多线程

优逸客 让天下没有难学的技术

官网：http://www.sxuek.com/

# 第一章 线程概述：

1）程序(program)，是为完成特定任务、用某种语言编写的一组指令的集合。即指一段静态的代码，静态对象。

2）进程(process)，是程序的一次执行过程，或是正在运行的一个程序。动态过程：有它自身的产生、存在和消亡的过程。

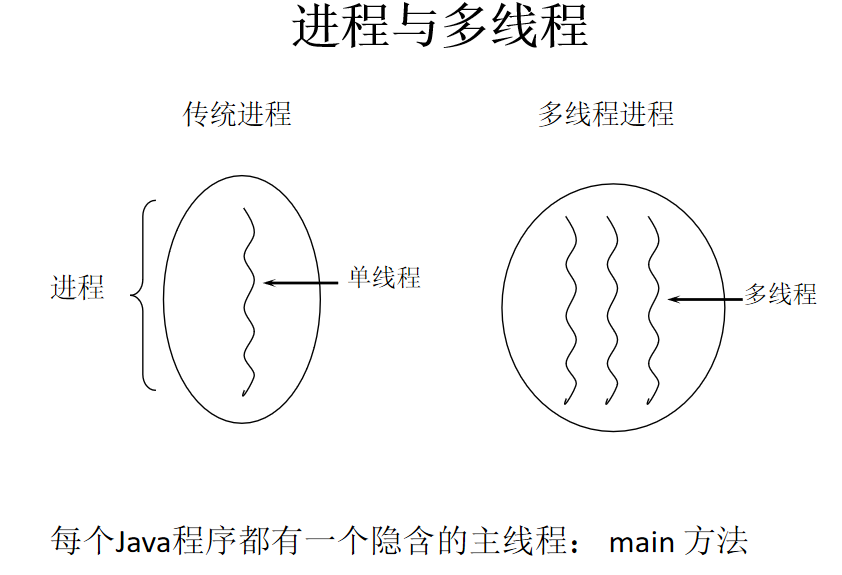
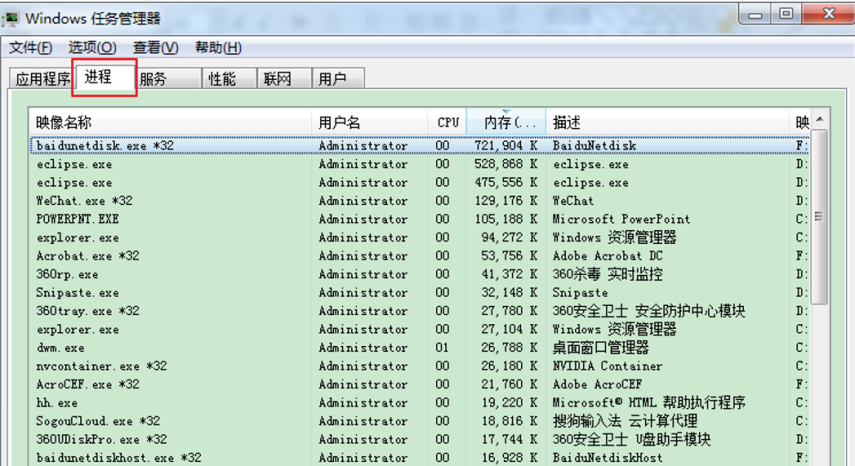
如：运行中的QQ，运行中的MP3播放器；

程序是静态的，进程是动态的；

进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域。

3）线程(thread)，进程可进一步细化为线程，是一个程序内部的一条执行路径。若一个程序可同一时间执行多个线程，就是支持多线程的。线程作为调度和执行的单位，每个线程拥有独立的运行栈和程序计数器(pc)，线程切换的开销小。

一个进程中的多个线程共享相同的内存单元/内存地址空间它们从同一堆中分配对象，可以访问相同的变量和对象。这就使得线程间通信更简便、高效。但多个线程操作共享的系统资源可能就会带来安全的隐患。



单核CPU和多核CPU的理解：

单核CPU，其实是一种假的多线程，因为在一个时间单元内，也只能执行一个线程的任务。例如：虽然有多车道，但是收费站只有一个工作人员在收费，只有收了费才能通过，那么CPU就好比收费人员。如果有某个人不想交钱，那么收费人员可以把他“挂起”（晾着他，等他想通了，准备好了钱，再去收费）。但是因为CPU时间单元特别短，因此感觉不出来。

如果是多核的话，才能更好的发挥多线程的效率。（现在的服务器都是多核的）

一个Java应用程序java.exe，其实至少有三个线程：main()主线程，gc() 垃圾回收线程，异常处理线程。当然如果发生异常，会影响主线程。

并行与并发：

并行：多个CPU同时执行多个任务。比如：多个人同时做不同的事。

并发：一个CPU(采用时间片)同时执行多个任务。比如：秒杀、多个人做同一件事。

框架：pc端ssm、微服务springboot、spingclid

使用多线程的优点：

背景：只使用单个线程完成多个任务（调用多个方法），肯定比用多个线程来完成用的时间更短，为何仍需多线程呢？

多线程程序的优点：

1）提高应用程序的响应。对图形化界面更有意义，可增强用户体验；

2）提高计算机系统CPU的利用率；

3）改善程序结构。将既长又复杂的进程分为多个线程，独立运行，利于理解和修改；

何时需要多线程：

1）程序需要同时执行两个或多个任务。

2）程序需要实现一些需要等待的任务时，如用户输入、文件读写操作、网络操作、搜索等。

3）需要一些后台运行的程序时。

# 第二章 线程的创建和启动

|  |
| --- |
| *public class Test{*  *public void method1(String str) {*  *System.out.println(str);*  *}*  *public void method2(String str) {*  *method1(str);*  *}*  *public static void main(String[] args) {*  *Test test = new Test();*  *test.method2("hello World!");*  *}*  *}* |

查看JavaAPI帮助文档→Thread类。

Java语言的JVM允许程序运行多个线程，它通过*java.lang.Thread*类来实现。

Thread类的特性：

每个线程都是通过某个特定Thread对象的run()方法来完成操作的，经常把run()方法的主体称为线程体，通过该Thread对象的start()方法来调用这个线程。

Thread类构造器：

|  |  |
| --- | --- |
| *Thread()* | *创建新的Thread对象* |
| *Thread(String threadname)* | *创建线程并指定线程实例名* |
| *Thread(Runnable target)* | *指定创建线程的目标对象，它实现了Runnable接口中的run* |
| *Thread(Runnable target, String name)* | *创建新的Thread对象* |

