第一章 Java 并发编程实践基础

1.1 进程与线程

进程和线程是两个既有关系，又有重大区别的计算机概念，本届首先回顾一下进程和线程的基本概念，然后讲解一下他们的区别，最后是 Java 线程概念模型。

1.1.1 进程

讲解进程的概念时，首先会提到与之相关的另一个概念：程序。首先介绍程序的概念，然后引入进程。

1.1.1.1程序与资源共享

1. 程序的封闭性与可再现性

在程序设计中，程序员习惯于用顺序方式编制程序。例如，一个比较典型的顺序程序是：先从某一外部设备（例如磁盘）上输入数据，随之一步一步进行计算，最后将计算结果输出。计算机中的这种程序活动有如下几个特点：

（1）一个程序在机器中运行时独占全机资源，因此除了初始状态外，只有程序本身规定的动作才能改变这些资源的状态。

（2）机器严格地顺序执行程序规定的动作。每个动作都必须在前一动作结束后才能开始，除了人为干预造成机器暂时停顿外，前一动作的结束就意味着后一动作的开始。程序和机器执行程序的严格一一对应。

（3）程序的执行结果与它的运行速度无关。也就是说，处理机在执行程序两个动作之间的停顿不会影响程序的执行结果。

上述特点概况起来就是程序的封闭性和可再现性。所谓封闭性指的是程序一旦开始运行，其计算结果就只取决于程序本身，除了人为地改变机器的运行状态或机器故障以外，没有其它因素能够对程序的运行过程施加影响。所谓再现性就是当机器在同一数据集上重复执行同一程序时，机器内部的动作系列完全相同，最后获得的结果也相同。这种工作方式的特点是简单、清晰、便于调试程序。

2. 资源共享与并行

为了提高计算机系统内各种资源的使用效率，现代计算机系统设计中普遍采用了多道程序技术。与单道程序相比，多道程序的工作环境发生了很大变化，主要表现在下列两个方面：

（1）资源共享

资源共享指的是系统中的软、硬件资源不再为单个用户程序独占，而由几道用户程序共同使用。于是，这些资源的状态就不再取决于一道程序，而是由多道程序的活动所决定。这就从根本上打破了了一道程序封闭于一个系统中运行的局面。

（2）程序的并发运行

系统中各个部分不再以单纯的串行方式工作。换言之，在任一时刻系统中不再只有一个活动，而是存在着许多并行的活动。从硬件方面看，处理机、各种外设、存储部件常常并行地进行着工作。从程序方面看，则可能有若干个作业程序或者同时、或者互相穿插在系统中并行运行。这时，机器不再是简单地顺序执行一道程序。也就是说，一道程序的前一动作结束后，系统不一定立即执行其后续操作，而可能转而执行其它程序的某一操作。对于程序中可以执行的操作也可能不需要等待另一操作结束，系统就开始执行它们。这样也就打破了程序执行的顺序性。同时，多个程序活动可能是在不同的数据集上执行同一个程序，所以程序以及机器执行程序的活动不再有严格的一一对应关系。

1.1.1.2进程与并发

1. 进程的引入

在多道程序工作环境下，一个程序活动不再能独占系统资源，因此也就不再能单独决定这些资源的状态；程序和机器执行程序的活动之间也不再有一一对应关系。总之，程序活动不再处于一个封闭的系统中，而是和其它程序活动之间存在着相互依赖和制约的关系，因而呈现出并发、动态以及相互制约这些新的特征。在这种情况下，程序这个静态的概念已经不能如实地反映程序活动的这些特征。为此，六十年代中期 MULTICS 操作系统的设计者和 E.W.Dijkstra 为首的 T.H.E 操作系统的设计者开始广泛应用进程(process)这一新的概念来描述系统和用户的程序活动。

“进程”是操作系统的最基本的，也是最重要的概念之一。这个概念对于操作系统的理解、描述和设计都具有极其重要的意义。但是迄今为止对这一概念还没有一个确切统一的描述。有人称进程是可以并行运动的计算部分(S.E.Madnick,J.J.Donovan);有人称进程是一个程序与其数据一道在计算机上顺序执行时所产生的活动(A.C.Shaw);有人从调度组织角度出

