Lecture12-1 多线程编程&网页

1. 多线程概述

介绍

经典的编程模型中，有一个单一的**中央处理器 central processor** 处理和执行所有的指令，然而，现代计算机有多核处理器，可以同时执行多个任务，充分利用这些处理器，你需要写一些程序来做**并行处理 parallel processing**

对于 Java 来说，这意味着学习 `Thread`，单一的线程与之前学的一样，而超过一个 `Thread` 可以同时运行（并行处理）

并行程序中的线程很少是彼此完全独立的，他们通常需要合作和沟通，学习管理和控制线程之间的合作是主要难点

使用线程的原因

- 计算速度更快（多个线程同时工作）
- 使用线程来处理“阻塞”的操作，例如，程序在等待数据通过网络连接到达时被阻塞，多线程使程序的一部分继续执行有用的工作

进程与线程

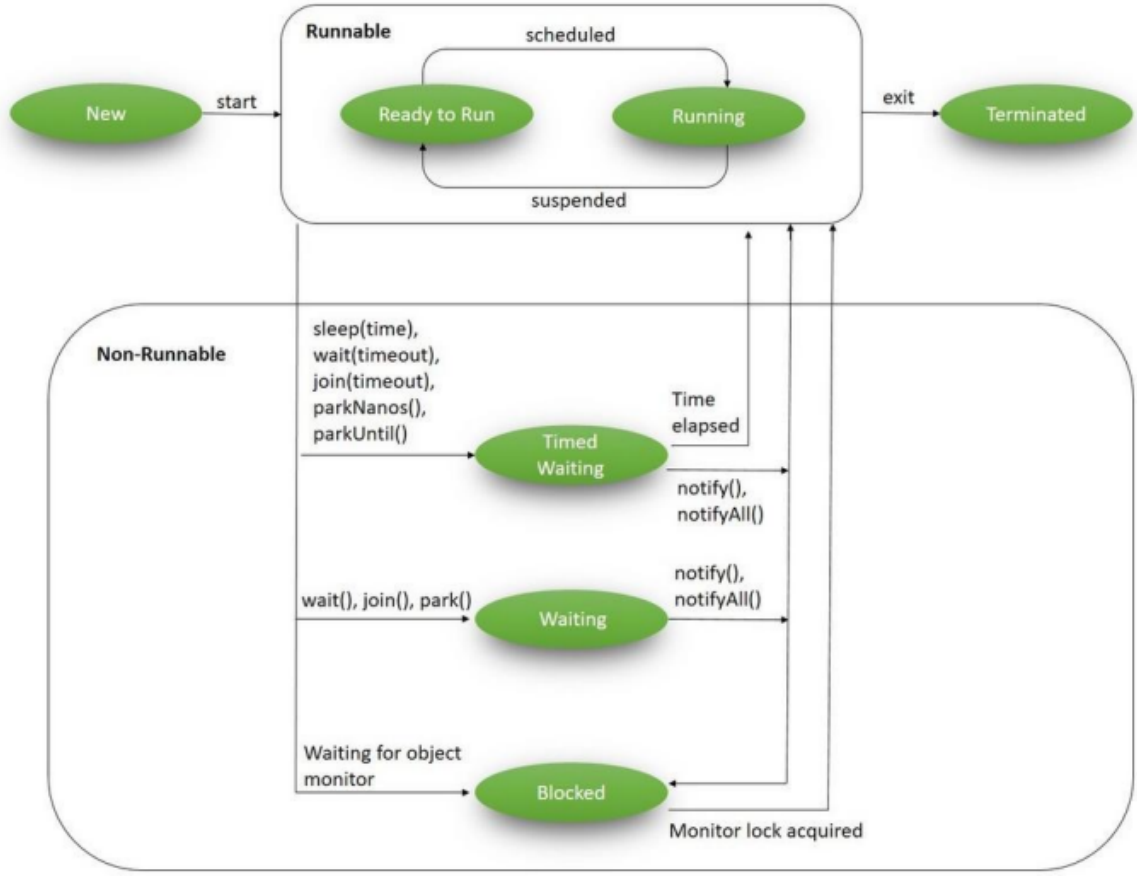
进程 Process

- 运行在电脑上的程序实例（所有的独立的可执行程序）

线程 Threads

- 进程内可分派的工作单元
- 多线程处理器中的 Threads 可能和软件中的 Threads 相同，也可能不同
- 进程内可分派的工作单元
    - 包含处理器内容（包括程序计数器和堆栈指针）和堆栈的数据区
    - 线程按顺序执行
    - 可中断：处理器可以转向另一个线程
- **线程切换 Thread Switch**
    - 处理器在**同一进程**内完成的线程之间切换
    - 成本通常比进程切换的成本要低

