# Java多线程

1. 实现多线程的两种方法：

继承Thread类 ：

实现Runnable接口 ：避免了单继承的局限性。

1. Thread类的常用方法：

start() :使该线程开始执行；Java 虚拟机调用该线程的 run 方法。

多次启动一个线程是非法的。特别是当线程已经结束执行后，不能再重新启动。会抛出[IllegalThreadStateException](mk:@MSITStore:C:\Users\Koterwong\Desktop\JDK_API_1_6_zh_CN.CHM::/java/lang/IllegalThreadStateException.html)异常

run() :存放线程执行的代码

Thread.yield() : 让出处理器时间，并执行其他线程。没有执行完毕会继续抢占CPU

join() : 让其他线程等待该线程结束。当A线程执行到了B线程的.join()方法时，A就会等待。等B线程都执行完，A才会执行。

setDaemon(boolean on) :将该线程标记为守护线程或用户线程。后台线程，在线程开始前调用，当所有线程都是后台线程时，则该线程结束。和interrupt()都可以结束线程。

interrupt() :打断线程的等待获睡眠状态。并让线程收到[InterruptedException](mk:@MSITStore:C:\Users\Koterwong\Desktop\JDK_API_1_6_zh_CN.CHM::/java/lang/InterruptedException.html)。

static Thread currentThread() 返回正在执行的线程对象引用

getName() : 对应setName 可在创建线程的时候，在构造方法中直接传name。

toString() : 返回该线程的字符串表示形式，包括线程名称、优先级和线程组。

1. 多线程的安全问题

导致的安全问题 :当多条语句在操作同一个线程共享数据时，一个线程对多条语句只执行了一部分，还没有执行完，另一个线程参与进来执行。导致共享数据的错误。

Java提供的解决方法:

1. 同步代码块
2. 同步函数(给被线程调用的函数加锁，锁是this)。
3. 静态同步函数，使用的对象是 类名.class文件。

synchronized(对象) //对象如同锁。

{

需要被同步的代码 //即使被CPU执行权被抢走，代码块被锁，其他线程进不来。

}

同步的前提：

1. 必须要有两个或者两个以上的线程。
2. 必须是多个线程使用同一个锁。

好处：解决了多线程的安全问题。

弊端：多个线程需要判断锁，较为消耗资源，

1. 死锁

//同步中嵌套同步

//两个线程分别持有对方的锁，都进不去同步函数。

class Test implements Runnable{

private boolean flag;

Test(boolean flag){

this.flag = flag;

}

public void run(){

if(flag){

while(true){

synchronized(MyLock.locka){ //locka锁住。

synchronized(MyLock.lockb){

}

}

}

}

else{

while(true){

synchronized(MyLock.lockb){ //lockb锁住

synchronized(MyLock.locka){

}

}

}

}

}

}

//定义锁

class MyLock{

static Object locka = new Object();

static Object lockb = new Object();

}

class DeadLockTest{

public static void main(String[] args) {

Thread t1 = new Thread(new Test(true));

Thread t2 = new Thread(new Test(false));

t1.start();

t2.start();

}

}

线程间通讯

多个线程在操作同一个资源，但是操作的动作不同。

都使用在同步中，因为要对持有监视器(锁)的线程操作。只有同步才具有锁。

wait()、notify() :为什么要定义Object类中呢？

wait()、notify() 必须是同一个锁对象调用。才能等待和唤醒对象。

而锁可以是任意对象，所以可以被任意对象调用的方法定义Object类中。

**//写入和读取操作，一个进行完另一进行。**

**class** Res{

**private** String name = "";

**private** String sex = "";

**public** **boolean** flag = **false**;

**public** **synchronized** **void** set(String name, String sex) {

**if** (flag) {

**try** {

**this**.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**this**.name = name;

**this**.sex = sex ;

flag = **true**; //写入线程执行

**this**.notify();

}

**public** **synchronized** **void** get() {

**if** (!flag) {

**try** {

**this**.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.***out***.println(name+"....."+sex);

flag = **false**;

**this**.notify();

}

}

//定义写入线程

**class** Input **implements** Runnable{

Res res = **null**;

