Java 多线程

天津卓讯科技

第7章 多线程

- 7.1 线程概述
- 7.2 创建和启动线程
- 7.3 多线程的同步
- 7.4 使用Java Timer调度任务

7.1 进程和线程的概念

- 进程(Process)：每个独立执行的程序称为进程。
  - 多进程: 在操作系统中能同时运行多个任务(程序)
- 线程(Thread)：程序内部的一条执行路径。
  - 多线程: 在同一应用程序中多条执行路径同时执行
- 线程和进程的区别
  - 每个进程都有独立的代码和数据空间(进程上下文)，进程间的切换开销大
  - 同一进程内的多个线程共享相同的代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换的开销小

7.2 线程的创建和启动

- 两种创建新线程的方式
  - 第一种：将类声明为java.lang.Thread的子类并重写run()方法
    - class MyThread extends Thread{ public void run(){...} }
  - 创建此线程类的实例并启动：
    - MyThread thread1 = new MyThread();
    - thread1.start();
  - 第二种：定义实现java.lang.Runnable接口的类
  - Runnable接口中只有一个方法public void run();用来定义线程运行体：
    - class MyRun implements Runnable{ public void run(){...} }
  - 创建Thread实例时将这个类的实例作为参数传递到线程实例内部。然后再启动：
    - Thread thread1 = new Thread(new MyRun());
    - thread1.start();

- 使用实现Runnable接口的方式创建线程时可以为相同程序代码的多个线程提供共享的数据。

7.2.2 线程的基本使用方法

- public void start() //启动该线程
- public static Thread currentThread() //返回对当前线程对象的引用
- public ClassLoader getContextClassLoader()
  - 返回该线程的上下文ClassLoader
- public final boolean isAlive() //线程是否处于活动状态
- public Thread.State getState() //返回该线程的状态
- public final String getName() //返回该线程的名称
- public final void setName(String name) //改变线程名称
- public final void setDaemon(boolean on)
  - 将该线程标记为守护线程或用户线程

示例

```
public class Thread1Test {
    public static void main(String[] args) {
        Runner runner = new Runner();
        Thread thread1 = new Thread(runner);
        thread1.start();
        for(int i = 0; i < 100; i++){
            System.out.println("-------Main Thread:" + i);
        }
    }
}
class Runner implements Runnable{
    public void run() {
        for(int i = 0; i < 100; i++){
            System.out.println("runner:" + i);
        }
    }
}
```

示例2

```
public class Thread2Test {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread();
        thread1.start();
        for(int i = 0; i < 100; i++){
            System.out.println("-------Main Thread:" + i);
        }
    }
}

class MyThread extends Thread{
    public void run() {
        for(int i = 0; i < 100; i++){
            System.out.println("MyThread:" + i);
        }
    }
}
```

思考与练习

- 用多线程程序模拟铁路售票系统：
  - 实现通过5个售票点发售某日某次列车的1000张车票，一个售票点用一个线程表示。

思考与练习

```
class TicketOffice implements Runnable{
    private int tickets = 1000;
    public void run() {
        while(tickets > 0){
            tickets--;
            System.out.println(Thread.currentThread().getName() + ":卖出第"+ tickets+ "张票");
        }
    }
}
```

```
public class TicketOfficeTest {
    public static void main(String[] args) {
        TicketOffice off = new TicketOffice();
        Thread thread1 = new Thread(off);
        thread1.setName("售票点1");
        thread1.start();   …
    }
}
```