线程的状态



2. 多线程的介绍

在Java中，单个任务被称为**线程** `thread`

术语“线程”指的是“控制线程”或“执行线程”，意思是一个接一个执行的指令序列——线程通过时间扩展，将每个指令连接到下一个指令，在多线程程序中，可以有许多控制线程，它们通过时间并行地编织并形成程序的完整结构

每一个 Java 程序都至少有一个线程，当 Java 虚拟机运行程序时，它会创建一个线程，负责执行程序的 `main` 线程，主线程可以创建其它的线程，这些线程甚至可以在主线程结束了后继续

Thread 类

- 在 Java 中，线程被表示成一个属于 `java.lang.Thread` 的对象（或者它的子类）
- `Thread` 对象仅执行**一个单一的方法一次**，这个方法代表着这个线程需要执行的任务
- 这个方法是归属于它自己的线程所控制的，所以可以并行的开多个线程
- 当线程执行的这个唯一的方法结束了之后，该 `Thread` 对象无法再一次被利用

创建和运行线程

有两种方式编写多线程

继承 Thread

一种是创建一个继承了 `Thread` 的子类，来重写 `run()` 方法

```java
public class NameThread extends Thread{
    private String name; // 线程名字
    public NameThread(String name){
        this.name = name;
    }

    public void run(){ // run 方法打印了线程的名字
        System.out.println("Greetings from thread '" + name + "'!");
    }
}
```

使用 `NameThread`，你必须创造一个该类的对象

```java
NameThread greetings = new NameThread("Fred");
```

但是，创建对象并不会自动启动线程运行或执行其 `run()` 方法

```java
greetings.start();
```

方法 `start()` 在启动新的控制线程（开始执行线程子类的 `run()` 方法）后立即返回，它不会等下线程终止，这意味着线程的 `run()` 方法中的代码与调用 `start()` 方法之后的语句同时执行

```java
NameThread greetings = new NameThread("Fred");
greetings.start();
System.out.println("Thread has been started");
```

实现 Runnable

`Thread` 类有一个构造器，它将 `Runnable` 作为参数

当一个实现了 `Runnable` 的对象传给了构造器，那么 `Thread` 对象的 `run()` 方法会直接调用实现了 `Runnable` 对象的 `run()` 方法

```java
public class NameRunnable implements Runnable{
    private String name;
    public NameRunnable(String name){
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println("Greetings from runnable '" + name + "'!");
    }
}
```

为了使用这个类，我们要创建一个 `NameRunnable` 对象，然后使用该对象来创造一个 `Thread` 对象

```java
NameRunnable greetings = new NameRunnable("Fred");
Thread greetingsThread = new Thread(greetings);
greetingsThread.start();
System.out.println("Thread has been started");
```

