

第十章 多线程

1 进程线程

进程

进程指一个内存中运行的应用程序，它是系统运行程序的基本单位。

一个程序从创建、运行到消亡，这样整个过程就是一个进程。

一个操作系统中可以同时运行多个进程，每个进程运行时，系统都会为其分配独立的内存空间。

任务管理器

文件(E) 选项(O) 查看(V)

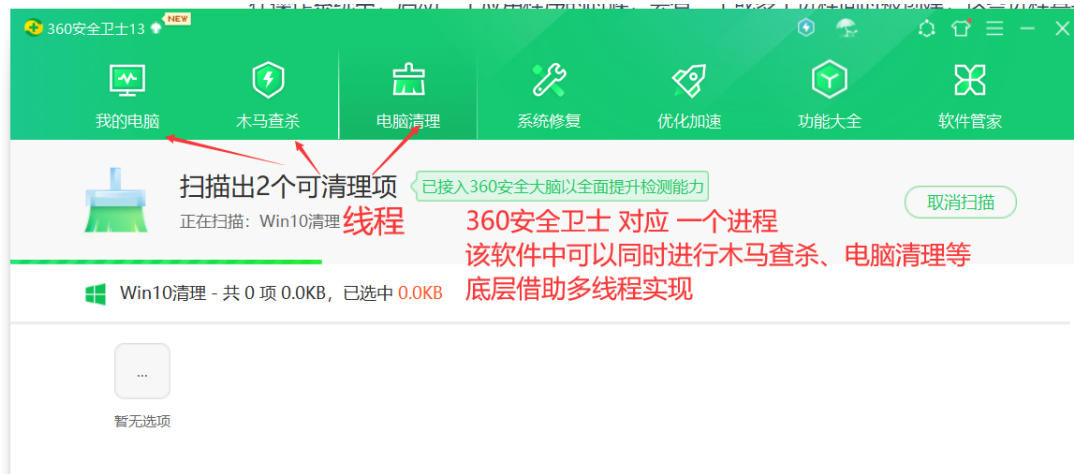
进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	3% CPU	40% 内存	2% 磁盘	0% 网络
应用 (13)				
> 360安全浏览器 (32 位)	0.1%	78.9 MB	0.1 MB/秒	0 Mbps
> Adobe Reader (32 位) (2)	0%	153.3 MB	0 MB/秒	0 Mbps
> EditPlus	0%	3.1 MB	0 MB/秒	0 Mbps
> Firefox	0%	81.0 MB	0 MB/秒	0 Mbps
> Foxmail 7.2 (32 位)	0.1%	20.7 MB	0 MB/秒	0 Mbps
> SpringToolSuite4.exe	0%	22.8 MB	0 MB/秒	0 Mbps
> TIM (32 位)	0.5%	33.5 MB	0 MB/秒	0 Mbps
> TIM (32 位)	0.2%	92.8 MB	0 MB/秒	0 Mbps

在操作系统中，启动一个应用程序的时候，会有一个或多个进程同时被创建，这些进程其实就表示了当前这个应用程序，在系统中的资源使用情况以及程序运行的情况。如果关闭这个进程，那么对应的应用程序也就关闭了。

