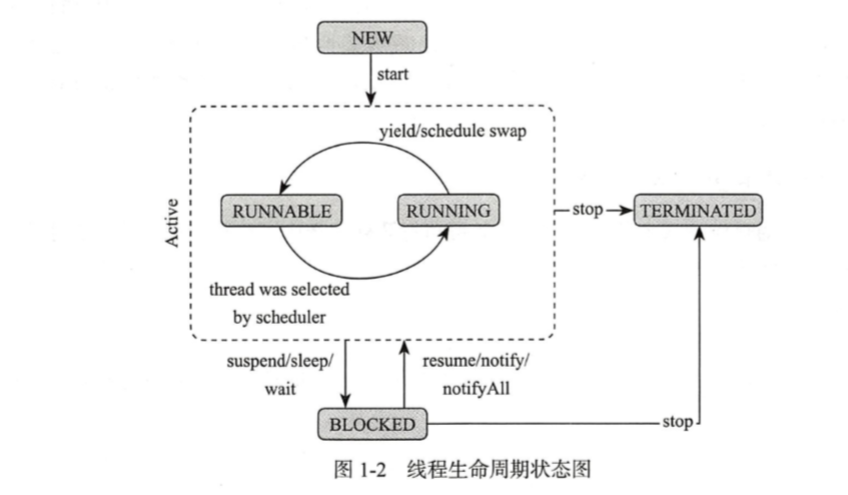
# 有关线程

线程的生命周期详解



线程的生命周期体可以分为如下5个主要的阶段

* NEW
* RUNNABLE
* RUNNING
* BLOCKED
* TERMINATED

**线程的NEW状态**

当我们用关键字new创建了一个Thread对象时，此时它并不处于执行状态，因为没有调用start方法启动该线程，那么线程的状态为NEW状态，准确的说，它只是Thread对象的状态，因为在没有start之前，该现线程根本不存在，与你用关键字new创建一个普通的java对象没什么区别。

New状态通过start方法进入RUNNABLE状态

**线程的RUNNABLE状态**

线程对象进入RUNNABLE状态必须调用start方法，此时才是真正地在JVM进程中创建了一个线程，线程一经启动是不会立即就得到执行的，线程的运行与否和进程一样都要听命于CPU的调度，所以这个中间状态称之为可执行状态，也就是RUNNABLE状态，也就是说它具备执行的资格，但是并没有真正的执行起来，而是在等待CPU的调度

由于存在Running状态，所以不会直接进入BLOCKED状态和TERMINATED状态，即使是在线程的执行逻辑中调用wait，sleep或者其他block的io操作等，也必须先获得CPU的调度执行权才可以，严格来讲，RUNNABLE的线程只能意外终止或者进入RUNNING状态

**线程的BLOCKED状态**

线程在BLOCKED状态中可以切换为如下几个状态：

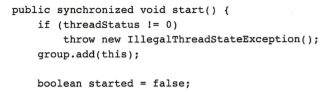
* 直接进入TERMINATED状态，比如调用JDK已经不推荐使用的stop方法或者意外死亡（JVM Crash）
* 线程阻塞的操作结束，比如读取了想要的数据字节进入到RUNNABLE状态
* 线程完成了指定时间的休眠，进入到了RUNNABLE状态
* Wait中的线程被其他线程notify/notifyall唤醒，进入RUNNABLE状态
* 线程获取到了某个锁资源，进入RUNNABLE状态
* 线程在阻塞过程中被打断，比如其他线程调用了interrupt方法，进入RUNNABLE状态

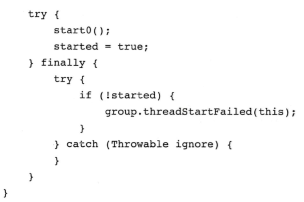
**线程的TERMINATED状态**

TERMINATED是一个线程的最终状态，在该状态中线程将不会切换到其他任何状态，线程会进入TERMINATED状态，意味着该线程的整个生命周期都结束了，下列这些情况将会使线程进入TERMINATED状态