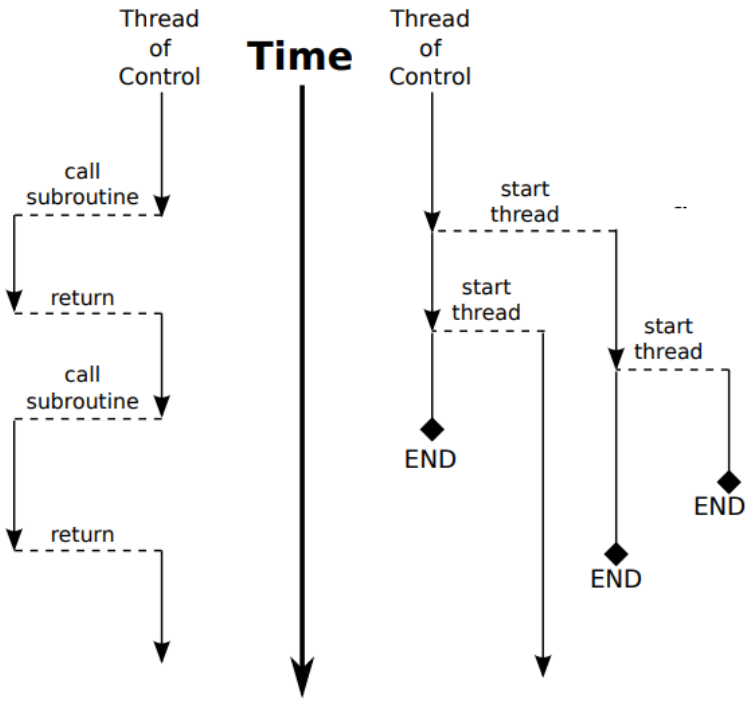
其实，`Runnable` 对象也可以是一个匿名对象，通过 Lambda 表达式传入 `Thread` 里

```java
Thread greetingFromFred = new Thread(() -> System.out.println("Greetings from Fred!"));
greetingFromFred.start();
```

调用线程与调用例程的区别

需要注意的是调用 `greeting.start()` 与调用 `greetings.run()` 有很大的区别

- 调用 `greetings.run()` 只会在当前线程中执行 `run()` 方法的内容，也就不是并行处理了
- 调用 `greetings.start()` 则会创建一个新的线程来执行 `run()` 方法的内容



- 当一个线程调用一个子程序 subroutine 时，仍然**只有一个控制线程**，它在子程序中待一段时间，直到子程序返回
    - 称左边的为同步通讯
- 当一个线程启动另一个线程 thread 时，会有**一个新的控制线程与原始控制线程并行运行**，甚至可以在原始线程终止后继续运行
    - 称右边的为异步通讯

线程完成的顺序与它们启动的顺序不相同，而且**顺序是不确定的**，也就是说，如果程序再次运行，线程完成的顺序可能会不同

每当要运行的线程数量超过要运行它们的处理器数量时，计算机就会通过快速地从一个线程切换到另一个线程，将其注意力分散到所有可运行线程中，也就是说，每个处理器运行一个线程一段时间，然后切换到另一个线程并运行该线程一段时间，被成为 “**上下文切换 context switches**”

第一个线程示例

```
1  public class GreetingRunnable implements Runnable{
2
3      private static final int REPETITIONS = 10;
4      private static final int DELAY = 1000;
5
6      private String greeting;
7
8      public GreetingRunnable(String aGreeting){
9          greeting = aGreeting;
10     }
11     @Override
12     public void run() {
13         try{
14             for(int i = 0; i < REPETITIONS; i++){
```

```
                Date now = new Date();
                System.out.println(now + " " + greeting);
                Thread.sleep(DELAY);
            }
        }catch (InterruptedException ignored){}
    }
}
```

```
public static void main(String[] args) {
    GreetingRunnable r1 = new GreetingRunnable("Hello");
    GreetingRunnable r2 = new GreetingRunnable("Goodbye");
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
    t1.start();
    t2.start();
}
```

```
Tue Nov 30 19:16:55 CST 2021 Goodbye
Tue Nov 30 19:16:55 CST 2021 Hello
Tue Nov 30 19:16:56 CST 2021 Goodbye
Tue Nov 30 19:16:56 CST 2021 Hello
Tue Nov 30 19:16:57 CST 2021 Goodbye
Tue Nov 30 19:16:57 CST 2021 Hello
Tue Nov 30 19:16:58 CST 2021 Hello
Tue Nov 30 19:16:58 CST 2021 Goodbye
Tue Nov 30 19:16:59 CST 2021 Hello
Tue Nov 30 19:16:59 CST 2021 Goodbye
Tue Nov 30 19:17:00 CST 2021 Goodbye
Tue Nov 30 19:17:00 CST 2021 Hello
Tue Nov 30 19:17:01 CST 2021 Hello
Tue Nov 30 19:17:01 CST 2021 Goodbye
Tue Nov 30 19:17:02 CST 2021 Goodbye
```

