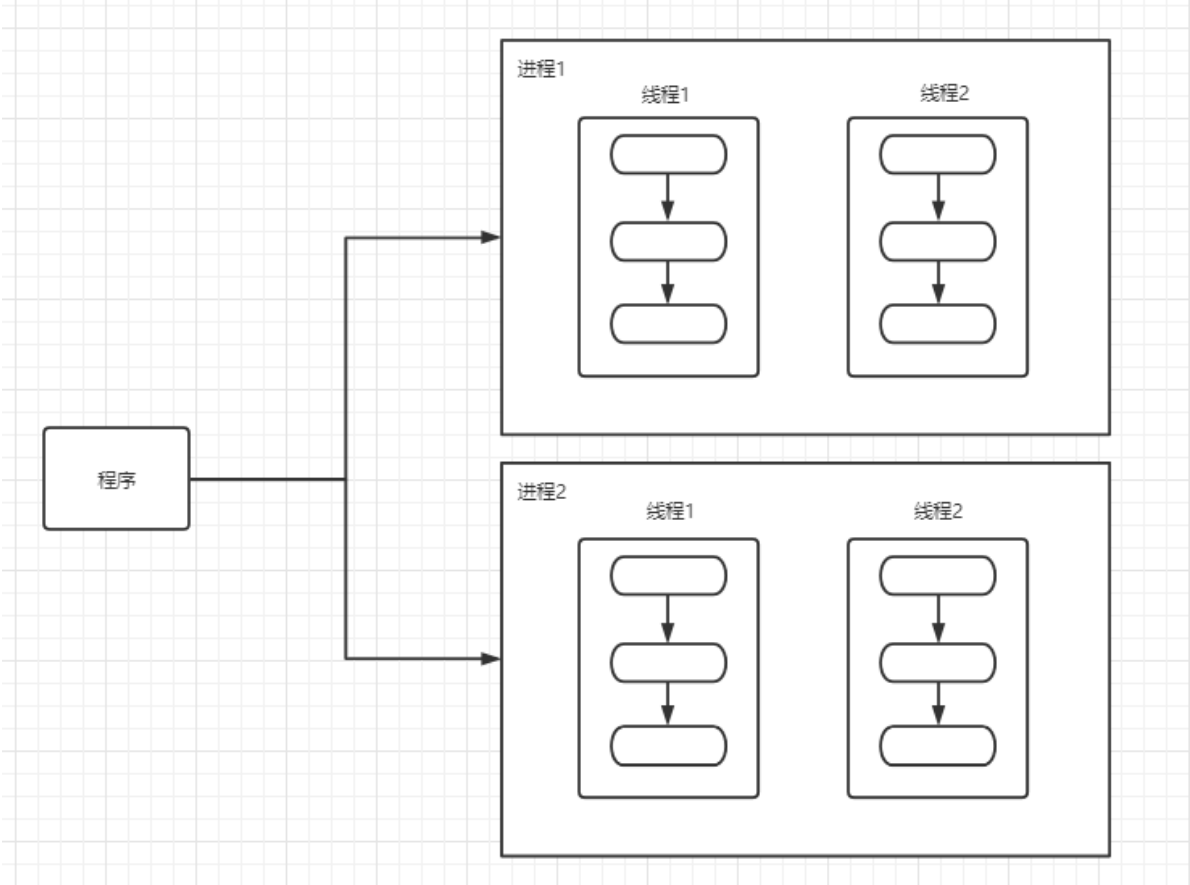
Chapter 30 多线程

30.1 线程的概念

30.1.1 程序、进程和线程

- 程序：一个静态的对象，内含指令和数据的文件，存储在磁盘或其他存储设备中
- 进程：一个动态的对象，是程序的一个执行过程，存在于系统内存中。一个进程对应于一个程序
- 线程：运行于某个进程中，用于完成某个具体任务的顺序控制流程，**程序中完成一个任务的有始有终的执行流，都有一个执行的起点，经过一系列指令后到达终点。**



不同进程的内存空间是隔离的，因此进程1中的变量i与进程2中的变量i属于不同的内存空间。进程切换和进程间通信开销大。进程间交换数据只能通过：共享内存、管道、消息队列、Socket通信等机制