* 线程运行正常结束，结束生命周期
* 线程运行出错意外结束
* JVM Crash，导致所有的线程都结束

Thread start方法源码：





、

其中最核心的部分数start0这个本地方法，也就是JNI方法



Start方法中会调用start0方法，而run方法是start0调用的

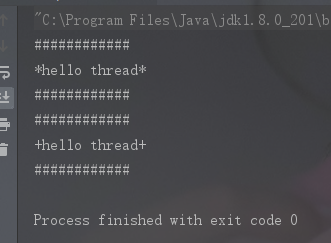
总结出以下几点：

* Thread被构造后的NEW状态，事实上threadStatus这个内部属性为0
* 不能两次启动Thread，否则就会出现IllegalThreadStateException异常
* 线程启动后将会被加入到一个ThreadGroup中
* 一个线程生命周期结束，也就是到了TERMINATED状态，再次调用start方法是不允许的，也就是说TERMINATED状态是没有办法回到RUNNABLE/RUNNING状态的

模板设计模式在Thread中的应用

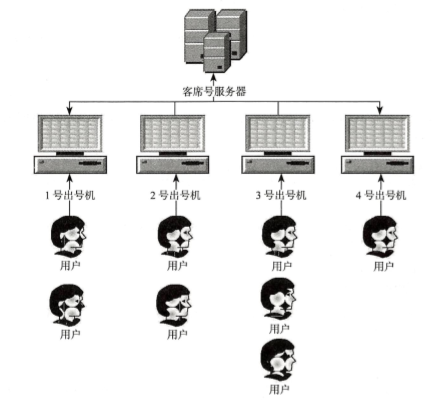
package com.winkilee.Thread;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/4/27 14:46  
 \** ***@Description*** *模板设计模式在Thread中的应用  
 \*/*public class TemplateMethod {  
  
 public final void print(String message){  
 System.*out*.println("############");  
 warpPrint(message);  
 System.*out*.println("############");  
 }  
  
 protected void warpPrint(String message){  
 }  
  
 public static void main(String[] args) {  
 TemplateMethod t1=new TemplateMethod(){  
 @Override  
 protected void warpPrint(String message){  
 System.*out*.println("\*"+message+"\*");  
 }  
 };  
 t1.print("hello thread");  
  
 TemplateMethod t2=new TemplateMethod(){  
 @Override  
 protected void warpPrint(String message){  
 System.*out*.println("+"+message+"+");  
 }  
 };  
 t2.print("hello thread");  
 }  
}

运行结果



Print方法类似于Thread的start方法，而warpPrint则类似于run方法，这样做的好处是，程序结构由父类控制，并且是final修饰的，不允许被重写，子类只需要实现想要的逻辑任务即可。

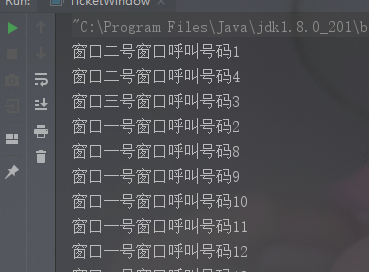
**Thread模拟营业大厅叫号机程序**



假设大厅共有四台出号机，就意味着四个线程在工作，约定当天最多受理50笔业务，也就是号码最多可以出到50

package com.winkilee.Thread;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/4/27 15:03  
 \** ***@Description*** *多线程模拟营业厅叫号机  
 \*/*public class TicketWindow extends Thread{  
  
 private final String name;  
  
 private static final int *MAX*=50;  
  
 private static int *index*=1;  
  
 public TicketWindow(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public void run(){  
 while (*index*<=*MAX*){  
 System.*out*.println("窗口"+name+"呼叫号码"+(*index*++));  
 }  
 }  
  
  
 public static void main(String[] args) {  
 TicketWindow ticketWindow1=new TicketWindow("一号窗口");  
 ticketWindow1.start();  
  
 TicketWindow ticketWindow2=new TicketWindow("二号窗口");  
 ticketWindow2.start();  
  