3. 线程的操作

获得本电脑处理器的数量

```
Runtime.getRuntime().availableProcessors();
```

Start()

当调用了 `Start()` 后，线程会立马执行 `run()` 方法，直到它因为某种情况结束了

判断线程是否活跃

```
// Thread thrd
bool alive = thrd.isAlive();
```

当线程的终止时，它被称为死了 dead

线程休眠

```
 Thread.sleep(milliseconds); // 一个静态方法
```

使执行此方法的线程“休眠”指定的毫秒数

- 休眠的线程仍然是活跃的 alive，但是它没有运行
- 当一个线程休眠时，计算机仍然可以处理其它正在运行的线程
- 线程休眠可以用来给执行的代码插入一段暂停的时间

`sleep()` 方法会抛出一个 `InterruptedException`，它是一个需要强制异常处理的**受控异常 checked exception**

在实践上，该方法通常要被 `try..catch` 语句包裹

```
try {
    Thread.sleep(lengthOfPause);
}catch (InterruptedException e) {
}
```

线程干扰 interrupt

一个线程可以干扰 interrupt 其它的线程

- 让它停止休眠，活跃起来
- 让它暂停

```
thrd.interrupt();
```

这样做可以方便地将信号从一个线程发送到另一个线程

当线程捕获 `InterruptedException` 异常时，它知道自己被中断了

除了 catch 到的异常意外，线程也可以通过调用静态方法

```
Thread.interrupted();
```

来检查自己是否被中断

join()

有些时候，一个线程需要等待另外的线程死掉后再跑，这是通过 `Thread` 类的 `join()` 方法调用的

如果一个线程调用了 `thrd.join()` 那么其它的线程会休眠，等到 thrd 完成之后再继续

```
CountPrimesThread[] worker = new CountPrimesThread[numberOfThreads];
long startTime = System.currentTimeMillis();
for (int i = 0; i < numberOfThreads; i++) {
    worker[i] = new CountPrimesThread();
    worker[i].start();
}
for (int i = 0; i < numberOfThreads; i++) {
    try {
        worker[i].join(); // Wait until worker[i] finishes, if it hasn't already.
    }catch (InterruptedException e) {
    }
}
// At this point, all the worker threads have terminated.
long elapsedTime = System.currentTimeMillis() - startTime;
System.out.println("Total elapsed time: " + (elapsedTime/1000.0) + " seconds");
```

这段代码假设了没有 `InterruptedException` 发生，如果要绝对确定 `worker[i]` 停止了，需要做出一些改变

```
while (worker[i].isAlive()) {
    try {
        worker[i].join();
    }catch (InterruptedException e) {
    }
}
```

4. 资源竞争问题 Race Condition

问题示例

我们创建一个银行账户，开始的账户余额是 0 元，我们创建两种线程

- 一个线程每次存 $100
- 一个线程每次取 $100

BankAccount

```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount {
   private double balance;

   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount() {
      balance = 0;
   }

   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   public void deposit (double amount) {
      System.out.print( "Depositing " + amount );
      double newBalance = balance + amount;
      System.out.println( ", new balance is " + newBalance );
      balance = newBalance;
   }

   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
   */
   public void withdraw (double amount) {
      System.out.print("Withdrawing " + amount);
      double newBalance = balance - amount;
      System.out.println(", new balance is " + newBalance);
      balance = newBalance;
   }

   /**
      Gets the current balance of the bank account.
      @return the current balance
   */
   public double getBalance() {
      return balance;
   }
}

```

DepositRunnable

```
/**
   A deposit runnable makes periodic deposits to a bank account.
*/
public class DepositRunnable implements Runnable {
   private static final int DELAY = 1;
   private BankAccount account;
   private double amount;
   private int count;

   /**
      Constructs a deposit runnable.
      @param account the account into which to deposit money
      @param amount the amount to deposit in each repetition
      @param count the number of repetitions
   */
   public DepositRunnable (BankAccount account, double amount, int count) {
      this.account = account;
      this.amount = amount;
      this.count = count;
   }

   public void run() {
      try {
         for (int i = 1; i <= count; i++) {
            account.deposit( amount );
            Thread.sleep( DELAY );
         }
      } catch (InterruptedException exception) {}
   }
}
```

