了解JAVA多线程编程的人都知道，要产生一个线程有两种方法，一是类直接继承Thread类并实现其run()方法；二是类实现Runnable接口并实现其run()方法，然后新建一个以该类为构造方法参数的Thread，类似于如下形式: Thread t=new Thread(myRunnable)。而最终使线程启动都是执行Thread类的start()方法。

在JAVA中，一个线程一旦运行完毕，即执行完其run()方法，就不可以重新启动了。此时这个线程对象也便成了无用对象，等待垃圾回收器的回收。下次想再启动这个线程时，必须重新new出一个线程对象再start之。频繁地创建和销毁对象不仅影响运行效率，还可能因无用线程对象来不及被回收而产生大量的垃圾内存，在存储空间和处理速度都相对受限的移动平台上这种影响尤为显著。那么，能否重新设计一种线程类，使其能够被反复启动而无需频繁地创建和销毁对象呢？

当然可以。下面我就介绍一下对这个“可重启线程”类的设计。

首先必须明确，如果仍是把想要线程去做的任务直接放在线程的run()方法中，是无论如何无法达成目的的，因为就像上面已经说的，JAVA的线程类一旦执行完run()方法就无法再启动了。所以唯一可行的办法是，把用户程序要做的run()方法（不妨称作“用户过程”）套在线程实际的run()方法内部的while循环体内，当用户过程执行完后使线程wait。当调用restart方法重启线程时，实际就是唤醒等待中的线程使之开始下一次while循环。大致的思想确定了，下面的代码就很好理解了：

public class ReusableThread implements Runnable {

//线程状态监听者接口

public interface ThreadStateListener {

public abstract void onRunOver(ReusableThread thread);//当用户过程执行完毕后调用的方法

}

public static final byte STATE\_READY=0; //线程已准备好，等待开始用户过程

public static final byte STATE\_STARTED=1; //用户过程已启动

public static final byte STATE\_DESTROYED=2; //线程最终销毁

byte mState; //标示可重启线程的当前状态

Thread mThread; //实际的主线程对象

Runnable mProc; //用户过程的run()方法定义在mProc中

ThreadStateListener mListener; //状态监听者，可以为null

/\*\* Creates a new instance of ReusableThread \*/

public ReusableThread(Runnable proc) {

mProc = proc;

mListener = null;

mThread = new Thread(this);

mState = STATE\_READY;

}

public byte getState() {return mState;}

public void setStateListener(ThreadStateListener listener) {

mListener = listener;

}

/\*\*可以在处于等待状态时调用该方法重设用户过程\*/

public synchronized boolean setProcedure(Runnable proc) {

if (mState == STATE\_READY) {

mProc = proc;

return true;

}

else

return false;

}

/\*\*开始执行用户过程\*/

public synchronized boolean start() {

if (mState == STATE\_READY) {

mState = STATE\_STARTED;

if (!mThread.isAlive()) mThread.start();

notify(); //唤醒因用户过程执行结束而进入等待中的主线程

return true;

}

else

return false;

}

/\*\*结束整个线程，销毁主线程对象。之后将不可再次启动\*/

public synchronized void destroy() {

mState = STATE\_DESTROYED;

notify();

mThread = null;

}

public void run() {

while (true) {

synchronized (this) {

try {

while (mState != STATE\_STARTED) {

if (mState == STATE\_DESTROYED) return;

wait();

}

} catch(Exception e) {e.printStackTrace();}

}

if (mProc != null) mProc.run();

if (mListener != null) mListener.onRunOver(this); //当用户过程结束后，执行监听者的onRunOver方法

synchronized (this) {

if (mState == STATE\_DESTROYED) return;

mState = STATE\_READY;

}

}

}

}

代码很好懂是不是？但是要解释一下为什么要有一个“状态监听者”接口。有时候我们可能想要在用户过程结束后得到一个及时的通知，好进行另外的处理，这时状态监听者的onRunOver方法就有了用处。一个直观的例子是，在下面要提到的“线程池”类中，一个可重启线程执行完一次用户过程后应当自动回收入池，这时就可以把回收入池的动作放在onRunOver方法中，而它的参数就是该可重启线程对象，于是就可以把参数所指示的对象回收进线程池中。

至于线程池类，其实就是以前提到的对象池类的一个子类，其中的对象全是ReusableThread类的。另外它实现了ReusableThread.ThreadStateListener接口，以便可以在用户过程结束时及时收到通知，执行回收线程的工作：

public class ThreadPool extends ObjectPool implements ReusableThread.ThreadStateListener {

public static final int DefaultNumThreads = 16; //默认池容量

public ReusableThread getThread() {

return (ReusableThread)fetch();

