第三章 多线程

一、线程概述

线程是进程的组成部分，一个进程可以拥有多个线程，一个线程必须有一个父进程。线程可以拥有自己的堆栈、自己的程序计数器和自己的局部变量，但不再拥有系统资源，它与父进程的其他线程共享该进程所拥有的全部资源。

线程的执行是抢占式的，也就是说，当前运行的线程在任何时候都可能被挂起，以便另一个线程可以运行。一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行。

一个程序运行后至少有一个进程，一个进程里可以包含多个线程，但至少要包含一个线程。

二、线程的创建和启动

1、继承Thread类创建线程类

Thread类创建并启动线程步骤：

①、创建线程子类（SubThread）继承Thread类；

②、子类中重写run（）方法；

③、主线程（main方法）中，实例化当前子线程类；

④、调用start（）方法，以启动子线程，并执行run()方法。

示例：

**class** SubThread **extends** Thread{

**public** **void** run(){

**for**(**int** i=1;i<=100;i++){

System.*out*.println(Thread.*currentThread*().getName()+":"+i);

}

}

}

**public** **class** TestThread{

**public** **static** **void** main(String[] args){

SubThread subThread = **new** SubThread();

subThread.start();

**for**(**int** i=1;i<=100;i++){

System.*out*.println(Thread.*currentThread*().getName()+":"+i);

}

}

}

**Tip:**

main方法的方法体就是java程序主线程的线程执行体。

继承Thread类创建的线程类，多条线程之间无法共享线程类的实例变量。

* Thread.*currentThread*():返回当前正在执行的线程对象；
* getName()：返回调用该线程的名字。

2、实现Runnable接口创建线程类

实现Runnable接口创建并启动多条线程的步骤：

①、定义实现Runnable接口的实现类；

②、重写run()方法；

③、创建一个Runnable接口实现类的对象；

④、将该对象作为形参传递给Thread类的构造器中，创建Thread类的对象，该 Thread类对象即为一个线程对象；

⑤、调用start()，启动线程：执行Thread对象构造器中形参对象的run()方法。

示例：

//第二种方式实现多线程：实现Runnable 接口的方式

**class** PrintThread **implements** Runnable{

**public** **void** run(){

**for**(**int** i=1;i<=100;i++){

**if**(i%2==0){

System.*out*.println(Thread.*currentThread*().getName()+":"+i);

}

}

}

}

**public** **class** TestThreadImp{

**public** **static** **void** main(String[] args) {

PrintThread p1 = **new** PrintThread();

Thread t1 = **new** Thread(p1);

t1.start();

}

}

3、两种方式的比较

1.联系：都必须实现Runnable接口，并实现run()方法；

2.实现方式更好：

①、规避了java单继承的局限性；

②、如果多个线程要同时操作一份资源（或数据），实现方式更优。

4、Thread类常用方法

|  |  |
| --- | --- |
| 1 | **synchronized** **void** start()：启动线程，并调用run()方法 |
| 2 | **void** run()：实现子类必须重写，所有任务执行体都在run()中 |
| 3 | **static** **native** Thread currentThread()：获取当前线程对象 |
| 4 | **final** **void** setName(String name)：设置线程名 |
| 5 | **final** String getName()：获取线程名 |
| 6 | **static** **native** **void** yield();：调用此方法的线程释放当前CPU的执行权 |
| 7 | **final** **void** join()：  在A线程中调用B线程的join(),表示：当执行此方法，A线程停止执行，直至B线程执行完毕，A线程再接着执行 |
| 8 | **final** **synchronized** **void** join(**long** millis)：  强制让调用此方法的的线程执行millis毫秒 |
| 9 | **final** **native** **boolean** isAlive()：判断线程是否还存活 |
| 10 | **static** **native** **void** sleep(**long** millis)：让当前线程睡眠ms毫秒 |
| 11 | **final** **void** setPriority(**int** newPriority)：改变线程优先级 |
| 12 | **final** **int** getPriority()：返回线程优先值 |

三、线程的生命周期

1、**new** （新创建）

当用new操作符实例化一个线程对象的时候，该线程处于该状态，该线程对象只是被实例化了，还没有运行。

2、**Runnable** （可运行）

当调用start()方法时，线程处于可运行状态，一个可运行的线程可能正在运行也可能没有运行，这取决于操作系统给线程提供运行的时间。

**Tip:**

不要对已经处于启动状态的线程再次调用start方法，否则将引发IllegalThreadStateException异常。

3、**Blocked** （被阻塞）

当一个线程试图获取一个内部的对象锁，而该锁被其他线程持有，则该线程进入阻塞状态。

4、**Waiting** （等待）

当线程等待另一个线程通知调度器一个条件时，它自己进入等待状态。调用Object.wait()和Thread.join()方法时，或者是调用java.util.concurrent库中的Lock或Condition时，就会出现这种情况。