WithdrawRunnable

```
/**
   A withdraw runnable makes periodic withdrawals from a bank account.
*/
public class WithdrawRunnable implements Runnable {
   private static final int DELAY = 1;
   private BankAccount account;
   private double amount;
   private int count;

   /**
      Constructs a withdraw runnable.
      @param account the account from which to withdraw money
      @param amount the amount to withdraw in each repetition
      @param count the number of repetitions
```

```
15      */
16      public WithdrawRunnable (BankAccount account, double amount, int count) {
17         this.account = account;
18         this.amount = amount;
19         this.count = count;
20      }
21
22      public void run() {
23         try {
24            for (int i = 1; i <= count; i++) {
25               account.withdraw( amount );
26               Thread.sleep( DELAY );
27            }
28         } catch (InterruptedException exception) {}
29      }
30   }
```

BankAccountThreadRunner

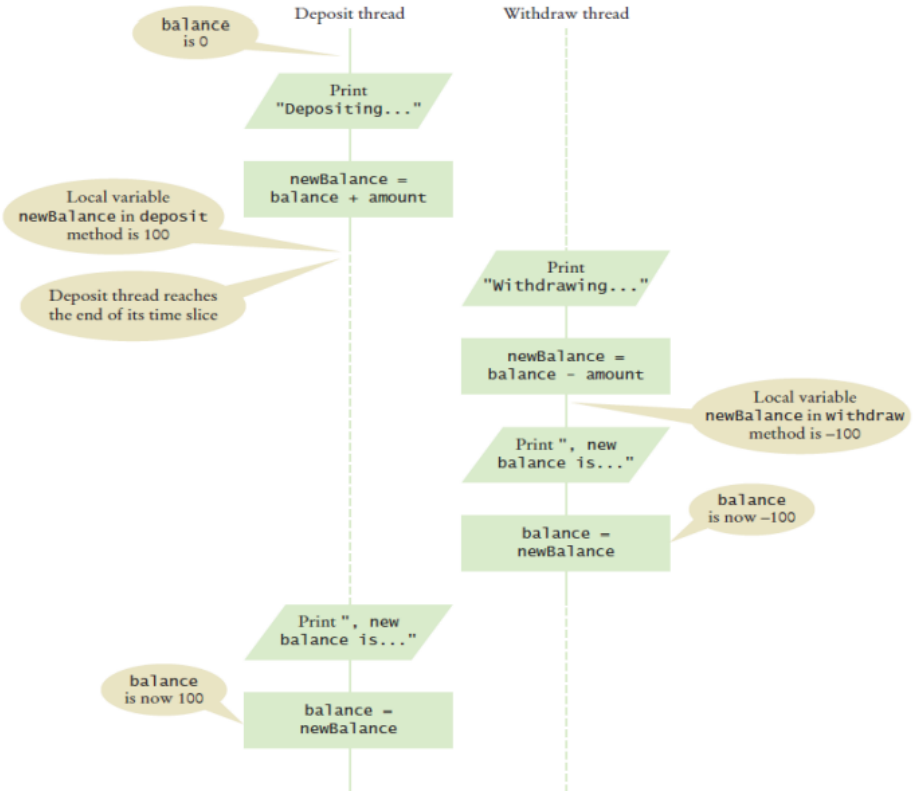
```
1   /**
2      This program runs threads that deposit and withdraw
3      money from the same bank account.
4   */
5   public class BankAccountThreadRunner {
6      public static void main (String[] args) {
7         BankAccount account = new BankAccount();
8         final double AMOUNT = 100;
9         final int REPETITIONS = 100;
10        final int THREADS = 100;
11
12        for (int i = 1; i <= THREADS; i++) {
13           DepositRunnable d =
14              new DepositRunnable( account, AMOUNT, REPETITIONS );
15           WithdrawRunnable w =
16              new WithdrawRunnable( account, AMOUNT, REPETITIONS );
17
18           Thread dt = new Thread(d);
19           Thread wt = new Thread(w);
20
21           dt.start();
22           wt.start();
23        }
24     }
25  }
```