创建线程的两种方式：

## 2.1 继承Thread类

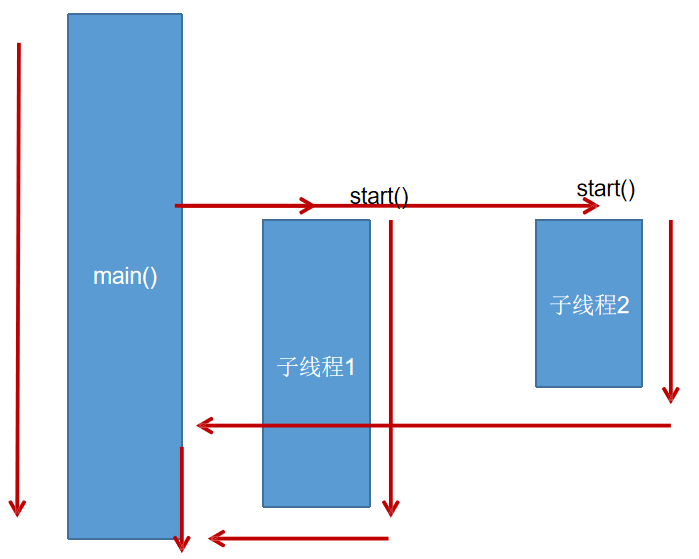
步骤：

1) 定义子类继承Thread类；

2) 子类中重写Thread类中的run方法；

3) 创建Thread子类对象，即创建了线程对象；

4) 调用线程对象start方法：启动线程、调用run方法；



案例：

|  |
| --- |
| *class MyThread extends Thread{*  *public MyThread() {}*  *@Override*  *public void run() {*  *for (int i = 0; i < 100; i++) {*  *System.out.println("子线程：" + i);*  *}*  *}*  *}*  *public class Test{*  *public static void main(String[] args) {*  *MyThread myThread = new MyThread();*  *myThread.start();*  *//myThread.run(); //不是新启动了一个线程，而是方法调用*  *// 不可以让已经start()的线程去执行，会报IllegalThreadStateException*  *myThread.start();*  *//再启动一个线程：*  *MyThread myThread1 = new MyThread();*  *myThread1.start();*  *for (int i = 0; i < 100; i++) {*  *System.out.println("主线程：" + i);*  *}*  *}*  *}*  */\*\**  *\* 需求：创建两个分线程，其中一个线程遍历100以内的偶数，另一个线程遍历100以内的奇数*  *\*/*  *class MyThread1 extends Thread{*  *public MyThread1() {}*  *@Override*  *public void run() {*  *for (int i = 0; i < 100; i++) {*  *if (i % 2 == 0) {*  *System.out.println(Thread.currentThread().getName() +" - "+ i);*  *}*  *}*  *}*  *}*  *class MyThread2 extends Thread{*  *public MyThread2() {}*  *@Override*  *public void run() {*  *for (int i = 0; i < 100; i++) {*  *if (i % 2 != 0) {*  *System.out.println(Thread.currentThread().getName() + " - "+ i);*  *}*  *}*  *}*  *}*  *public class Test{*  *public static void main(String[] args) {*  *//第一种：常规写法*  *//MyThread1 myThread1 = new MyThread1();*  *//MyThread2 myThread2 = new MyThread2();*  *//myThread1.start();*  *//myThread2.start();*  *//第二种：匿名子类的方式*  *new Thread() {*  *@Override*  *public void run() {*  *for (int i = 0; i < 100; i++) {*  *if (i % 2 == 0) {*  *System.out.println(Thread.currentThread().getName() +" - "+ i);*  *}*  *}*  *}*  *}.start();*  *new Thread() {*  *public void run() {*  *for (int i = 0; i < 100; i++) {*  *if (i % 2 != 0) {*  *System.out.println(Thread.currentThread().getName() +" - "+ i);*  *}*  *}*  *};*  *}.start();*  *}*  *}* |

### 2.1.1 Thread类方法

|  |  |
| --- | --- |
| 方法声明 | 功能 |
| *void start()* | 启动线程，并执行对象的run()方法 |
| *run()* | 线程在被调度时执行的操作 |
| *String getName()* | 返回线程的名称 |
| *void setName(String name)* | 设置该线程名称 |
| *static Thread currentThread()* | 返回当前线程。在Thread子类中就是this，通常用于主线程和Runnable实现类 |
| *Static void yield()* | 线程让步；暂停当前正在执行的线程，把执行机会让给优先级相同或更高的线程；若队列中没有同优先级的线程忽略此方法； |
| *join()* | 当某个程序执行流中调用其他线程的join()方法时，调用线程将被阻塞，直到 join() 方法加入的 join 线程执行完为止，低优先级的线程也可以获得执行； |
| *Static void sleep(long millis)* | (指定时间:毫秒)  令当前活动线程在指定时间段内放弃对CPU控制,使其他线程有机会被执行,时间到后 重排队。  抛出InterruptedException异常 |
| *stop()* | 强制线程生命期结束，不推荐使用 |
| *boolean isAlive()* | 返回boolean，判断线程是否还活着 |

1）设置线程名称，获取线程名称：

|  |
| --- |
| */\*\**  *\* 此类用于演示线程的常见方法一：设置线程名称，获取线程名称*  *\* setName*  *\* getName*  *\* currentThread:获取当前线程对象，是一个静态方法，直接用类名调用即可*  *\* 注意：新线程没有设置名称，也有默认名。*  *\* Thread-0*  *\* Thread-1*  *\* ...*  *\*/*  *public class TestThreadMethod1 {*  *public static void main(String[] args) {*  *A a = new A("小黄");//方式一*  *// a.setName("小黄");//方式二*  *a.start();*  *// a.run();*  *}*  *}*  *class A extends Thread{*  *public A(String name){*  *super(name); //调用父类构造器设置名称*  *}*  *@Override*  *public void run() {*  *for(int i=1;i<=100;i++){*  *System.out.println(getName()+"----"+i);*  *}*  *}*  *}* |

