Java 基础提升篇：深入浅出 Java 多线程

初遇

Java 给多线程编程提供了内置的支持。一个多线程程序包含两个或多个能并发运行的部分。程序的每一部分都称作一个线程，并且每个线程定义了一个独立的执行路径。

多线程是多任务的一种特别的形式，但多线程使用了更小的资源开销。

这里定义和线程相关的另一个术语 - **进程：一个进程包括由操作系统分配的内存空间，包含一个或多个线程。一个线程不能独立的存在，它必须是进程的一部分。一个进程一直运行，直到所有的非守候线程都结束运行后才能结束。**

多线程能满足程序员编写高效率的程序来达到充分利用 CPU 的目的。

1.多线程基础概念介绍

进程是程序（任务）的执行过程，它持有资源（共享内存，共享文件）和线程。

分析：

1) **执行过程是动态性的**，你放在电脑磁盘上的某个 eclipse 或者 QQ 文件并不是我们的进程，只有当你双击运行可执行文件，使 eclipse 或者 QQ 运行之后，这才称为进程。它是一个执行过程，是一个动态的概念。
2) **它持有资源（共享内存，共享文件）和线程：我们说进程是资源的载体**，也是线程的载体。这里的资源可以理解为内存。我们知道程序是要从内存中读取数据进行运行的，所以每个进程获得执行的时候会被分配一个内存。
3) 线程是什么？

如果我们把进程比作一个班级，那么班级中的每个学生可以将它视作一个线程。学生是班级中的最小单元，构成了班级中的最小单位。一个班级有可以多个学生，这些学生都使用共同的桌椅、书籍以及黑板等等进行学习和生活。

在这个意义上我们说：

线程是系统中最小的执行单元；同一进程中可以有多个线程；线程共享进程的资源。

4) 线程是如何交互？

就如同一个班级中的多个学生一样，我们说多个线程需要通信才能正确的工作，

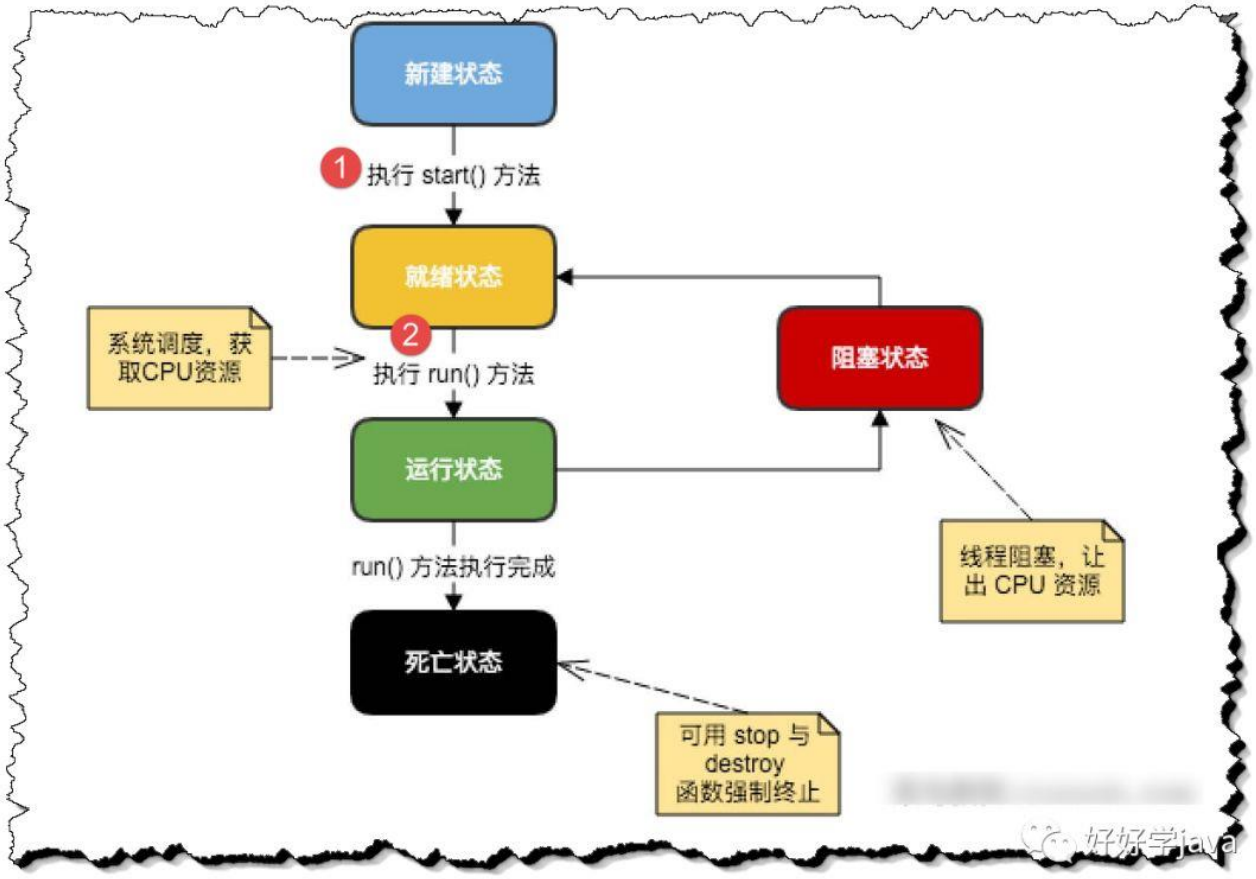
这种通信，我们称作**线程的交互。**

5) 交互的方式：互斥、同步

类比班级，就是在同一班级之内，同学之间通过相互的协作才能完成某些任务，有时这种协作是需要竞争的，比如学习，班级之内公共的学习资料是有限的，爱学习的同学需要抢占它，需要竞争，当一个同学使用完了之后另一个同学才可以使用；如果一个同学正在使用，那么其他新来的同学只能等待；另一方面需要同步协作，就好比班级六一需要排演节目，同学需要齐心协力相互配合才能将节目演好，这就是进程交互。

一个线程的生命周期

线程经过其生命周期的各个阶段。下图显示了一个线程完整的生命周期。



- **新建状态:**

使用 new 关键字和 Thread 类或其子类建立一个线程对象后，该线程对象就处于新建状态。它保持这个状态直到程序 start() 这个线程。

- **就绪状态:**

当线程对象调用了 start()方法之后，该线程就进入就绪状态。就绪状态的线程处于就绪队列中，要等待 JVM 里线程调度器的调度。

- **运行状态:**

如果就绪状态的线程获取 CPU 资源，就可以执行 run()，此时线程便处于运行状态。处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状态和死亡状态。