一个进程里的线程切换开销小的多，因为它们位于同一内存空间里。线程1、2线程位于同一内存空间使得线程之间数据交换非常容易。变量i可以被线程1、2访问（但要考虑同步）。因此线程又叫轻量级进程

当一个进程被创建，自动地创建了一个主线程。因此，一个进程至少有一个主线程。

30.1.2 线程作用

- 一个进程的多个子线程可以**并发运行。**
- 多线程可以使程序 反应更快、交互性更强、执行效率更高。
- 应用：Server端程序，GUI程序
    - Server端的程序，都是需要启动多个线程来处理大量来自客户端的请求

- GUI程序：GUI线程：处理UI消息循环，如鼠标消息、键盘消息；Worker线程：后台的数据处理工作，比如打印文件，大数据量的运算

30.2 Runnable接口和Thread线程类

30.2.1 通过实现Runnable几口创建线程

具体步骤：

1. 实现Runnable接口，实现唯一的接口方法run
2. 创建实现Runnable接口的类的具体对象
3. 利用Thread类的构造函数创建线程对象
4. 通过线程对象的start方法启动线程

代码举例：

```
public class RunnableDemo implements Runnable{
    public RunnableDemo(){
        // constructor function
    }

    @Override
    public void run() {
        // do something
    }

    public static void main(String[] args) {
        // 创建一个实现了Runnable接口的实例
        Runnable task = new RunnableDemo();
        // 创建一个线程，注意new Thread()中的参数必须是实现了Runnable接口的实例
        Thread thread1 = new Thread(task);
        // 启动线程，会执行task的run方法
        thread1.start();
    }
}
```

注意：任何线程只能启动依次，多次调用产生`IllegalThreadStateException`异常

30.2.2 通过继承Thread类创建线程

具体步骤：

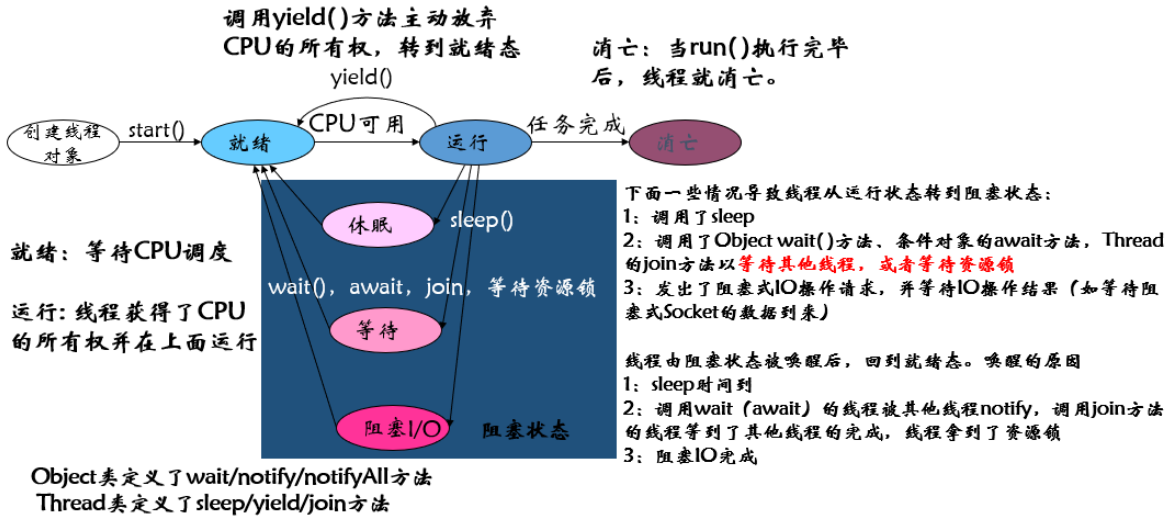
1. 定义Thread类的扩展类，在扩展类中重写run方法
2. 通过扩展类创建线程对象
3. 通过线程对象的start方法启动线程

代码举例：

```
public class ThreadDemo {
    public static void main(String[] args) {
        Thread thread1 = new PrintCharDemo2('a');
        Thread thread2 = new PrintCharDemo2('b');
        thread1.start();
        thread2.start();
```

```
    }

}

// 扩展类继承了Thread类
class PrintCharDemo2 extends Thread{
    private char printedChar;
    public PrintCharDemo2(char a){
        this.printedChar = a;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(this.printedChar);
        }
    }
}
```