资源竞争问题

执行代码，会发现出现了很多的问题

```
Withdrawing 100.0Depositing 100.0, new balance is 100.0
Depositing 100.0Withdrawing 100.0, new balance is 0.0
Depositing 100.0Depositing 100.0Withdrawing 100.0, new balance is -100.0
, new balance is -100.0
Withdrawing 100.0, new balance is -200.0
Withdrawing 100.0, new balance is -300.0
Withdrawing 100.0, new balance is -400.0
Depositing 100.0Depositing 100.0, new balance is -300.0
Depositing 100.0, new balance is -200.0
Withdrawing 100.0, new balance is -300.0
Withdrawing 100.0, new balance is -400.0
Withdrawing 100.0, new balance is -500.0
Withdrawing 100.0, new balance is -600.0
Withdrawing 100.0, new balance is -700.0
Withdrawing 100.0Withdrawing 100.0, new balance is -800.0
Withdrawing 100.0, new balance is -900.0
Withdrawing 100.0Depositing 100.0Withdrawing 100.0, new balance is -800.0
Withdrawing 100.0, new balance is -300.0
Depositing 100.0Depositing 100.0, new balance is -200.0
Withdrawing 100.0, new balance is 100.0
Withdrawing 100.0Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
, new balance is 100.0
```

这是由于出现了资源竞争



锁

`Lock` 对象用于控制想要操作共享资源的线程

Java 库定义了一个 `Lock` 接口和几个实现该接口的类， `ReentrantLock` 类是最常用的锁类

通常，一个 `Lock` 对象被添加到一个方法访问共享资源的类中，如下所示

```
public class BankAccount{
    private Lock balanceChangeLock;
    // ...
    public BankAccount(){
        balanceChangeLock = new ReentrantLock();
        // ...
    }
}
```

所有操作共享资源的代码都被 `Lock` 对象的 `lock()` 和 `unlock()` 方法包围

```
public void deposit(double amount){
    balanceChangeLock.lock(); // lock
    try{
        System.out.print("Depositing" + amount);
        double newBalance = balance + amount;
        System.out.print(", new balance is " + newBalance);
        balance = newBalance;
    }finally{
        balanceChangeLock.unlock(); // unlock
    }
}
```

死锁 - condition 对象

可以使用锁对象来确保在多个线程访问共享数据时，共享数据处于一致的状态

然而，锁可能会导致另一个问题。可能发生的情况是，一个线程获得了一个锁，然后等待另一个线程做一些基本的工作，如果另一个线程当前正在等待获取相同的锁，那么这两个线程都不能继续

这种情况被称为**死锁 deadlock** 或者 **deadly embrace**

假设我们希望在我们的程序中不允许有负的账户余额，这里有一个简单的方法

在 `WithdrawRunnable` 类的 `run` 方法中，我们可以在取款前检查余额

```
if(account.getBalance() >= amount){
    account.withdraw(amount);
}
```

显然，其实这个验证应该移动到 `withdraw()` 方法内，这样就确保了测试资金充足与实际取款不能分离。因此，`withdraw()` 方法看起来像这样：

```
public void withdraw(double amount){
    balanceChangeLock.lock();
    try{
        while(balance < amount){
            // wait for the balance to grow..
        }
        //...
    }finally{
        balanceChangeLock.unlock();
    }
}
```

为了克服这个问题，我们使用一个 `condition` 对象

`condition` 对象允许一个线程暂时释放锁，以便另一个线程可以继续，并在稍后的时间重新获得锁

```
public class BankAccount{
    private Lock balanceChangeLock;
    private Condition sufficientFoudsCondition;
    // ...
    public BankAccount(){
        balanceChangeLock = new ReentrantLock();
        sufficientFoudsCondition = balanceChangeLock.newCondition();
        // ...
    }
}
```

习惯上是给 `condition` 对象一个描述您想要测试的条件的名称（例如“ sufficient funds ”）

您需要实现一个适当的测试，只要测试没有完成，就调用条件对象的 `await()` 方法

```
public void withdraw(double amount){
    balanceChangeLock.lock();
    try{
        while(balance < amount){
            sufficientFoudsCondition.await();
        }
        //...
    }finally{
        balanceChangeLock.unlock();
    }
}
```

