized(锁对象){//获取锁 我们将锁还可以称为锁旗舰或者监听器

  同步的代码

  }//释放锁

  2.Lock:从jdk1.5开始使用的同步方法-称为显示同步

- 原理:Lock本身是接口,要通过他的子类创建对象干活儿
- 常用子类:ReentrantLock
- 使用过程:

首先调用lock()方法获取锁

进行同步的代码块儿

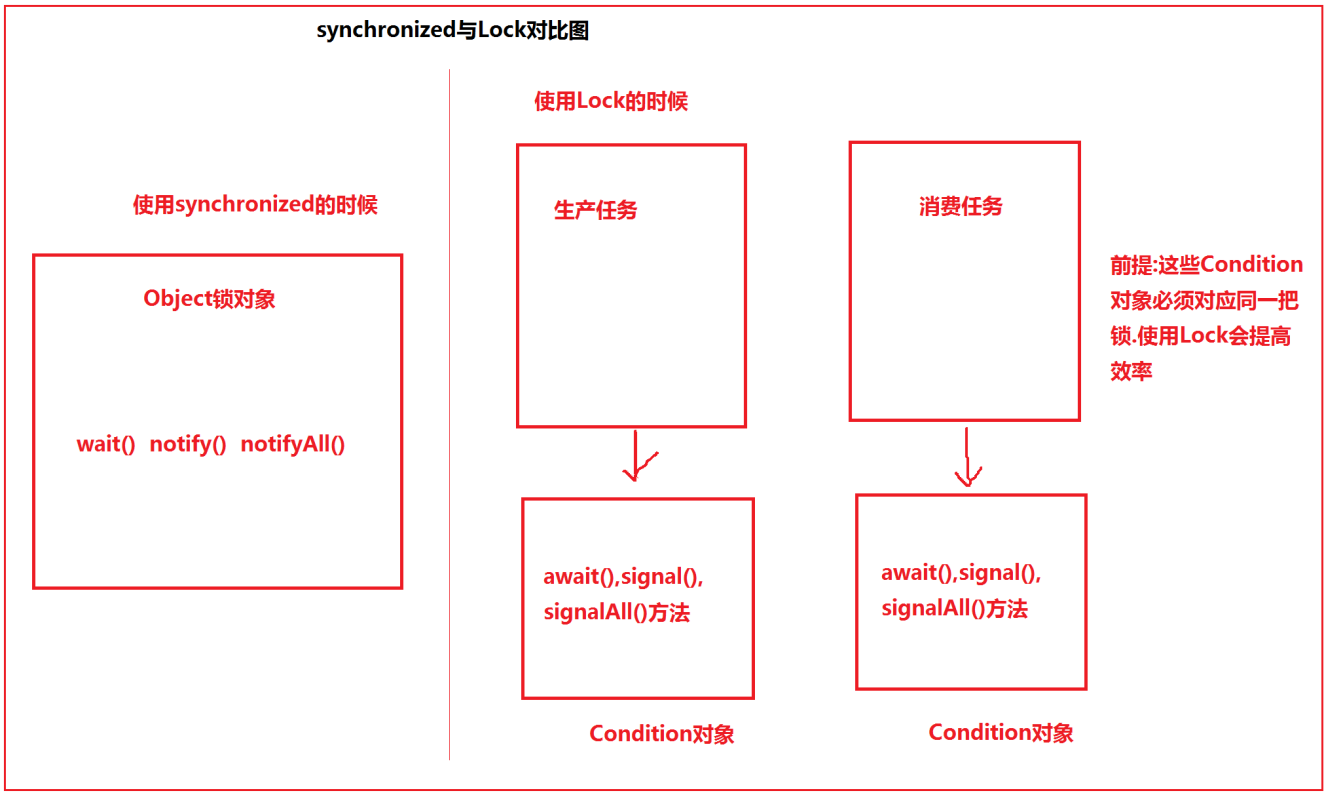
使用unlock()方法释放锁

- 使用的场景:

  当进行多生产者多消费者的功能时,使用Lock,其他的都使用synchronized

- 使用效率:Lock高于synchronized

- 比较Object多wait,notify和Condition的await,signal



- 示例代码

  说明:通过对多生产者多消费者代码的改进,实现用lock替换wait,notify

```
1 package com.qf.test;
2
3 import java.util.concurrent.locks.Condition;
```

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

public class Demo7 {
    public static void main(String[] args) {
        //准备数据
        Product2 product = new Product2();
        //准备任务
        Producer2 producer = new Producer2(product);
        Consumer2 consumer = new Consumer2(product);
        //准备线程
        Thread proThread1 = new Thread(producer);
        Thread proThread2 = new Thread(producer);
        Thread conThread1 = new Thread(consumer);
        Thread conThread2 = new Thread(consumer);
        //开启线程
        proThread1.start();
        conThread1.start();
        proThread2.start();
        conThread2.start();
    }
}

//创建产品
class Product2{
    String name;//产品的名字
    double price;//产品的价格
    int count;//生产的产品数量

    //标识
    boolean flag = false;

    //创建锁对象
```

```
37      Lock lock = new ReentrantLock();
38      //用于生产任务的Condition
39      Condition proCon = lock.newCondition();
40      //用于消费任务的Condition
41      Condition conCon = lock.newCondition();
42
43      //准备生产
44      public  void setProduce(String name,double price){
45          try {
46              lock.lock();//获取锁
47              while (flag == true) {
48                  try {
49                      //wait();//让生产线程等待
50                      proCon.await();
51                  } catch (InterruptedException e) {
52                      // TODO Auto-generated catch block
53                      e.printStackTrace();
54                  }
55              }
56
57              this.name = name;
58              this.price = price;
59
System.out.println(Thread.currentThread().getName()+"
生产了:"+this.name+"    产品的数量:"+this.count+"    价
格:"+this.price);
60
61              count++;
62              flag = ! flag;
63              //notify();//唤醒消费线程
64              //notifyAll();
65              conCon.signal();
66          }finally {
```

```
            lock.unlock();//释放锁
        }

    }
    //准备消费
    public   void getConsume() {
        try {
            lock.lock();
            while (flag == false) {
                try {
                    //wait();//让消费线程等待
                    conCon.await();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }

System.out.println(Thread.currentThread().getName()+"消费了:"+this.name+"   产品的数量:"+this.count+"   价格:"+this.price);
            //唤醒生产线程
            flag = ! flag;
            //notify();
            //notifyAll();
            proCon.signal();
        }finally {
            lock.unlock();
        }
    }
}
//创建生产任务
class Producer2 implements Runnable{
```