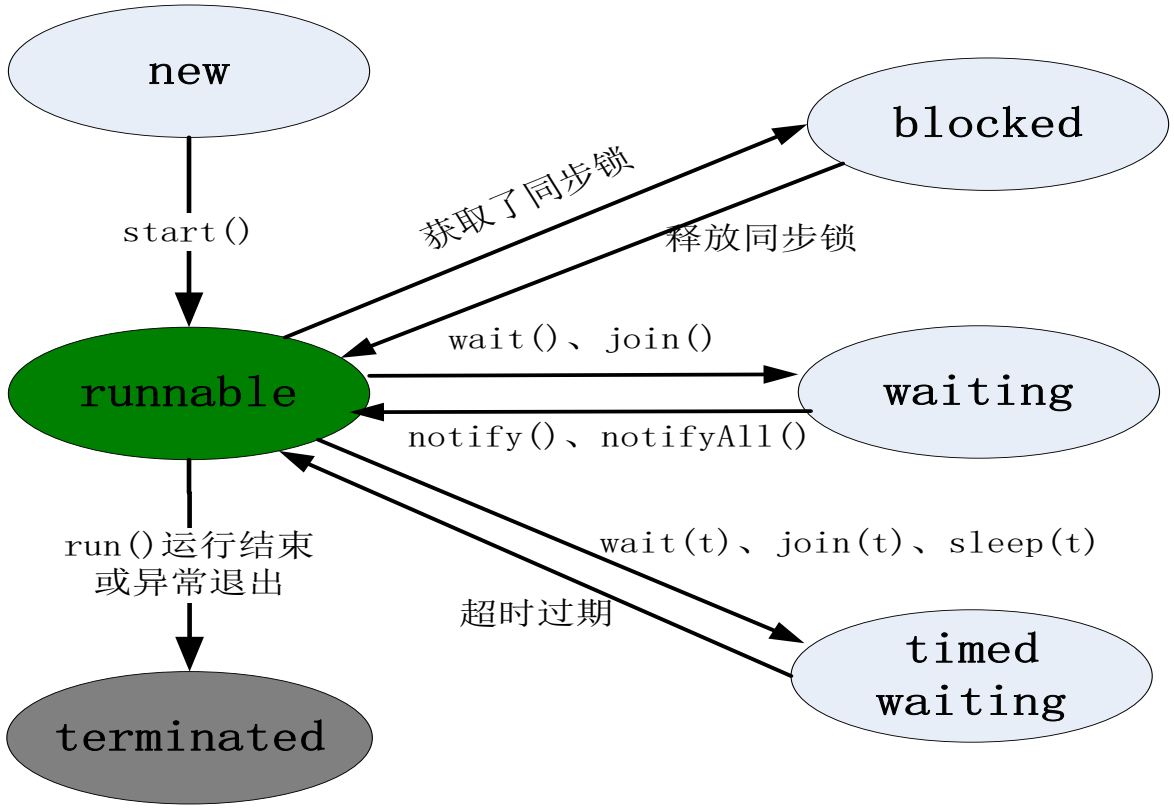
本示例还有线程不同步问题

7.2.3 线程分类

- 用户线程：默认创建的都是用户线程
  - 用于完成指定任务的一个线程。
- 守护线程：在没有其它用户线程在运行时会自动退出
  - 用于服务用户线程的。
  - 通过Thread类的setDaemon(true)来完成。
  - Java垃圾回收线程就是一个典型的守护线程

7.3 线程的状态及转换

- Thread.State枚举表示出了这6种状态
  - NEW / RUNNABLE / TERMINATED
  - BLOCKED / WAITING / TIMED_WAITING



7.4.1 线程的调度原理

- 线程调度器
  - 抢占式模型：根据多个线程的优先级、饥饿时间(已经多长时间没有被执行了)，计算出每个线程的总优先级，线程调度器就会让总优先级最高的这个线程来执行。
  - 分时间片模型：所有线程排成一个队列。线程调度器按照它们的顺序，给每个线程分配一段时间(毫秒级)。
    - 如果在时间片结束时线程还没有执行完，则它的执行权会被调度器剥夺并分配给另一个线程；
    - 如果线程在时间片结束前阻塞或结束，则调度器立即进行切换。

7.4.2 线程的优先级

- 为了表示不同线程对操作系统和用户的重要性，Java定义了10个等级的优先级。
  - 用1-10之间的数字表示，数字越大表明线程的级别越高
  - 三个静态成员变量：MIN_PRIORITY、MAX_PRIORITY和NORMAL_PRIORITY（默认值）
- 优先级的操作
  - void setPriority(int newPriority); //要在start()前调用
- 注意：不要依赖优先级来控制线程的执行顺序。