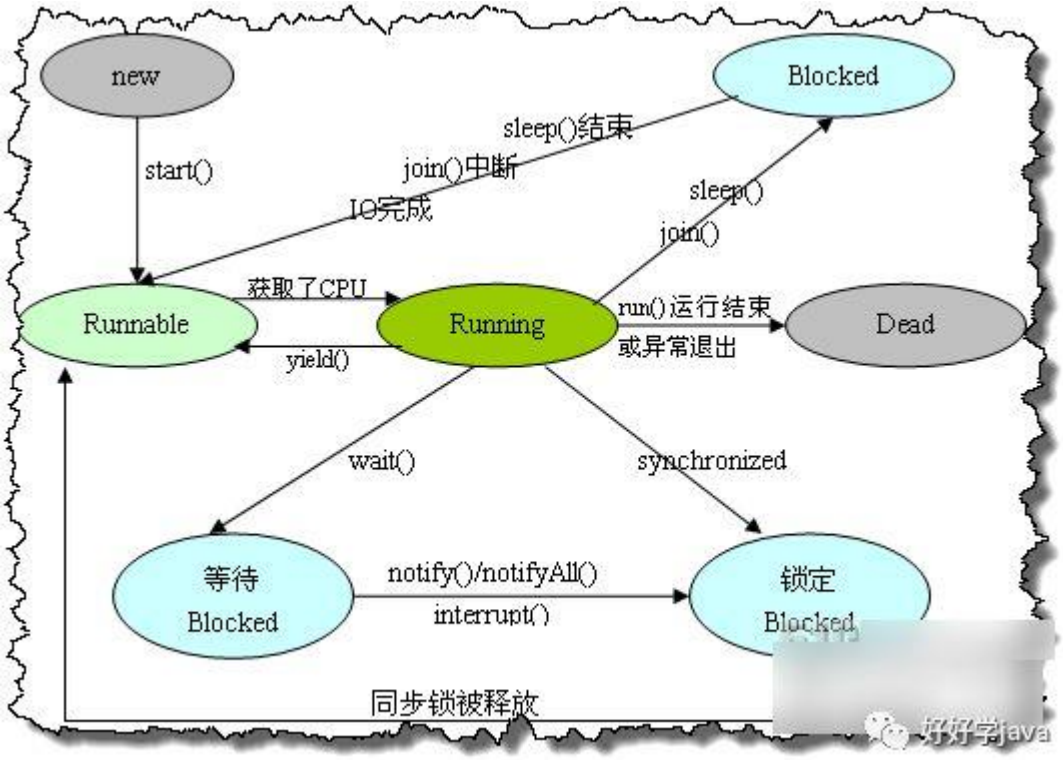
- **阻塞状态:**

如果一个线程执行了 sleep（睡眠）、suspend（挂起）等方法，失去所占用资源之后，该线程就从运行状态进入阻塞状态。在睡眠时间已到或获得设备资源后可以重新进入就绪状态。

- **死亡状态:**

一个运行状态的线程完成任务或者其他终止条件发生时，该线程就切换到终止状态。

线程的状态转换图



1、新建状态（New）: 新创建了一个线程对象。

2、就绪状态（Runnable）: 线程对象创建后，其他线程调用了该对象的 start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取 CPU 的使用权。

3、运行状态（Running）: 就绪状态的线程获取了 CPU，执行程序代码。

4、阻塞状态（Blocked）: 阻塞状态是线程因为某种原因放弃 CPU 使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

- 等待阻塞：运行的线程执行 wait()方法，JVM 会把该线程放入等待池中。
- 同步阻塞 :运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线程放入锁池中。
- 其他阻塞：运行的线程执行 sleep()或 join()方法，或者发出了 I/O 请求时，JVM 会把该线程置为阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。

5、死亡状态（Dead）: 线程执行完了或者因异常退出了 run()方法，该线程结束生命周期。

线程的调度

1、调整线程优先级：

每一个 Java 线程都有一个优先级，这样有助于操作系统确定线程的调度顺序。Java 线程的优先级用整数表示，取值范围是 1~10，Thread 类有以下三个静态常量：

```
static int MAX_PRIORITY
    线程可以具有的最高优先级，取值为 10。

static int MIN_PRIORITY
    线程可以具有的最低优先级，取值为 1。

static int NORM_PRIORITY
    分配给线程的默认优先级，取值为 5。
```

Thread 类的 setPriority()和 getPriority()方法分别用来设置和获取线程的优先级。

每个线程都有默认的优先级。主线程的默认优先级为 Thread.NORM_PRIORITY。

线程的优先级有继承关系，比如 A 线程中创建了 B 线程，那么 B 将和 A 具有相同的优先级。

JVM 提供了 10 个线程优先级，但与常见的操作系统都不能很好的映射。如果希望程序能移植到各个操作系统中，应该仅仅使用 Thread 类有以下三个静态常量作为优先级，这样能保证同样的优先级采用了同样的调度方式。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器资源。但是，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

2、线程睡眠

Thread.sleep(long millis)方法，使线程转到阻塞状态。millis 参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。

3、线程等待

Object 类中的 wait()方法，导致当前的线程等待，直到其他线程调用此对象的 notify()方法或 notifyAll() 唤醒方法。这个两个唤醒方法也是 Object 类中的方法，行为等价于调用 wait(0) 一样。

4、线程让步

Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

5、线程加入

join()方法，等待其他线程终止。在当前线程中调用另一个线程的 join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

6、线程唤醒

Object 类中的 notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发

生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。类似的方法还有一个 notifyAll() ,唤醒在此对象监视器上等待的所有线程。

注意：Thread 中 suspend()和 resume()两个方法在 JDK1.5 中已经废除，不再介绍。因为有死锁倾向。

7、常见线程名词解释

主线程：JVM 调用程序 main()所产生的线程。

当前线程 这个是容易混淆的概念。一般指通过 Thread.currentThread()来获取的进程。

后台线程：指为其他线程提供服务的线程，也称为守护线程。JVM 的垃圾回收线程就是一个后台线程。

前台线程：是指接受后台线程服务的线程，其实前台后台线程是联系在一起，就像傀儡和幕后操纵者一样的关系。傀儡是前台线程、幕后操纵者是后台线程。由前台线程创建的线程默认也是前台线程。可以通过 isDaemon()和 setDaemon()方法来判断和设置一个线程是否为后台线程。