30.2.3 线程的状态切换



30.2.3 Thread类常见的方法

- start();开始一个线程，进入Ready状态，如无其他线程等待，则立即Run进入running状态
- isAlive();获取线程当前是否在运行
- setPriority(p:int);为该线程指定优先值p:1~10
- join();等待线程结束
- sleep(millis:long);让当前线程休眠若干ms，监视器自动恢复其运行
- yield();将线程从running变为ready，允许其他线程执行（自己也可能立即执行）
- interrupt();中断该线程

yield用法：

```
public class YieldDemo extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println('a');
            Thread.yield(); // 挂起进入ready，给其他进程调度机会
        }
    }

}
```

sleep用法：

```
public class SleepDemo extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println('a');
            if (i>50){
                // 处于阻塞状态（如在睡眠，在wait，在执行阻塞式IO）的线程，
                // 如果被其他线程打断（即处于阻塞的线程的interupt方法被其它线程调用），
                // 会抛出InterruptedException，是一个必检异常
                try {
                    Thread.sleep(5);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

join用法：

```
public class JoinDemo extends Thread{
    public static void main(String[] args) {
        Thread threadA = new Thread(new PrintChar('a'));
        Thread threadB = new Thread(new PrintChar('b'));
        threadA.start();        // 启动线程A
        try {
            threadA.join();     // 主线程被阻塞，等待线程A执行完毕
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        threadB.start();        // 主线程被唤醒，启动线程B
    }
}
class PrintChar implements Runnable{
    private char printedChar;

    public PrintChar(char printedChar) {
        this.printedChar = printedChar;
```

```
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(printedChar);
        }
    }

}
```

join方法的作用：在A线程中调用了B线程（对象）的join()方法时，表示A线程放弃控制权（被阻塞了），只有当B线程执行完毕时，A线程才被唤醒继续执行。

程序在main线程中调用A线程（对象）的join方法时，main线程放弃cpu控制权（被阻塞），直到线程A执行完毕，main线程被唤醒执行threadB.start();

运行结果是全部a打印完才开始打印b

30.3 线程池

线程池适合大量线程任务的并发执行。线程池通过有效管理线程、“复用” 线程来提高性能。从JDK 1.5 开始使用Executor接口（执行器）来执行线程池中的任务，Executor的子接口ExecutorService管理和控制任务。我们只需要把实例化好的线程任务对象（Runnable接口实例）交给执行器Executor就可以了。Thread由线程池内部来创建和维护。

代码举例：

```
public class ThreadPoolDemo {
    public static void main(String[] args) {
        // 创建一个线程池，最大容量为3个
        ExecutorService es = Executors.newFixedThreadPool(3);
        // 提交runnable的任务给线程池
        es.execute(new PrintString("Thread A running"));
        es.execute(new PrintString("Thread B running"));
        es.execute(new PrintString("Thread C running"));
        // 关闭线程池
        es.shutdown();
    }
}

class PrintString implements Runnable{
    private String s;

    public PrintString(String s) {
        this.s = s;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(s);
        }
    }
}
```

区分任务和线程的区别：

任务：实现了Runnable接口的类的实例，任务的逻辑由run方法实现

线程：是Thread类的实例，是任务的运行载体，任务必须通过线程来运行

`Thread thread = new Thread(task)`中thread是线程，task是任务。

30.4 线程同步

如果多个线程对一个对象进行操作，如果操作的顺序不一致，会导致结果不一致，因此需要同步线程。

临界区：可能被多个线程同时进入的程序的一部分区域，可以是方法，也可以是语句块。因此我们需要对临界区进行同步，保证任何时候只有一个线程进入临界区。

30.4.1 synchronized关键字实现同步

synchronized同步方法

synchronized是通过加锁来实现方法同步的：一个线程要进入同步方法，首先拿到锁，进入方法后立刻上锁，导致其他要进入这个方法的线程被阻塞（等待锁）。

对于synchronized实例方法，是对**该方法的对象加锁**

对于synchronized静态方法，是对**该方法的类加锁**

当进入方法的线程执行完方法后，锁被释放，会唤醒等待这把锁的其他线程

基本语法：`public synchronized void deposit(double amount)`

synchronized同步语句块

基本语法：`synchronized (expr) {  statements;  }`

表示式expr结果必须是对一个对象的引用，因此可以通过对任何对象加锁来同步语句块。

- 如果expr指向的对象没有被加锁，则第一个执行到同步块的线程对该对象加锁，线程执行该语句块，然后解锁；
- 如果expr指向的对象已经加了锁，则执行到同步块的其它线程将被阻塞
- expr指向的对象解锁后，所有等待该对象锁的线程都被唤醒

同步语句块允许同步方法中的部分代码，而不必是整个方法，增强了程序的并发能力

任何同步的实例方法都可以转换为同步语句块