```
97      Product2 product;
98      public Producer2(Product2 product) {
99          super();
100         this.product = product;
101     }
102     public void run() {
103         while (true) {
104             product.setProduce("bingbing", 10);
105         }
106
107     }
108 }
109 //创建消费任务
110 class Consumer2 implements Runnable{
111     Product2 product;
112     public Consumer2(Product2 product) {
113         super();
114         this.product = product;
115     }
116     public void run() {
117         while (true) {
118             product.getConsume();
119         }
120     }
121 }
122
```

4.1.4. 唤醒等待机制(了解)

4.1.4.1. 方法简介

Object类中几个方法如下：

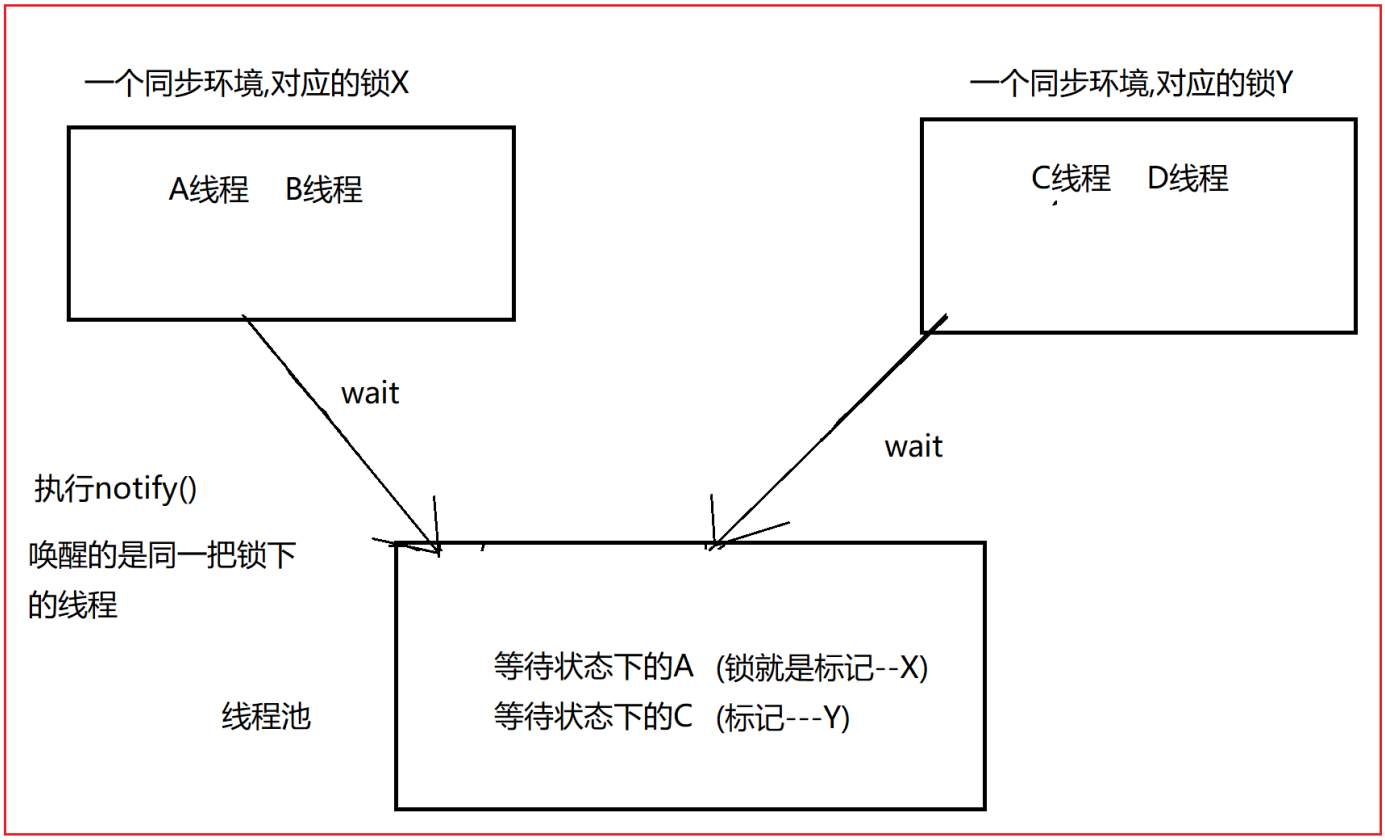
- wait()
  - 等待，让当前的线程，释放自己持有的指定的锁标记，进入到等待队列。
  - 等待队列中的线程，不参与CPU时间片的争抢，也不参与锁标记的争抢。
- notify()
  - 通知、唤醒。唤醒等待队列中，一个等待这个锁标记的随机的线程。
  - 被唤醒的线程，进入到锁池，开始争抢锁标记。
- notifyAll()
  - 通知、唤醒。唤醒等待队列中，所有的等待这个锁标记的线程。
  - 被唤醒的线程，进入到锁池，开始争抢锁标记。

4.1.4.2. wait和sleep的区别

- sleep()方法，在休眠时间结束后，会自动的被唤醒。 而wait()进入到的阻塞态，需要被notify/notifyAll手动唤醒。
- wait()会释放自己持有的指定的锁标记，进入到阻塞态。sleep()进入到阻塞态的时候，不会释放自己持有的锁标记。

4.1.4.3. 注意事项

- 无论是wait()方法，还是notity()/notifyAll()方法，在使用的时候要注意，一定要是自己持有的锁标记，才可以做这个操作。否则会出现 IllegalMonitorStateException 异常。
- 为什么wait,notify方法要使用锁调用?



4.4.5. 死锁 (了解)

出现的情况有两种

- 所有的线程处于等待状态

  大家都处于等待状态,没有人获取cpu使用

- 锁之间进行嵌套调用

  多个线程， 同时持有对方需要的锁标记， 等待对方释放自己需要的锁标记。此时就是出现死锁。 线程之间彼此持有对方需要的锁标记， 而不进行释放， 都在等待。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Program {
```

```
7       public static void main(String[] args) {
8           Runnable runnable1 = () -> {
9               synchronized ("a") {
10                  System.out.println("线程A，持有了a锁，在等
    待b锁");
11                  synchronized ("b") {
12                      System.out.println("线程A同时持有了a锁
    和b锁");
13                  }
14              }
15          };
16
17          Runnable runnable2 = () -> {
18              synchronized ("b") {
19                  System.out.println("线程B，持有了b锁，在等
    待a锁");
20                  synchronized ("a") {
21                      System.out.println("线程B同时持有了a锁
    和b锁");
22                  }
23              }
24          };
25          new Thread(runnable1, "A").start();
26          new Thread(runnable2, "B").start();
27      }
28  }
```