一些常见问题

1) 线程的名字，一个运行中的线程总是有名字的，名字有两个来源，一个是虚拟机自己给的名字，一个是你自己的定的名字。在没有指定线程名字的情况下，虚拟机总会为线程指定名字，并且主线程的名字总是 main，非主线程的名字不确定。
2) 线程都可以设置名字，也可以获取线程的名字，连主线程也不例外。
3) 获取当前线程的对象的方法是：Thread.currentThread()；
4) 每个线程都将启动，每个线程都将运行直到完成。一系列线程以某种顺序启动并不意味着将按该顺序执行。对于任何一组启动的线程来说，调度程序不能保证其执行次序，持续时间也无法保证。
5) 当线程目标 run()方法结束时该线程完成。
6) 一旦线程启动，它就永远不能再重新启动。只有一个新的线程可以被启动，并且只能一次。一个可运行的线程或死线程可以被重新启动。
7) 线程的调度是 JVM 的一部分，在一个 CPU 的机器上，实际上一次只能运行一个线程。一次只有一个线程栈执行。JVM 线程调度程序决定实际运行哪个处于可运行状态的线程。

   众多可运行线程中的某一个会被选中做为当前线程。可运行线程被选择运行的顺序是没有保障的。
8) 尽管通常采用队列形式，但这是没有保障的。队列形式是指当一个线程完成“一轮”时，它移到可运行队列的尾部等待，直到它最终排队到该队列的前端为止，它才能被再次选中。事实上，我们把它称为可运行池而不是一个可运行队列，目的是帮助认识线程并不都是以某种有保障的顺序排列唱呢个一个队列的事实。
9) 尽管我们没有无法控制线程调度程序，但可以通过别的方式来影响线程调度的方式。

2. Java 中线程的常用方法介绍

Java 语言对线程的支持

主要体现在 Thread 类和 Runnable 接口上，都继承于 java.lang 包。它们都有个共同的方法：public void run()。

run 方法为我们提供了线程实际工作执行的代码。

下表列出了 Thread 类的一些重要方法：

序号	方法描述
1	public void start()使该线程开始执行；Java 虚拟机调用该线程的 run 方法。
2	public void run()如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。
3	public final void setName(String name)改变线程名称，使之与参数 name 相同。
4	public final void setPriority(int priority)更改线程的优先级。
5	public final void setDaemon(boolean on)将该线程标记为守护线程或用户线程。
6	public final void join(long millisec)等待该线程终止的时间最长为 millis 毫秒。
7	public void interrupt()中断线程。
8	public final boolean isAlive()测试线程是否处于活动状态。

测试线程是否处于活动状态。上述方法是被 Thread 对象调用的。下面的方法是 Thread 类的静态方法。

序号	方法描述
1	public static void yield()暂停当前正在执行的线程对象，并执行其他线程。
2	public static void sleep(long millisec)在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。
3	public static boolean holdsLock(Object x)当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。
4	public static Thread currentThread()返回对当前正在执行的线程对象的引用。
5	public static void dumpStack()将当前线程的堆栈跟踪打印至标准错误流。

Thread 常用的方法

类别	方法签名	简介
线程的创建	Thread()	
	Thread(String name)	
	Thread(Runnable target)	
	Thread(Runnable target, String name)	
线程的方法	void start()	启动线程
	static void sleep(long millis)	线程休眠
	static void sleep(long millis, int nanos)	
	void join()	使其他线程等待当前线程终止
	void join(long millis)	
	void join(long millis, int nanos)	
	static void yield()	当前运行线程释放处理器资源
获取线程引用	static Thread currentThread()	返回当前运行的线程引用

3. 线程初体验（编码示例）

创建线程的方法有两种：

1) 继承 Thread 类本身
2) 实现 Runnable 接口

线程中的方法比较有特点，比如:启动（start），休眠（sleep），停止等，多个线程是交互执行的（cpu 在某个时刻。只能执行一个线程，当一个线程休眠了或者执行完毕了，另一个线程才能占用 cpu 来执行）因为这是 cpu 的结构来决定的，在某个时刻 cpu 只能执行一个线程，不过速度相当快，对于人来将可以认为是并行执行的。

在一个 java 文件中，可以有多个类（此处说的是外部类），但只能有一个 public 类。

这两种创建线程的方法本质没有任何的不同，一个是实现 Runnable 接口，一个是继承 Thread 类。

使用实现 Runnable 接口这种方法：

1) 可以避免 java 的单继承的特性带来的局限性；
2) 适合多个相同程序的代码去处理同一个资源情况，把线程同程序的代码及数据有效的分离，较好的体现了面向对象的设计思想。开发中大多数情况下都使用实现 Runnable 接口这种方法创建线程。

实现 Runnable 接口创建的线程最终还是要通过将自身实例作为参数传递给 Thread 然后执行。

```
语法： Thread actress=new Thread(Runnable target ,String name);
```