```
public synchronized void xMethod() {
  // method body
}
```

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

30.4.2 加锁来实现同步

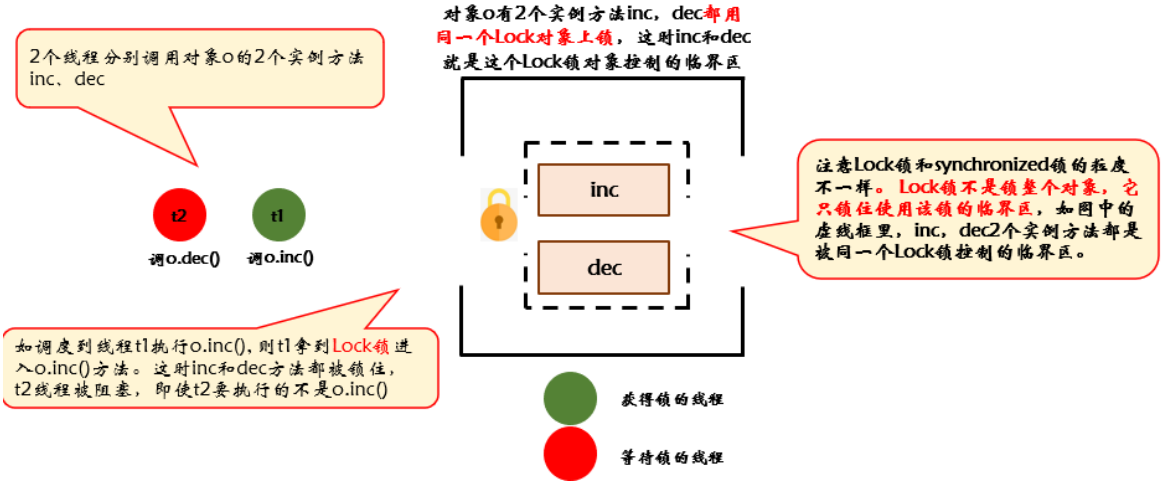
采用synchronized关键字的同步要隐式地在对象实例或类上加锁，粒度较大影响性能，JDK 1.5 可以显式地加锁，能够在更小的粒度上进行线程同步，一个锁是一个Lock接口的实例，类ReentrantLock是Lock的一个具体实现：可重入的锁

伪代码：

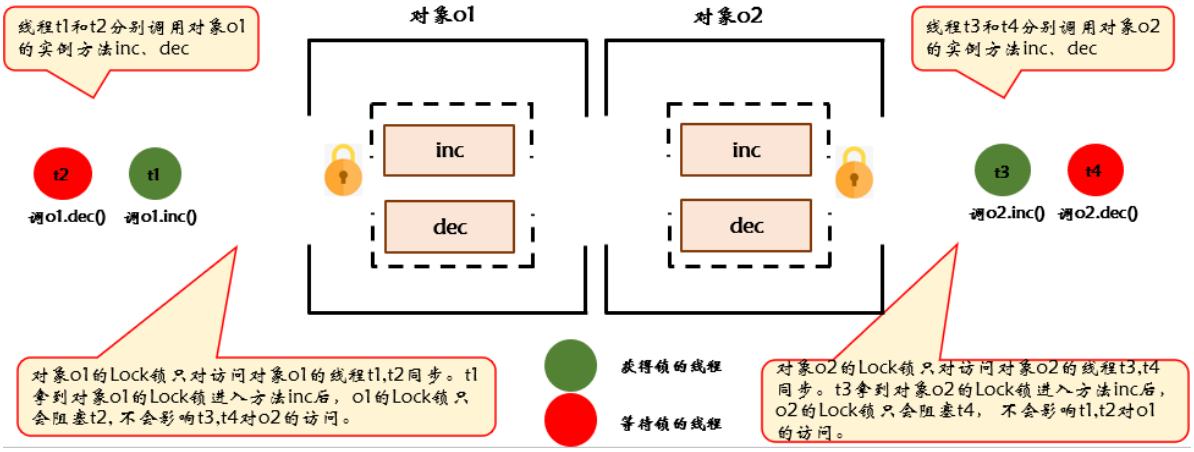
```
void methodA(){
    lock.lock(); // 获取锁
    methodB();
    lock.unlock() // 释放锁
}

