Java中的多线程

**方式一：继承Thread类创建线程**

**public** **class** FirstThread **extends** Thread{

**private** **int** i;

@Override

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

**for**(i = 0; i < 100; i++){

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

}

}

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

**for**(**int** i = 0; i < 100; i++){

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

**if**(i == 20){

**new** FirstThread().start();

**new** FirstThread().start();

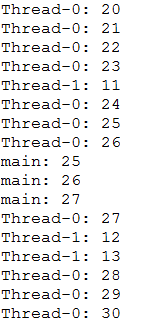
}

}

}

}

运行结果：



在Java中有如下两个方法：

**Thread.currentThread()：**currentThread是Thread类的静态方法，该方法总是返回当前正在执行的线程对象

**getName()：**该方法返回调用该方法的线程名字，在默认情况下，主线程的名字为main，用户启动的多个线程的名字依次为Thread-0、Thread-1…等

**方式二：实现Runnable接口创建线程**

**public** **class** multiThread **implements** Runnable{

**private** **int** i;

@Override

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

**for**(; i < 100; i++){

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

}

}

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

**for**(**int** i = 0; i < 100; i++){

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

**if**(i == 20){

multiThread firstThreadRunnable = **new** multiThread();

**new** Thread(firstThreadRunnable).start();

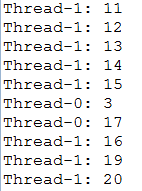
**new** Thread(firstThreadRunnable).start();

}

}

}

}



在这种情况下会发现两个线程共享了变量，导致结果比较乱

如果将里面的改为：

multiThread firstThreadRunnable = **new** multiThread();

multiThread secondThreadRunnable = **new** multiThread();

**new** Thread(firstThreadRunnable).start();

**new** Thread(secondThreadRunnable).start();

一切又恢复正常

通过上面的比较会发现：

采用继承Thread方式创建线程可以直接使用this.currentThread方法，采用实现Runnable接口的方式只能使用Thread.currentThread方法，前者创建的Thread子类即可代表线程本身，后者创建的Runnable对象只能作为线程对象的target。同时会发现采用Runnable接口创建的多个线程可以共享线程类的实例属性，这是因为在这种方法下，程序所创建的Runnable对象只是线程的target，而多个线程可以共享同一个target，所以多个线程可以共享同一个线程类（实际上是线程的target类）的实例属性

注意：在启动线程时，要调用start方法，而不要直接调用run方法，如果直接调用run方法，则run方法就变为一个普通函数，而不会产生线程

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

**for**(**int** i = 0; i < 100; i++){

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

**if**(i == 20){

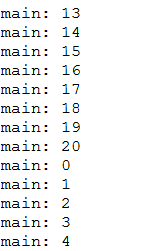
**new** FirstThread().run();

**new** FirstThread().run();

}

}

}

可以看出完全是主线程在运行

**线程死亡：**

线程会以如下三种方式死亡：

（1）run()或call()方法执行完成，线程正常结束

（2）线程抛出一个未捕获的Exception或Error

（3）直接调用该线程的stop()方法来结束该线程—--该方法容易导致死锁，不推荐使用

**控制线程：**

Join线程：Thread提供了让一个线程等待另一个线程完成的方法—join方法，当在某个程序执行流中调用其它线程的join方法时，调用线程将被阻塞，知道被join()方法加入的join线程执行完为止

Join方法有三种重载方式：

1. join：等待被join的线程执行完成
2. join(long millis)：等待被join的线程时间最长为millis毫秒，如果在millis毫秒内被join的线程还没有执行结束，则不再等待
3. join(long millis, int nanos)：等待被join的线程时间最长为millis毫秒加nanos毫微秒

**public** **class** FirstThread **extends** Thread{

**private** **int** i;

@Override

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

**for**(i = 0; i < 100; i++){

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

}

}

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

**new** FirstThread().run();

**for**(**int** i = 0; i < 100; i++){

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

**if**(i == 20){

FirstThread firstThread = **new** FirstThread();

firstThread.start();

**try**{

**firstThread.join();**

}**catch**(Exception e){

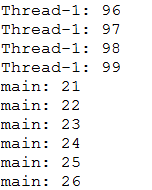
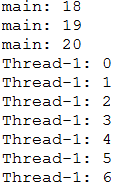
}