 TicketWindow ticketWindow3=new TicketWindow("三号窗口");  
 ticketWindow3.start();  
  
 TicketWindow ticketWindow4=new TicketWindow("四号窗口");  
 ticketWindow4.start();  
 }  
}

运行结果如下：

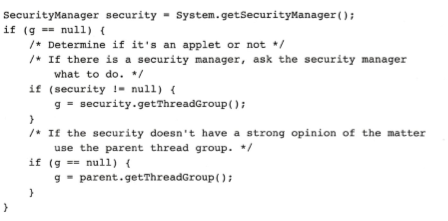


**线程与线程组**

**Thread与ThreadGroup**

在Thread的构造函数中，可以显示地指定线程的Group，也就是ThreadGroup

Thread init方法源码

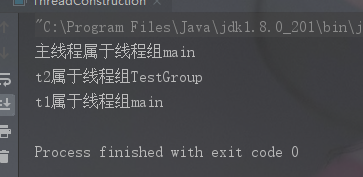


可以看出，如果在构造Thread的时候没有显示地指定一个ThreadGroup，那么子线程将会被加入到父线程所在的线程组

Demo：

package com.winkilee.test;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/4/27 15:33  
 \** ***@Description*** *线程组测试用例  
 \*/*public class ThreadConstruction {  
  
 public static void main(String[] args) {  
 Thread t1=new Thread("t1");  
  
 ThreadGroup group=new ThreadGroup("TestGroup");  
  
 Thread t2=new Thread(group,"t2");  
  
 ThreadGroup mainThreadGroup=Thread.*currentThread*().getThreadGroup();  
  
 System.*out*.println("主线程属于线程组"+mainThreadGroup.getName());  
 System.*out*.println("t2属于线程组"+t2.getThreadGroup().getName());  
 System.*out*.println("t1属于线程组"+t1.getThreadGroup().getName());  
  
 }  
}

运行结果

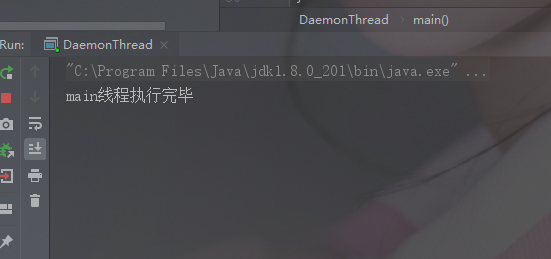


**有关守护线程**

守护线程demo

package com.winkilee.demo;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/4/27 15:57  
 \** ***@Description*** *守护线程demo  
 \*/*public class DaemonThread {  
  
 public static void main(String[] args) throws InterruptedException {  
 //main线程开始  
  
 Thread thread = new Thread(() -> {  
 while (true) {  
 try {  
 Thread.sleep(1);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 //thread.setDaemon(true);  
  
 thread.start();  
 Thread.*sleep*(2\_000L);  
 System.*out*.println("main线程执行完毕");  
  
 }  
}

运行结果



发现程序并没有退出执行

即使main线程正常的结束了自己的生命周期，原因就是因为JVM进程里还存在一个非守护线程在运行

如果打开注释//thread.setDaemon(true);  
也就是将thread设置为了守护线程，那么main线程在结束生命周期后，JVM也会随之退出，当然thread线程也会结束

设置守护线程setDaemon方法只在线程启动之前才能生效，如果一个线程已经死亡，那么设置setDaemon就会抛出IllegalThreadStateException异常

**守护线程使用场景**

守护线程的典型代表是垃圾回收，这是很多人说守护进程非常有用的理由，但实际上守护进程在用户开发上的应用场景几乎用处不大，可能的应用场景：

1. 内存资源或者线程的管理，但是非守护线程也可以做

2. 守护线程负责一个可以将当前的JVM退出的功能，即将非damon的线程都退出，然后jvm自动退出，感觉用的也非常少，可以直接通知相关线程退出不就可以了，考虑设计上优雅一些，可能有点好处。