4.3. 线程其他内容(了解)

4.3.1. 线程的停止

```
package com.qf.test;
/*
 * 线程的停止:3种
 * 1.通过一个标识结束线程
 * 2.调用stop方法---因为有固有的安全问题,所以系统不建议使用.
 * 3.调用interrupt方法----如果目标线程等待很长时间（例如基于一个条件变量），则应使用 interrupt 方法来中断该等待。
 */
// * 1.通过一个标识结束线程
//public class Demo8 {
//    public static void main(String[] args) {
//        MyTest myTest = new MyTest();
//        Thread t1 = new Thread(myTest);
//        t1.start();
//
//        try {
//            Thread.sleep(100);
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }
//
//        int i = 0;
//        while (true){
//            if (++i == 10){
//                myTest.flag = false;
//
//                break;//关闭主线程
//            }
//        }
//    }
//}
```

```
31 //
32 //class  MyTest implements  Runnable{
33 //    boolean flag = true;
34 //    @Override
35 //    public void run() {
36 //        while (flag){
37 //
   System.out.println(Thread.currentThread().getName()+"ha
   ha");
38 //        }
39 //    }
40 //}
41
42 //3.调用interrupt方法----如果目标线程等待很长时间（例如基于一个
   条件变量），则应使用 interrupt 方法来中断该等待。
43 public class Demo8 {
44     public static void main(String[] args) {
45         MyTest myTest = new MyTest();
46         Thread t1 = new Thread(myTest);
47         t1.start();
48
49         try {
50             Thread.sleep(100);
51         } catch (InterruptedException e) {
52             e.printStackTrace();
53         }
54
55         int i = 0;
56         while (true){
57             if (++i == 10){//当i==10的时候,我就让子线程结
   束,直接调用interrupt方法
58                 t1.interrupt();
59
```

```
60                  break;//关闭主线程
61              }
62          }
63      }
64  }
65
66  class  MyTest implements  Runnable{
67      boolean flag = true;
68      @Override
69      public synchronized void run() {
70          while (flag){
71              try {
72                  wait();
73              } catch (InterruptedException e) {
74                  e.printStackTrace();
75                  flag = false;
76              }
77
    System.out.println(Thread.currentThread().getName()+"ha
    ha");
78          }
79      }
80  }
81
```

