**细说多线程之Thread VS Runnable**

1.继承 Thread 类

MyThread my = new MyThread();

my.start();

2.实现 Runable 接口:（代码多出一行）

MyThread my = new MyThread();

Thread thread = new Thread(my);

thread.start();

启动线程调用start()方法[注:并没有开始执行]，之后线程进入线程队列中，等待cpu服务，

一旦获取到时间片立即进入run()方法开始执行，run()方法逻辑完成，本线程结束





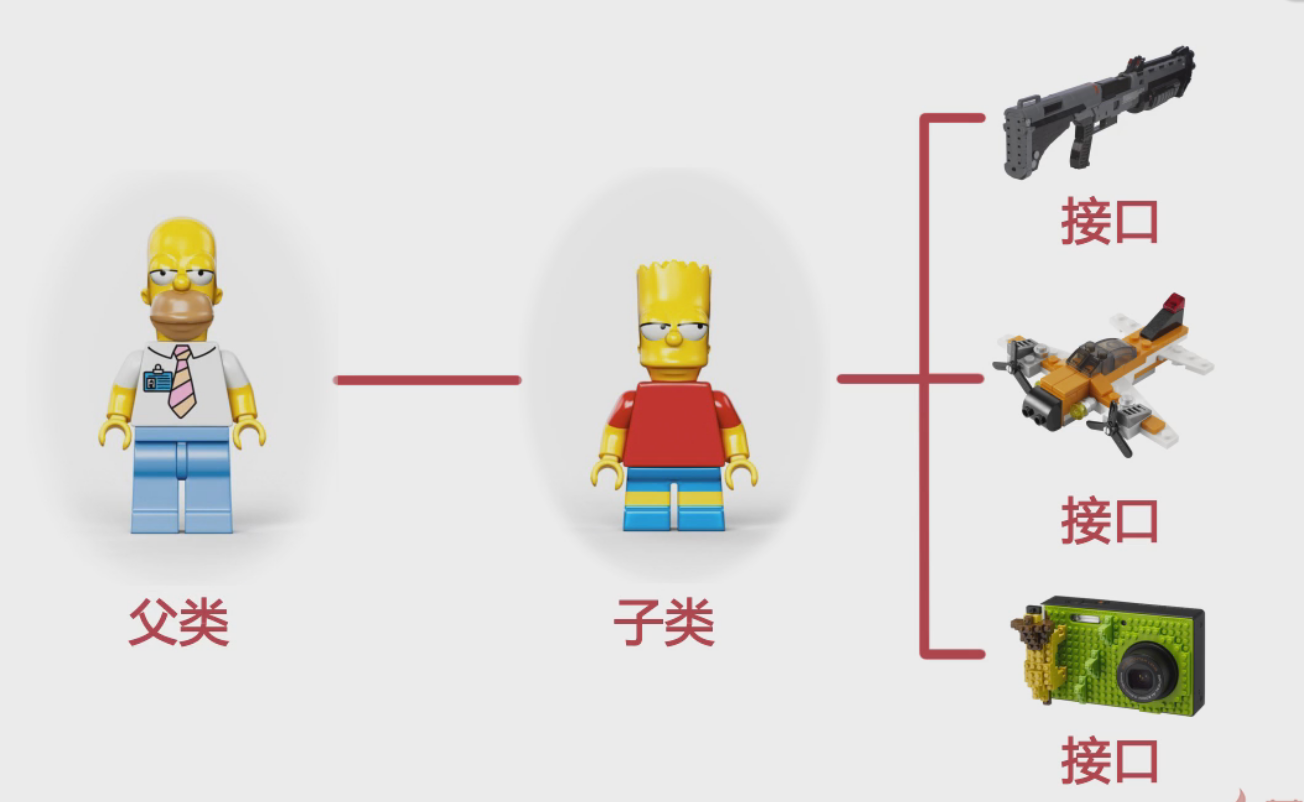


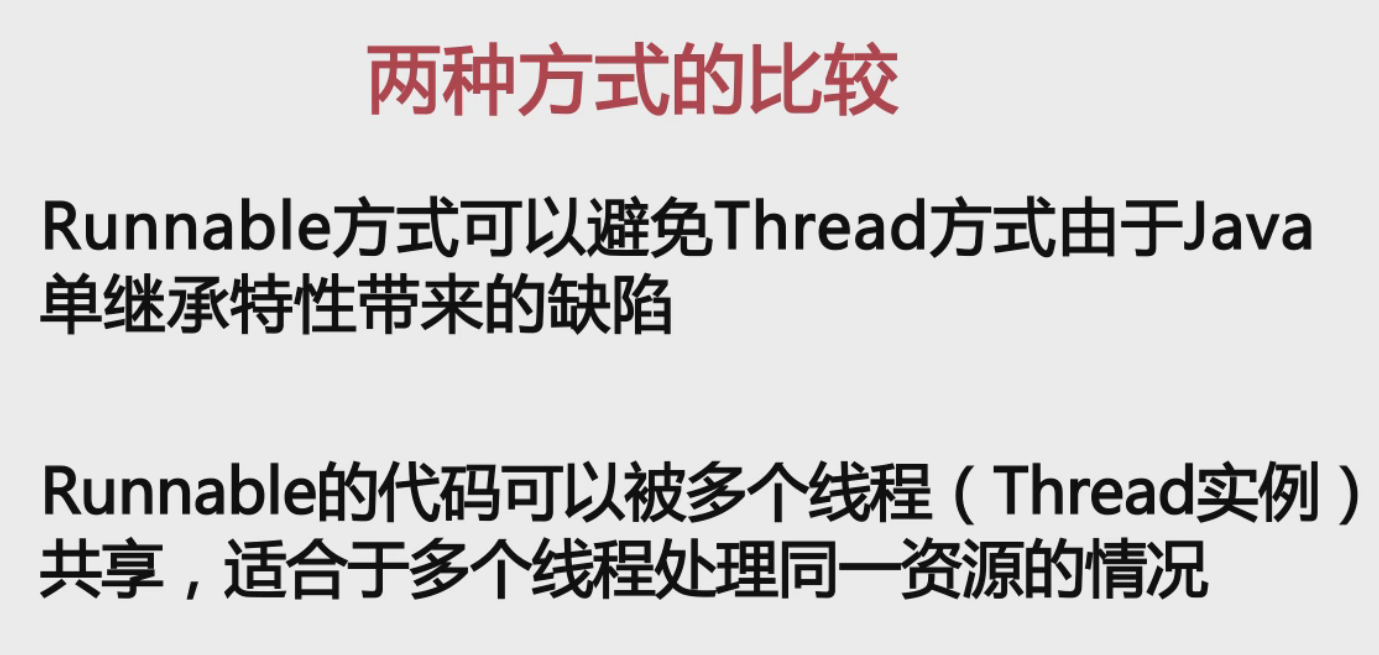
在Java中一个子类可以实现多个接口，但只能继承一个父类。单继承

两种创建线程方式的简单比较：<br>

1：在Java世界中一个类可以同时实现多个接口但是只能集成一个类，这就是单继承

2：使用继承Thread类的方式来创建线程，不适合多个线程处理同一资源的情况，这种情况对于使用实现Runnable接口的方式创建的线程是适合的。





**package** com.createThread;

/\*创建线程的两种方式：

\* 1.继承Thread类

\* 2.实现Runnable接口

\* \*/

//定义一个线程类

**class** MyThread **extends** Thread{

**private** **int** ticketsCount = 5; //一共有5张火车票

**private** String name; //窗口，就是线程的名字

**public** MyThread(String name){

**this**.name = name;

}

//创建启动线程的方法

@Override

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

//如果ticketsCount不为0，说明还有火车票，可以继续卖

**while**(ticketsCount > 0){

ticketsCount--; //如果还有票，就卖掉一张

System.***out***.println(name +"卖了1张票，剩余票数为"+ticketsCount);

}

}

}

**public** **class** TicketsThread {

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

//创建3个线程，模拟三个窗口卖票

MyThread mt1 = **new** MyThread("窗口1");