发，称进程是一个独立的可以调度的活动(Ellis.Cohen,DavidJofferson);有人则从资源共享和竞争方面观察，认为进程是一个抽象的实体，当它执行一个任务时将要求分配和释放各种资源(Peterdenning)。这些描述都注意到了进程的动态性质，但侧重面不同。为了突出进程和程序两个概念的区别和联系，我们对进程作如下描述：进程是一种活动，它是由一个动作系列组成，每个动作是在某个数据集上执行一段程序，整个活动的结果是提供一种系统或用户功能。

2. 进程与程序的区别

我们再为进程和程序之间的区别和联系作以下几点说明。

1）进程是程序的一次运行活动，属于一种动态的概念。程序是一组有序的静态指令，是一种静态的概念。但是，进程离开了程序也就没有了存在的意义。因此，我们可以这样说：进程是执行程序的动态过程，而程序是进程运行的静态文本。如果我们把一部动画片的电影拷贝比拟成一个程序，那么这部动画片的一次放映过程就可比为一个进程。

2）一个进程可以执行一个或多个程序。例如：一个进程进行 C 源程序编译时，它要执行前处理、词法语法分析、代码生成和优化等几个程序。反之，同一程序也可能由多个进程同时执行，例如：上述 C 编译程序可能同时被几个程序执行，它们对相同或不同的源程序分别进行编译，各自产生目标程序。我们再次以动画片及其放映活动为例，一次电影放映活动可以连续放映几部动画片，这相当于一个进程可以执行几个程序。反之，一部动画片可以同时在若干家电影院中放映，这相当于多个进程可以执行几个同一程序。不过要注意的是，几家电影院放映同一部电影，如果使用的是同一份拷贝，那么实际上是交叉进行的。但在多处理机情况下，几个进程却完全可以同时使用一个程序副本。

3）程序可以作为一种软件资源长期保持着，而进程则是一次执行过程，它是暂时的，是动态地产生和终止的。这相当于电影拷贝可以长期保存，而一次放映活动却只延续 1～2 小时。

进程需要使用一种机构才能执行程序，这种机构称之为处理机(Processor)。处理机执行指令，根据指令的性质，处理机可以单独用硬件或软、硬件结合起来构成。如果指令是机器指令，那么处理机就是我们一般所说的中央处理机(CPU)。

3. 进程的并发性和不确定性

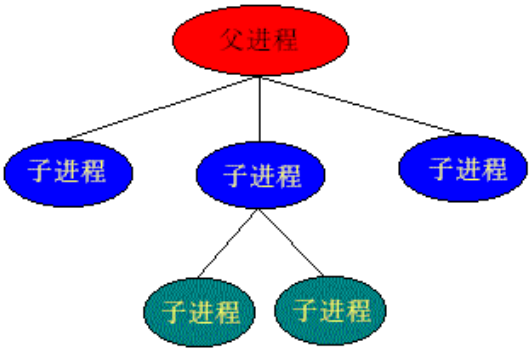
并发性：并发可以看成是在系统中同时有几个进程在活动着，也就是同时存在几个程序的执行过程。如果进程数与处理机数相同，则每个进程都占用一个处理机。但更一般的情况是是处理机数少于进程数，于是处理机就应被共享，在进程间进行切换使用。如果相邻两次切换的时间间隔非常短，而观察时间又相当长，那么各个进程都在前进，造成一种宏观上并

行运行的效果。所以并发处理的真正含义是：如果我们把系统作为一个整体来观察，则在任一时刻有若干进程存在于系统的这一部分或那一部分，这些进程都处在其起点和终点之间。我们把所有这些进程都看成是正在系统中运行着、活跃着。

不确定性：我们把进程看成是一个动作系列，而每个动作是执行一段程序。处理机要检测是否已接获某种需要立即处理的中断信号。如果已经接到这种信号，则立即停止正在执行的程序段，转而执行相应的中断处理程序。在此以后，还要按情况或者恢复继续执行被中断的程序，或者调度执行另一个进程的程序。因为中断发生的时间以及频繁程度与系统中许多经常变化着的不确定因素有关，例如，系统中活跃着的进程的数量以及它们的工作情况，各种硬件工作速度的细微变化等，所有它们都是不可预测的。因此，各个进程(也就是各个动作序列)也就在不可预测的次序中前进。如果由于进程间相互制约关系造成了某一进程或某些进程异常情况，那么由于这种制约关系是与一定的活动序列紧密相关的，而这种动作序列又不易复现。于是它所造成的进程的异常运行情况也就不易复现。可见，操作系统外部表现出来的不确定性就是内部动作序列不可预测、不易复现的反应。