void methodB(){
    lock.lock(); // 再次获取该锁
     // 其他业务
    lock.unlock();// 释放锁
}
```

30.4.3 锁的总结

- 情景1：假设一个类有多个用**synchronized**修饰的同步实例方法，如果多个线程访问这个类的**同一个对象**，当一个线程获得了该对象锁进入到其中一个同步方法时，**这把锁会锁住这个对象所有的同步实例方法**
- 情景2：假设一个类有多个用**synchronized**修饰的同步实例方法，如果多个线程访问这个类的**不同对象**，那么**不同对象的**synchronized**锁不一样**，**每个对象的锁只能对访问该对象的线程同步**
- 情景3：如果采用Lock锁进行同步，**一旦Lock锁被一个线程获得，那么被这把锁控制的所有临界区都被上锁，**这时所有其他访问这些临界区的线程都被阻塞。



- 情景4：如果一个类采用Lock锁对临界区上锁，而且这个**Lock**锁也是该类的实例成员，**那么这个类的二个实例的**Lock**锁就是不同的锁**，下面的动画演示了这种场景：对象o1的Lock锁和对象o2的Lock锁是不同的锁对象。



如果采用synchronized关键字对类 A的实例方法进行同步控制，这时等价于synchronized(this){ }

一旦一个线程进入类A的对象o的synchronized实例方法，对象o被加锁，对象o所有的synchronized实例方法都被锁住，从而阻塞了要访问对象o的synchronized实例方法的线程，但是与访问A类其它对象的线程无关

如果采用synchronized关键字对类 A的静态方法进行同步控制，这时等价于synchronized(A.class){ }。一旦一个线程进入A的一个静态同步方法，A所有的静态同步方法都被锁（这个锁是类级别的锁），这个锁对所有访问该类静态同步方法的线程有效，不管这些线程是通过类名访问静态同步方法还是通过不同的对象访问静态同步方法。

如果通过Lock对象进行同步，首先看Lock对象对哪些临界区上锁，一旦Lock锁被一个线程获得，那么被这把锁控制的所有临界区都被上锁（如场景3）；另外要区分Lock对象本身是否是不同的：不同的Lock对象能阻塞的线程是不一样的（如场景4）。

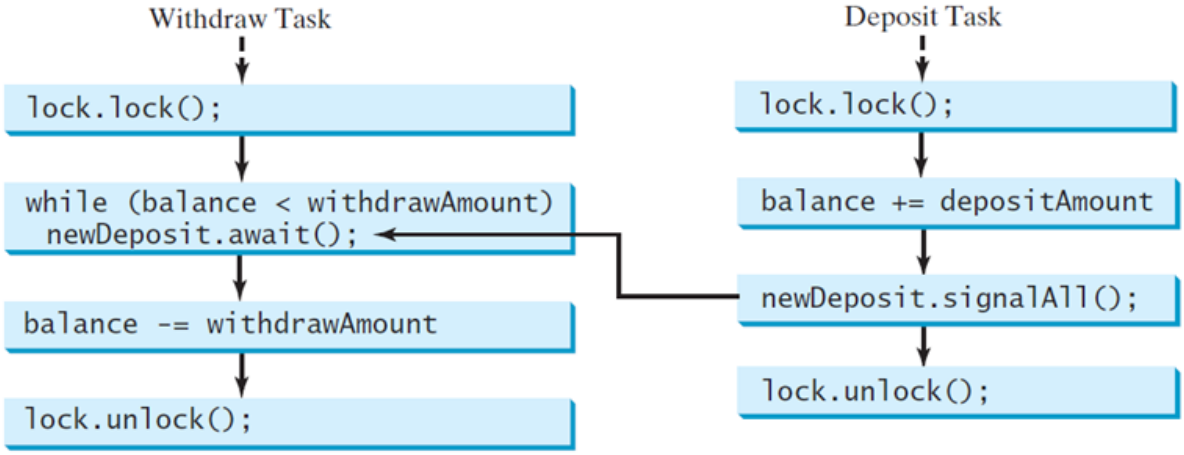
30.4.4 线程合作

线程之间有资源竞争，synchronized和Lock锁这些同步机制解决的是资源竞争问题

线程之间还有相互协作的问题

假设创建并启动两个任务线程：

- 存款线程用来向账户中存款
- 提款线程从同一账户中提款
- 当提款的数额大于账户的当前余额时，提款线程必须等待存款线程往账户里存钱
- 如果存款线程存入一笔资金，必须通知提款线程重新尝试提款，如果余额仍未达到提款的数额，提款线程必须继续等待新的存款



线程之间的相互协作：可通过Condition对象的await/signal/signalAll来完成

- Condition (条件)对象是通过调用Lock实例的newCondition( )方法而创建的对象
- Condition对象可以用于协调线程之间的交互（使用条件实现线程间通信）
- 一旦创建了条件对象condition，就可以通过调用condition.await()使当前线程进入等待状态，
- 其它线程通过同一个条件对象调用signal和signalAll()方法来唤醒等待的线程，从而实现线程之间的相互协作

线程合作举例：