2）Sleep、interrupt：

|  |
| --- |
| */\*\**  *\* 此类用于演示线程的常见方法二：*  *\* sleep:让当前线程休眠指定毫秒数，参数单位：ms,是一个静态方法，可以直接通过类名调用。一般来讲，用于模拟线程交错效果*  *\* interrupt：中断线程的休眠、等待状态，如果中断的线程正在休眠，则会抛InterruptedException*  *\*/*  *public class TestThreadMethod2 {*  *public static void main(String[] args) {*  *System.out.println("main:"+Thread.currentThread().getPriority());*  *SleepThread st = new SleepThread();*  *st.start();*  *System.out.println("st:"+st.getPriority());*  *for(int i=1;i<=100;i++){*  *System.out.println("岳灵珊小师妹在练剑"+i);*  *if(i==10){*  *st.interrupt();*  *break;*  *}*  *}*  *}*  *}*  *class SleepThread extends Thread{*  *@Override*  *public void run() {*  *for(int i=1;i<=50;i++){*  *try {*  *Thread.sleep(100);*  *} catch (InterruptedException e) {*  *// e.printStackTrace();*  *System.out.println("令狐冲睡醒啦，开始好好学习");*  *break;*  *}*  *}*  *}*  *}* |

3）Yield、 join：

|  |
| --- |
| */\*\**  *\* 此类用于演示线程的常见方法四：*  *\* yield:线程的礼让，但是礼让的时间不确定。 是一个静态方法，直接通过类名调用即可*  *\* join：线程的插队，当前线程已经抢到cpu占用权，让其他线程插队在自己前面执行，如果其他线程插队成功，则肯定其他线程先执行完*  *\* 注意：其他线程有可能没有插队成功*  *\*/*  *public class Test {  public static void main(String[] args) {   /\*YieldDemo yd = new YieldDemo();  yd.start();\*/  JoinDemo jd = new JoinDemo("子线程：");  jd.start();  for(int i=1;i<=500;i++){  System.out.println("主线程："+i);  if(i>=5){  try {  jd.join(); //让jd线程插队在当前线程前面去，先执行  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  } } class YieldDemo extends Thread{  @Override  public void run() {  long start = System.currentTimeMillis();  String str="";  for(int i=1;i<=50000;i++){  str+="hello"+i;  Thread.yield();//让当前线程礼让  }  long end = System.currentTimeMillis();  System.out.println("耗时时间："+(end-start));  /\*  \* 不加礼让：耗时时间：10427  \* 加上礼让 ： 耗时时间：10602  \*/  } } class JoinDemo extends Thread{  public JoinDemo(String name){  super(name);  }  @Override  public void run() {  for(int i=1;i<=100;i++){  try {  Thread.sleep(5);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName()+","+i);  }  } }* |

### 2.1.2 线程的调度

调度策略：（1）时间片；（2）抢占式：高优先级的线程抢占CPU；

Java的调度方法：

（1）同优先级线程组成先进先出队列（先到先服务），使用时间片策略；

（2）对高优先级，使用优先调度的抢占式策略；

线程的优先级等级：

*MAX\_PRIORITY：10*

*MIN \_PRIORITY：1*

*NORM\_PRIORITY：5*

涉及的方法：

*getPriority() ：返回线程优先值*

*setPriority(int newPriority) ：改变线程的优先级*

注意：

线程创建时继承父线程的优先级；

低优先级只是获得调度的概率低，并非一定是在高优先级线程之后才被调用；

setPriority、getPriority：

|  |
| --- |
| */\*\**  *\* 此类用于演示线程的常见方法三：*  *\* @author liyuting*  *\* setPriority:设置优先级 1\_10*  *\* getPriority：获取优先级*  *\*/*  *public class TestThreadMethod3 {*  *public static void main(String[] args) {*  *PriorityDemo pd1 = new PriorityDemo("A");*  *pd1.setPriority(Thread.MAX\_PRIORITY);*  *pd1.start();*  *PriorityDemo pd2 = new PriorityDemo("B");*  *pd2.setPriority(Thread.MIN\_PRIORITY);*  *pd2.start();*  *PriorityDemo pd3 = new PriorityDemo("C");*  *pd3.setPriority(Thread.NORM\_PRIORITY);*  *pd3.start();*  *}*  *}*  *class PriorityDemo extends Thread{*  *public PriorityDemo(String name) {*  *super(name);*  *}*  *@Override*  *public void run() {*  *for(int i=1;i<=100;i++){*  *System.out.println(Thread.currentThread()+"-----"+i);*  *}*  *}*  *}* |

### 2.1.3 线程的分类

Java中的线程分为两类：一种是守护线程，一种是用户线程。

它们在几乎每个方面都是相同的，唯一的区别是判断JVM何时离开。守护线程是用来服务用户线程的，通过在start()方法前调用*thread.setDaemon(true)*可以把一个用户线程变成一个守护线程。

Java垃圾回收就是一个典型的守护线程。若JVM中都是守护线程，当前JVM将退出。

形象理解：兔死狗烹，鸟尽弓藏；

案例：

|  |
| --- |
| */\**  *\* 此类用于演示线程的常见方法五：*  *\* setDaemon(true) 设置守护线程*  *\* 线程分类： 线程停止的时机*  *\* 用户线程： 线程的任务体正常执行完*  *\* 守护线程： 所有用户线程执行结束，哪怕守护线程中的任务体还没有执行完毕，*  *也会伴随着结束，经典的守护线程：垃圾回收机制；*  *\*/*  *public class TestThreadMethod5 {*  *public static void main(String[] args) {*  *DaemonDemo dd = new DaemonDemo();*  *dd.setDaemon(true); //当main线程结束，守护线程也结束*  *dd.start();*  *for(int i=1;i<=100;i++){*  *System.out.println("王宝强正在认真的拍戏~"+i);*  *}*  *}*  *}*  *class DaemonDemo extends Thread{*  *@Override*  *public void run() {*  *while(true){*  *System.out.println("宋喆和马蓉愉快的聊天呢,哈哈哈哈");*  *}*  *}*  *}* |

### 2.1.4 线程停止的正确方式：

|  |
| --- |
| */\*\**  *\* 此类用于演示线程的停止*  *\* 1.stop 已经过时*  *\* 2.Interrupt 只能中断睡眠、等待这些状态，会抛出InterruptedException异常,并没有真正的结束线程*  *\* 3.采用通知的方法 ★*  *\* 步骤1：让需要停止的线程中添加一个循环标记，默认值为true*  *\* 步骤2：让需要停止的线程中添加一个公共的set方法，用于更新循环标记*  *\* 步骤3：需要停止该线程时，调用set方法即可*  *\*/*  *public class TestThreadStop {*  *public static void main(String[] args) {*  *StopThread st = new StopThread();*  *st.start();*  *for(int i=1;i<=100;i++){*  *System.out.println("小花在看电影~"+i);*  *if(i==5){*  *//中断StopThread线程*  *st.setFlag(false);*  *break;*  *}*  *}*  *}*  *}*  *class StopThread extends Thread{*  *boolean flag = true;*  *@Override*  *public void run() {*  *while(flag){*  *System.out.println("肖强和小黄约会ing,O(∩\_∩)O哈哈~");*  *}*  *}*  */\**  *\* 功能：可以更新循环变量标记*  *\*/*  *public void setFlag(boolean flag){*  *this.flag=flag;*  *}*  *}* |