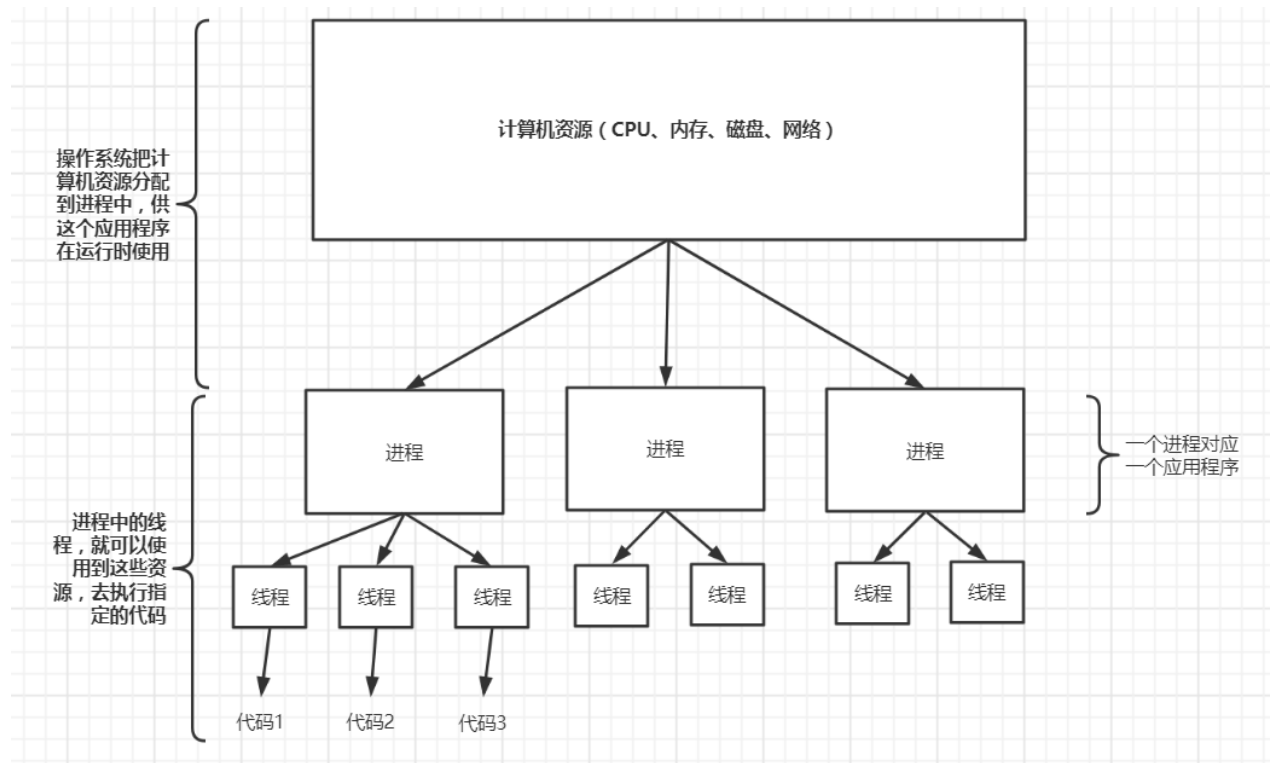
线程

线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程，一个进程中也可以有多个线程，此时这个应用程序就可以称之为多线程程序。



结论：一个程序运行后至少有一个进程，一个进程中可以包含一个(main线程)或多个线程！

当一个进程中启动了多个线程去分别执行代码（同时完成多个功能）的时候，这个程序就是多线程程序，内存等资源使用情况如下：



思考：JVM是多线程的吗？

是，JVM可以在运行程序的同时，进行GC垃圾回收，同一时刻做不同事情。

案例：

```
1 public class Test01_JVM {
2     public static void main(String[] args) throws
    InterruptedException {
3         for(int i = 1; i < 1000000; i++)
4             new Test(i);
5
6         //main线程休眠3s
7         Thread.sleep(3000);
8         System.err.println("main end...");
9     }
10 }
11
12 class Test {
13     int n;
```

```

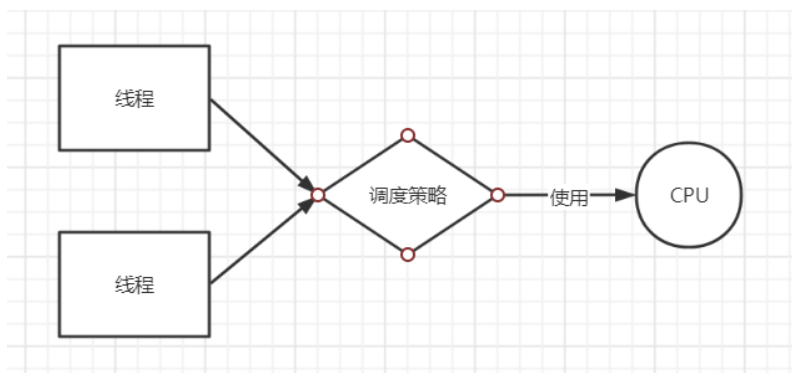
14     public Test(int n) {
15         this.n = n;
16     }
17
18     //当GC进行垃圾回收指定对象的时候，对象的finalize方法会被自动调用
19     @Override
20     protected void finalize() throws Throwable {
21         System.out.println("Test被销毁, n: " + n);
22         super.finalize();
23     }
24 }