4. 进程的结构

在 **UNIX 或者 Linux** 中，进程是通过 **FORK 系统调用**被创建的。在调用了 **FORK** 之后，父进程可以和子进程**并行**。父进程还可以创建多个子进程，也就是说，在同一时刻，一个父进程可以有多个正在运行的子进程。子进程也可以执行 FORK 调用。这样就可以在系统中生成一个**进程树**。



进程层次图

进程通常由三部分组成。一部分是程序，一部分数据集合，另一部分被称为进程控制块(ProcessControlBlock,简记 PCB)。

进程的程序部分描述了进程所要完成的功能。数据集合部分则有两方面的内容，即程序运行时所需要的数据部分和工作区。如果一个程序能为多个进程同时共享执行，它是进程执

行时不可修改的部分。而数据集合部分则通常为一个进程独占，为进程的可修改部分。程序和数据集合是进程存在的物质基础，是进程的实体。

进程控制块有时也称为进程描述块，它包含了进程的描述信息和控制信息，是进程动态特性的集中反映。它所包含的信息类型和数量随操作系统而异。在小型的比较简单的操作系统中，PCB 只占用十几个单元，而在比较复杂的大型操作系统中，PCB 则可能占用数十甚至数百个单元。但是不管哪一种情况，PCB 一般都应包含如下信息：

总之，每个进程基本上有自己独立的代码和数据空间，独立的程序计数器等上下文环境，进程切换的开销是比较大的。

1.1.2 线程

进程具备并发性的特点，这种并发性是不同的进程之间反映出来的，不同的进程有不同进程空间，进程之间的切换消耗比较大。那么就考虑到引入线程的概念，在进程的内部引入并发性，一个进程可以创建多个线程，线程之间具备并发性。不同的线程之间可以共享进程的地址空间和数据。

一般的讲，线程是一个程序，或者进程内部的一个顺序控制流。线程本身不能独立运行，必须在进程中执行，使用进程的地址空间。每个线程有自己单独的程序计数器。

一个进程内部包含多个顺序控制流，或者并发执行多种运算，就是多线程。

每个程序执行时都会产生一个进程，而每一个进程至少要有一个**主线程**。这个线程其实是进程执行的一条**线索**（Thread），除了主线程外你还可以给进程增加其它的线程，也即增加其它的执行线索，由此在某种程度上可以看成是给一个应用程序增加了多任务功能。当程序运行后，您可以根据各种条件挂起或运行这些线程，尤其在多 CPU 的环境中，这些线程是可以并发或者并行运行的。

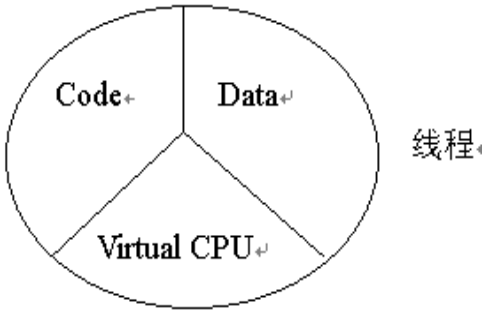
多线程就是在一个进程内有多个线程。从而使一个应用程序有了多任务的功能。有人会问：多进程技术不是也可以实现这一点吗？但是创建进程的高消耗（每个进程都有独立的数据和代码空间），进程之间通信的不方便（消息机制），进程切换的时间太长，这些导致了多线程的提出。对于单 CPU 来说（没有开启超线程），在同一时间只能执行一个线程，所以如果想实现多任务，那么就只能每个进程或线程获得一个时间片，在某个时间片内，只能一个线程执行，然后按照某种策略换其他线程执行。由于时间片很短，这样给用户的感觉是同时有好多线程在执行。但是线程切换是有代价的，因此如果采用多进程，那么就需要将线程所