## 2.2 实现Runnable接口

步骤：

1）定义子类，实现Runnable接口。

2）子类中重写Runnable接口中的run方法。

3）通过Thread类含参构造器创建线程对象。

4）将Runnable接口的子类对象作为实际参数传递给Thread类的构造器中。

5）调用Thread类的start方法：开启线程，调用Runnable子类接口的run方法。

语法：

|  |
| --- |
| *1、声明任务类：*  *class MyRunnable implement Runnable{*  *public void run(){*  *//线程体*  *}*  *}*  *2、创建线程，并启动：*  *new Tread(new MyRunnable).start();* |

案例：

|  |
| --- |
| */\*\**  *\* 此类用于演示线程的创建并启动*  *\*创建两个子线程，让其中一个输出1-100之间的偶数，另一个输出1-100之间的奇数*  *\*/*  *public class TestThreadStart {*  *public static void main(String[] args) {*  *//创建子线程1*  *A a = new A();*  *a.start();*  *//创建子线程2*  *new Thread(new B()).start();*  *}*  *}*  *class A extends Thread{*  *@Override*  *public void run() {*  *for(int i=1;i<=100;i++){*  *try {*  *Thread.sleep(10);*  *} catch (InterruptedException e) {*  *e.printStackTrace();*  *}*  *if (i%2==0) {*  *System.out.println("----------------------"+i);*  *}*  *}*  *}*  *}*  *class B implements Runnable{*  *public void run() {*  *for(int i=1;i<=100;i++){*  *try {*  *Thread.sleep(10);*  *} catch (InterruptedException e) {*  *e.printStackTrace();*  *}*  *if (i%2!=0) {*  *System.out.println(i);*  *}*  *}*  *}*  *}*  */\*\**  *\* 此类用于演示线程创建并启动方式的区别*  *\*案例：三个售票窗口卖票*  *\*/*  *public class TestThreadStartDiff {*  *public static void main(String[] args) {*  *//创建并启动 方式一 ：会出现数据问题*  *// SellTicket s1 = new SellTicket();*  *// s1.start();*  *// SellTicket s2 = new SellTicket();*  *// s2.start();*  *// SellTicket s3 = new SellTicket();*  *// s3.start();*    *//创建并启动 方式二*  *SellTicket2 st = new SellTicket2();*  *new Thread(st).start();*  *new Thread(st).start();*  *new Thread(st).start();*  *}*  *}*  *//方式一：*  *class SellTicket extends Thread{*  *int tickets = 100;//总票数*  *@Override*  *public void run() {*  *while(true){*  *if(tickets<=0){*  *System.out.println("票已经售完！");*  *break;*  *}*  *System.out.println(Thread.currentThread().getName()+"卖了一张票，票数余额："+(--tickets));*  *}*  *}*  *}*  *//方式二：*  *class SellTicket2 implements Runnable{*  *int tickets = 100;//总票数*  *@Override*  *public void run() {*  *while(true){*  *if(tickets<=0){*  *System.out.println("票已经售完！");*  *break;*  *}*  *System.out.println(Thread.currentThread().getName()+"卖了一张票，票数余额："+(--tickets));*  *}*  *}*  *}* |

## 2.3 继承方式和实现方式的联系与区别：

*public class Thread extends Object implements Runnable*

区别：

继承Thread: 线程代码存放Thread子类run方法中。

实现Runnable：线程代码存在接口的子类的run方法。

实现方法的好处：

1）避免了单继承的局限性；

2）多个线程可以共享同一个接口子类的对象，非常适合多个相同线程来处理同一份资源；

案例：

|  |
| --- |
| */\*\**  *在main方法中创建并启动两个线程。*  *第一个线程循环随机打印100以内的整数，直到第二个线程从键盘读取了“Q”命令。*  *\*/*  *public class TestStop {*  *public static void main(String[] args) {*  *MyThread1 m1 = new MyThread1();*  *MyThread2 m2 = new MyThread2(m1);*  *m1.start();*  *m2.start();*  *}*  *}*  *class MyThread1 extends Thread{*  *boolean flag =true;*  *public void run() {*  *while(flag){*  *try {*  *Thread.sleep(30);*  *} catch (InterruptedException e) {*  *e.printStackTrace();*  *}*  *int num = (int)(Math.random()\*100);*  *System.out.println("随机数："+num);*  *}*  *}*  *public void setFlag(boolean flag) {*  *this.flag = flag;*  *};*  *}*  *class MyThread2 extends Thread{*  *Scanner input = new Scanner(System.in);*  *MyThread1 m ;*  *public MyThread2(MyThread1 m){*  *this.m=m;*  *}*  *@Override*  *public void run() {*  *while(true){*  *System.out.println("请输入：");*  *char key = input.next().toUpperCase().charAt(0);*  *if(key=='Q'){*  *//1. 中断MyThread1（通过MyThread1对象调用setFlag方法）*  *m.setFlag(false);*  *//2. 本身线程停止*  *break;*  *}*  *}*  *}*  *}* |

# 第三章 线程的生命周期

JDK中用*Thread.State枚举*表示了线程的几种状态；

要想实现多线程，必须在主线程中创建新的线程对象。Java语言使用Thread类及其子类的对象来表示线程，在它的一个完整的生命周期中通常要经历如下的五种状态：

|  |
| --- |
| *public enum State {*  *NEW,*  *RUNNABLE,*  *BLOCKED,*  *WAITING,*  *TIMED\_WAITING,*  *TERMINATED;*  *}* |