例如：

```
Thread actressThread=new Thread(new Actress(),"Ms.runnable");
```

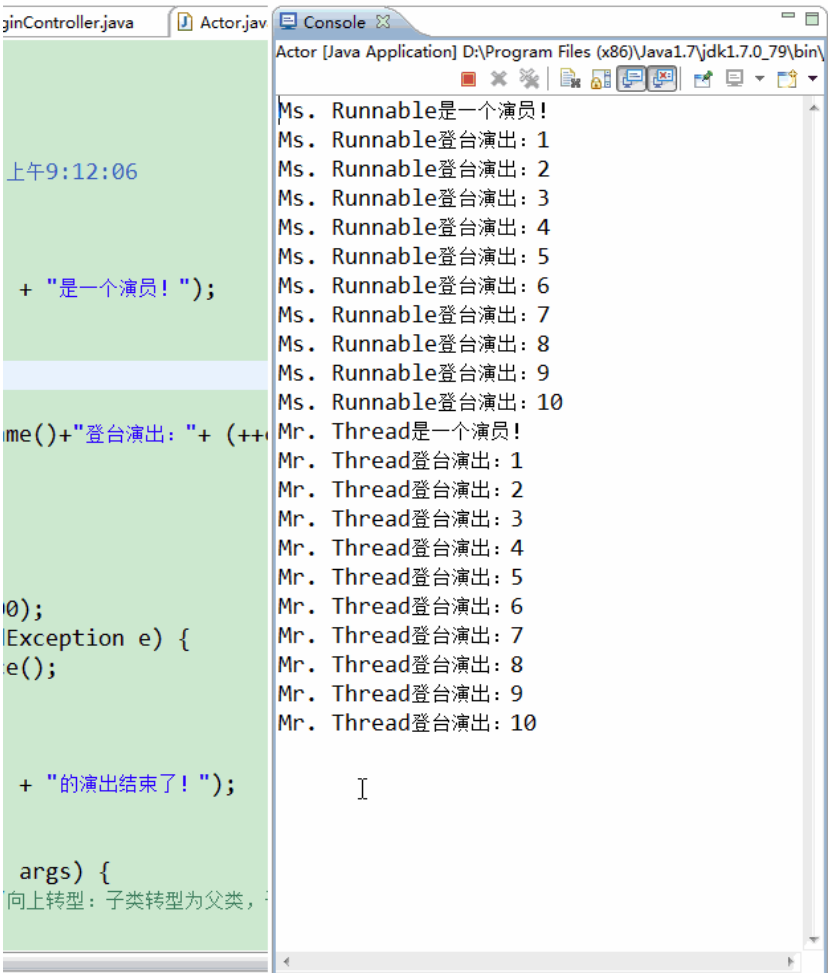
```
actressThread.start();
```

代码示例：

```
public class Actor extends Thread{
    public void run() {
        System.out.println(getName() + "是一个演员！");
        int count = 0;
        boolean keepRunning = true;
        while(keepRunning){
            System.out.println(getName()+"登台演出："+ (++count));
            if(count == 100){
                keepRunning = false;
            }
            if(count%10== 0){
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        System.out.println(getName() + "的演出结束了！");
    }
    public static void main(String[] args) {
       Thread actor = new Actor();//向上转型：子类转型为父类，子类对象就会遗失和父类不同的方法。向上转型符合 Java 提倡的面向抽象编程思想，还可以减轻编程工作量
       actor.setName("Mr. Thread");
       actor.start();
       //调用 Thread 的构造函数 Thread(Runnable target, String name)
       Thread actressThread = new Thread(new Actress(), "Ms. Runnable");
       actressThread.start();
    }
}
//注意：在“xx.java”文件中可以有多个类，但是只能有一个 Public 类。这里所说的不是内部类，都是一个个独立的外部类
class Actress implements Runnable{
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "是一个演员！");//Runnable 没有 getName()方法，需要通过线程的 currentThread()方法获得线程名称
        int count = 0;
```

```
        boolean keepRunning = true;
        while(keepRunning){
            System.out.println(Thread.currentThread().getName()+"登台演出："+
(++count));
            if(count == 100){
                keepRunning = false;
            }
            if(count%10== 0){
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        System.out.println(Thread.currentThread().getName() + "的演出结束了！");
    }
}
/**
 *运行结果 Mr. Thread 线程和 Ms. Runnable 线程是交替执行的情况
 *分析：计算机 CPU 处理器在同一时间同一个处理器同一个核只能运行一条线程，
 *当一条线程休眠之后，另外一个线程才获得处理器时间
 */
```

运行结果：



示例 2：

ArmyRunnable 类：

```
/**
 * 军队线程
 * 模拟作战双方的行为
 */
public class ArmyRunnable implements Runnable {
    /* volatile 关键字
     * volatile 保证了线程可以正确的读取其他线程写入的值
     * 如果不写成 volatile，由于可见性的问题，当前线程有可能不能读到这个值
     * 关于可见性的问题可以参考 JMM（Java 内存模型），里面讲述了：happens-before 原则、可见性
     * 用 volatile 修饰的变量，线程在每次使用变量的时候，都会读取变量修改后的值
     */
    volatile boolean keepRunning = true;
    @Override
    public void run() {
        while (keepRunning) {
            //发动 5 连击
            for(int i=0;i<5;i++){
```

```
                System.out.println(Thread.currentThread().getName()+"进攻对方
["+i+"]");
                //让出了处理器时间，下次该谁进攻还不一定呢！
                Thread.yield();//yield()当前运行线程释放处理器资源
            }
        }
        System.out.println(Thread.currentThread().getName()+"结束了战斗！");
    }
}
```

KeyPersonThread 类：

```
public class KeyPersonThread extends Thread {
    public void run(){
        System.out.println(Thread.currentThread().getName()+"开始了战斗！");
        for(int i=0;i<10;i++){
            System.out.println(Thread.currentThread().getName()+"左突右杀,攻击隋
军...");
        }
        System.out.println(Thread.currentThread().getName()+"结束了战斗！");
    }
}
```

Stage 类：

```
/**
 * 隋唐演义大戏舞台 6  */
public class Stage extends Thread {
    public void run(){
        System.out.println("欢迎观看隋唐演义");
        //让观众们安静片刻，等待大戏上演
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
        System.out.println("大幕徐徐拉开");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
        System.out.println("话说隋朝末年，隋军与农民起义军杀得昏天黑地...");
        ArmyRunnable armyTaskOfSuiDynasty = new ArmyRunnable();
```

```
        ArmyRunnable armyTaskOfRevolt = new ArmyRunnable();
        //使用 Runnable 接口创建线程
        Thread  armyOfSuiDynasty = new Thread(armyTaskOfSuiDynasty,"隋军");
        Thread  armyOfRevolt = new Thread(armyTaskOfRevolt,"农民起义军");
        //启动线程，让军队开始作战
        armyOfSuiDynasty.start();
        armyOfRevolt.start();
        //舞台线程休眠，大家专心观看军队厮杀
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("正当双方激战正酣，半路杀出了个程咬金");
        Thread  mrCheng = new KeyPersonThread();
        mrCheng.setName("程咬金");
        System.out.println("程咬金的理想就是结束战争，使百姓安居乐业！");
        //停止军队作战
        //停止线程的方法
        armyTaskOfSuiDynasty.keepRunning = false;
        armyTaskOfRevolt.keepRunning = false;
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        /*
         * 历史大戏留给关键人物
         */
        mrCheng.start();
        //万众瞩目，所有线程等待程先生完成历史使命
        try {
            mrCheng.join();//join()使其他线程等待当前线程终止
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("战争结束，人民安居乐业，程先生实现了积极的人生梦想，为人民
作出了贡献！");
        System.out.println("谢谢观看隋唐演义，再见！");
    }
```

```
    public static void main(String[] args) {
        new Stage().start();
    }
}
```

4. Java 线程的正确停止

如何正确的停止 Java 中的线程？

stop 方法：该方法使线程戛然而止（突然停止），完成了哪些工作，哪些工作还没有做都不清楚，且清理工作也没有做。

stop 方法不是正确的停止线程方法。线程停止不推荐使用 stop 方法。

正确的方法---设置退出标志

使用 volatile 定义 boolean running=true,通过设置标志变量 running，来结束线程。如本文:volatile boolean keepRunning=true;