5、**Timed\_waiting** （计时等待）

有几个方法有一个超时参数，调用它们将进入计时等待状态。这些方法为:

* **Thread.sleep(long l),**
* **Object.wait(long l),**
* **Thread.join(long l),**
* **Lock.TryLock(long l),**
* **Condition.awiat(long l)。**

6、**Terminated** （被终止）

有两个原因导致线程被终止：1、因为run方法正常退出而自然死亡；2、因为一个没有被捕获的异常终止了run方法而意外死亡。

要确定一个线程的状态，可以调用getState()方法。线程的阻塞和等待没有太大区别，只是一个是主动，一个是被动。

四、控制线程

1、join线程

在A线程中调用B线程的join方法，会让A线程等待B线程完成之后再运行。A线程进入阻塞状态，直到B线程运行完成。

2、线程睡眠：**sleep**

调用当前线程类的sleep方法会使当前线程从运行状态进入阻塞状态，在sleep时间段内，该线程不会获得执行的机会，即使系统中没有其他可运行的线程，处于sleep中的线程也不会运行，因此sleep方法常用来暂停程序的执行。

3、线程让步：**yield**

yield方法可以让当前正在执行的线程暂停，但是不会阻塞该线程，它只是将线程转入就绪状态，让系统的线程调度器重新调度一次。

当某个线程调用了yield方法后，只有优先级与当前线程相同或者优先级比当前线程更高的就绪状态的线程才会获得执行机会。

4、线程优先级

每个线程执行时都具有一定的优先级，优先级高的线程获得较多的执行机会，而优先级低的线程则获得较少的执行机会。

每个线程默认的优先级都与创建它的父线程具有相同的优先级，在默认情况下，main线程具有普通优先级。

Thread类提供了三个关于优先级静态常量，分别是：

/\* The minimum priority that a thread can have. \*/

**public** **final** **static** **int** *MIN\_PRIORITY* = 1;

/\* The default priority that is assigned to a thread. \*/

**public** **final** **static** **int** *NORM\_PRIORITY* = 5;

/\* The maximum priority that a thread can have. \*/

**public** **final** **static** **int** *MAX\_PRIORITY* = 10;

5、后台线程

有一种线程，它是在后台运行的，它的任务是为其他的线程提供服务，这种线程被称为“后台线程(Daemon Thread)，”又称“守护线程”。Jvm的垃圾回收线程就是典型的后台线程。

后台线程有个特征：如果前台线程都死亡，后台线程会自动死亡。

示例：

**public** **class** DaemonRunable **implements** Runnable {

@Override

**public** **void** run() {

**for**(**int** i=0;i<999;i++){

System.*out*.println(Thread.*currentThread*().getName()+":"+i);

}

}

**public** **static** **void** main(String[] args) {

DaemonRunable dr = **new** DaemonRunable();

Thread tr = **new** Thread(dr);

//将此线程设置成后台线程

tr.setDaemon(**true**);

tr.start();

**for**(**int** i=0;i<10;i++){

System.*out*.println(Thread.*currentThread*().getName()+":"+i);

}

}

}

本来tr后台线程应该运行到i=999才结束，但运行时发现后台线程无法运行到999，

因为当主线程，上述程序中唯一的前台进程运行结束后，JVM会自动退出，因而后台线程也就被结束了。

**Tip:**

前台线程死亡后，JVM会通知后台线程死亡，但从它接收指令，到它做出响应，需要一定时间。setDaemon()方法必须在start之前运行，否则抛出异常。

6、join,sleep,yeild综合实例

**class** TestSleep **extends** Thread{

@Override

**public** **void** run() {

setName("Sleep");

**for**(**int** i=0;i<50;i++){

**try** {

//让当前线程睡眠100ms

TestSleep.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.*out*.println(getName()+":"+i);

}

}

}

**class** TestYield **extends** Thread{

@Override

**public** **void** run(){

setName("Yield");

**for**(**int** i=0;i<50;i++){

//暂停单前线程，让系统调度器重新调度

TestYield.*yield*();