1. 新建：

当一个Thread类或其子类的对象被声明并创建时，新生的线程对象处于新建状态；

1. 就绪：

处于新建状态的线程被*start()*后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件；

1. 运行：

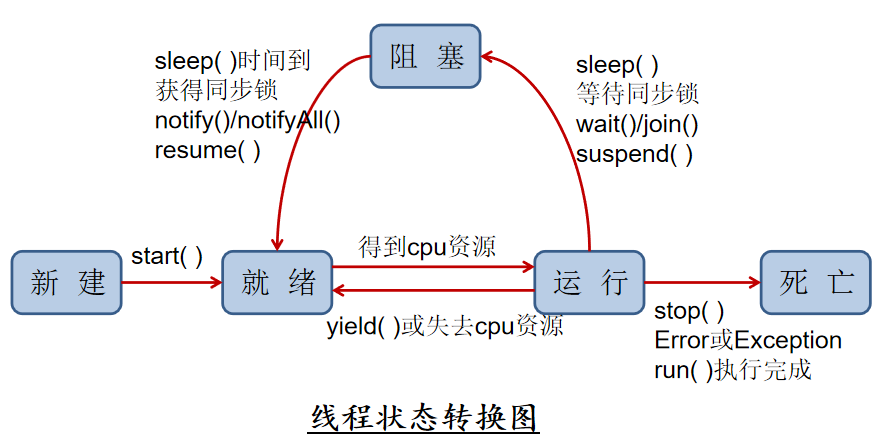
当就绪的线程被调度并获得CPU资源时,便进入运行状态，run()方法定义了线程的操作和功能；

1. 阻塞：

在某种特殊情况下，被人为挂起或执行输入输出操作时，让出 CPU 并临时中止自己的执行，进入阻塞状态；

1. 死亡：

线程完成了它的全部工作或线程被提前强制性地中止；



案例：

|  |
| --- |
| */\*\**  *\* 1. 案例题目描述:编写龟兔赛跑多线程程序，设赛跑长度为30米，乌龟和兔子每跑完10米输出一次结果。*  *兔子的速度是10米每秒,兔子每跑完10米休眠的时间10秒*  *乌龟的速度是1米每秒,乌龟每跑完10米的休眠时间是1秒*  *2. 案例完成思路要求：*  *1)乌龟定义一个线程,兔子定义一个线程*  *2)两个线程同时开启*  *3)提示：可以使用Thread.sleep(毫秒数)来模拟耗时*  *\*/*  *public class Test {  public static void main(String[] args) {  Rabbit r = new Rabbit();  r.setName("兔子");  r.start();  Torist t = new Torist();  t.setName("乌龟");  t.start();  } }  class Rabbit extends Thread{  @Override  public void run() {  for(int i=1;i<=30;i++){ //i:米数  try {  Thread.sleep(100); // 100毫秒1米  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(getName()+"跑了"+i+"米！");  if(i%10==0){  try {  Thread.sleep(10000); // 10米休息10秒  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  } }  class Torist extends Thread{  @Override  public void run() {  for(int i=1;i<=30;i++){ //i:米数  try {  Thread.sleep(1000); //1秒1米  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(getName()+"跑了"+i+"米");  if(i%10==0){  try {  Thread.sleep(1000); // 10米休息1秒  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  } }* |

# 第四章 线程的同步

线程的同步是指线程的安全问题，如果多个线程处理共享资源时，出现了数据安全问题，称为线程的同步问题。

1）线程的同步如何发生的？

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；

2）如何避免线程的同步问题：

对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可以参与执行；

比如：厕所的坑位上锁；

Synchronized的使用方法：

|  |
| --- |
| 1. 同步代码块   *synchronized (对象){*  *// 需要被同步的代码；*  *}*   1. *同步方法。*   *public synchronized void show (String name){*  *….*  *}* |

注意：普通的同步方法，锁对象：this;

静态的同步方法，锁对象：当前类.class

要求：多个线程使用的锁对象必须是同一个对象；

3）线程同步的前提：

1. 多线程环境；
2. 有共享资源或者线程的任务体中有对共享资源的处理语句；

4）使用线程同步的思路：

（1）先分析需要上锁的代码；

（2）分析需要同步代码块，还是同步方法；

（3）检测多线程使用的锁对象是否为同一个；

案例：

|  |
| --- |
| */\*\*  \* 此类用于演示线程的同步问题  \*/ public class Test {  public static void main(String[] args) {  SellTicket2 s = new SellTicket2();  Thread t1 = new Thread(s);  t1.setName("窗口A");  t1.start();  Thread t2 = new Thread(s);  t2.setName("窗口B");  t2.start();  Thread t3 = new Thread(s);  t3.setName("窗口C");  t3.start();  } }*  *//方式一：使用同步代码块实现同步 class SellTicket implements Runnable{  int tickets = 1000;//总票数  @Override  public void run() {  while(true){  //当某一个线程执行时，其他线程处于排队状态，  synchronized (this) {  if (tickets <= 0) {  System.out.println("票已经售完！");  break;  }  try {  Thread.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + "卖了一张票，票数余额：" + (--tickets));  }  }  } }*  *//方式二：使用同步方法实现同步 class SellTicket2 implements Runnable{  int tickets = 1000;//总票数  boolean loop=true;  @Override  public void run() {  while(loop){  sellTicket();  }  }  private synchronized void sellTicket() {  if (tickets <= 0) {  System.out.println("票已经售完！");  loop=false;  return;  }  try {  Thread.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + "卖了一张票，票数余额：" + (--tickets));  } }*  */\*\**  *\* 有2个账户分别向同一个卡上取钱（10000），每次都取1000*  *\*/*  *public class TestSyn {*  *public static void main(String[] args) {*  *WithDraw w1 = new WithDraw();*  *Thread t1 = new Thread(w1);*  *t1.setName("周芷若");*  *t1.start();*  *Thread t2 = new Thread(w1);*  *t2.setName("赵敏");*  *t2.start();*  *}*  *}*  *class WithDraw implements Runnable{*  *int money = 10000;*  *@Override*  *public void run() {*  *while(true){*  *synchronized (this) {*  *if (money <= 0) {*  *System.out.println("钱已经取完");*  *break;*  *}*  *try {*  *Thread.sleep(10);*  *} catch (InterruptedException e) {*  *e.printStackTrace();*  *}*  *System.out.println(Thread.currentThread().getName() + "取了1000，还剩余：" + (money -= 1000));*  *}*  *}*  *}*  *}*  */\*\**  *\* 此类用于演示同步在单例模式中的应用*  *\*/*  *public class TestSingle {*    *public static void main(String[] args) {*  *A a = new A();*  *A a2 = new A();*  *a.start();*  *a2.start();*  *}*  *}*  *class A extends Thread{*  *@Override*  *public void run() {*  *while(true){*  *try {*  *Thread.sleep(10);*  *} catch (InterruptedException e) {*  *e.printStackTrace();*  *}*  *System.out.println(Single.getInstance());*  *}*  *}*  *}*  *//懒汉式*  *class Single{*  *//1.构造器私有化*  *private Single(){}*  *//2.本类内部创建Single的引用*  *private static Single single;*  *//3.提供一个公共的方法，对外暴露该对象*  *public static Single getInstance(){*  *if(single==null){//提高效率*  *synchronized (Single.class) {*  *if (single == null) {//线程的安全*  *try {*  *Thread.sleep(50);*  *} catch (InterruptedException e) {*  *e.printStackTrace();*  *}*  *single = new Single();*  *}*  *}*  *}*  *return single;*  *}*  *}* |