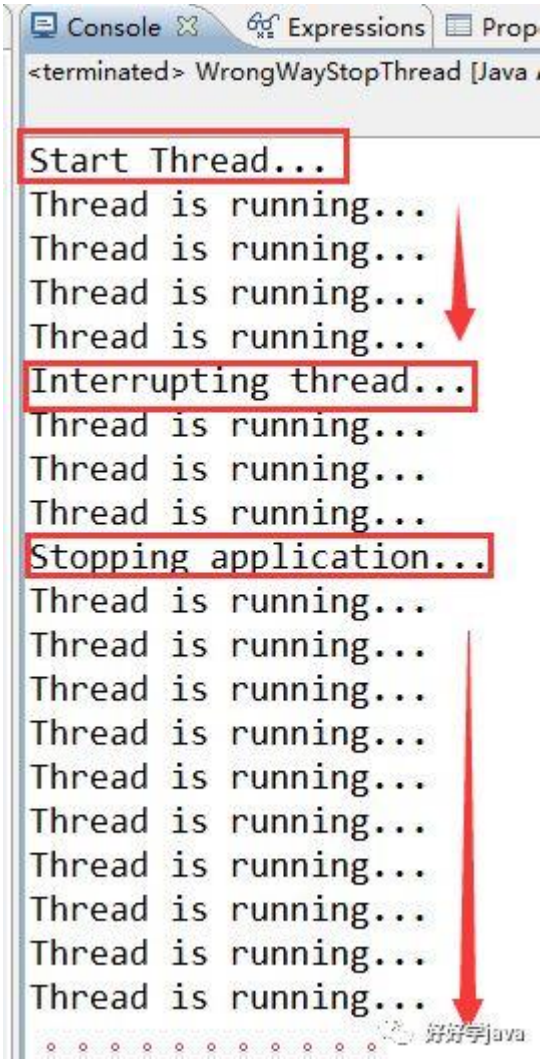
这样做的好处是：使得线程有机会使得一个完整的业务步骤被完整地执行，在执行完业务步骤后有充分的时间去做代码的清理工作，使得线程代码在实际中更安全。

广为流传的错误方法---interrupt 方法

当一个线程运行时，另一个线程可以调用对应的 Thread 对象的 interrupt()方法来中断它，该方法只是在目标线程中设置一个标志，表示它已经被中断，并立即返回。这里需要注意的是 ,如果只是单纯的调用 interrupt()方法 ,线程并没有实际被中断 ,会继续往下执行。代码示例：

```
/**
 * 错误终止进程的方式—interrupt
 */
public class WrongWayStopThread extends Thread {
    public static void main(String[] args) {
        WrongWayStopThread thread = new WrongWayStopThread();
        System.out.println("Start Thread...");
        thread.start();
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Interrupting thread...");
        thread.interrupt();
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Stopping application...");
    }
    public void run() {
        while(true){
            System.out.println("Thread is running...");
            long time = System.currentTimeMillis();
            while ((System.currentTimeMillis()-time) <1000) {//这部分的作用大致相当于 Thread.sleep(1000)，注意此处为什么没有使用休眠的方法
                //减少屏幕输出的空循环（使得每秒钟只输出一行信息）
            }
        }
    }
}
```

运行结果：



由结果看到 interrupt()方法并没有使线程中断，线程还是会继续往下执行。

Java API 中介绍：

interrupt

`public void interrupt()`

Interrupts this thread.

Unless the current thread is interrupting itself, which is always permitted, the `checkAccess` method of this thread is invoked, which may cause a `SecurityException` to be thrown.

If this thread is blocked in an invocation of the `wait()`, `wait(long)`, or `wait(long, int)` methods of the `Object` class, or of the `join()`, `join(long)`, `join(long, int)`, `sleep(long)`, or `sleep(long, int)`, methods of this class, then its interrupt status will be cleared and it will receive an `InterruptedException`.

If this thread is blocked in an I/O operation upon an `InterruptibleChannel` then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a `ClosedByInterruptException`.

If this thread is blocked in a `Selector` then the thread's interrupt status will be set and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector's `wakeup` method were invoked.

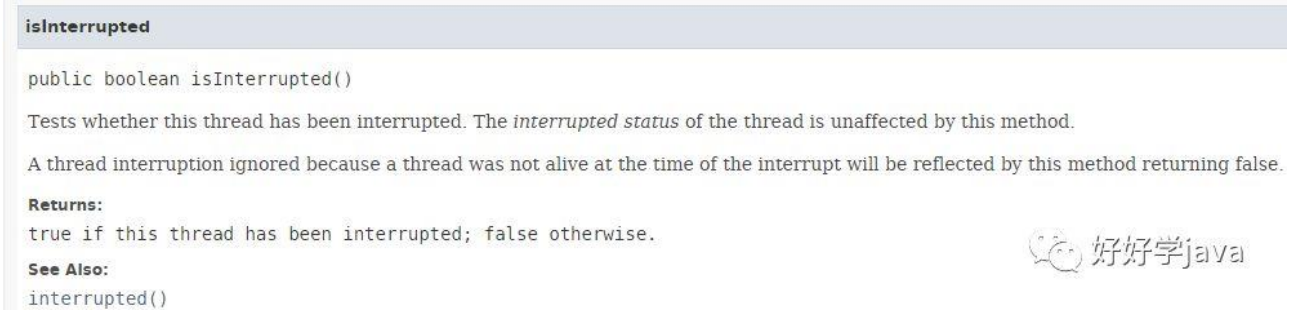
If none of the previous conditions hold then this thread's interrupt status will be set.

Interrupting a thread that is not alive need not have any effect.

Throws:

`SecurityException` - if the current thread cannot modify this thread

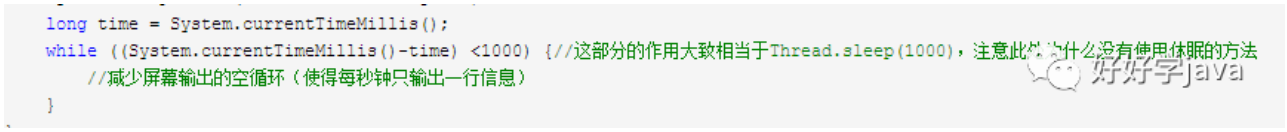
但是 interrupt()方法可以使我们的中断状态发生改变，可以调用 isInterrupted 方法