}

}

}

}



**后台线程：**

后台线程运行于后台，它的任务是为其它线程提供服务，JVM的垃圾回收线程就是典型的后台线程，调用Thread对象的setDawmon(true)方法就可以将指定线程设置为后台线程，当所有的前台线程都死亡以后，后台线程将随之死亡，当整个虚拟机中只剩下后台线程时，程序就没有继续运行的必要了，所以虚拟机就退出了

**public** **class** FirstThread **extends** Thread{

**private** **int** i;

@Override

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

**for**(i = 0; i < 10000; i++){

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

}

}

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

**for**(**int** i = 0; i < 100; i++){

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

**if**(i == 20){

FirstThread firstThread = **new** FirstThread();

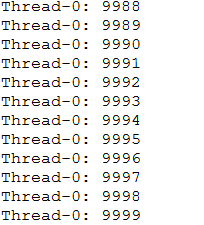
firstThread.start();

}

}

}

}

我们发现在主线程死亡后，子线程仍然在运行

**public** **class** FirstThread **extends** Thread{

**private** **int** i;

@Override

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

**for**(i = 0; i < 10000; i++){

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

}

}

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

**for**(**int** i = 0; i < 100; i++){

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

**if**(i == 20){

FirstThread firstThread = **new** FirstThread();

**firstThread.setDaemon(true);**

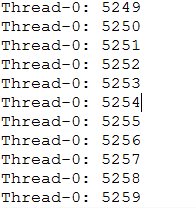
firstThread.start();

}

}

}

}



在将子线程设置为后台线程后，主线程死亡后，子线程（后台）也跟着死亡了

从上面的程序可以看出，主线程默认是前台线程，产生的子线程默认也是前台线程，并不是所有的线程默认都是前台线程，有些线程默认就是后台线程---前台线程创建的子线程默认是前台线程，后台线程创建的子线程默认是后台线程

**注意一点：**

前台线程死亡后，JVM会通知后台线程死亡，但从它接受指令到做出响应，需要一定时间，而且要将某个线程设置为后台线程，必须在该线程启动之前设置，也就是说，setDaemon(true)必须在start()方法调用之前调用，否则会引起IllegalThreadStateException异常，例如：

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

Thread.*currentThread*().setDaemon(**true**);

}

在main函数中将主线程设置为后台线程发生异常：



**线程休眠与线程让步：**

线程休眠sleep，线程让步yield

线程让步与线程休眠有点相同，它也可以让当前正在执行的线程暂停，但它不会阻塞该线程，它只是将该线程转入就绪状态，yield只是让当前线程暂停一下，让系统的线程调度器重新调度一次，完全可能的情况是：当某个线程调用了yield方法暂停后，线程调度器又将其调度出来重新执行。

关于sleep与yield方法的区别如下：

1. sleep暂停当前线程后会给其它线程机会，不会理会其它线程的优先级，但yield方法只会给优先级相同或者优先级更高的线程执行机会
2. sleep会将线程转入阻塞状态，直到经过阻塞时间才会转入就绪状态，而yield不会讲线程转入阻塞状态，只是强制当前线程转入就绪状态，因此完全有可能某个线程调用yield暂停后，立即再次获得处理器资源被执行
3. sleep方法生命抛出了InterrupterdException异常，所以调用sleep方法要捕获该异常，要么显式生命抛出该异常，而yield方法则没有声明抛出任何异常
4. sleep比yield方法有更好的移植性，通常不建议使用yield

**多线程的同步**

（1）同步代码块

Java的多线程支持引入同步监视器来解决同步问题，使用同步监视器的通用方法就是同步代码块，同步代码块如下：

Synchronized(obj){

}

上面的Synchronized后面括号里的obj就是同步监视器，上面代码的含义是：线程开始执行同步代码块之前，必须先获得同步监视器的锁定。任何时刻只能有一个线程可以获得对同步监视器的锁定，当同步代码块执行完成后，该线程会释放对该同步监视器的锁定

（2）同步方法

与同步代码块对应，Java的多线程还提供了同步方法，同步方法就是使用synchronized修饰的方法，对于同步方法而言，无须显式指定同步监听器，同步方法的同步监视器就是this，也就是该对象本身。通常我们只对竞争资源或修改了竞争资源的方法进行同步

Public synchronized void draw()

（3）同步锁

Lock与ReadWriteLock是Java5新提供的两个根接口，并为Lock提供了ReentrantLock(可重入锁)实现类，在实现线程安全的控制中，比较常用的是ReentrantLock，一般用法为：

private final ReentrantLock lock = new ReentrantLock();

try{

lock.lock();