隶属的该进程所需要的内存进行切换，这时间代价是很多的。而线程切换代价就很少，线程是可以共享内存的。所以采用多线程在切换上花费的比多进程少得多。但是，线程切换还是需要时间消耗的。所以采用一个拥有两个线程的进程执行所需要的时间比一个线程的进程执行两次所需要的时间要多一些。即采用多线程不会提高程序的执行速度，反而会降低速度，但是对于用户来说，可以减少用户的响应时间。上述结果只是针对单 CPU，如果对于多 CPU 或者 CPU 采用超线程技术的话，采用多线程技术还是会提高程序的执行速度的。因为单线程只会映射到一个 CPU 上，而多线程会映射到多个 CPU 上，超线程技术本质是多线程硬件化，所以也会加快程序的执行速度。

总之，进程内的同一类线程可以共享代码和数据空间，每个线程有独立的运行栈和程序计数器，切换的开销比较小，灵活性高。在支持超线程和多核的 CPU 上，多线程能够并发或者并行执行，可以在同一时间段内完成不同的任务，或者加快程序的执行。同一进程内的多个线程，调度比较灵活，可以相互协调和协作共同完成特定任务，

1.2 创建多线程

在 Java 中创建多线程是一件非常简单的事情。Java 定义了一个线程的概念模型，把一个线程分为三部分：虚拟 CPU（java.lang.Thread 类），虚拟 CPU 执行的代码和数据。



创建一个 java.lang.Thread 的对象，就意味着创建了一个线程。一个由 main 方法开始执行的 Java 程序，至少包含一个线程，即主线程。创建多个 Thread 的对象，就创建了多个线程。

Thread 类通过其 run（）方法来完成起任务，方法 run()为线程体。一般在 java 中有两种比较典型的构造线程的方法：1）继承 Thread 类，重写 run（）方法；2）把线程体从 Thread 类中独立出来，形成单独的线程目标对象，就是实现 Runnable 接口及其 run（）方法。

这两种方法都是通过 Thread 类的 start()方法启动线程的。

JDK5.0 提供了创建线程池并执行线程的方法。

1.2.1 继承 Thread 创建线程

继承 java.lang.Thread 类创建线程是最简单的一种方法，也最直接。下面创建一个 MyThread1 类，继承 Thread，重写其 run()方法。并在 main()方法中创建多个并发线程。

```
package simplethread;
public class MyThread1 extends Thread {
    public MyThread1(String name) {
        super(name);//传递线程的名字
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        for (int i = 0; i < 5; i++) { //创建5个线程
            new MyThread1("thread" + i).start();
        }
    }
    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {//输出线程名字和i
            System.out.println(this.getName() + ":" + i);
        }
    }
}
```

这种创建方式，把线程执行的逻辑代码直接写在了 Thread 的子类中，这样根据线程的概念模型，虚拟 CPU 和代码混合在一起了。并且 java 是单继承机制，线程体继承 Thread 类后，就不能继承其他类了，线程的扩展受影响。

1.2.2 实现 Runnable 接口创建线程

为了构建结构清晰线程程序，可以把代码独立出来形成线程目标对象，然后传给 Thread 对象。通常，实现 Runnable 接口的类创建的对象，称作线程的目标对象。下面创建一个类 MyThread2 实现 Runnable 接口，然后创建线程目标对象，传递给虚拟的 CPU。

```
package simplethread;
public class MyThreadTarget implements Runnable {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            //创建线程目标对象
```

```
            Runnable r = new MyThreadTarget();
            //把目标对象传递个Thread，即虚拟的cpu
            new Thread(r, "thread" + i).start();
        }
    }
    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            System.out.println(Thread.currentThread().getName() + ":"
+ i);
        }
    }
}
```

从程序中可以看出线程目标对象和 Thread 分开了，并传递给了 Thread。如果有比较复杂的数据要处理，可以在线程目标对象中引入数据。使用这种方式获得线程的名字就稍微复杂一些，需要使用到 Thread 中的静态方法，获得当前线程对象，然后再调用 getName（）方法。

这种方式在较复杂的程序中用的比较普遍。

1.2.3 线程池

线程有时称为轻量级进程。与进程一样，它们拥有通过程序运行的独立的并发路径，并且每个线程都有自己的程序计数器，称为堆栈和本地变量。然而，线程存在于进程中，它们与同一进程内的其他线程共享内存、文件句柄以及每进程状态。