当一个线程调用 `await()` 时，它并不是简单地像一个线程到达它的时间片结束时那样去激活它，相反，它处于**阻塞状态 block state**，并且在解除阻塞之前它不会被线程调度程序激活，要解除阻塞，另一个线程必须在相同的条件对象上执行 `signalAll()` 方法

`signalAll()` 方法将解除阻塞等待该条件的所有线程，然后，它们可以与所有其他正在等待锁对象的线程竞争

最终，其中一个将获得对锁的访问权，并且它将从 `await()` 方法中退出

在实例中，`deposit()` 方法调用 `signalAll()`

```
public void deposit(double amount){
    balanceChangeLock.lock();
    try{
        //...
        sufficientFoudsCondition.signalAll();
    }finally{
        balanceChangeLock.unlock();
    }
}
```

`singalAll()` 的调用通知了等待的线程余额可能可以用了，所以可以尝试是否可以取款

5. 线程冲突的解决 synchronized

线程之间的问题是，它们可能会同时调用某一个资源，这样有可能会产生资源冲突问题，也叫做 **race condition**

要解决竞争条件的问题，必须有某种方法让线程获得对共享资源的**独占访问 exclusive access**

Java 提供 `synchronized` 方法和 `synchronized` 语句，通过确保每次只有一个线程试图访问资源，它们用于保护共享资源

同步在 Java 中的实际规则是：两个线程不能同时同步到同一个对象上，即它们不能同时执行在该对象上同步的代码段，如果一个线程在一个对象上同步，而第二个线程试图在同一个对象上同步，那么第二个线程将被迫等待，直到第一个线程完成该对象

同步方法

```
public class ThreadSafeCounter {
    private int count = 0; // The value of the counter.
    synchronized public void increment() {
        count = count + 1;
    }
    synchronized public int getValue() {
        return count;
    }
}
```

如果 tsc 是 ThreadSafeCounter 类的一个对象，那么任何一个调用 `tsc.invrement()` 的线程来给 counter + 1 的话都是绝对的线程安全的

同步语句

但是，这个 ThreadSafeCounter 并没有完全解决所有的资源冲突问题，考虑这种情况

```
if ( tsc.getValue() == 10 ) {
    doSomething();
}
```

`doSomething()` 需要 counter 等于 10，它可以在判定为 10 的时候进入，但如果这时别的线程修改了它，那么执行到 `doSomething()` 的时候就会出现问题

我们可以通过同步语句来解决这种问题

```
synchronized(tsc) {
    if ( tsc.getValue() == 10 )
        doSomething();
}
```

同步语句将某个对象（如这里的 tsc）作为类似一种输入参数

同步语句的格式为：

```
synchronized( <object> ) {
    <statements>
}
```

6. 易变变量 Volatile Variables

同步的**开销其实是很大**的，应该避免过度使用它，因此，在某些情况下不通过同步的方式访问某些共享变量是可以的

当共享变量的值在一个线程中设置而在另一个线程中使用时，会出现一个微妙的问题

由于Java中线程的实现方式，第二个线程可能不会立即看到变量的更改值，在共享变量的值被另一个线程更改后，线程可能会在一段时间内继续看到该变量的旧值。这是因为允许线程缓存共享数据

每个线程都可以**保留共享数据的本地副本**。当一个线程更改共享变量的值时，其他线程缓存中的本地副本不会立即更改，因此其他线程可以继续看到旧值，至少是**短暂地看到旧值**

可以在同步代码之外安全地使用共享变量，但在这种情况下，必须将该变量声明为 `volatile`，`volatile` 关键字是一个修饰符，可以添加到全局变量声明中

```
private volatile int count;
```

如果一个变量被声明为 volatile，那么没有线程会在其缓存中保留该变量的局部副本，这意味着对变量的任何更改将立即对所有线程可见，使得线程即使在同步代码之外也可以安全地引用volatile共享变量

但是，它并不能解决出现资源竞争的情况