System.*out*.println(getName()+":"+i);

}

}

}

**class** TestJoin **extends** Thread{

**private** TestSleep testSleep;

**private** TestYield testYield;

**public** TestJoin(TestSleep testSleep, TestYield testYield){

**this**.testSleep = testSleep;

**this**.testYield = testYield;

}

@Override

**public** **void** run() {

setName("Join");

**for**(**int** i=0;i<50;i++){

System.*out*.println(getName()+":"+i);

**if**(i==5){

**try** {

//暂停当前线程直至testYield运行完成

testYield.join();

} **catch** (InterruptedException e){

e.printStackTrace();

}

}**else** **if**(i==10){

**try**{

//暂停当前线程直至testSleep运行完成

testSleep.join();

}**catch**(InterruptedException e){

e.printStackTrace();

}

}

}

}

//getter,setter方法省略

}

**public** **class** TestThreadAllMeans {

**public** **static** **void** main(String[] args) {

TestSleep ts = **new** TestSleep();

TestYield ty = **new** TestYield();

TestJoin tj = **new** TestJoin(ts,ty);

ts.start();

ty.start();

tj.start();

**for**(**int** i=0;i<50;i++){

System.*out*.println(tj.getName()+":"+tj.getState());

System.*out*.println(ts.getName()+":"+ts.getState());

System.*out*.println(ty.getName()+":"+ty.getState());

System.*out*.println(Thread.*currentThread*().getName()+":"+i);

}

}

}

五、线程同步

1、同步代码块

**synchronized**(同步监视器){

//需要被同步的代码块（即为操作共享数据的代码）

}

共享数据：多个线程共同操作同一个数据（变量）

同步监视器：由任意一个类的对象来充当，哪个线程获取该监视器，谁就执行大括号里被同步的代码。俗称：锁；

要求：所有的线程必须共用一个同步锁！

注：在实现的方式中，考虑同步的话，可以使用this来充当锁。但是在继承方式中，不能用this充当锁。

2、同步方法

将操作共享数据的的方法声明为**synchronized**。即此方法为同步方法，能够保证当其中的一个线程执行此方法时，其它线程等待。

同步方法也有同步监听器即锁，该锁为隐式的this对象。

3、同步锁（Lock）

该方式为显示定义同步锁对象来实现同步，同步锁使用Lock对象充当。Lock提供了比synchronized方法和代码块更广泛的锁定操作，支持多个相关的Condition对象。ReentrantLock锁具有重入性，ReentrantLock对象会维持一个计数器来追踪lock方法的嵌套调用，线程在每次调用lock（）加锁后，必须通过unlock()释放锁，所以一段被锁保护的代码可以调用另一被相同锁保护的方法。

**private** **final** Lock lock = **new** ReentrantLock();

//…

**public** **void** needLockMethod(){

lock.lock();

**try**{

//critical section

}**catch**(Exception e) { //… }

**finally**{

lock.unlock();

}

}

锁和条件：

* 锁用来保护代码片段，任何时刻只能有一个线程执行被保护的代码块；
* 锁可以管理试图进入被保护的代码段的线程；
* 锁可以拥有一个或多个相关的条件对象；
* 每个条件对象管理那些进入被保护的代码段但还不能运行的线程。

4、同步方法，同步代码块，同步锁实例代码

**public** **class** TestGetBanlance {

**public** **static** **void** main(String[] args) {

Account2 xm = **new** Account2("xiaoming",3000);

GetPerson gp = **new** GetPerson(xm);

Thread t1 = **new** Thread(gp);

Thread t2 = **new** Thread(gp);

Thread t3 = **new** Thread(gp);

t1.start();

t2.start();

t3.start();

}

}

**class** GetPerson **implements** Runnable{

**private** Account2 account;

**public** GetPerson(Account2 account){

**this**.account = account;

}

**public** **synchronized** **void** run() {

account.draw1(500);

account.draw2(200);

account.draw3(100);

}

//getter，setter方法省略

}

**class** Account2{

**private** String name;

**private** **int** balance;

**private** **final** Lock lock = **new** ReentrantLock();

**public** Account2(String name,**int** balance){

**this**.name = name;

**this**.balance = balance;

}

//同步锁

**public** **void** draw3(**int** aimCash){

lock.lock();

**try** {

**if**(balance<aimCash){

System.*out*.println("当前余额不足！");

}**else**{

balance=balance-aimCash;