## 4.1 同步锁 Lock：

从Java 5开始，Java提供了更强大的线程同步机制—通过显式定义同步锁对象来实现同步。同步锁使用Lock对象充当。

Lock是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象。

在实现线程安全的控制中，比较常用的是*ReentrantLock(可重入锁)*，可以显式加锁、释放锁。

## 4.2 互斥锁 synchronized：

在Java语言中，引入了对象互斥锁的概念，来保证共享数据操作的完整性。

每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。

关键字synchronized 来与对象的互斥锁联系。当某个对象用synchronized修饰时，表明该对象在任一时刻只能由一个线程访问。

同步的局限性：导致程序的执行效率要降低

同步方法（非静态的）的锁为this。

同步方法（静态的）的锁为当前类本身。

## 4.3 Lock与synchronized 区别：

 synchronized与Lock两者区别：

   1）Lock是一个接口，而Synchronized是关键字。

  2）Synchronized会自动释放锁，而Lock必须手动释放锁。

  3）Lock可以让等待锁的线程响应中断，而Synchronized不会，线程会一直等待下去。

   4）通过Lock可以知道线程有没有拿到锁，而Synchronized不能。

   5）Lock能提高多个线程读操作的效率。

   6）Synchronized能锁住类、方法和代码块，而Lock是块范围内的；

synchronized：在需要同步的对象中加入此控制，synchronized可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。  
 lock：需要显示指定起始位置和终止位置。一般使用ReentrantLock类做为锁，多个线程中必须要使用一个ReentrantLock类作为对象才能保证锁的生效。且在加锁和解锁处需要通过lock()和unlock()显示指出。所以一般会在finally块中写unlock()以防死锁。  
 synchronized原始采用的是CPU悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在CPU转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起CPU频繁的上下文切换导致效率很低  
 Lock用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。  
 乐观锁实现的机制就是CAS操作（Compare and Swap）。我们可以进一步研究ReentrantLock的源代码，会发现其中比较重要的获得锁的一个方法是compareAndSetState。这里其实就是调用的CPU提供的特殊指令。  
案例：

|  |
| --- |
| *class LockTest{*  *//构建同步锁对象：*  *private Lock lock = new ReentrantLock();*  */\**  *\* 当前线程使用完毕释放后，其他线程才能获取锁*  *\*/*  *public void lockTest(Thread thread) {*  *//当前线程获取同步锁：*  *lock.lock();*  *System.out.println("线程：" + thread.getName() + "获取同步锁！");*  *try {*  *//为看出执行效果，是线程此处休眠2秒*  *Thread.sleep(2000);*  *} catch (InterruptedException e) {*  *System.out.println("线程"+thread.getName() + "发生了异常释放锁");*  *}finally {*  *System.out.println("线程"+thread.getName() + "执行完毕释放锁");*  *//当前线程释放锁*  *lock.unlock();*  *}*  *}*  *}*  *public class Test {*  *public static void main(String[] args) {*  *LockTest lockTest = new LockTest();*  *//声明一个线程 “线程一”*  *Thread thread1 = new Thread(new Runnable() {*  *@Override*  *public void run() {*  *lockTest.lockTest(Thread.currentThread());*  *}*  *} , "子线程1");*  *//声明一个线程 “线程二”*  *Thread thread2 = new Thread(new Runnable() {*  *@Override*  *public void run() {*  *lockTest.lockTest(Thread.currentThread());*  *}*  *} , "子线程2");*  *thread1.start();*  *thread2.start();*  *}*  *}*    https://www.cnblogs.com/zhaosq/archive/2019/04/16/10694597.html |

## 4.4 释放锁的操作：

1）当前线程的同步方法、同步代码块执行结束；

2）当前线程在同步代码块、同步方法中遇到break、return终止了该代码块、该方法的继续执行；

3）当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致异常结束；

4）当前线程在同步代码块、同步方法中执行了线程对象的wait()方法，当前线程暂停，并释放锁；

## 4.5 不会释放锁的操作：

1）线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行；

2）线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁（同步监视器）；

3）应尽量避免使用suspend()和resume()来控制线程；

## 4.6 线程的死锁问题：

1）死锁：

不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁。

2）解决方法：

专门的算法、原则；

尽量减少同步资源的定义；

案例：

|  |
| --- |
| */\*\*  \* 此类用于演示线程的死锁问题  \* 案例：绑匪绑了田雨  \* 张飞凡：你把田雨还给我，我把钱给你  \* 绑匪：你把钱给我，我把田雨还给你  \*/ public class Test {  public static void main(String[] args) {  DeadDemo d1 = new DeadDemo(true); //张飞凡的线程  DeadDemo d2 = new DeadDemo(false); //绑匪的线程  d1.start();  d2.start();  } } class DeadDemo extends Thread{  //共享资源：  static ZhangFeiFan zhangFeiFan = new ZhangFeiFan();  static BangFei bb = new BangFei();  private boolean flag;  public DeadDemo(boolean flag){  this.flag = flag;  }  @Override  public void run() {  if (flag) { //张飞凡的线程  synchronized (bb) {  zhangFeiFan.say();  synchronized (zhangFeiFan) {  zhangFeiFan.get();  }  }  } else { //绑匪的线程  synchronized (zhangFeiFan) {  bb.say();  synchronized (bb) {  bb.get();  }  }  }  } } class BangFei{  public void say(){  System.out.println("绑匪说：你把钱给我，我把田雨还给你");  }  public void get(){  System.out.println("绑匪终于拿到了想要的钱！");  } } class ZhangFeiFan{  public void say(){  System.out.println("张飞凡说：你把田雨还给我，我把钱给你");  }  public void get(){  System.out.println("张飞凡终于救回了田雨");  } }* |

# 第五章 线程通信

线程通信：在线程之间”交流“时，通过一些方法，实现对线程的控制；

线程通信方法：

1. wait()方法：

语法：锁对象.wait();

特点：

（1）让当前线程等待，直到其他线程调用了该锁对象的notify()或者notifyAll()方法为止，否则将一直等待；

（2）wait()方法的调用，必须是在同步的条件下；

（3）wait()方法的调用，会导致锁的释放；

（4）如果等待结束（被唤醒）后，则继续从断点处往下执行；

2）notify() 、notifyAll():