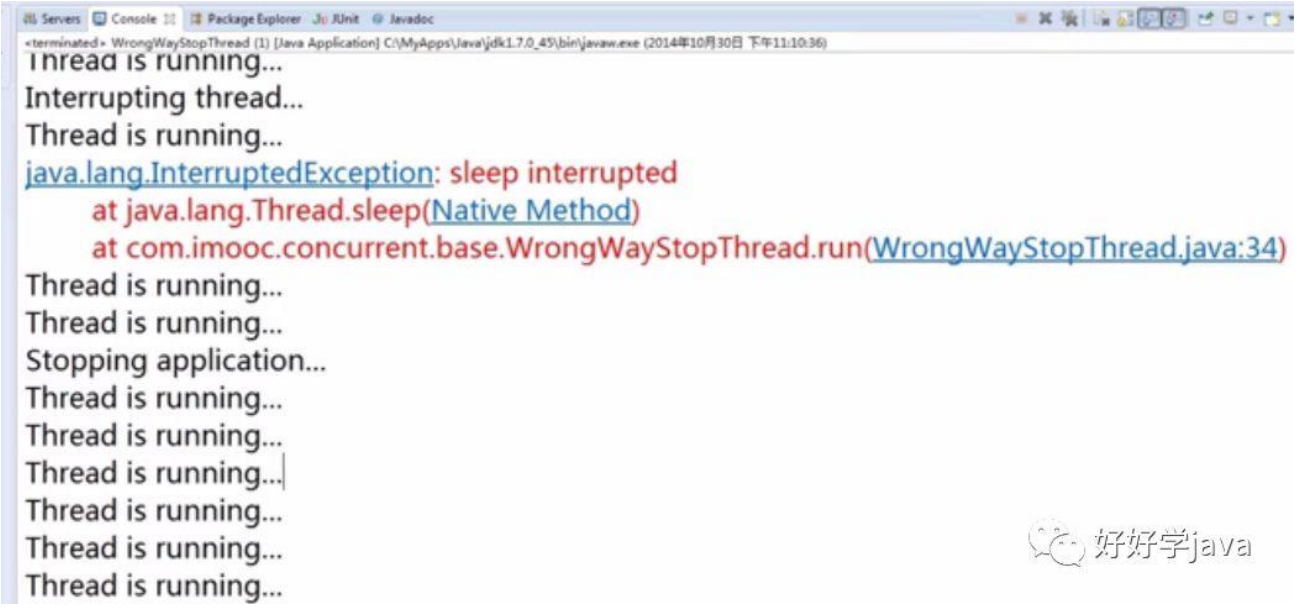
将上处 run 方法代码改为下面一样，程序就可以正常结束了。

```
public void run() {
        while(!this.isInterrupted()){//interrupt()可以使中断状态放生改变，调用isInterrupted()
            System.out.println("Thread is running...");
            long time = System.currentTimeMillis();
            while ((System.currentTimeMillis()-time) <1000) {//这部分的作用大致相当于Thread.sleep(1000)，注意此处为什么没有使用休眠的方法
                //减少屏幕输出的空循环（使得每秒钟只输出一行信息）
            }
        }
    }
```

但是这种所使用的退出方法实质上还是前面说的使用退出旗标的方法，不过这里所使用的退出旗标是一个特殊的标志“线程是否被中断的状态”。



这部分代码相当于线程休眠 1 秒钟的代码。但是为什么没有使用 Thread.sleep(1000)。如果采用这种方法就会出现



线程没有正常结束，而且还抛出了一个异常，异常抛出位置在调用 interrupt 方法之后。为什么会有这种结果？

在 API 文档中说过：如果线程由于调用的某些方法（比如 sleep，join。。。）而进入一种

阻塞状态时，此时如果这个线程再被调用 interrupt 方法，它会产生两个结果：第一，它的中断状态被清除 clear，而不是被设置 set。那 isInterrupted 就不能返回是否被中断的正确状态，那 while 函数就不能正确的退出。第二，sleep 方法会收到 InterruptedException 被中断。

interrupt()方法只能设置 interrupt 标志位（且在线程阻塞情况下，标志位会被清除，更无法设置中断标志位），无法停止线程。

5. 线程交互

争用条件：

1) 当多个线程同时共享访问同一数据（内存区域）时，每个线程都尝试操作该数据，从而导致数据被破坏（corrupted)，这种现象称为争用条件
2) 原因是，每个线程在操作数据时，会先将数据初值读【取到自己获得的内存中】，然后在内存中进行运算后，重新赋值到数据。
3) 争用条件：线程 1 在还【未重新将值赋回去时】，线程 1 阻塞，线程 2 开始访问该数据，然后进行了修改，之后被阻塞的线程 1 再获得资源，而将之前计算的值覆盖掉线程 2 所修改的值，就出现了数据丢失情况。

互斥与同步：守恒的能量

1. 线程的特点，共享同一进程的资源，同一时刻只能有一个线程占用 CPU
2. 由于线程有如上的特点，所以就会存在多个线程争抢资源的现象，就会存在争用条件这种现象
3. 为了让线程能够正确的运行，不破坏共享的数据，所以，就产生了同步和互斥的两种线程运行的机制
4. 线程的互斥（加锁实现）：**线程的运行隔离开来，互不影响，使用 synchronized 关键字实现互斥行为**，此关键字即可以出现在方法体之上也可以出现在方法体内，以一种块的形式出现，在此代码块中有线程的等待和唤醒动作，用于支持线程的同步控制
5. 线程的同步（线程的等待和唤醒：wait()+notifyAll()）：**线程的运行有相互的通信控制，运行完一个再正确的运行另一个**
6. 锁的概念：比如 private final Object lockObj=new Object();
7. 互斥实现方式：synchronized 关键字
   synchronized(lockObj){---执行代码----}加锁操作
   lockObj.wait();线程进入等待状态，以避免线程持续申请锁，而不去竞争 cpu 资源
   lockObj.notifyAll();唤醒所有 lockObj 对象上等待的线程
8. 加锁操作会开销系统资源，降低效率

同步问题提出

线程的同步是为了防止多个线程访问一个数据对象时，对数据造成的破坏。

例如：两个线程 ThreadA、ThreadB 都操作同一个对象 Foo 对象，并修改 Foo 对象上的数据。

```
public class Foo {
    private int x = 100;
    public int getX() {
        return x;
    }
    public int fix(int y) {
        x = x - y;
        return x;
    }
}
public class MyRunnable implements Runnable {
    private Foo foo = new Foo();
    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread ta = new Thread(r, "Thread-A");
        Thread tb = new Thread(r, "Thread-B");
        ta.start();
        tb.start();
    }
    public void run() {
        for (int i = 0; i < 3; i++) {
            this.fix(30);
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + " : 当前 foo 对象的 x 值= " + foo.getX());
        }
    }
    public int fix(int y) {
        return foo.fix(y);
    }
```

```
}
```