7.5 线程的其它控制

- 线程睡眠：
  - Thread.sleep(long millis) throws InterruptedException 方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪(Runnable)状态。sleep()平台移植性好。
- 线程让步：
  - Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给别的线程。
- 线程加入：
  - void join() throws InterruptedException方法在当前线程中调用另一个线程的join()方法，则当前线程转入WAITING状态，直到另一个线程运行结束，当前线程再由阻塞转为就绪状态。

示例

```
public class ThreadYieldTest {
    public static void main(String[] args) {
        System.out.println("主线程:" + Thread.currentThread().getName());
        Thread thread1 = new Thread(new MyThread2());
        thread1.start();
        Thread thread2 = new Thread(new MyThread2());
        thread2.start();
    }
}
class MyThread2 implements Runnable{
    public void run() {
        for(int i = 0; i < 100; i++){
            System.out.println(Thread.currentThread().getName()+ ":" + i);
            if(i % 10 == 0){        Thread.yield(); //线程让步            }
        }
    }
}
```

示例2

```
public class ThreadJoinTest {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyThread3());
        thread1.start();
        for (int i = 1; i <= 100; i++) {
            System.out.println(Thread.currentThread().getName() + ":" + i);
            if (i == 50) {
                try {    thread1.join();    //线程合并    } catch (InterruptedException e) {    e.printStackTrace();    }
            }
        }
    }
}
class MyThread3 implements Runnable{
    public void run() {
        for (int i = 1; i <= 50; i++) {
            System.out.println(Thread.currentThread().getName() + ":" + i);
            try {    Thread.sleep(100);  } catch (InterruptedException e) {     e.printStackTrace();    }
        }
    }
}
```

7.5.2 线程的停止

- 如果线程的run()方法中执行的是一个重复执行的循环，可以提供一个循环结束标记来控制。
  - 示例：ThreadEndTest.java
- 如果线程因为执行sleep()或是wait()而进入了阻塞状态，此时要想停止它，可能使用interrupt()，程序会抛出InterruptException异常而离开run()方法。
  - 示例：ThreadEndTest2.java
- 如果程序因为输入/输出的等待而阻塞，基本上必须等待输入/输出的动作完成才能离开阻塞状态。无法用interrupt()方法来使得线程离开run()方法，要想离开，只能通过引发一个异常。

示例

```
public class ThreadEndTest {
    public static void main(String[] args) {
        SomeThread some = new SomeThread();
        Thread thread1 = new Thread(some);
        thread1.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {        e.printStackTrace();    }
        some.terimate();
    }
}
class SomeThread implements Runnable {
    private boolean flag = true;
    public void terimate() {        this.flag = false;    }
    public void run() {
        while (flag) {
            System.out.println("...run...");
        }
    }
}
```

示例2

```
public class ThreadEnd2Test {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new SomeThread2());
        thread1.start();
        try {       Thread.sleep(2000); } catch (InterruptedException e) { e.printStackTrace(); }
        thread1.interrupt();
    }
}
class SomeThread2 implements Runnable{
    public void run(){
        System.out.println("sleep.....");
        try {
            Thread.sleep(9999);
        } catch (InterruptedException e) {
            System.out.println(" interrupted...");
            e.printStackTrace();
        }
    }
}
```

7.6 多线程同步

- 7.6.1 原理
  - 多个线程需要共享对同一个数据的访问。如果每个线程都会调用一个修改共享数据状态的方法，那么，这些线程将会互相影响对方的运行。
- 在Java语言中，引入对象互斥锁的概念，保证共享数据操作的完整性。
  - 每个对象都对应一个可称为“互斥锁”的标记，这个标记保证在任一时刻，只能有一个线程访问对象。

7.6.2多线程同步的实现

- synchronized同步代码块
  - 当某个对象用synchronized修饰时，表明该对象在任一时刻只能由一个线程访问。

```
synchronized(this){
    需要同步的代码;
}
```