4.3.2. 线程的休眠

线程休眠， 就是让当前的线程休眠指定的时间。 休眠的线程进入到阻塞状态， 直到休眠结束。 阻塞的线程， 不参与CPU时间片的争抢。

注: 线程休眠的时间单位是毫秒。

```java
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Program {
    public static void main(String[] args) {
        // 使用接口的方式进行线程的实例化
        Runnable runnable = () -> {
            for (int i = 0; i < 10; i++) {

System.out.println(Thread.currentThread().getName() + "
: " + i);
                try {
                    // 线程休眠
                    Thread.sleep(1000);
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        // 实例化两个线程,  处理的逻辑完全相同
        Thread thread0 = new Thread(runnable, "t0");
        Thread thread1 = new Thread(runnable, "t1");

        thread0.start();
        thread1.start();
    }
}
```

4.3.3. 线程的合并

将一个线程中的任务， 合并入到另外一个线程中执行， 此时， 合并进来的线程有限执行。 类似于： 插队。

注意:优先级只比main线程的高.对其他的线程没有影响.

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Program {
    // 记录余票数量
    static int ticketCount = 100;

    public static void main(String[] args) {
        // 循环卖票
     while (ticketCount > 0) {
            // 票数 -1
        System.out.println("窗口卖出一张票给散客，剩余： " + --ticketCount);
        // 卖出 30 张
        if (ticketCount == 70) {
            // 实例化一个 VIP 团体线程
            Thread vip = new Thread(() -> {
                for (int i = 0; i < 50; i++) {
                    System.out.println("窗口卖出一张票给VIP团队，剩余： " + --ticketCount);
                }
            });
        // 先开启
        vip.start();
        // 再合并
```

```
26          try {
27            vip.join();
28          } catch (InterruptedException e) {
29            e.printStackTrace();
30          }
31            }
32         }
33      Thread thread = new Thread(runnable);
34      thread.start();
35      }
36  }
```

4.3.4. 线程的优先级设置

设置线程的优先级， 可以决定这个线程能够抢到CPU时间片的概率。 线程的优先级范围在 [1, 10]， 默认的优先级是5。 数值越高， 优先级越高。 但是要注意， 并不是优先级高的线程一定能抢到CPU时间片， 也不是优先级的线程一定抢不到CPU时间片。 线程的优先级只是决定了这个线程能够抢到CPU时间片的概率。 即便是优先级最低的线程， 依然可以抢到CPU时间片。

```
1  /**
2   * @Author 千锋大数据教学团队
3   * @Company 千锋好程序员大数据
4   * @Description
5   */
6  public class Program {
7      public static void main(String[] args) {
8          // 使用接口的方式进行线程的实例化
9          Runnable runnable = () -> {
10             for (int i = 0; i < 100; i++) {
```

```
11
System.out.println(Thread.currentThread().getName() + "
: " + i);
12              }
13          };
14          // 实例化两个线程，  处理的逻辑完全相同
15          Thread thread0 = new Thread(runnable, "t0");
16          Thread thread1 = new Thread(runnable, "t1");
17
18          // 设置线程的优先级，  必须在这个线程启动之前
19          thread0.setPriority(1);
20          thread1.setPriority(10);
21
22          thread0.start();
23          thread1.start();
24      }
25  }
```

4.3.5. 守护线程

守护线程， 又叫后台线程。 是一个运行在后台， 并且会和前台线程争抢CPU时间片的线程。

- 守护线程依然会和前台线程争抢CPU时间片， 实现并发的任务。
- 在一个进程中， 如果所有的前台线程都结束了， 后台线程即便任务没有执行结束， 也会自动结束。

```
1  /**
2   * @Author 千锋大数据教学团队
3   * @Company 千锋好程序员大数据
4   * @Description
5   */
```

```java
public class Program {
    public static void main(String[] args) {
        // 实例化一个线程
        Thread thread = new Thread(() -> {
            while (true) {
                System.out.println("守护线程在运行");
                try {
                  Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        // 将一个线程设置为守护线程
        thread.setDaemon(true);
        // 开启线程
        thread.start();

        for (int i = 0; i < 10; i++) {
            System.out.println("主线程: " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

4.4. 线程池(了解)

