**JAVA基础**

# 线程同步

同步方法

|  |
| --- |
| **public** **class** SyncMethod {  **public** **static** **void** main(String[] args) {  }  // 同步方法，整个方法体都是同步代码块  // 同步监视器 this-调用该方法的当前实例  **public** **synchronized** **void** sync1() {  **try** {  wait();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  // sync1等价于sync3  **public** **void** sync3() {  **synchronized** (**this**) {  }  }    // 静态同步方法 整个方法体都是同步代码块  // 同步监视器 SyncMethod.class，当前类二进制文件  // 在类被加载时，SyncMethod.class被加载到内存，  // 只会被加载一次，在内存中只存在一份  **public** **static** **synchronized** **void** sync2() {    }  // sync2等价于sync4  **public** **static** **void** sync4() {  **synchronized**(SyncMethod.**class**) {  }    }  } |

## 同步问题解决

* 线程安全类具有特征--account

1.该类的对象可以被多个线程安全地访问

2.每个线程调用该对象的任意方法之后都将得到正确的结果

3.每个线程调用该对象的任意方法后，该对象状态依然保持合理状态

|  |
| --- |
| **public** **class** Account {  **private** String accountId;  **private** **double** balance;        **public** Account() {  **super**();  // **TODO** Auto-generated constructor stub  }  **public** Account(String accountId, **double** balance) {  **super**();  **this**.accountId = accountId;  **this**.balance = balance;  }  **public** String getAccountId() {  **return** accountId;  }  **public** **void** setAccountId(String accountId) {  **this**.accountId = accountId;  }  **public** **double** getBalance() {  **return** balance;  }  **public** **void** setBalance(**double** balance) {  **this**.balance = balance;  }  @Override  **public** String toString() {  **return** "Account [accountId=" + accountId + ", balance=" + balance + "]";  }  } |
| **public** **class** SyncProm {  // synchronized解决线程并发资源共享信息不一致的问题  **public** **static** **void** main(String[] args) {  Account account = **new** Account("123456", 20000);  DrawRunnable dr = **new** DrawRunnable(account);  Thread t1 = **new** Thread(dr);  Thread t2 = **new** Thread(dr);  // 线程并发，所带来的数据信息不一致的问题  t1.start();  t2.start();  }  }  **class** DrawRunnable **implements** Runnable {  **private** Account account;  **public** DrawRunnable(Account account) {  **this**.account = account;  }  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **double** bala = account.getBalance() - 100;  account.setBalance(bala);  System.***out***.println(bala);  }  }  } |