System.*out*.println("M3当前余额："+balance);

}

}**catch**(Exception e) {

e.printStackTrace();

}**finally**{

lock.unlock();

}

}

//同步方法

**public** **synchronized** **void** draw1(**int** aimCash){

**if**(balance<aimCash){

System.*out*.println("当前余额不足！");

}**else**{

balance=balance-aimCash;

System.*out*.println("M1当前余额："+balance);

}

}

//同步代码块

**public** **void** draw2(**int** aimCash){

**synchronized**(**this**){

**if**(balance<aimCash){

System.*out*.println("当前余额不足！");

}**else**{

balance=balance-aimCash;

System.*out*.println("M2当前余额："+balance);

}

}

} }

六、线程通信

1、概述

当线程在系统内运行时，线程的调度具有一定的透明性，程序通常无法准确控制线程的轮换执行，但我们可以通过一些机制来保证线程协调运行。

2、线程协调运行

当使用同步方法和同步代码块的时候，可以使用wait(),notify(),notifyAll()使线程协调运行。

对于**synchronized**方法，同步监听器就是该类本身（**this**）,所以在同步方法中直接调用这三个方法；

对于**synchronized**代码块，同步监听器是括号内的对象，所以必须通过该对象调用这三个方法。

Java.lang.Object下的此三个方法：

**final** **void** wait():

导致当前线程等待，直到其他线程调用该同步监听器的notify()或notifyAll()方法来唤醒该线程。调用该方法的当前线程会释放对该同步监听器的锁定。

**final** **native** **void** notify():

唤醒在此同步监听器上等待的单个线程。如果所有线程都在此同步监听器上等待，则会唤醒其中一个线程。选择是任意的。只有当前线程放弃对该同步监听器的锁定后（使用wait()方法），才可以执行被唤醒的线程。

**final** **native** **void** notifyAll():

唤醒在此同步监听器上等待的所有线程。

3、条件变量控制协调

Condition使用方法如下代码所示：

**private** **final** Lock lock = **new** ReentrantLock();

**private** Condition condition = lock.newCondition();//进行线程通信

//…

**public** **void** needLockMethod(){

lock.lock();

**try** {

condition.await();

//critical section

condition.signalAll();

}**catch**(Exception e) { //… }

**finally**{

lock.unlock();

}

}

当使用同步锁（Lock）对象来保证线程同步时，Java提供了一个Condition类来保持协调。使用Condition可以让那些已经得到Lock对象却无法继续执行的线程释放Lock对象，Condition对象也可以唤醒其他处于等待的线程。

4、线程通信示例：

\*使用wait()协调线程运行：

**class** Account{

**private** String name;

**private** **double** balance;

//标识账户中是否已有存款的旗标

**private** **boolean** flag = **false**;

**public** Account(String name,**double** balance){

**this**.name = name;

**this**.balance = balance;

}

**public** **synchronized** **void** draw(**double** drawAmount){

**try**{

**if**(!flag){

**wait();**

}**else**{

balance-=drawAmount;

System.*out*.println("账户余额为："+balance);

flag = **false**;

**notifyAll();**//唤醒其他线程

}

}**catch**(InterruptedException ex){

ex.printStackTrace();

}

}

**public** **synchronized** **void** deposit(**double** depositAmount){

**try**{

**if**(flag){

**wait();**

}**else**{

balance+=depositAmount;

flag = **true**;

**notifyAll();**

}

}**catch**(InterruptedException ex){

ex.printStackTrace();

}

}

}

\*使用Condition对象控制协调

**class** Account{

**private** String name;

**private** **double** balance;

//标识账户中是否已有存款的旗标

**private** **boolean** flag = **false**;

**private** **final** Lock lock = **new** ReentrantLock();

**private** **final** Condition cdt = lock.newCondition();

**public** Account(String name,**double** balance){

**this**.name = name;

**this**.balance = balance;

}

**public** **void** draw(**double** drawAmount){

lock.lock();

**try**{

**if**(!flag){

**cdt.await();**

}**else**{

balance-=drawAmount;

System.*out*.println("账户余额为："+balance);

flag = **false**;

**cdt.signalAll();**//唤醒其他线程

}

}**catch**(InterruptedException ex){

ex.printStackTrace();

}

}

**public** **void** deposit(**double** depositAmount){

lock.lock();

**try**{

**if**(flag){

**cdt.await();**

}**else**{

balance+=depositAmount;

flag = **true**;