一个进程中的线程是在同一个地址空间中执行的，所以多个线程可以同时访问相同对象，并且它们从同一堆栈中分配对象。

创建线程会使用相当一部分内存，其中包括有堆栈，以及每线程数据结构。**如果创建过多线程，其中每个线程都将占用一些 CPU 时间，结果将使用许多内存来支持大量线程，每个线程都运行得很慢。这样就无法很好地使用计算资源。**

Java 自从 5.0 以来，提供了线程池。线程的目标执行对象可以共享线程池中有限书目的线程对象。

一般的服务器都需要线程池，比如 Web、FTP 等服务器，不过它们一般都自己实现了线程池，比如 Tomcat、Resin 和 Jetty 等，现在 JDK 本身提供了，我们就没有必要重复造车轮了，直接使用就可以，何况使用也很方便，性能也非常高。

下面是使用线程池创建的多线程程序，100 个线程目标对象共享 2 个线程。

```
package pool;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class TestThreadPool {
    public static void main(String args[]) throws InterruptedException {
        // 在线程池中创建 2 个线程
        ExecutorService exec = Executors.newFixedThreadPool(2);
        // 创建 100 个线程目标对象
        for (int index = 0; index < 100; index++) {
            Runnable run = new Runner(index);
            // 执行线程目标对象
            exec.execute(run);
        }
        // shutdown
        exec.shutdown();
    }
}
// 线程目标对象
class Runner implements Runnable {
    int index = 0;
    public Runner(int index) {
        this.index = index;
    }
    @Override
    public void run() {
        long time = (long) (Math.random() * 1000);
        // 输出线程的名字和使用目标对象及休眠的时间
        System.out.println("线程： " + Thread.currentThread().getName() + "(目标对象"
                + index + ")" + ":Sleeping " + time + "ms");
```

```
                try {
                    Thread.sleep(time);
                } catch (InterruptedException e) {
                }
            }
        }
```

执行结果的片段如下：

```
线程：pool-1-thread-1(目标对象 23):Sleeping 938ms
线程：pool-1-thread-2(目标对象 24):Sleeping 352ms
线程：pool-1-thread-2(目标对象 25):Sleeping 875ms
线程：pool-1-thread-1(目标对象 26):Sleeping 607ms
线程：pool-1-thread-1(目标对象 27):Sleeping 543ms
线程：pool-1-thread-2(目标对象 28):Sleeping 520ms
线程：pool-1-thread-1(目标对象 29):Sleeping 509ms
线程：pool-1-thread-2(目标对象 30):Sleeping 292ms
```

从执行结果可以看出，线程池中只生成了两个线程对象，100 个线程目标对象共享他们。

从程序中可以看出，使用 JDK 提供的线程池一般分为 3 步：1）创建线程目标对象，可以是不同的，例如程序中的 Runnner；2）使用 Executors 创建线程池，返回一个 ExecutorService 类型的对象；3）使用线程池执行线程目标对象，exec.execute(run)，最后，结束线程池中的线程，exec.shutdown()。

API：java.util.concurrent.Executors extends Object

该类主要定义了一些工厂方法和工具方法，其中最重要的就是创建各种线程池。

1) public static ExecutorService newFixedThreadPool(int nThreads)

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程，在需要时使用提供的 ThreadFactory 创建新线程。在任意点，在大多数 nThreads 线程会处于处理任务的活动状态。如果在所有线程处于活动状态时提交附加任务，则在有可用线程之前，附加任务将在队列中等待。如果在关闭前的执行期间由于失败而导致任何线程终止，那么一个新线程将代替它执行后续的任务（如果需要）。在某个线程被显式地关闭之前，池中的线程将一直存在。

2) public static ThreadFactory defaultThreadFactory()

返回用于创建新线程的默认线程工厂。此工厂创建同一个线程组(ThreadGroup)中 Executor 使用的所有新线程。如果有 SecurityManager，则它使用 System.getSecurityManager() 返回的组，其他情况则使用调用 defaultThreadFactory 方法的组。每个新线程都作为非守护程序而创建，并且具有设置线程优先级为 Thread.NORM_PRIORITY 与线程组中允许的最大优先级的较小者。新线程具有可通过 *pool-N-thread-M* 的 Thread.getName() 来访问的名称，其中 *N* 是此工厂的序列号，*M* 是此工厂所创建线程的序列号。