## 存钱者与取钱者交替执行

线程不能显式释放同步监视器，几种释放同步监视器情况

1.同步代码块执行结束

2.同步方法中遇到break return终止了代码块

3.同步代码块中出现Error或Exception

4.执行同步代码块时，程序执行了同步监视器的wait()方法，当前线程暂停

下面的情况不会释放同步监视器的锁定

1.执行同步方法或同步代码块时，程序调用sleep() yield()来暂停当前线程

wait()、notify()和notifyAll()三个方法，这三个方法不属于Thread类，而是属于Object类。但这三个方法必须由同步监视器对象来调用

* 3个方法的解释

1.wait():导致当前线程等待，直到其也线程调用该同步监视器的notify()或notifyAll()来唤醒。

2.notify():唤醒在此同步监视器上等待的单个线程。如果有多个线程在此监视器上等待，则会选择其中一个进行唤醒，选择哪个线程是任意的。

3.notifyAll():唤醒在此同步监视器上等待的所有线程

|  |
| --- |
| **public** **class** Account {  **private** String accountId;  **private** **double** balance;        **public** Account() {  **super**();  // **TODO** Auto-generated constructor stub  }  **public** Account(String accountId, **double** balance) {  **super**();  **this**.accountId = accountId;  **this**.balance = balance;  }  **public** String getAccountId() {  **return** accountId;  }  **public** **void** setAccountId(String accountId) {  **this**.accountId = accountId;  }  **public** **double** getBalance() {  **return** balance;  }  **public** **void** setBalance(**double** balance) {  **this**.balance = balance;  }  @Override  **public** String toString() {  **return** "Account [accountId=" + accountId + ", balance=" + balance + "]";  }    **boolean** hasMoney = **false**;    /\*\*  \* 取钱  \*  \* 20次存钱 20次取钱 交替执行  \* 账户初始没有钱，先调用取钱，取钱线程会执行wait，等待状态，进入阻塞，释放资源  \* 存钱线程执行，hasMoney为false，不会执行wait，像账户中存钱，  \* 存完钱后调用notify方法，唤醒等待的线程，即唤醒取钱线程，释放资源(同步监视器)，  \* 在开始竞争资源，如果存钱者竞争到，便会执行wait方法，进入等待，阻塞执行，释放资源  \* 取钱者线程开始取钱，取完后释放资源（同步监视器），唤醒等待的线程，唤醒存钱者，开始竞争资源  \* .....  \*/  **public** **synchronized** **void** draw() {    **if**(!hasMoney) {  **try** {  wait();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  **this**.balance = balance - 100;  System.***out***.println(**this**.balance);  hasMoney = **false**;  // 唤醒在这个同步监视器上等待的某个线程，唤醒存钱线程  notify();    }  /\*\*  \* 存钱  \*/  **public** **synchronized** **void** save() {  **if** (hasMoney) {  **try** {  wait();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  **this**.balance = balance + 100;  System.***out***.println(**this**.balance);  hasMoney = **true**;  // 唤醒在这个同步监视器上等待的某个线程， 唤醒取钱线程  notify();  // 唤醒在这个同步监视器上等待的所有线程  // notifyAll();    }  } |
| **public** **class** MainAccount {  **public** **static** **void** main(String[] args) {  // 新建一个账号，使两个线程都在这个账号上进行操作  Account account = **new** Account("12345678",0.0);  // 存钱者，将账号作为参数传入  SaveRunnable sr = **new** SaveRunnable(account);  // 取钱者，将账号作为参数传入  DrawerRunnable dr = **new** DrawerRunnable(account);  // 存钱者线程启动  Thread st = **new** Thread(sr);  st.start();  // 取钱者线程启动  Thread dt = **new** Thread(dr);  dt.start();  }  }  /\*\*  \* 存钱者  \* **@author** tzhang  \*  \*/  **class** SaveRunnable **implements** Runnable {    **private** Account account;  **public** SaveRunnable(Account account) {  **this**.account = account;  }    @Override  **public** **void** run() {  **for** (**int** i = 0; i < 20; i ++) {  account.save();  }    }    }  /\*\*  \* 取钱者  \* **@author** tzhang  \*  \*/  **class** DrawerRunnable **implements** Runnable {  **private** Account account;  **public** DrawerRunnable(Account account) {  **this**.account = account;  }    @Override  **public** **void** run() {  **for** (**int** i = 0; i < 20; i ++) {  account.draw();  }  }    } |

## 死锁

|  |
| --- |
| **public** **class** DeadLock {  // A 与 B互相鞠躬，当同事发生时  **public** **static** **void** main(String[] args) {  // 创建a、b好友  Friend a = **new** Friend("a");  Friend b = **new** Friend("b");  // 创建线程的简写，不需要操作线程对象，所以直接这么写  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  a.bow(b);  }  }).start();    **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  b.bow(a);  }  }).start();    }  }  **class** Friend {  String name;    **public** Friend(String name) {  **this**.name = name;  }  // a.bow(b)-》同步监视器是a,在同步方法没有执行完之前，a处于锁定状态  // b.bow(a)-》同步监视器是b,在同步方法没有执行完之前，b处于锁定状态  // 在这0.5秒发生了什么事情？  // a.bow(b),b.bow(a)都执行了（因为cpu运算速度是非常快的），此时a与b都被  // 作为同步监视器而锁定，那么锁定了之后，b.bowBack(a)需要获得b这个同步  // 监视器才能继续执行，a.bowback(b)需要获得a的同步监视器才能执行  // 又因为a与b都处于锁定状态，所以他们互相等待，发生了死锁    **public** **synchronized** **void** bow(Friend b) {  System.***out***.println(**this**.name + " say good day!");      **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    b.bowBack(**this**);    System.***out***.println(**this**.name + " raise up");  }    // b.bowBack(a) 同步监视器是b 在同步方法没有执行完之前，b处于锁定状态  **public** **synchronized** **void** bowBack(Friend a) {  System.***out***.println(**this**.name + " say thank you!");  }    } |

private Lock lock = new ReentrantLock();

private Condition condition = lock.newCondition();

condition

* await()相当于wait()方法
* await(long time, TimeUnit unit)相当于wait(long)方法
* signal()相当于notify()方法
* signalAll()相当于notifyAll()方法