MyThread mt2 = **new** MyThread("窗口2");

MyThread mt3 = **new** MyThread("窗口3");

//启动这三个线程，就是窗口开始卖票

mt1.start();

mt2.start();

mt3.start();

}

/\*

\* 总共只有5张火车票，每个窗口卖了5次，结果卖出15张票，这是不合理的

\*

\* \*/

}

**package** com.createThread;

/\*创建线程的两种方式：

\* 1.继承Thread类

\* 2.实现Runnable接口

\* \*/

//创建一个线程类

**class** MyThread1 **implements** Runnable{

**private** **int** ticketsCount = 5; //一共有5张火车票

@Override

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

//如果ticketsCount不为0，说明还有火车票，可以继续卖

**while**(ticketsCount > 0 ){

ticketsCount--;

//Thread.currentThread().getName() 获得当前线程的名字

System.***out***.println(Thread.*currentThread*().getName() +"卖了1张票，剩余票数为"+ticketsCount);

}

}

}

**public** **class** TicketsThread1 {

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

MyThread1 mt = **new** MyThread1();

//创建三个线程,来模拟三个售票窗口

Thread th1 = **new** Thread(mt,"窗口1");

Thread th2 = **new** Thread(mt,"窗口2");

Thread th3 = **new** Thread(mt,"窗口3");

//启动三个线程，就是三个窗口开始卖票

th1.start();

th2.start();

th3.start();

}

}

线程的生命周期

创建 -> 就绪 -> 运行 -> (阻塞->解除阻塞->就绪->运行) ->消亡

创建状态:新建了一个线程对象,如Thread t = new Thread();

就绪状态:调用线程的start()方法启动线程后,线程就进入了就绪状态,此时线程只是进入了线程队列,等待获取cpu资源,具备了运行的条件,但并不一定已经开始运行了,因为当前cpu可能在执行别的线程.