4.4.1. 线程池的简介

线程池， 其实就是一个容器， 里面存储了若干个线程。

使用线程池， 最主要是解决线程复用的问题。 之前使用线程的时候， 当我们需要使用一个线程时， 实例化了一个新的线程。 当这个线程使用结束后， 对这个线程进行销毁。 对于需求实现来说是没有问题的， 但是如果频繁的进行线程的开辟和销毁， 其实对于CPU来说， 是一种负荷， 所以要尽量的优化这一点。

可以使用复用机制解决这个问题。 当我们需要使用到一个线程的时候， 不是直接实例化， 而是先去线程池中查找是否有闲置的线程可以使用。 如果有， 直接拿来使用； 如果没有， 再实例化一个新的线程。 并且， 当这个线程使用结束后， 并不是马上销毁， 而是将其放入到线程池中， 以便下次继续使用。

4.4.2. 线程池的开辟

在Java中， 使用ThreadPoolExecutor类来描述线程池， 在这个类的对象实例化的时候， 有几个常见的参数：

参数	描述
int corePoolSize	核心线程的数量
int maximunPoolSize	线程池最大容量（包含了核心线程和临时线程）
long keepAliveTime	临时线程可以空闲的时间
TimeUnit unit	临时线程保持存活的时间单位
BlockingQueue<Runnable> workQueue	任务等待队列
RejectedExecutionHandler handler	拒绝访问策略

> - BlockingQueue
>   - ArrayBlockingQueue
>   - LinkedBlockingQueue
>   - SynchronouseQueue
> - RejectedExecutionHandler
>   - ThreadPoolExecutor.AbortPolicy : 丢弃新的任务，并抛出异常 RejectedExecutionException
>   - ThreadPoolExecutor.DiscardPolicy： 丢弃新的任务，但是不会抛出异常
>   - ThreadPoolExecutor.DiscardOldestPolicy： 丢弃等待队列中最早的任务
>   - ThreadPoolExecutor.CallerRunsPolicy : 不会开辟新的线程，由调用的线程来处理

4.4.3. 线程池的工作原理

线程池中的所有线程， 可以分为两部分： **核心线程** 和 **临时线程**

核心线程：

核心线程常驻于线程池中， 这些线程， 只要线程池存在， 他们不会被销毁。 只有当线程池需要被销毁的时候， 他们才会被销毁。

临时线程：

就是临时工。 当遇到了临时的高密度的线程需求时， 就会临时开辟一些线程， 处理一些任务。 这些临时的线程在处理完自己需要处理的任务后， 如果没有其他的任务要处理， 就会闲置。 当闲置的时间到达了指定的时间之后， 这个临时线程就会被销毁。

任务分配逻辑：

1. 当需要处理并发任务的时候， 优先分配给核心线程处理。
2. 当核心线程都已经分配了任务， 又有新的任务出现时，会将这个新的任务存入等待队列。
3. 当等待队列被填满后， 再来新的任务时， 会从开辟一个临时线程，处理这个新的任务。
4. 当临时线程加核心线程数量已经到达线程池的上限，再来新的任务的时候，就会触发拒绝访问策略。

4.4.4. 线程池的常用方法

方法	描述
execute(Runnable runnable)	将任务提交给线程池， 由线程池分配线程来并发处理。
shutdown()	向线程池发送一个停止信号，这个操作并不会停止线程池中的线程，而是在线程池中所有的任务都执行结束后，结束线程和线程池。
shutdownNow()	立即停止线程池中的所有的线程和线程池。

4.4.5. 线程池的工具类

线程池的开辟， 除了可以使用构造方法进行实例化， 还可以通过Executors工具类进行获取。 实际应用中， 大部分的场景下， 可以不用前面的构造方法进行线程池的实例化， 而是用Executors工具类中的方法进行获取。

方法	描述
Executors.newSingleThreadExecutor()	核心线程1 最大线程1 等待队列容量 Integer.MAX_VALUE
Executors.newCachedThreadPool()	核心线程0 最大线程Integer.MAX_VALUE 闲置时间 60
Executors.newFixedThreadPool（int size)	核心线程size 最大线程 size 等待队列容量 Integer.MAX_VALUE
submit()	向线程池中添加任务
shutdown()	停止线程池