3) public static ExecutorService newCachedThreadPool()

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。注意，可以使用 ThreadPoolExecutor 构造方法创建具有类似属性但细节不同（例如超时参数）的线程池。

4) public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

5) void execute(Runnable command)

在未来某个时间执行给定的命令。该命令可能在新的线程、已入池的线程或者正调用的线程中执行，这由 Executor 实现决定。

6) void shutdown()

启动一次顺序关闭，执行以前提交的任务，但不接受新任务。如果已经关闭，则调用没有其他作用。

1.3 线程的基本控制

线程创建后，可以执行 start（）方法启动线程，根据线程任务的特性和线程之间的协调性要求，需要对线程进行控制。对线程的控制通常是通过调用 Thread 对象的方法实现的，主要有 sleep（）、suspend（）、resume（）、join（）、interrupt（）和 stop 方法。一般情况下方法的调用会引起线程状态的转变。

1.3.1 使用 Sleep 暂停执行

Thread.sleep()使当前线程的执行暂停一段指定的时间，这可以有效的使应用程序的其他线程或者运行在计算机上的其他进程可以使用处理器时间。该方法不会放弃除 CPU 之外的其它资源。

Sleep 有两个重载的版本，一个以毫秒指定睡眠时间，另一个以纳秒指定睡眠时间，但并不保证这些睡眠时间的精确性，因为他们受到系统计时器和调度程序精度和准确性的影响。另外中断（interrupt）可以终止睡眠时间，在任何情况下，都不能假设调用 sleep 就会按照指定的时间精确的挂起线程。

```
package control;
public class SleepTest {
    public static void main(String[] arg) {
        String[] args = { "one", "two", "three", "for" };
        long start=System.nanoTime();
        for (int i = 0; i < args.length; i++) {
            try {
                System.out.println(args[i]);
                // 休眠主线程
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        long end=System.nanoTime();
        System.out.println("总的时间: "+(end-start)/1000000);
    }
}
```

需要注意的是，sleep（）方法声明可以抛出 InterruptedException 异常，当另一个线程中断了已经启动 sleep 的当前线程时机会抛出这个异常。上面的程序只有主线程，不需要考虑这个问题。

1.3.2 使用 join 等待另外一个线程结束

Join 方法让一个线程等待另一个线程的完成，如果 t1，t2 是两个 Thread 对象，在 t1 中调用 t2.join（），会导致 t1 线程暂停执行，直到 t2 的线程终止。Join 的重载版本允许程序员指定等待的时间，但是和 sleep 一样，这个时间是不精确的。

```
package control;
public class JoinTest extends Thread {
    static int result = 0;
    public JoinTest(String name) {
        super(name);
    }
    public static void main(String[] args) {
        System.out.println("主线程执行");
        Thread t = new JoinTest("计算线程");
        t.start();
        System.out.println("result: " + result);
        try {
            long start = System.nanoTime();
            t.join();
            long end = System.nanoTime();
            System.out.println((end - start) / 1000000 + "毫秒后:" +
result);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void run() {
        System.out.println(this.getName() + "开始计算...");
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        result = (int) (Math.random() * 10000);
        System.out.println(this.getName() + "结束计算: ");
    }
}
```

执行结果如下：

```
主线程执行
result：0
计算线程开始计算...
计算线程结束计算：
4000 毫秒后:6155
```

上面的程序中，计算线程在计算的时候休眠了 4000 毫秒，在主线程中调用了 t.join（）

后，主线程等待计算线程执行结束，然后输出结果。

可以把 t.join()修改为 t.join(2000)。观察输出结果，发现主线程并没有等待计算线程执行结束，就输出结果了。

```
主线程执行
result：0
计算线程开始计算...
1999 毫秒后:0
计算线程结束计算：
```

1.3.3 使用中断(Interrupt)取消线程

已经启动的线程是活跃的，即 isAlive()方法返回 true，线程终止之前一直是活跃的。有三种方法可以使线程终止：1）run()方法正常返回；2）run（）方法意外结束；3）应用程序终止。