}finally{

lock.unlock();

}

总结：

Synchronized方法控制对类成员变量的访问：每个类实例对应一把锁，每个synchronized方法都必须获得调用该方法的类实例的锁才能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程才能获得该锁，重新进入可执行状态，也就是同一时刻对于每一个类实例至多只有一个能够获得该类实例对应的锁。在Java中，不仅每个类实例对应一把锁，每个类也对应一把锁，这牙膏我们也可将类的静态成员函数声明为synchronized，以控制其对类的静态成员变量的访问。

Synchronized方法的缺陷：若将一个大的方法声明为synchronized将会大大影响效率，典型地，若将线程类的方法run()声明为synchronized，则在该线程运行期间，其它线程就没有机会运行该方法了，当然我们可以将访问类成员变量的方法提出来放到另外一个synchronized方法中，但是这样我们可以使用synchronized代码块

**注意：**synchronized是不能继承的，也就是说，基类的方法synchronized f()在继承类中并不自动是synchronized f()，而是变成了f()，继承类需要显式的指定它的某个方法为synchronized方法

**线程通信**

（1）传统的线程通信：

[synchronized、Object.wait()、Object.notify()/notifyAll()原理](http://blog.csdn.net/achellies/article/details/7094467)

**Sychronized与互斥锁：**

synchronized用来锁定某个对象，体现的是标准的互斥锁mutex的机制，synchronized代码块的开始即是lock该mutex，代码块的结束即是unlock。

**Object.wait()与条件变量**

Object.wait()在使用时通常要判断是否满足某个条件，不满足某个外部条件cond时调用wait()，来让线程阻塞同时释放被synchronized锁定的mutex；从这个过程看来Object.wait()实际上是起到条件变量的作用，wait()内部实际上先将synchronized锁定的锁释放，之后将当前线程阻塞在某个内置的条件condition上（注意：此condition为内置的，与外部判断的条件cond并非同一个，外部的cond需要程序员根据程序逻辑来判断改变，而这个condition只能被Object.notify()/notifyAll()改变），直到内置条件condition被Object.notify()/notifyAll()修改时才会重新锁定该mutex，继续执行wait()后的代码。

**Object.notify()、notifyall与条件变量**

Object.notify()/notifyAll()实际上只起到一个sinal内置条件变量的作用，调用Object.notify()/notifyAll()之后，这个时候其他处于wait()中的线程所等待的内置条件变量已经满足，但是由于wait()中仍然需要lock mutex，而在Object.notify()/notifyAll()中没有把mutex释放掉，故阻塞在wait()处的线程继续等待，但等待的条件不再是内置条件变量而是锁mutex；直到synchronized代码块结束时，由于会自动释放被synchronized锁定的mutex，故此时所有在wait()中等待mutex的线程开始竞争mutex，得到该mutex的会继续执行，否则继续等待mutex。

**相关代码：**

synchronized (obj) {// 此处相当于mutex=obj，lock(mutex)

    while (!cond) {// 判断外部条件cond，不满足时让线程wait();

        obj.wait();

    }

    // .....执行满足条件cond时的逻辑过程

    obj.notifyAll();// 更改内置条件condition

}// 此处相当于mutex=obj，unlock(mutex)

**obj.wait()内部实现(伪代码)：**

wait() {

    unlock(mutex);//解锁mutex

    wait\_condition(condition);//等待内置条件变量condition

    lock(mutex);//竞争锁

}

**obj.notify()/notifyAll()内部实现(伪代码)：**

obj.notify()/notifyAll(){

    condition=true;//只起到把内置条件变量置为true的作用

}

注意:该代码中之所以用while循环判断该外部条件cond，是因为，第一次Object.notify()/notifyAll()被调用之后，所有线程所等待的内部条件condition都已经满足。此时所有线程竞争mutex，而mutex只会被其中一个线程获得，其余线程继续等待在mutex上。当获得mutex的那个线程执行结束时，所有线程又会竞争mutex，此时，某个线程因为获得了mutex而继续执行。 但这个过程中该线程并没有判断外部条件cond是否成立。 也许在第一次获得mutex的线程中已经将外部条件cond改变为false，而当前获得mutex的线程没有判断cond是否为true而直接执行了后续的代码，相当于通过漏洞执行了后续的代码，必然导致程序逻辑的错误。故，必须用while判断外部条件。同时，由此得出，wait和notify所等待和改变的内置条件变量condition一定在obj对象中，只有这样所有被锁定在obj上的线程才能共享该condition。故可以做这样的推断：  
被synchronized锁定的mutex为mutex =obj.mutex；  
被wait和notify共享的内置条件变量condition = obj.condition；