运行状态:当线程获取了cpu资源时就进入了运行状态,执行线程的run方法.

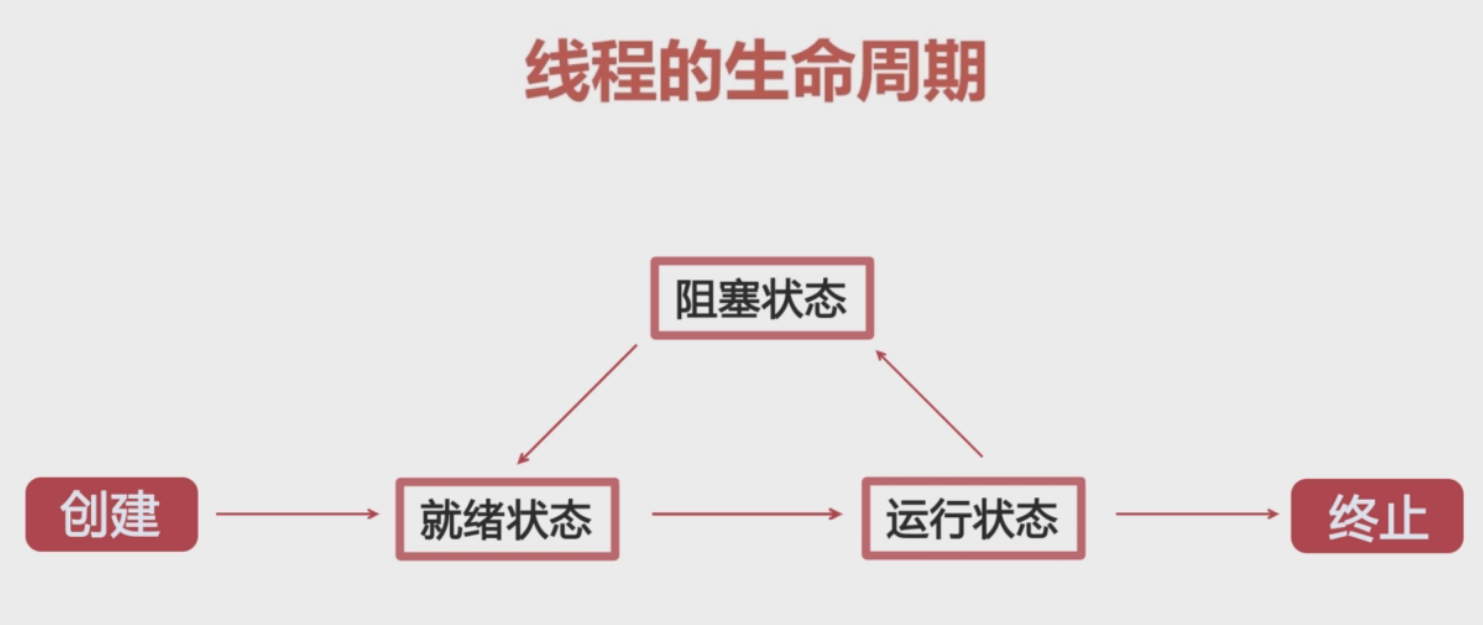
消亡:一旦线程的run方法执行完毕后线程将会消亡.

阻塞状态:当线程在运行状态时,可能会遇到一些阻塞事件,此时当前线程会让出cpu资源,暂停了自己的执行,并且进入阻塞状态.

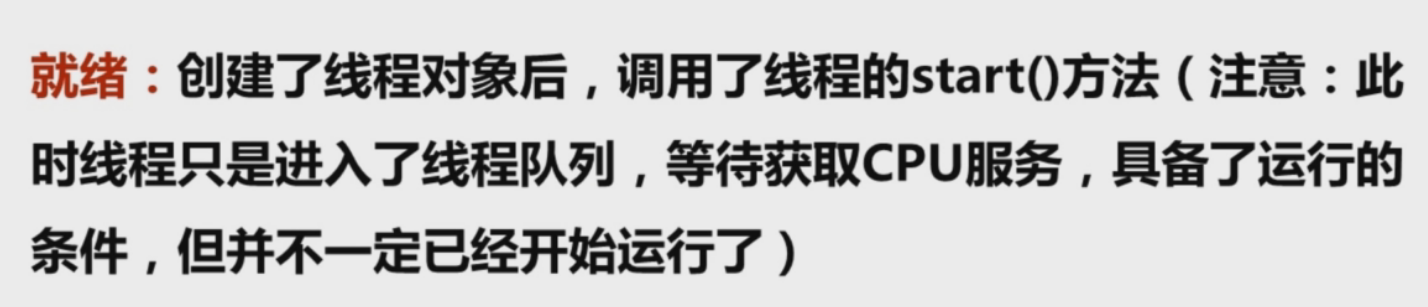
当阻塞解除时便会重新进入就绪状态,继续进入线程队列等待获取cpu资源.当重新获取了cpu资源后,会从上次已执行的run方法代码后继续执行.