经常会碰到这样的情况，我们创建了执行某项工作的线程，然后在他完成之前需要取消这项工作。要使线程在完成任务之前可取消，必须采取一定的措施，但应该是一个清晰而安全的机制使线程终止。

我们可以通过**中断**（Thread.interrupt）线程来请求取消，并且让线程来监视并响应中断。中断请求通常是用户希望能够终止线程的执行，但并不会强制终止线程，但是它会中断线程的睡眠状态，比如调用 sleep 和 wait 方法后。

线程自己检查中断状态并终止线程比直接调用 stop()放要安全很多，因为线程可以保存自己的状态。并且 stop()方法已经不推荐使用了。

和中断线程有关的方法有：1）interrupt，向线程发送中断，2）isInterrupted，测试线程是否已经被中断；3）Interrupted，测试当前线程是否已经被中断，随后清楚线程“中断”状态的静态方法。

线程的中断状态只能有线程自己清除，当线程侦测到自己被中断时，经常需要在响应中断之前做某些清除工作，这些清除工作可能涉及那些在线程仍然保持中断状态时会受到影响的操作。

如果被中断的线程正在执行 sleep，或者 wait 方法，就会抛出 InterruptedException 异常。

这种抛出异常的中断会清除线程的中断状态。

大体上任何执行阻塞操作的方法，都应该通过 Interrupt 来取消阻塞操作。

下面的程序，主线程在等待计算线程 2000 毫秒后，中断计算线程，计算线程由于正在执行 sleep，就会抛出 InterruptedException 异常，终止休眠状态，然后进入异常处理，在 catch 中可以做一些清理工作（如果需要），然后线程执行结束。

这是一种典型的终止线程执行的方法。

```
package control;
public class InterruptTest extends Thread {
    static int result = 0;
    public InterruptTest(String name) {
        super(name);
    }
    public static void main(String[] args) {
        System.out.println("主线程执行");
        Thread t = new InterruptTest("计算线程");
        t.start();
        System.out.println("result: " + result);
        try {
            long start = System.nanoTime();
            t.join(2000);
            long end = System.nanoTime();
            t.interrupt();
            System.out.println((end - start) / 1000000 + "毫秒后:" +
result);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void run() {
        System.out.println(this.getName() + "开始计算...");
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            System.out.println(this.getName()+"被中断,结束");
            return;
        }
        result = (int) (Math.random() * 10000);
        System.out.println(this.getName() + "结束计算");
    }
}
```

下面是输出结果

```
主线程执行
result：0
计算线程开始计算...
1999 毫秒后:0
计算线程被中断,结束
```

从输出结果中可以看出，计算线程被中断后，run()方法中的最后两行语句没有执行。没有产生计算结果。

如果一个线程长时间没有调用能够抛出 InterruptedException 异常的方法，那么线程就必须定期的调用 Thread.interrupted 方法，如果接收到中断就返回 true，然后就可以退出线程。

```
package control;
public class InterruptTest2 extends Thread {
    static int result = 0;
    public InterruptTest2(String name) {
        super(name);
    }
    public static void main(String[] args) {
        System.out.println("主线程执行");
        Thread t = new InterruptTest2("计算线程");
        t.start();
        System.out.println("result: " + result);
        try {
            long start = System.nanoTime();
            t.join(10);
            long end = System.nanoTime();
            t.interrupt();
            System.out.println((end - start) / 1000000 + "毫秒后:" +
result);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void run() {
        System.out.println(this.getName() + "开始计算...");
        for (int i = 0; i < 100000; i++) {
            result++;
            if (Thread.interrupted()) {
                System.out.println(this.getName() + "被中断");
```

```
                return;
            }
        }
        System.out.println(this.getName() + "结束计算");
    }
}
```

输出结果如下：

```
主线程执行

result：0

计算线程开始计算...

计算线程被中断

10 毫秒后:18555
```

上面的程序，计算线程原计划执行 100000 次循环，主线程等待 10 毫秒后，中断计算线程，计算线程接收到中断后，就可以结束执行了。

在更加复杂的应用程序中，当线程收到中断信号后，抛出 InterruptedException 异常可能更有意义。把中断处理代码集中在 catch 子句中。

```
if (Thread.interrupted()) {
                System.out.println(this.getName() + "被中断");
                throw new InterruptedException();
}
```