运行结果：

```
Thread-A : 当前 foo 对象的 x 值= 40
Thread-B : 当前 foo 对象的 x 值= 40
Thread-B : 当前 foo 对象的 x 值= -20
Thread-A : 当前 foo 对象的 x 值= -50
Thread-A : 当前 foo 对象的 x 值= -80
Thread-B : 当前 foo 对象的 x 值= -80
Process finished with exit code 0
```

从结果发现，这样的输出值明显是不合理的。原因是两个线程不加控制的访问 Foo 对象并修改其数据所致。

如果要保持结果的合理性，只需要达到一个目的，就是将对 Foo 的访问加以限制，每次只能有一个线程在访问。这样就能保证 Foo 对象中数据的合理性了。

在具体的 Java 代码中需要完成一下两个操作：

- 把竞争访问的资源类 Foo 变量 x 标识为 private；
- 同步哪些修改变量的代码，使用 synchronized 关键字同步方法或代码。

同步和锁定

1. 锁的原理

Java 中每个对象都有一个内置锁

当程序运行到非静态的 synchronized 同步方法上时，自动获得与正在执行代码类的当前实例（this 实例）有关的锁。获得一个对象的锁也称为获取锁、锁定对象、在对象上锁定或在对象上同步。

当程序运行到 synchronized 同步方法或代码块时才该对象锁才起作用。

一个对象只有一个锁。所以，如果一个线程获得该锁，就没有其他线程可以获得锁，直到第一个线程释放（或返回）锁。这也意味着任何其他线程都不能进入该对象上的 synchronized 方法或代码块，直到该锁被释放。

释放锁是指持锁线程退出了 synchronized 同步方法或代码块。

关于锁和同步，有一下几个要点：

1) 只能同步方法，而不能同步变量和类；
2) 每个对象只有一个锁；当提到同步时，应该清楚在什么上同步？也就是说，在哪个对象上同步？
3) 不必同步类中所有的方法，类可以同时拥有同步和非同步方法。
4) 如果两个线程要执行一个类中的 synchronized 方法，并且两个线程使用相同的实例来调用方法，那么一次只能有一个线程能够执行方法，另一个需要等待，直到锁被释放。也就是说：如果一个线程在对象上获得一个锁，就没有任何其他线程可以进入（该对象的）类中的任何一个同步方法。
5) 如果线程拥有同步和非同步方法，则非同步方法可以被多个线程自由访问而不受锁

的限制。

6) 线程睡眠时，它所持的任何锁都不会释放。
7) 线程可以获得多个锁。比如，在一个对象的同步方法里面调用另外一个对象的同步方法，则获取了两个对象的同步锁。
8) 同步损害并发性，应该尽可能缩小同步范围。同步不但可以同步整个方法，还可以同步方法中一部分代码块。
9) 在使用同步代码块时候，应该指定在哪个对象上同步，也就是说要获取哪个对象的锁。

例如：

```
public int fix(int y) {
        synchronized (this) {
            x = x - y;
        }
        return x;
}
```

当然，同步方法也可以改写为非同步方法，但功能完全一样的。
例如：

```
public synchronized int getX() {
        return x++;
}
```

与

```
public int getX() {
        synchronized (this) {
            return x;
        }
}
```

效果是完全一样的。

静态方法同步

要同步静态方法，需要一个用于整个类对象的锁，这个对象是就是这个类（XXX.class)。
例如：

```
public static synchronized int setName(String name){
      Xxx.name = name;
}
```

等价于

```
public static int setName(String name){
      synchronized(Xxx.class){
```

```
            Xxx.name = name;
        }
}
```

线程同步小结

1) 线程同步的目的是为了保护多个线程访问一个资源时对资源的破坏。
2) 线程同步方法是通过锁来实现，每个对象都有切仅有一个锁，这个锁与一个特定的对象关联，线程一旦获取了对象锁，其他访问该对象的线程就无法再访问该对象的其他同步方法。
3) 对于静态同步方法，锁是针对这个类的，锁对象是该类的 Class 对象。静态和非静态方法的锁互不干预。一个线程获得锁，当在一个同步方法中访问另外对象上的同步方法时，会获取这两个对象锁。
4) 对于同步，要时刻清醒在哪个对象上同步，这是关键。
5) 编写线程安全的类，需要时刻注意对多个线程竞争访问资源的逻辑和安全做出正确的判断，对“原子”操作做出分析，并保证原子操作期间别的线程无法访问竞争资源。
6) 当多个线程等待一个对象锁时，没有获取到锁的线程将发生阻塞。
7) 死锁是线程间相互等待锁锁造成的，在实际中发生的概率非常的小。真让你写个死锁程序，不一定好使，呵呵。但是，一旦程序发生死锁，程序将死掉。

深入剖析互斥与同步

互斥的实现(加锁) :synchronized(lockObj); 保证的同一时间 ,只有一个线程获得 lockObj.
同步的实现：wait()/notify()/notifyAll()
注意：wait()、notify()、notifyAll()方法均属于 Object 对象，而不是 Thread 对象。

- void notify()：唤醒在此对象监视器上等待的单个线程。
- void notifyAll()：唤醒在此对象监视器上等待的所有线程。
- void wait()：导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。

当然，wait()还有另外两个重载方法：

- void wait(long timeout)

导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量。

- void wait(long timeout, int nanos)

导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量。

notify()唤醒 wait set 中的一条线程，而 notifyall()唤醒所有线程。

同步是两个线程之间的一种交互的操作（一个线程发出消息另外一个线程响应）

关于等待/通知，要记住的关键点是：

必须从同步环境内调用 wait()、notify()、notifyAll()方法。线程不能调用对象上等待或通知的方法，除非它拥有那个对象的锁。

wait()、notify()、notifyAll()都是 Object 的实例方法。与每个对象具有锁一样，每个对象可以有一个线程列表，他们等待来自该信号（通知）。线程通过执行对象上的 wait()方法获得这个等待列表。从那时候起，它不再执行任何其他指令，直到调用对象的 notify()方法为止。如果多个线程在同一个对象上等待，则将只选择一个线程（不保证以何种顺序）继续执行。如果没有线程等待，则不采取任何特殊操作。

下面看个例子就明白了：

```
/**
* 计算输出其他线程锁计算的数据
*/
public class ThreadA {
    public static void main(String[] args) {
        ThreadB b = new ThreadB();
        //启动计算线程
        b.start();
        //线程 A 拥有 b 对象上的锁。线程为了调用 wait()或 notify()方法，该线程必须是那个对象锁的拥有者
        synchronized (b) {
            try {
                System.out.println("等待对象 b 完成计算。。。");
                //当前线程 A 等待
                b.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("b 对象计算的总和是：" + b.total);
        }
    }
}
/**
* 计算 1+2+3 ... +100 的和
*/
public class ThreadB extends Thread {
    int total;
    public void run() {
        synchronized (this) {
            for (int i = 0; i < 101; i++) {
                total += i;
```

```
            }
            //（完成计算了）唤醒在此对象监视器上等待的单个线程，在本例中线程 A 被唤醒
            notify();
        }
    }
}
```