```
public class ConditionDemo {
    private static Lock lock = new ReentrantLock();     // 创建lock对象
    private static Condition condition = lock.newCondition();       // 创建lock的
Condition对象
    private int balance = 0;

    public int getBalance() {
        return balance;
    }

    public void withdraw(int amount) {
        lock.lock();        // 获取锁

        try {
            // while循环{ //必须是while， 不能用if
            // 只要余额小于取钱数额，就调用condition.wait，使得当前线程（进入withDraw方法
的线程）被挂起；
            // 如果当前线程被唤醒，如果余额还小于取钱数额，继续等待
            // }
            // 当执行到while循环的下一条语句，余额一定>=取钱数额
            // condition.wait会导致当前线程被挂起同时锁被释放（和sleep不一样），否则存钱线
程永远没机会进入deposit方法
            while (balance < amount) {
                System.out.println("\t\t\t\tWaiting for deposit");
                condition.await();
            }
            balance -= amount;
            System.out.println("\t\t\t\tWithdraw " + amount + "\t\t\t\t" +
getBalance());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void deposit(int amount) {
        lock.lock();

        try {
            balance += amount;
            System.out.println("deposit" + amount + "\t\t\t\t\t\t\t\t" +
getBalance());
            // 进入deposit的是另外一个线程，往账户存钱后，
            // 调用newDeposit.signalAll去唤醒所有因调用condition.wait而被挂起的线程（二
者配套使用）
```

```
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

```
public class ConditionDemoTest {
    private static ConditionDemo conditionDemo = new ConditionDemo();

    public static class DepositTask implements Runnable {
        @Override
        public void run() {
            try {
                while (true) {
                    conditionDemo.deposit((int) (Math.random() * 10) + 1);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static class WithdrawTask implements Runnable{
        @Override
        public void run() {
            try {
                while (true) {
                    conditionDemo.withdraw((int) (Math.random() * 10) + 1);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        ConditionDemo conditionDemo = new ConditionDemo();
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(new DepositTask());
        executor.execute(new WithdrawTask());
        System.out.println("Thread1\t\t\tThread2\t\t\t\tBalance");
    }
}
```

```
Thread1         Thread2             Balance
deposit5                            5
                Withdraw 2          3
                Withdraw 2          1
deposit7                            8
                Waiting for deposit
deposit3                            11
                Withdraw 10         1
                Waiting for deposit
deposit4                            5
                Waiting for deposit
deposit6                            11
```