}

public void onRunOver(ReusableThread thread) {

recycle(thread); //当用户过程结束时，回收线程

}

private void init(int size) {

ReusableThread thread;

//初始化线程池内容

for (int i=0; i<size; i++) {

thread = new ReusableThread(null);

thread.setStateListener(this);

setElementAt(thread, i);

}

}

public ThreadPool(int size) {

; super(size);

init(size);

}

public ThreadPool() {

super(DefaultNumThreads);

init(DefaultNumThreads);

}

}

当然，还有一些可能需要添加的功能，因为既然只是比普通线程多了一个可重启的“增强”型线程类，那么原来Thread类具有的功能也应该具有，比如线程的sleep()。不过那些比较简单，这里就略去了。

下面编写测试程序。我准备这样进行：并不用到线程池类，而是对对象池类和可重启线程类进行联合测试，该对象池中的对象所属的类CharEmitter实现了Runnable接口和线程状态监听者接口，并且含有一个可重启线程成员对象，它并不包含在任何线程池对象中，而是独立使用的。当此线程的用户过程（定义在CharEmitter类中）结束后，onRunOver方法执行回收本CharEmitter对象入池的动作。这样就同时起到了间接测试线程池类的作用，因为它与对象池的区别也不过是在onRunOver中执行回收动作而已。

还是直接上代码说得清楚：

TestThreadPool.java :

/\*\*字符放射器\*/

class CharEmitter implements Runnable, ReusableThread.ThreadStateListener {

char c; //被发射的字符

boolean[] isEmitting; //标示某字符是否正被发射（直接以字符对应的ASCII码作下标索引）

ReusableThread thread; //可重启线程对象

ObjectPool myHomePool; //为知道应把自己回收到哪里，需要保存一个到自己所在对象池的引用

CharEmitter(ObjectPool container, boolean[] isCharEmitting) {

isEmitting=isCharEmitting;

myHomePool=container;

thread=new ReusableThread(this); //新建可重启线程对象，设其用户过程为CharEmitter类自己定义的

}

/\*\*开始“发射”字符\*/

public void emit(char ch) {

//字符被要求只能是'0'到'9'之间的数字字符

if (ch>='0' && ch<='9') {

c=ch;

}

else c=' ';

thread.start(); //启动线程

}

public void run() {

if (c==' ') return; //若不是数字字符直接结束

//为便于观察，不同数字之前的空格数目不同，以便将其排在不同列上

int spaceLen=c-'0';

StringBuffer s=new StringBuffer(spaceLen+1);

for (int i=0; i<spaceLen; i++) s.append(' ');

s.append(c);

while (isEmitting[c]) {

System.out.println(s); //不断地向屏幕写字符

}

}

/\*\*实现线程状态监听者接口中的方法\*/

public void onRunOver(ReusableThread t) {

myHomePool.recycle(this); //回收自身入池

}

}

public class TestThreadPool {

public static void main(String[] args) {

// TODO Auto-generated method stub

//标示字符是否正被发射的标志变量数组

boolean[] isEmitting=new boolean[256];

for (int i=0; i<256; i++) isEmitting[i]=false;

ObjectPool emitters=new ObjectPool(10); //新建对象池，容量为10

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

//用CharEmitter对象填满池子

emitters.setElementAt(new CharEmitter(emitters, isEmitting), i);

}

byte[] c=new byte[1];

CharEmitter emitter;

while(true) {

try {

System.in.read(c); //从键盘读入一个字符，以回车键表示输入结束

} catch(Exception e) {e.printStackTrace();}

if (isEmitting[c[0]]) {

isEmitting[c[0]]=false; //若字符正被发射，则结束其发射

}

else {

isEmitting[c[0]]=true;

emitter=(CharEmitter)emitters.fetch(); //向池中索取一个CharEmitter对象

emitter.emit((char)c[0]); //发射用户输入的字符

}

}

}

}

执行后，从键盘上敲进0到9之间的任意数字并按回车，之后会不断地在屏幕上滚动显示该数字；再次输入同样的数字则不再显示该数字。同时存在多个数字被发射时，可以明显看出不同数字的显示是交错进行的，这正是由于虚拟机在各线程间调度的结果。运行结果表明，我们设计的类功能完全正确。

在以后要说的J2ME中蓝牙通讯的辅助类中，将会看到，线程池与可重启线程起到了不可替代的作用。