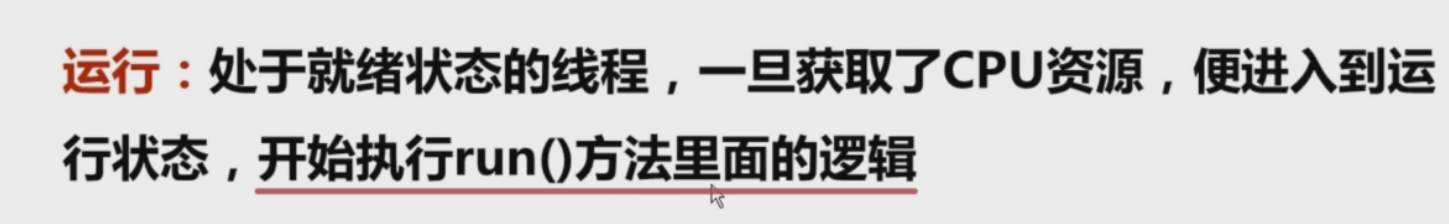
阻塞事件:线程的sleep()方法、线程的join()方法等.

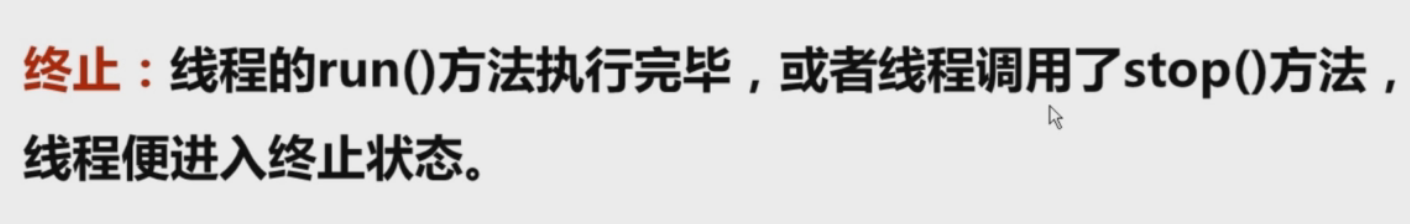
大多数情况下,线程是因为执行完run方法就会进入终止状态,除非人为调用了stop方法.