**cdt.signalAll();**

}

}**catch**(InterruptedException ex){

ex.printStackTrace();

}

}

}

\*线程通信两种方式测试代码：

**class** DrawThread **extends** Thread{

**private** Account account;

**private** **double** drawAmount;

**public** DrawThread(String name,Account account,**double** drawAmount){

**this**.account = account;

**this**.drawAmount = drawAmount;

}

**public** **void** run(){

**for**(**int** i=0;i<10;i++){

account.draw(drawAmount);

}

}

}

**class** DepositThread **extends** Thread{

**private** Account account;

**private** **double** depositAmount;

**public** DepositThread(String name,Account account,**double** depositAmount){

**this**.account = account;

**this**.depositAmount = depositAmount;

}

**public** **void** run(){

**for**(**int** i=0;i<10;i++){

account.deposit(depositAmount);

}

}

}

**public** **class** TestSynNotify {

**public** **static** **void** main(String[] args) {

Account acct = **new** Account("1234567",0);

**new** DrawThread("取钱者",acct,800).start();

**new** DepositThread("存款者甲",acct,800).start();

}

}

七、线程池

1、概述

系统启动一个新线程的成本是比较高的，所以在这种情况下，使用线程池可以大大提高性能。线程池在系统启动时即创建大量空闲的线程，将一个Runnable对象传给线程池，线程池就会启动一条线程来执行该对象的run方法，当run方法执行结束后，该线程并不会死亡，而是再次返回线程池中成为空闲状态，等待执行下个对象的run方法。

2、线程池的使用

①、调用Executors类的静态工厂方法创建一个ExecutorService对象，该对象 代表一个线程池；

②、创建Runnable实现类或Callable实现类的实例，作为线程执行任务；

③、调用ExecutorService对象的submit方法来提交Runnable实例或Callable 实例；

④、当不想提交任何任务时调用ExecutorService对象的shutdown方法来关闭 线程池。

3、示例代码

\*对线程通信实例的测试方法采用线程池的方式：

**public** **class** TestSynNotify {

**public** **static** **void** main(String[] args) {

Account acct = **new** Account("1234567",0);

//创建一个具有固定线程数的线程池

ExecutorService pool = Executors.*newFixedThreadPool*(6);

//向线程池中提交两个线程

pool.submit(**new** DrawThread1("取钱者",acct,800));

pool.submit(**new** DepositThread1("存款者",acct,800));

//关闭线程池

pool.shutdown();

}

}

八、ThreadLocal模式（以空间换时间）

**Tip:**

ThreadLoacl模式用来保证线程安全，当并非实现线程同步；ThreadLocal不是用来解决共享对象的多线程访问问题的，一般情况下，通过ThreadLocal.set()到线程中的对象是该线程自己使用的对象，其他线程是不需要访问的，也访问不到的**。**各个线程中访问的是不同的对象。

实现步骤：

①、建立一个类，并在其中封装一个静态的ThreadLocal变量，使其成为一个共享数据环境。

②、在类中实现访问静态ThreadLocal变量的静态方法（设值和取值）。

实例代码：

**class** Account5{

**private** **static** ThreadLocal<Integer> *tdll* =

**new** ThreadLocal<Integer>(){

**protected** Integer initialValue(){

**return** 2000;

}

};

**private** String name;

**public** **static** **int** getBalance(){

**return** *tdll*.get();

}

**public** **static** **void** setBalance(Integer value){

*tdll*.set(value);

}

**public** **static** **void** draw(**int** aimCash){

**if**(*tdll*.get()==0){

System.*out*.println("帐户余额为0！");

}**else** **if**(*tdll*.get()<aimCash){

System.*out*.println("帐户余额不足");

}**else**{

*tdll*.set(*tdll*.get()-aimCash);

System.*out*.println("取款金额："+aimCash);

System.*out*.println("帐户余额："+*tdll*.get());

}

}

}

其他部分无异同，在此不在赘述。

ThreadLocal并不能替代同步机制，两者面向的问题领域不同。同步机制是为了同步多个线程对相同资源的并发访问，是多个线程之间进行通信的有效方式；而ThreadLocal是隔离多个线程的数据共享，从根本上避免了多个线程之间共享资源，也就不需要对多个线程进行同步了。

通常我们认为：如果需要进行多个线程之间共享资源，以达到线程之间通信的功能，就使用同步机制，如果仅仅需要隔离多个线程之间共享冲突，可以使用ThreadLocal。