语法：锁对象.notify() 、锁对象.notifyAll();

特点：

（1）notity : 唤醒当前锁对象下等待的单个线程。优先级较高的优先被唤醒，如果优先级一样，则随机；

（2）notityAll : 唤醒当前锁对象下所有等待线程；

（3）Notity和notityAll的调用，必须是在同步的条件下；

注意：wait、notify、notifyAll和当前同步代码的锁对象必须是同一个，否则会报异常，*java.lang.IllegalMonitorStateException*；

案例：

|  |
| --- |
| */\*\**  *\* 此类用于演示线程的通信*  *\*/*  *public class TestCommunication {*  *public static void main(String[] args) {*  *WithDraw w1 = new WithDraw();*  *Thread t1 = new Thread(w1);*  *t1.setName("周芷若");*  *t1.start();*  *Thread t2 = new Thread(w1);*  *t2.setName("赵敏");*  *t2.start();*  *}*  *}*  *class WithDraw implements Runnable {*  *int money = 10000;*  *@Override*  *public void run() {*  *while (true) {*  *synchronized (this) {*  *if (money <= 0) {*  *System.out.println("钱已经取完");*  *break;*  *}*  *try {*  *Thread.sleep(10);*  *} catch (InterruptedException e) {*  *e.printStackTrace();*  *}*  *System.out.println(Thread.currentThread().getName() + "取了1000，还剩余：" + (money -= 1000));*  *//通知对方线程可以取钱了*  *this.notify();*  *//当前线程等待*  *try {*  *this.wait();*  *} catch (InterruptedException e) {*  *e.printStackTrace();*  *}*  *}*  *}*  *}*  *}*  */\*\*  \* 此类用于演示线程的通信  \* 案例：一个生产者和一个消费者  \*/ public class Test {  public static void main(String[] args) {  Clerk c = new Clerk();  Productor p = new Productor(c);  p.start();  Consumer con = new Consumer(c);  con.start();  } } //消费者： class Consumer extends Thread{  private Clerk c;  public Consumer(Clerk c){  this.c=c;  }  @Override  public void run() {  while(true){  //消费产品（调用Clerk对象的get方法）  c.get();  }  } } //生产者： class Productor extends Thread{  private Clerk c;  public Productor(Clerk c){  this.c=c;  }  @Override  public void run() {  while(true){  //生产产品（调用Clerk对象的save方法）  c.save();  }  } } // 记录员 class Clerk{  int count = 0; //产品数量  //生产  public synchronized void save(){ //默认锁对象：this  if(count>=100){ //库存已满  //等待  try {  this.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  try {  Thread.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println("生产者生产了一件产品，目前库存为："+(++count));  this.notify(); //唤醒消费者线程去消费  }  //消费  public synchronized void get(){  if(count<=0){  try {  this.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  try {  Thread.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println("消费者消费了一件产品，目前库存为："+(--count));  this.notify(); //唤醒生产者去生产  } }*  */\*\*  \* 此类用于演示线程的通信  \* 案例：多个生产者和多个消费者  \* 修改：  \* ①notify——>notifyAll  \* ②wait的条件判断，改成循环判断  \*/ public class Test {  public static void main(String[] args) {  Clerk2 c = new Clerk2();  Productor2 p1 = new Productor2(c,"张无忌");  p1.start();  Productor2 p2 = new Productor2(c,"令狐冲");  p2.start();  Consumer2 con1 = new Consumer2(c,"赵敏");  con1.start();  Consumer2 con2 = new Consumer2(c,"小昭");  con2.start();  Consumer2 con3 = new Consumer2(c,"依琳小师妹");  con3.start();  } } class Consumer2 extends Thread{  private Clerk2 c;  public Consumer2(Clerk2 c,String name){  super(name);  this.c=c;  }  @Override  public void run() {  while(true){  //消费产品（调用Clerk对象的get方法）  c.get();  }  } } class Productor2 extends Thread{  private Clerk2 c;  public Productor2(Clerk2 c,String name){  super(name);  this.c=c;  }  @Override  public void run() {  while(true){  //生产产品（调用Clerk对象的save方法）  c.save();  }  } } class Clerk2{  int count = 0; //产品数量  //生产  public synchronized void save(){ //默认锁对象：this  while(count>=20){ //库存已满  //等待  try {  this.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  try {  Thread.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName()+"生产了一件产品，目前库存为："+(++count));  this.notifyAll(); //唤醒其他正在等待的所有线程  }  //消费  public synchronized void get(){  while(count<=0){  try {  this.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  try {  Thread.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName()+"消费了一件产品，目前库存为："+(--count));  this.notifyAll(); //唤醒其他正在等待的所有线程  } }* |

# JDK5.0新增线程创建方式：

## 6.1 实现Callable接口

与使用Runnable相比， Callable功能更强大：

（1）call()方法相比run()方法，可以有返回值；

（2）call()方法可以抛出异常；

（3）callable支持泛型的返回值；

（4）需要借助FutureTask类，比如获取返回结果；

Future接口：

可以对具体Runnable、Callable任务的执行结果进行取消、查询是否完成、获取结果等。FutrueTask是Futrue接口的唯一的实现类。FutureTask 同时实现了Runnable, Future接口。它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。

案例：

|  |
| --- |
| *//1.创建一个实现Callable接口的实现类：*  *class MyThread implements Callable{*  *//2.实现call()方法，类似于run()方法*  *@Override*  *public Object call() throws Exception {*  *int sum = 0;*  *for (int i = 0; i < 100; i++) {*  *if (i % 2 == 0) {*  *System.out.println(i);*  *sum += i;*  *}*  *}*  *return sum; //自动装箱*  *}*  *}*  *public class Test {*  *public static void main(String[] args) {*  *//3.创建callable接口实现类：*  *MyThread mt = new MyThread();*  *//4.将callable接口实现类对象传递到FutureTask构造器中，创建FutureTask对象*  *FutureTask futureTask = new FutureTask(mt);*  *//5.将FutureTask对象传递到Thread构造器中,构建线程类对象；*  *Thread thread = new Thread(futureTask);*  *thread.start();*  *//6.获取callable中call方法的返回值：*  *try {*  *//get方法返回值即为call方法的返回值*  *Object sum = futureTask.get();*  *System.out.println("结果：" + sum);*  *} catch (InterruptedException | ExecutionException e) {*  *e.printStackTrace();*  *}*  *}*  *}* |