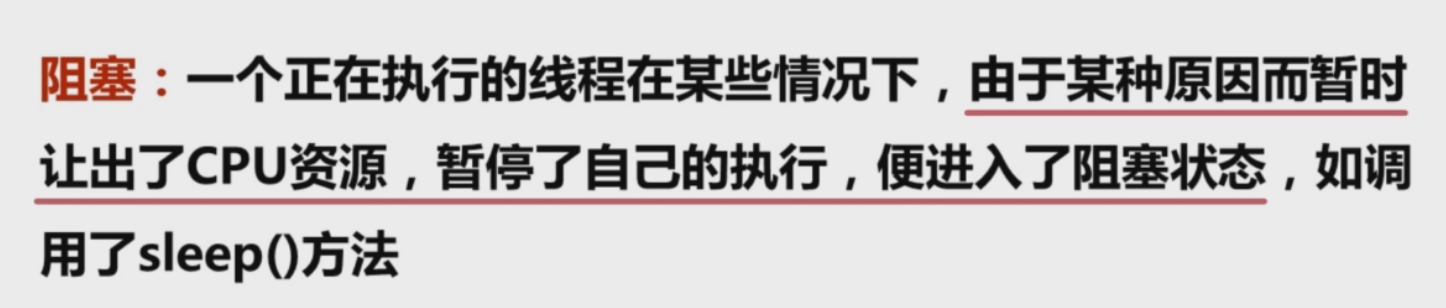


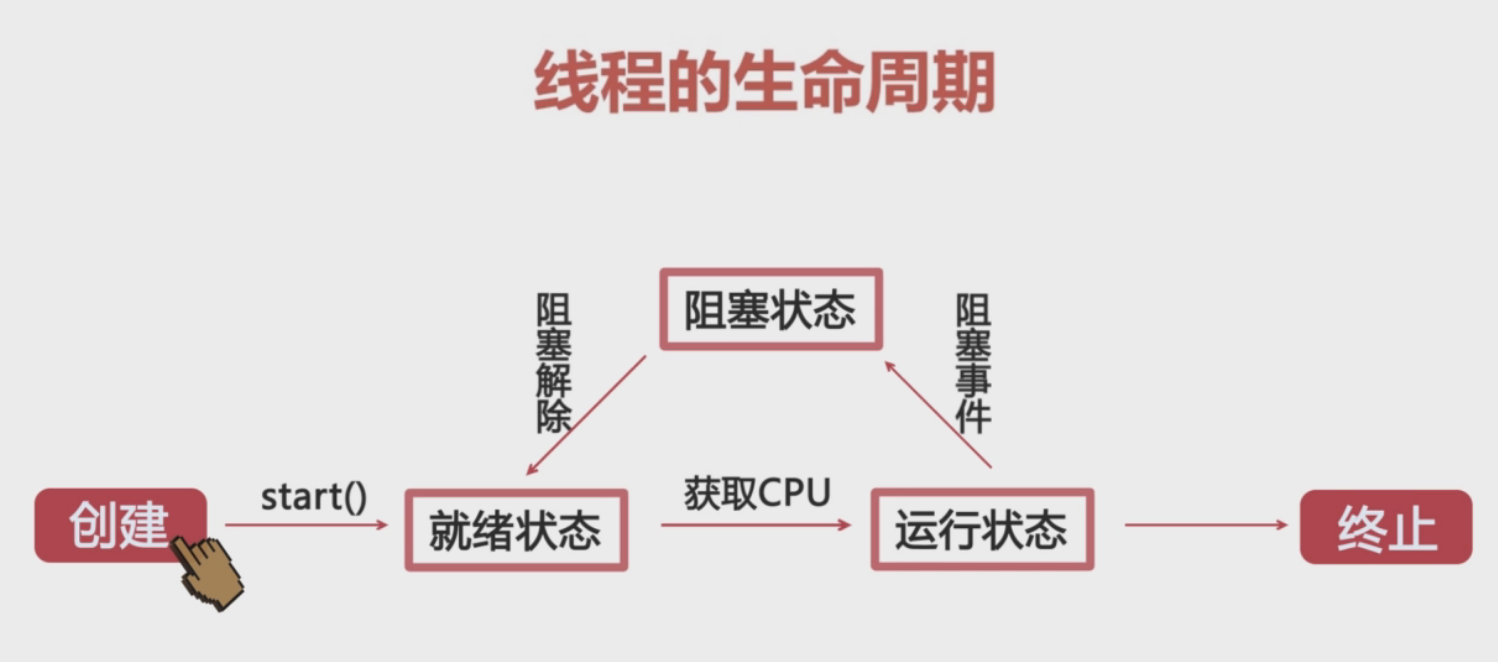












线程分为两类：

1、用户线程：运行在前台，执行具体任务

如程序的主线程、连接网络的子线程等都是用户线程

2、守护线程：运行在后台，为其他前台线程服务(也就是某些线程的服务者)