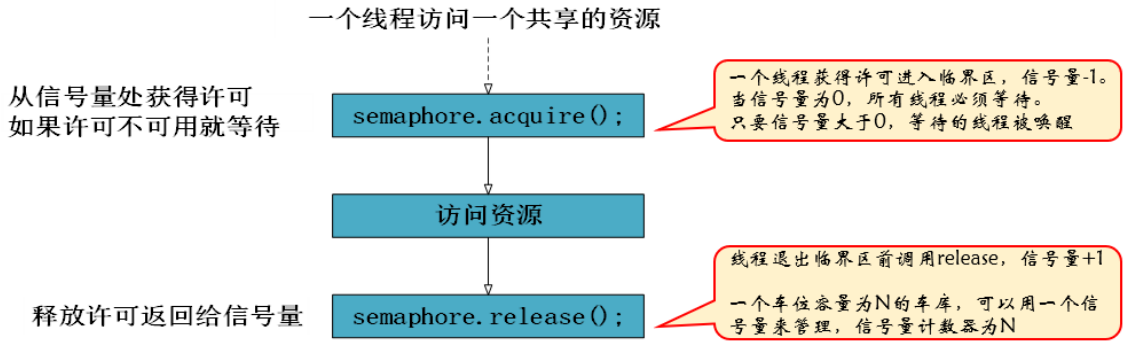
30.5 信号量

信号量用来限制访问一个共享资源的线程数，是一个有计数器的锁，访问资源之前，线程必须从信号量获取许可，访问完资源之后，该线程必须将许可返回给信号量

为了创建信号量，必须确定许可的数量（计数器最大值），同时可选用公平策略

任务通过调用信号量的acquire()方法来获得许可，信号量中可用许可的总数减1

任务通过调用信号量的release()方法来释放许可，信号量中可用许可的总数加1

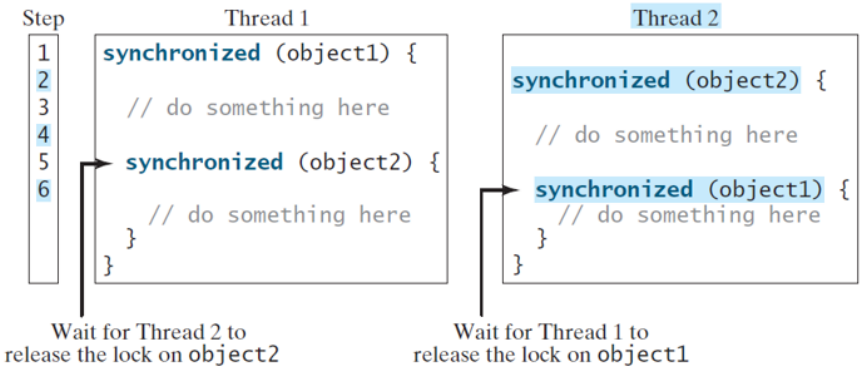



```
import java.util.concurrent.Semaphore;
  // An inner class for account
  private static class Account {
    // Create a semaphore
    private static Semaphore semaphore = new Semaphore(1);
    private int balance = 0;
    public int getBalance() {return balance;}

    public void deposit(int amount) {
      try {
        semaphore.acquire();
        int newBaance=balance+amount
         Thread.sleep(5);
        balance=new Balance;
      }
      finally {
        semaphore.release();
      }
```

```
    }
  }
}
```

30.6 避免死锁

- 死锁：如图所示，两个线程形成死锁



- 避免死锁：可以采用正确的资源排序来避免死锁
  - 给每一个需要上锁的对象指定一个顺序
  - 确保每个线程都按这个顺序来获取锁

线程2必须先获取object1上的锁，然后才能获取Object2上的锁

30.7 同步合集

■Java集合框架包括：List、Set、Map接口及其具体子类，都不是线程安全的。

■集合框架中的类不是线程安全的，可通过为访问集合的代码临界区加锁或者同步等方式来保护集合中的数据

■Collections类提供6个静态方法来将集合转成同步版本（即线程安全的版本）

■这些同步版本的类都是线程安全的，但是迭代器不是，因此使用迭代器时必须同步：

synchronized(要迭代的集合对象){// 迭代}

java.util.Collections	
+synchronizedCollection(c:Collection):Collection	从一个给定的合集返回一个同步集合
+synchronizedList(list:List):List	从一个给定的线性表返回一个同步线性表
+synchronizedMap(m:Map):Map	从一个给定的映射表返回一个同步映射表
+synchronizedSet(s:Set):Set	从一个给定的集合返回一个同步集合
+synchronizedSortedMap(s:SortedMap):SortedMap	从一个给定的排序映射表返回一个同步排序映射表
+synchronizedSortedSet(s:SortedSet):SortedSet	从一个给定的排序集合返回一个同步排序集合