## 6.2 使用线程池

背景：经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。

思路：提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。

好处：

（1）提高响应速度（减少了创建新线程的时间）；

（2）降低资源消耗（重复利用线程池中线程，不需要每次都创建）；

（3）便于线程管理：

*corePoolSize：核心池的大小*

*maximumPoolSize：最大线程数*

*keepAliveTime：线程没有任务时最多保持多长时间后会终止*

*…*

线程池相关API：

JDK 5.0起提供了线程池相关API：ExecutorService 和 Executors。

ExecutorService：

真正的线程池接口。常见子类*ThreadPoolExecutor*

*void execute(Runnable command)* ：执行任务/命令，没有返回值，一般用来执行

Runnable；

*<T> Future<T> submit(Callable<T> task)*：执行任务，有返回值，一般又来执行

Callable；

*void shutdown()* ：关闭连接池；

案例：

|  |
| --- |
| *//创建并使用多线程的第四种方法：使用线程池*  *class MyThread implements Runnable {*  *@Override*  *public void run() {*  *for (int i = 1; i <= 100; i++) {*  *System.out.println(Thread.currentThread().getName() + ":" + i);*  *}*  *}*  *}*  *public class ThreadPool {*  *public static void main(String[] args) {*  *// 1.调用Executors的newFixedThreadPool(),返回指定线程数量的ExecutorService*  *ExecutorService pool = Executors.newFixedThreadPool(10);*  *// 2.将Runnable实现类的对象作为形参传递给ExecutorService的submit()方法中，开启线程*  *// 并执行相关的run()*  *pool.execute(new MyThread());*  *pool.execute(new MyThread());*  *pool.execute(new MyThread());*  *// 3.结束线程的使用*  *pool.shutdown();*  *}*  *}* |

Executors：

工具类、线程池的工厂类，用于创建并返回不同类型的线程池；

系统启动一个新线程的成本是比较高的，因为它涉及与os交互。这种情况下，系统启动时即创建大量空闲的线程，就可以很好地提高性能，尤其是当程序需要创建大量生存期很短暂的线程时。

除此之外，使用线程池可以有效地控制系统中并发线程的数量。避免因并发创建的线程过多，导致系统性能下降，JVM崩溃。

Java 5以前，需要手动创建自己的线程池；Java 5开始，新增了Executors工厂类产生线程池。

使用线程池执行线程任务的步骤如下：

1.调用Executors 类的静态方法*newFixedThreadPool(int nThreads)*，创建一个可重用的、具有固定线程数的线程池ExecutorService对象；

2.创建Runnable实例，作为线程执行任务；

3.调用ExecutorService对象的submit()提交Runnable实例；

4.调用ExecutorService对象的shutDown()方法关闭线程池；

四种线程池：

Java通过Executors提供四种线程池，分别为：  
1）newCachedThreadPool：

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。  
2）newFixedThreadPool：

创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  
3）newScheduledThreadPool：

创建一个定长线程池，支持定时及周期性任务执行。  
4）newSingleThreadExecutor：

创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

注意：

线程池只是为了控制应用中处理某项业务中防止高并发问题带来的线程不安全的发生的概率。在我目前的测试用，还没有发现线程可以重用这个概念，因为线程开启后，用完就关闭了，不可以再次开启的，查看源码发现会每次新创建一个线程用来处理业务。我们可以通过线程池指定处理这项业务最大的同步线程数，比如：*Executors.newFixedThreadPool(3);*在线程池中保持三个线程可以同时执行，但是注意，并不是说线程池中永远都是这三个线程，只是说可以同时存在的线程数，当某个线程执行结束后，会有新的线程进来。*newFixedThreadPool.execute(new ThreadForpools());*这句话的含义并不是添加新的线程，而是添加新的处理业务请求进来。至少我当前是这么理解的，没有发现线程可以重复使用。

案例源码：

|  |
| --- |
| *public class ExecutorsTest {  public static void main(String[] args) { } } class ThreadForpools implements Runnable{  private Integer index;  public ThreadForpools() {  }  public ThreadForpools(Integer index) {  this.index = index;  }  @Override  public void run() {  /\*\*  \* 业务逻辑省略  \*/  try {  System.out.println("开始处理线程！");  Thread.sleep(index \* 100);  System.out.println("我的线程标识：" + this.toString());  } catch (InterruptedException e) {  e.printStackTrace();  }  } } /\*\*  \* Executors.newCachedThreadPool()  \* 创建一个可缓存线程池，应用中存在的线程数可以无限大；  \* 输出结果是：可以有无限大的线程数进来（线程地址不一样）  \*/ class ThreadPools1{  public static void main(String[] args) {  /\*\*  \* 获取4次线程，观察4个线程地址：  \*/  ExecutorService cachedThreadPool = Executors.newCachedThreadPool(); System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*newCachedThreadPool\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  for (int i=0;i<4;i++){  final int index = i;  cachedThreadPool.execute(new ThreadForpools(index));  }  } } /\*\*  \* Executors.newFixedThreadPool(num);  \* 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  \* 输出结果：每次只有两个线程在处理，当第一个线程执行完毕后，新的线程进来开始处理（线程地址不一样）  \*/ class ThreadPools2{  public static void main(String[] args) {  //线程池允许同时存在两个线程:  ExecutorService fixedThreadPool = Executors.newFixedThreadPool(2);  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*newCachedThreadPool\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  for (int i=0;i<4;i++){  final int index = i;  fixedThreadPool.execute(new ThreadForpools(index));  }  } } /\*\*  \* Executors.newScheduledThreadPool(num):  \* 创建一个定长线程池，支持定时及周期性任务执行。  \* 执行结果：延迟三秒之后执行，除了延迟执行之外和newFixedThreadPool基本相同，可以用来执行定时任务  \*/ class ThreadPools3{  public static void main(String[] args) {  //线程池允许同时存在两个线程:  ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(2); System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*newCachedThreadPool\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  for (int i=0;i<4;i++){  final int index = i;  //延迟三秒执行：  scheduledThreadPool.schedule(new ThreadForpools(index) , 3 , TimeUnit.SECONDS);  }  } } /\*\*  \*Executors.newSingleThreadExecutor():  \* 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。  \*执行结果：只存在一个线程，顺序执行  \*/ class ThreadPools4{  public static void main(String[] args) {  ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor(); System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*newCachedThreadPool\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  for (int i=0;i<4;i++){  final int index = i;  singleThreadExecutor.execute(new ThreadForpools(index));  }  } }* |