特点：一旦所有的用户线程都结束运行(没有了守护的对象)，守护线程会随JVM一起结束工作

如：数据库连接池中的监测线程（监测连接个数、监测连接超时时间）、

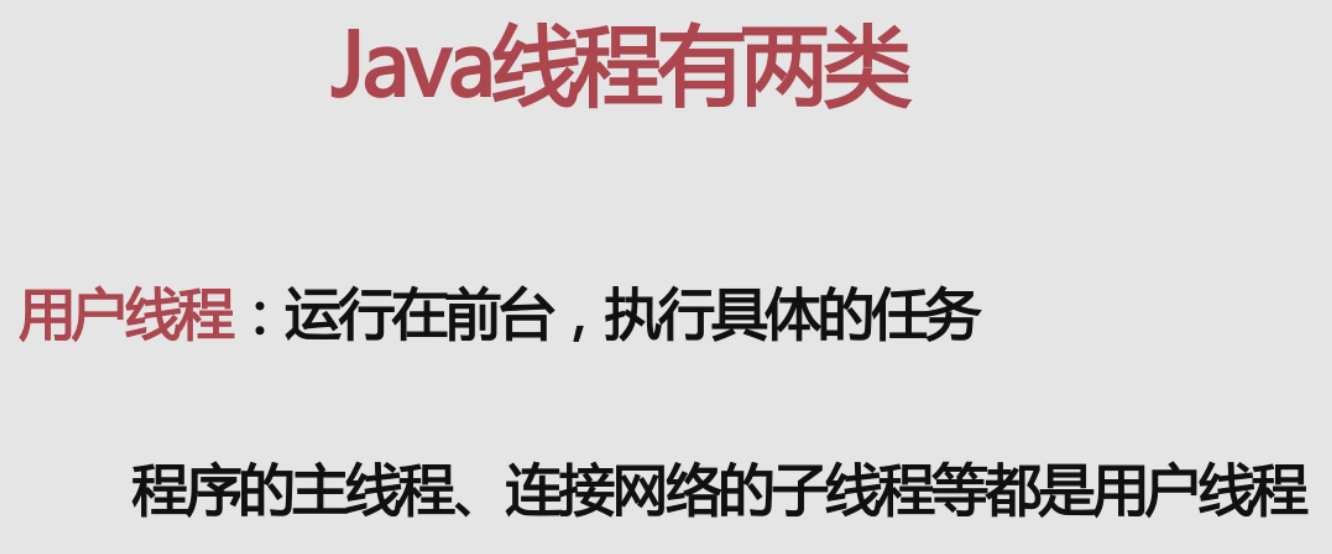
JVM虚拟机启动之后的监测线程

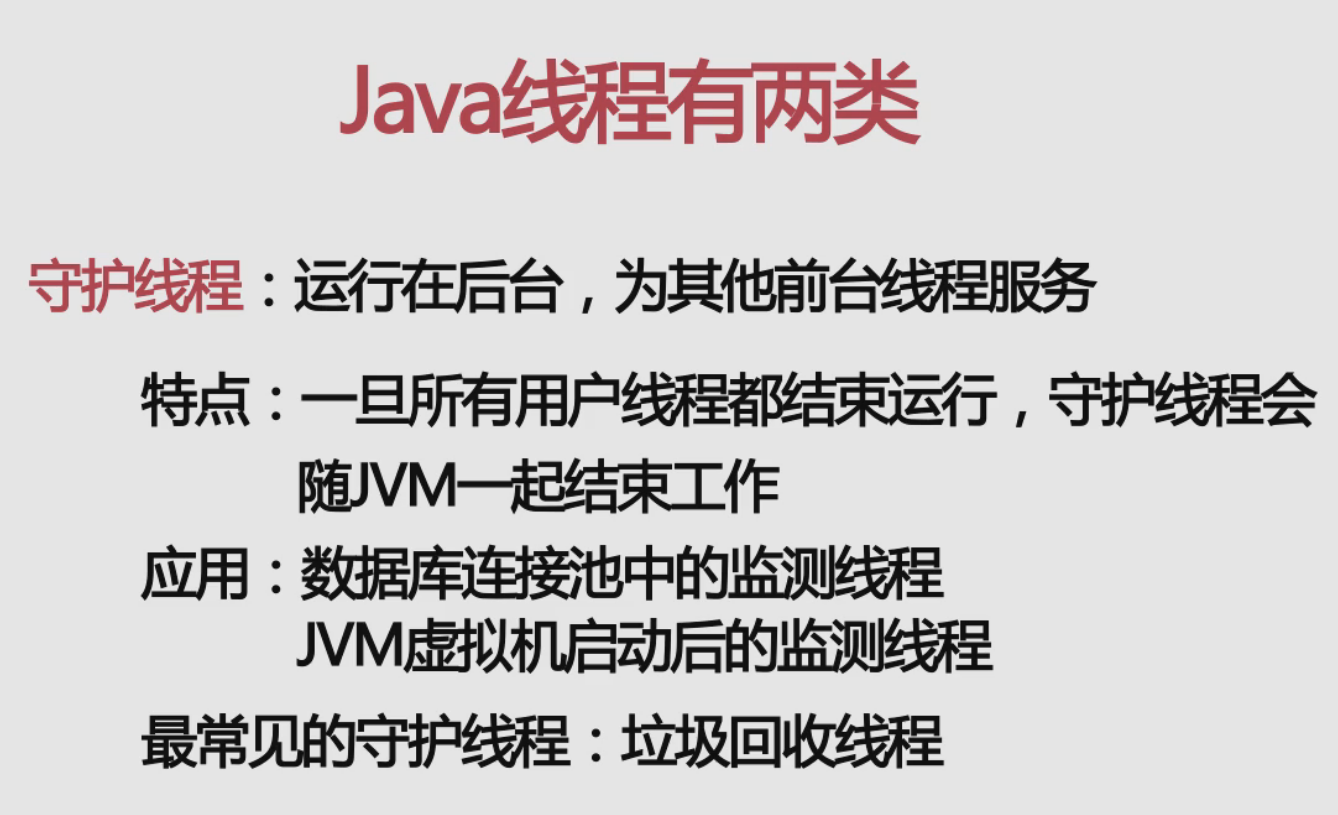
最常见的还是垃圾回收线程

注：1.daemon(英文守护的意思)设置是否是守护线程，必须在start方法前调用

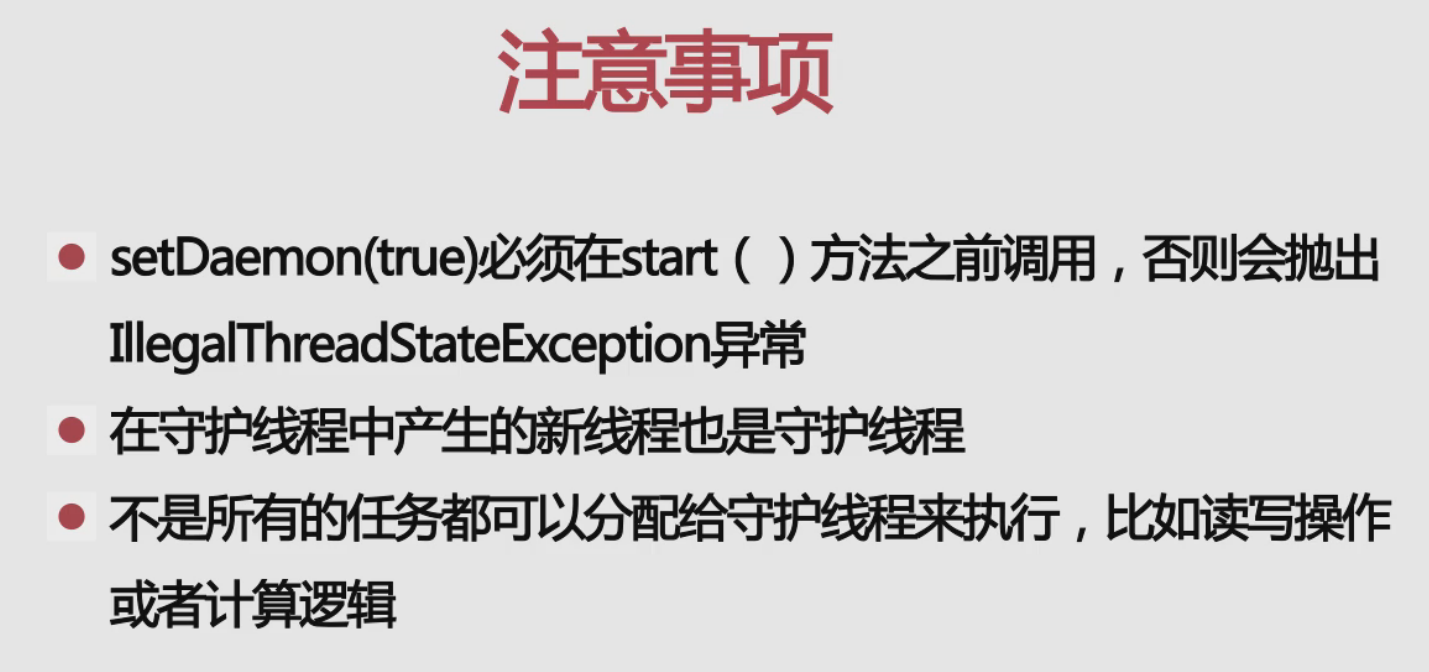
2.守护线程汇总产生的线程也是守护线程

3.不是所有的任务都能设置成守护线程，如读写操作、计算逻辑（也就是和实际业务逻辑相关的线程不能设置成守护线程）









**package** com.demo.daemon;

**import** java.io.File;

**import** java.io.FileOutputStream;

**import** java.io.OutputStream;

**import** java.util.Scanner;

/\*线程分为两类：

\* 1.用户线程

\* 2.守护线程

\*

\* \*/

//定义一个线程类

**class** DaemonThread **implements** Runnable{

@Override

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

System.***out***.println("进入守护线程"+Thread.*currentThread*().getName());

**try** {

writeToFile();

} **catch** (Exception e) {

e.printStackTrace();

}

System.***out***.println("退出守护线程"+Thread.*currentThread*().getName());

}

//该方法实现向文件中写数据

**private** **void** writeToFile() **throws** Exception{

//创建文件对象

File filename = **new** File("E:"+File.***separator***+"test.txt");

//向文件中写数据

OutputStream os = **new** FileOutputStream(filename,**true**);

**int** count = 0; //每写一次数据，count就++

**while**(count < 999){

os.write(("\r\nJava Thread"+count).getBytes());

System.***out***.println("守护线程"+ Thread.*currentThread*().getName()+"向文件中写入了Java Thread"+count++);

Thread.*sleep*(10000); //休眠1秒 每隔一秒写一次数据

}

}

}

**public** **class** DaemonThreadDemo {

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

System.***out***.println("程序进入了主线程"+Thread.*currentThread*().getName());

DaemonThread daemonThread = **new** DaemonThread();

Thread thread = **new** Thread(daemonThread);

thread.setDaemon(**true**); //设置为守护线程

thread.start(); //守护线程启动

//阻塞线程

Scanner sc = **new** Scanner(System.***in***);

sc.next();