结果：

```
等待对象 b 完成计算。。。
b 对象计算的总和是：5050
Process finished with exit code 0
```

千万注意：

当在对象上调用 wait()方法时，执行该代码的线程立即放弃它在对象上的锁。然而调用 notify()时，并不意味着这时线程会放弃其锁。如果线程荣然在完成同步代码，则线程在移出之前不会放弃锁。因此，只要调用 notify()并不意味着这时该锁变得可用。

多个线程在等待一个对象锁时候使用 notifyAll()：

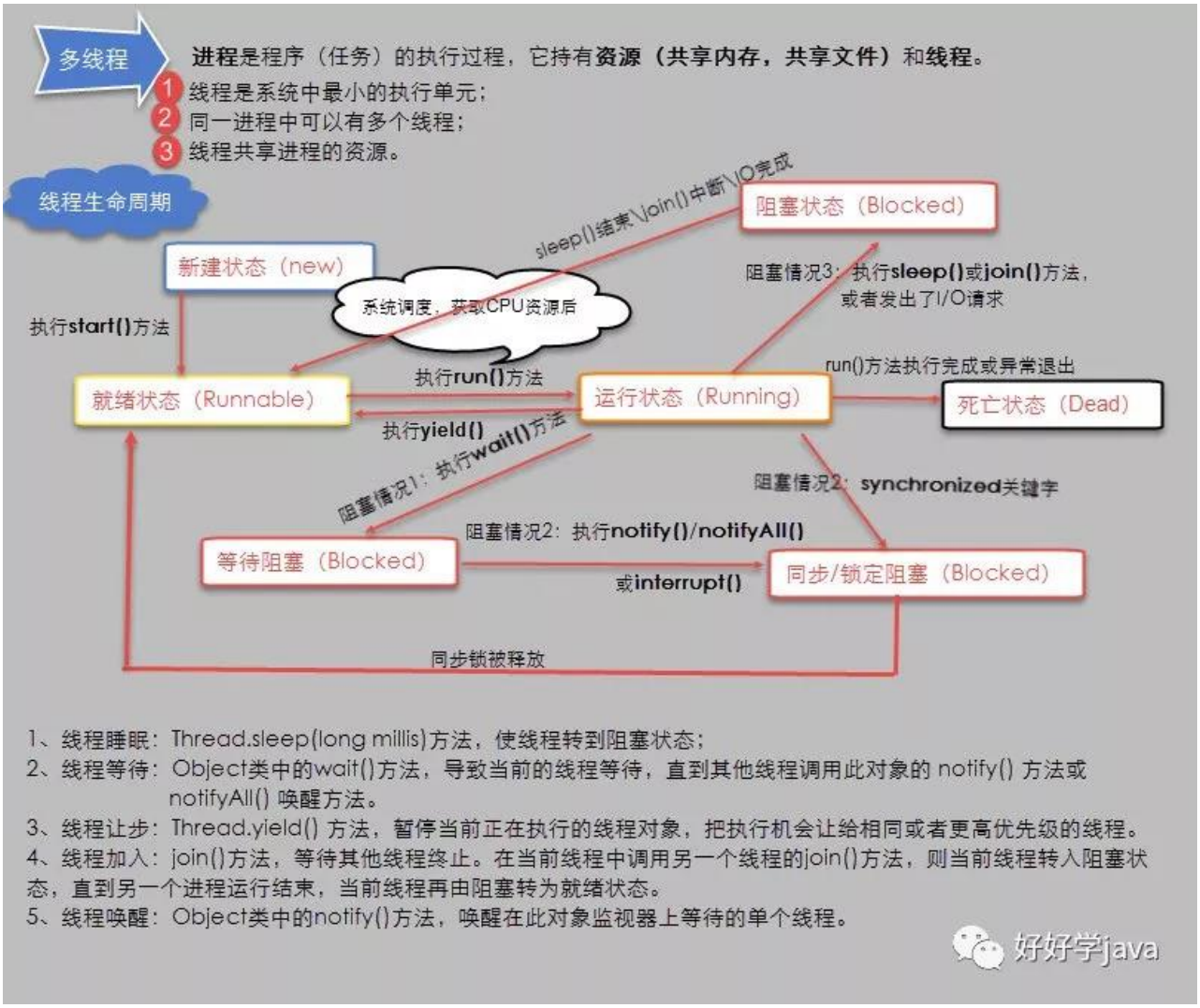
在多数情况下，最好通知等待某个对象的所有线程。如果这样做，可以在对象上使用 notifyAll()让所有在此对象上等待的线程冲出等待区，返回到可运行状态。

如何理解同步：Wait Set

Critical Section（临界资源）Wait Set（等待区域）

wait set 类似于线程的休息室，访问共享数据的代码称为 critical section。一个线程获取锁，然后进入临界区，发现某些条件不满足，然后调用锁对象上的 wait 方法，然后线程释放掉锁资源，进入锁对象上的 wait set。由于线程释放释放了理解资源，其他线程可以获取所资源，然后执行，完了以后调用 notify，通知锁对象上的等待线程。

Ps：若调用 notify();则随机拿出（这随机拿出是内部的算法，无需了解）一条在等待的资源进行准备进入 Critical Section；若调用 notifyAll();则全部取出进行准备进入 Critical Section。



扩展建议：如何扩展 Java 并发知识

1) Java Memory Mode：JMM 描述了 java 线程如何通过内存进行交互，了解 happens-before，synchronized,voliatile & final
2) Locks % Condition：Java 锁机制和等待条件的高层实现 java.util,concurrent.locks
3) 线程安全性：原子性与可见性， java.util.concurrent.atomic synchronized（锁的方法块）&volatile（定义公共资源） DeadLocks(死锁）--了解什么是死锁，死锁产生的条件
4) 多线程编程常用的交互模型
   · Producer-Consumer 模型（生产者-消费者模型）
   · Read-Write Lock 模型（读写锁模型）
   · Future 模型
   · Worker Thread 模型
   考虑在 Java 并发实现当中，有哪些类实现了这些模型，供我们直接调用
5) Java5 中并发编程工具：java.util.concurrent 包下的
   例如：线程池 ExcutorService 、Callable&Future 、BlockingQueue
6) 推荐书本：CoreJava 、 JavaConcurrency In Practice