**一个例子（一个存款，一个取款，且两次存款与取款不能连续进行）：**

**public** **class** Account {

**private** String accountNo;//账户ID

**private** **double** balance;//账号剩余的金额

**private** **boolean** flag = **false**;//账户是否有存款

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

**this**.accountNo = accountNo;

**this**.balance = balance;

}

**public** **double** getBalance(){

**return** balance;

}

**public** **synchronized** **void** Draw(**double** drawAmount){//取款

**try**{

**if**(!flag){

wait();

}

**if**{

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

balance -= drawAmount;

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

flag = **false**;

notifyAll();

}

}**catch**(InterruptedException e){

e.printStackTrace();

}

}

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

**try**{

**if**(flag){

wait();

}

**if(flag)** {

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

balance += depositAmount;

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

flag = **true**;

notifyAll();

}

}**catch**(InterruptedException e){

e.printStackTrace();

}

}

}

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

**private** Account account;

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

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

**super**(name);

**this**.account = account;

**this**.depositAmount = depositAmount;

}

@Override

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

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

account.deposit(depositAmount);

}

}

}

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

**private** Account account;

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

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

**super**(name);

**this**.account = account;

**this**.drawAmount = drawAmount;

}

@Override

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

**for**(**int** i = 0; i < 5; i++){

System.*out*.println("i = " + i);

account.Draw(drawAmount);

}

}

}

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

Account account = **new** Account("1234", 0);

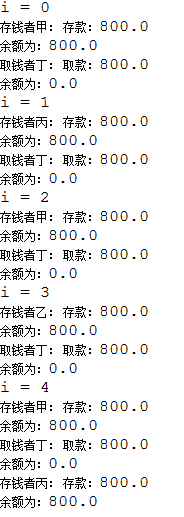
**new** DepositThread("存钱者甲：", account, 800).start();

**new** DepositThread("存钱者乙：", account, 800).start();

**new** DepositThread("存钱者丙：", account, 800).start();

**new** DrawThread("取钱者丁：", account, 800).start();

}



（2）使用Condition控制线程通信

如果不使用suychronized关键字保证同步，而是采用Lock对象来保证同步，则系统中不存在隐式同步监视器，也就不能使用wait、notify与notifyAll方式通信了。

当使用Lock时来实现同步时，Java提供了一个Condition类来保持协调，实际上就是Lock代替了同步方法与同步代码块，Condition代替了同步监视器的功能

取代后的代码为：

**public** **void** Draw(**double** drawAmount){//取款

**lock.lock();**

**try**{

**if**(!flag){

**condition.await();**

}

**if**(flag){

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

balance -= drawAmount;

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

flag = **false**;

**condition.signalAll();**

}

}**catch**(InterruptedException e){

e.printStackTrace();

}**finally**{

**lock.unlock();**

}

}

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

**lock.lock();**

**try**{

**if**(flag){

**condition.await();**

}

**if**(!flag){

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

balance += depositAmount;

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

flag = **true**;

**condition.signalAll();**

}

}**catch**(InterruptedException e){

e.printStackTrace();

}**finally{**

**lock.unlock();**

**}**

}

注意到黑体字区别，同时发现取钱与存钱函数没有synchronized关键字

**线程池**

使用线程池执行线程任务的步骤：

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方法关闭线程池

**public** **class** FirstThread **implements** Runnable{

@Override

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

**for**(**int** i = 0; i < 20; i++){

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

}

}

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

ExecutorService executor = Executors.*newFixedThreadPool*(6);//创建线程池

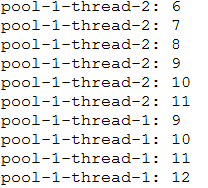
executor.submit(**new** FirstThread());//提交线程执行体

executor.submit(**new** FirstThread());

executor.shutdown();//关闭线程池

}

}



**ThreadLocal类**

ThreadLocal是线程局部变量的意思，它就是为每个使用该变量的线程都提供一个变量值的副本，使每一个线程都可以独立改变自己的副本，而不会和其他的线程冲突，从线程的角度上看，就像每个线程都拥有该变量

ThreadLocal不能代替同步机制，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是为了隔离多个线程的数据共享，从根本上避免多个线程之间对共享资源（变量）的竞争，也就不需要对多个线程进行同步了