- synchronized同步方法

```
public synchronized void sale(){
    …
}
```

- 显式加锁：JDK5.0。锁的粒度越细越好。

```
private static final Lock lock = new ReentrantLock();  //创建Lock实例
lock.lock();  //获取锁
…
lock.unlock();  //释放锁
```

synchronized关键字

```
class TicketOffice implements Runnable {
    private int tickets = 0;
    private boolean flag =true;
    public void run() {
        while (flag) {
            try {
               Thread.sleep(10);
            } catch (InterruptedException e) {    e.printStackTrace();    }
            if (tickets < 100) {
                tickets++;
            System.out.println(Thread.currentThread().getName()+":卖出第“+tickets+"张票");
            } else {
                flag = false;
            }
        }
    }
}
```

```
public class TestTicketOffice {
    public static void main(String[] args) {
        TicketOffice off = new TicketOffice();
        Thread thread1 = new Thread(off);
        thread1.setName("售票点1");
        thread1.start();   …
    }
}
```

修正

```
class TicketOffice implements Runnable {
    private int tickets = 0;
    private boolean flag = true;
    public void run() {
        while (flag) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {    e.printStackTrace();    }
          synchronized(this){
            if (tickets < 100) {
                tickets++;
            System.out.println(Thread.currentThread().getName()+":卖出第"+tickets+"张票");
            } else {
                flag = false;
            }
          }
        }
    }
}
```

```
public synchronized void sale(){
    if (tickets < 100) {
        tickets++;
        System.out.println(Thread.currentThread().getName()
                + ":卖出第" + tickets + "张票");
    } else {
        flag = false;
    }
}
```

7.6.3 多线程安全问题

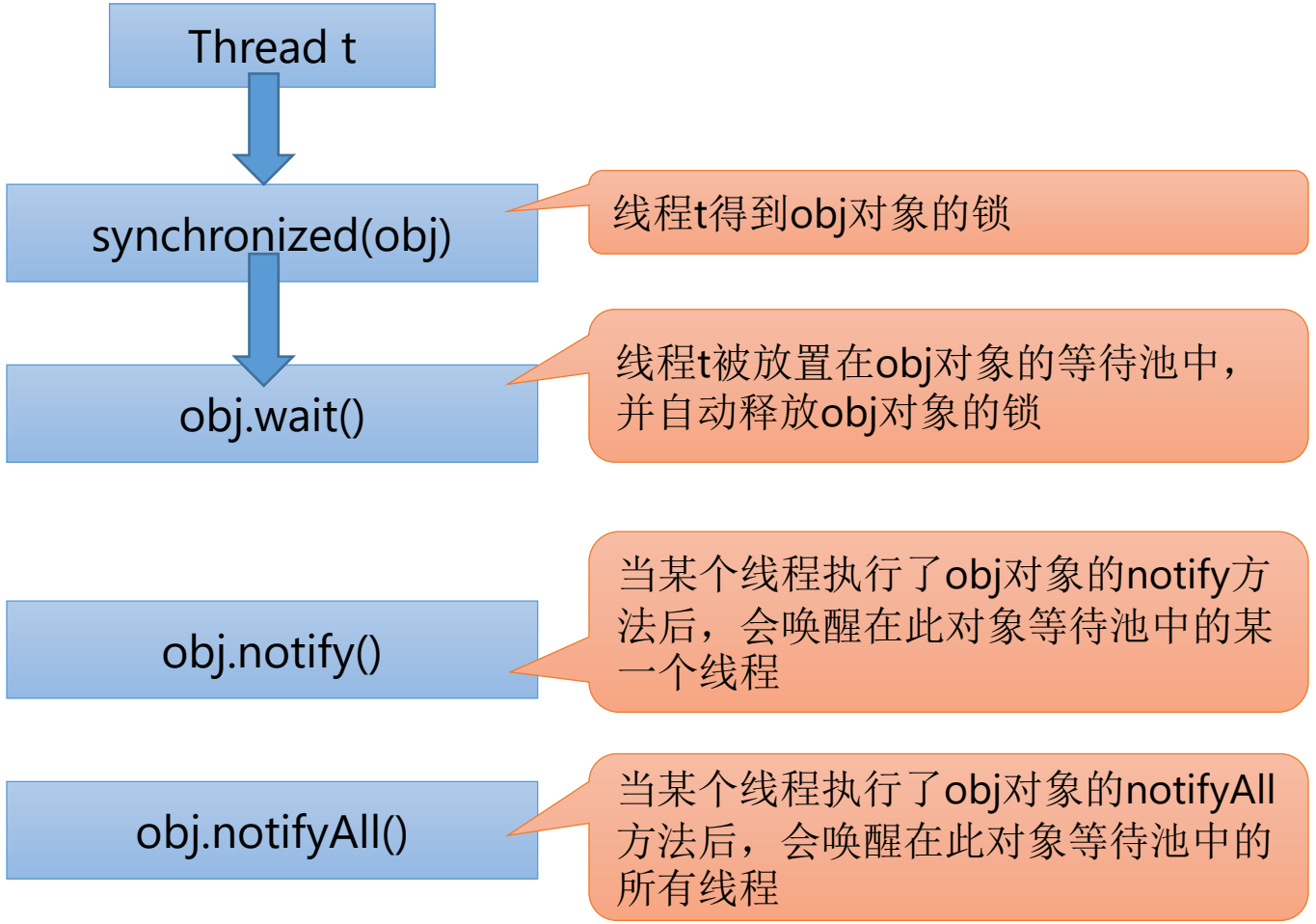
- 当run()方法体内的代码操作到了成员变量(共享数据)时，就可能会出现多线程安全问题(线程不同步问题)。