1.3.4 使用 Stop 终止线程

在 Thread 类中提供了 Stop 方法了强迫线程停止执行。但是现在已经过时了。

该方法具有固有的不安全性。用 Thread.stop 来终止线程将释放它已经锁定的所有监视器（作为沿堆栈向上传播的未检查 ThreadDeath 异常的一个自然后果）。如果以前受这些监视器保护的任何对象都处于一种不一致的状态，则损坏的对象将对其他线程可见，这有可能导致任意的行为。**stop 的许多使用方式都应由只修改某些变量以指示目标线程应该停止运行的代码来取代。目标线程应定期检查该变量，并且如果该变量指示它要停止运行，则从其运行方法依次返回。如果目标线程等待很长时间（例如基于一个条件变量），则应使用 interrupt 方法来中断该等待。**

有关更多信息，请参阅“为何不赞成使用 Thread.stop、Thread.suspend 和 Thread.resume？”

（JDK 文档中的 docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html）。

无论该线程在做些什么，它所代表的线程都被迫异常停止，并抛出一个新创建的 ThreadDeath 对象作为异常。停止一个尚未启动的线程是允许的。如果最后启动了该线程，它会立即终止。

应用程序通常不应试图捕获 ThreadDeath，除非它必须执行某些异常的清除操作（注意，抛出 ThreadDeath 将导致 try 语句的 finally 子句在线程正式终止前执行）。如果 catch 子句捕获了一个 ThreadDeath 对象，则重新抛出该对象很重要，因为这样该线程才会真正终止。

对其他未捕获的异常作出反应的顶级错误处理程序不会打印输出消息，或者另外通知应用程序未捕获到的异常是否为 ThreadDeath 的一个实例。

1.3.5 结束程序的执行

每个应用程序都从执行 main 的线程开始的，如果应用程序没有创建任何其他的线程，那么 main 方法返回时，应用程序就结束了，但是如果应用程序创建了其他线程，就要根据线程的类型分情况来考虑了。

线程一般分为两种：用户线程和守护线程。用户线程的存在可以使应用程序保持运行状态，而守护线程则不会。当最后一个用户线程结束时，所有守护线程都会被终止，应用程序也随之结束。守护线程的终止，很像调用 destroy 所产生的终止，事发突然，没有机会做任何清楚，所以应该考虑清楚，用守护线程执行哪种类型的任务。使用 Thread.setDaemon(true) 可以把线程标记为守护线程。默认情况下，线程的守护状态继承自创建它的线程。

一般 main 线程是程序运行时第一个启动的线程，称作初始线程。如果希望应用程序在初始线程消亡后就退出，就可以把所有创建出来的线程都标记为守护线程。

我们也可以通过调用 System，或者 Runtime 的 exit 方法来强制应用程序结束，这个方法将终止 Java 虚拟机的当前执行过程。

许多类会隐式的在应用程序中创建线程，比如图形用户界面，并创建了特殊的线程来处理事件。有些是守护线程，有些不是。如果没有更好的办法，那么就可以用 exit 方法。

1.4 并发编程实践简述

掌握了如何创建多线程的 Java 程序后，在后面的章节详细讲解如何创建线程安全的并发程序。

学习解决并发编程中遇到的常用问题，比如同步、互斥、死锁等；学习如何使用 JDK 提供的线程构造块创建并发程序；学习使用 Amino 开源软件提供的有关编写高效并发程序的数据结构、算法和调度模式；学习如何使用开源软件 MTRAT 诊断数据冲突；学习如何使用显示锁代替内在锁；学习如何使用原子量和无锁数据结构构建并发程序。

Linux公社（LinuxIDC.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

LinuxIDC.com提供包括Ubuntu，Fedora，SUSE技术，以及最新IT资讯等Linux专业类网站。

并被收录到Google 网页目录-计算机 > 软件 > 操作系统 > Linux 目录下。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

包括:

Ubuntu专题

Fedora专题

RedHat专题

SUSE专题

红旗Linux专题

Android专题