**public** Input(Res res) {

**this**.res = res;

}

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

**int** i = 0;

**while**(**true**){

**if** (i==0)

res.set("koter", "man");

**else**

res.set("lisa", "women");

i= (i+1)%2;

}

}

}

//定义读取线程

**class** Output **implements** Runnable{

Res res = **null**;

**public** Output(Res res) {

**this**.res = res;

}

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

**while**(**true**){

res.get();

}

}

}

**public** **class** Demo{

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

Res res = **new** Res();

**new** Thread(**new** Input(res)).start();

**new** Thread(**new** Output(res)).start();

}

}

# 多线程通讯(生产者与消费者)生产一个消费一个

JDK1.5 中将同步Synchronized替换成现实Lock操作。

将Object中的wait，notify notifyAll，替换了Condition对象。

该对象可以通过Lock进行获取。实现了本方只唤醒对方操作。

**class** Resourse{

**private** String name;

**private** **int** count = 1;

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

//定义Lock锁

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

//分别为生产者，消费者获取不同condition对象。

**final** Condition conditionPro = lock.newCondition();

**final** Condition conditionCus = lock.newCondition();

**public** **void** set(String name) **throws** InterruptedException {

lock.lock();

**try**{

**while** (flag) {

conditionPro.await();

}

**this**.name = name+(count++); System.***out***.println(Thread.*currentThread*().getName()+**this**.name);

flag = **true**;

//只唤醒消费者线程

conditionCus.signalAll();

}**finally**{

lock.unlock();

}

}

**public** **synchronized** **void** get() **throws** InterruptedException {

lock.lock();

**try** {

**while** (!flag) {

conditionCus.await();

} System.***out***.println(Thread.*currentThread*().getName()+"....."+name);

flag = **false**;

//只唤醒生产者线程

conditionPro.signalAll();

} **finally** {

lock.unlock();

}

}

}

**class** Product **implements** Runnable{

**private** Resourse resourse;

**public** Product(Resourse resourse) {

**this**.resourse = resourse;

}

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

**int** i = 0;

**while**(i<1000){

**try** {

resourse.set("商品");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

i++;

}

}

}

**class** Custom **implements** Runnable{

**private** Resourse resourse;

**public** Custom(Resourse resourse) {

**this**.resourse = resourse;

}

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

**int** i = 0;

**while**(i<1000){

**try** {

resourse.get();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

i++;

}

}

}

**public** **class** LockInsteadOfS{

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

Resourse resourse = **new** Resourse();

Product product = **new** Product(resourse);

Custom custom = **new** Custom(resourse);

**new** Thread(product,"生产者01").start();

**new** Thread(product,"生产者02").start();

**new** Thread(custom,"消费者01").start();

**new** Thread(custom,"消费者02").start();

}

}

# 停止线程。

线程执行的run方法结束，则该线程停止。一般run方法都是一循环结构存在。只要能控制循环结束，那么线程就结束。

特殊情况：

当线程处于了冻结状态。就不会读取到控制结束循环标记。那么线程就会一直处于等待。

这样就可以强制让线程恢复到运行状态中来。下一次循环读取标记，呢么线程结束。

1. Thread类提供该方法 interrupt()
2. 不用标记，设置线程为守护线程 ，当全部线程时守护线程时，程序结束(圣斗士星矢守护雅典娜)。

public class Demo{

public static void main(String[] args) {

StopThread st = new StopThread();

Thread t1 = new Thread(st,"线程一");

Thread t2 = new Thread(st,"线程二");

//设置守护线程，主线程运行结束，两个守护线程结束。

t1.setDaemon(true);

t2.setDaemon(true);

t1.start();

t2.start();

int num = 0;

while(true){

if(num++ == 60){

// t1.interrupt(); //打断线程的等待状态，并接受interruptException

// t2.interrupt();

break;

}

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

}

System.*out*.println("over");

}

}

class StopThread implements Runnable{

private boolean flag =true;

public synchronized void run(){

while(flag){

if (flag) {

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

// changeFlag(); //线程等待状态被强制打断，则抛出异常。catch到异常后，即可该停止线程。

}

}

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

}

}

//改变标记后，线程就结束。

public void changeFlag(){

flag = false;

}

}