- 编程技巧
  - 在方法中尽量少操作成员变量，多使用局部变量

7.7 线程交互

- 线程等待：
  - Object类中的wait() throws InterruptedException 方法，导致当前的线程等待，直到其他线程调用此对象的notify()方法或notifyAll()唤醒方法。
- 线程唤醒：
  - Object 类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的。
  - Object类中的notifyAll()方法，唤醒在此对象监视器上等待的所有线程。
- 这三个方法只能在被同步化(synchronized)的方法或代码块中调用

理解线程的等待和唤醒


Thread t
synchronized(obj)
线程t得到obj对象的锁
obj.wait()
线程t被放置在obj对象的等待池中，并自动释放obj对象的锁
obj.notify()
当某个线程执行了obj对象的notify方法后，会唤醒在此对象等待池中的某一个线程
obj.notifyAll()
当某个线程执行了obj对象的notifyAll方法后，会唤醒在此对象等待池中的所有线程

示例

- 生产者(Producer)与消费者(Consumer)的问题
  - 生产者将产品交给店员，而消费者从店员处取走产品，店员一次只能持有固定数量的产品，如果生产者生产了过多的产品，店员叫生产者等一下，如果店中有空位放产品了再通知生产者继续生产；如果店中没有产品了，店员会告诉消费者等一下，如果店中有产品了再通知消费者来取走产品。

7.8 使用Java Timer调度任务

- java.util.Timer类提供了基本的调度功能。这个类允许你调度一个任务（通过java.util.TimerTask子类定义）按任意周期运行。
  - java.util.Timer类：代表一个计时器
    - public void schedule(TimerTask task, long delay, long period)
      - 重复的以固定的延迟时间去执行一个任务
    - public void scheduleAtFixedRate(TimerTask task, long delay, long period)
      - 重复的以固定的频率去执行一个任务
    - public void cancel()
      - 终止此计时器，丢弃所有当前已安排的任务
  - java.util.TimerTask抽象类：表示一个计时器任务
    - public abstract void run()
      - 此计时器任务要执行的操作
    - public boolean cancel()
      - 取消此计时器任务

示例

```
public class TimerTest {
    public static void main(String[] args) {
        Timer timer = new Timer();
        //timer.schedule(new MyTask(), 0, 1000);
        timer.scheduleAtFixedRate(new MyTask(), 0, 1000);
        while(true){
            try {
                int ch = System.in.read();
                if(ch=='c'){
                    timer.cancel();    //退出任务
                }
            } catch (IOException e)  {              e.printStackTrace();     }
        }
    }
}
class MyTask extends java.util.TimerTask {
   public void run() {
        System.out.println("^_^");
   }
}
```

总结

- 线程 VS 进程
- 创建和启动线程的方式
- 线程的状态及转换
- 线程同步
- 线程交互：wait 、 notify / notifyAll
- 使用Java Timer调度任务