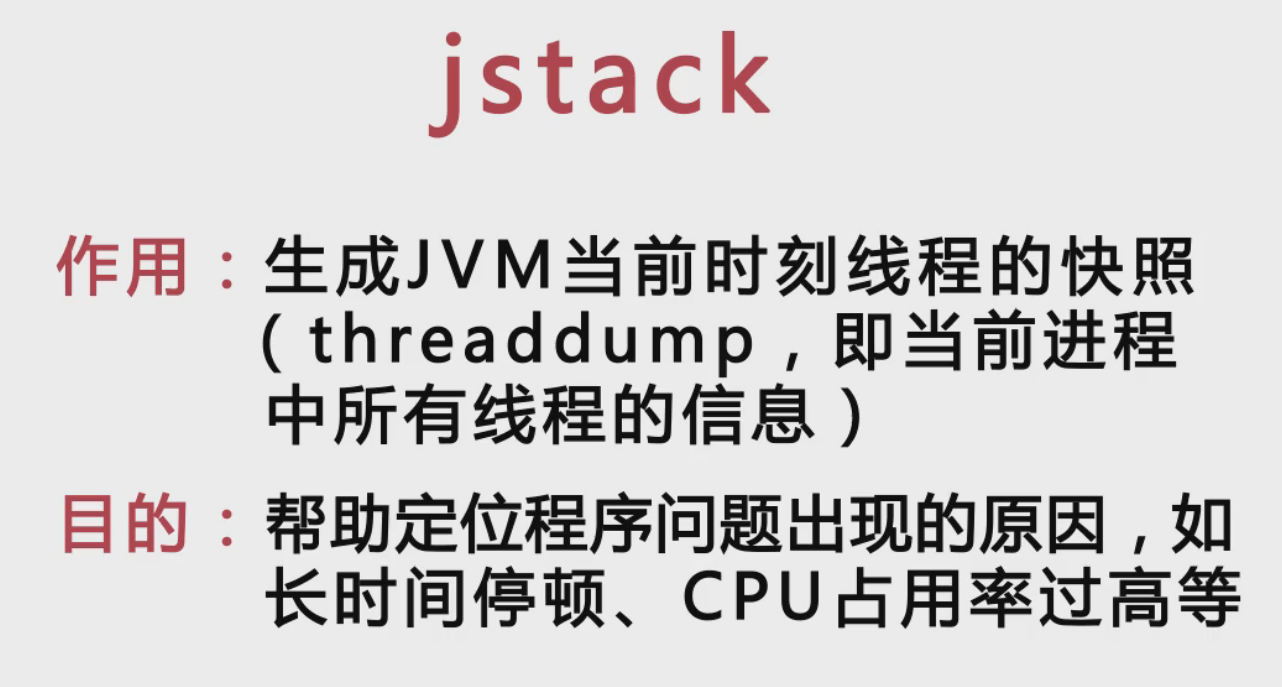
System.***out***.println("程序退出了主线程"+Thread.*currentThread*().getName());

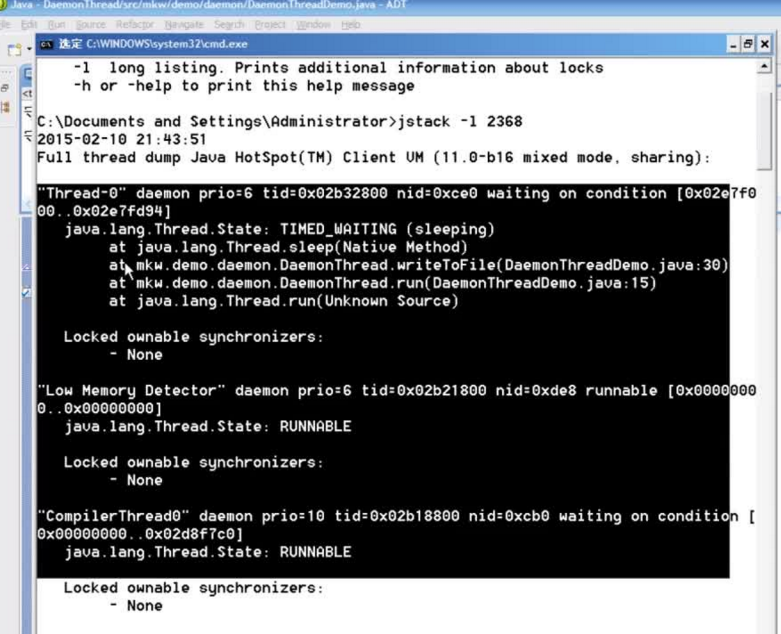
}

}



这两个工具在JDK安装目录下的bin中。



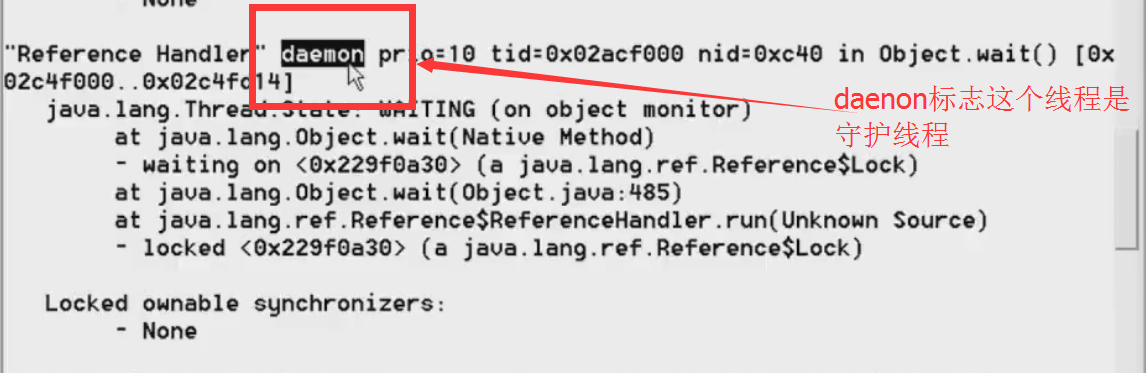


java.lang.Thread.State 线程的运行状态

Locked ownable synchronizers 是否线程同步

Low Memory Detector 为守护线程，自己启动，检测内存是否充足。

CompilerThread0 为守护线程，检测编译





程序中的同一资源指的是同一个Runnable对象

安全的买票程序中需要加入同步（synchronized）