```

2 并发并行

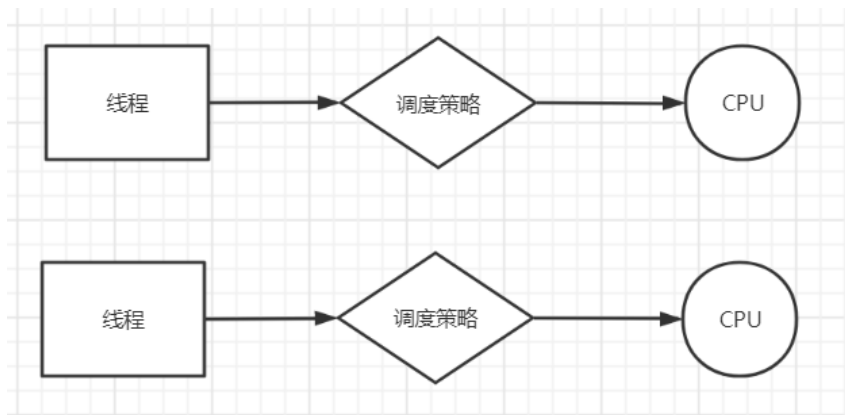
- **并发**：指两个或多个事件在**同一个时间段内**发生

线程的**并发**执行，是指在一个时间段内（微观），两个或多个线程，**使用同一个CPU交替运行**。



- **并行**：指两个或多个事件在**同一时刻**发生（同时发生）

线程的**并行**执行，是指在同一时刻，两个或多个线程，**各自使用一个CPU同时运行**。



如果计算机是单核CPU的话，那么同一时刻**只能**有一个线程使用CPU来执行代码

如果计算机是多核CPU的话，那么同一时刻**有可能**是两个线程同时使用不同的CPU执行代码

注意：当前章节讨论的是并发多线程。

补充内容：

如果我们的计算机是多核的，在程序中编写了两个线程，然后启动并运行它们，计算机会用一个CPU运行还是两个CPU去运行，我们无法知道也无法控制，因为计算机内核中有专门的资源调度算法负责资源的分配，我们从应用程序的层面无法干涉。

3 线程调度

时间片

并发多线程只有一个CPU，某个微观时刻，当指定线程拥有CPU的使用权，则该线程代码就可以执行，而其他线程阻塞等待。

一个线程不可能一直拥有CPU的使用权，不可能一直执行下去，它拥有CPU执行的时间是很短的，微秒纳秒级别，这个时间段我们就称之为CPU时间片。

线程执行时如果一个时间片结束了，则该线程就会停止运行，并交出CPU的使用权，然后等待下一个CPU时间片的分配。

在宏观上，一段时间内，我们感觉两个线程在同时运行代码，其实在微观中，这两个线程在使用一个CPU的时候，它们是**交替**着运行的，每个线程每次都是运行一个很小的时间片，然后就交出CPU使用权，只是它们两个交替运行的速度太快了，给我们的感觉，好像是它们两个线程在同时运行。

思考，生活中还有哪些是因为速度太快，从而通过我们的眼睛“欺骗”了我们的情况？

调度方式

当两个或多个线程使用一个CPU来运行代码的时候，在操作系统的内核中，就会有相应的算法来控制线程获取CPU时间片的方式，从而使得这些线程可以按照某种顺序来使用CPU运行代码，这种情况被称为线程调度。常见线程调度有：

- 时间片轮转

所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间。

- 抢占式调度

系统会让优先级高的线程优先使用 CPU（提高抢占到的概率），但是如果线程的优先级相同，那么会随机选择一个线程获取当前CPU的时间片。



JVM中的线程，使用的为抢占式调度。

4 线程创建

`java.lang.Thread` 是java中的线程类，所有线程对象都必须是Thread类或其子类的实例。

每个线程的作用，就是完成我们给它指定的任务，实际上就是执行一段我们指定的代码。我们只需要在 `Thread` 类的子类中重写 `run` 方法，完成相应的功能。

方法：

方法名	说明
<code>void run()</code>	在线程开启后，此方法将被调用执行
<code>void start()</code>	使此线程开始执行，Java虚拟机会调用run方法()

Java中通过继承Thread类来**创建并启动**一个新的线程的**步骤**如下：

- 定义 `Thread` 类的子类，重写 `run()` 方法，`run()` 方法中的代码就是线程的执行任务
- 创建 `Thread` 子类对象（**可以是匿名内部类对象**），这个对象就代表一个要独立运行的新线程
- 调用线程对象的 `start()` 方法来启动该线程

案例：

```
1  //1.子类继承父类Thread，并重写run方法（指定线程的执行任务）
2  class MyThread extends Thread {
3
4      //2.重写run方法
5      @Override
6      public void run() {
7          for (int i = 0; i < 10; i++) {
8              System.out.println("in run, hello thread");
9              try {
10                 //让当前执行代码的线程睡眠1000毫秒
11                 Thread.sleep(1000);
12             } catch (InterruptedException e) {
13                 e.printStackTrace();
14             }
15         }
16     }
17 }
18
19 public class Test04_Thread {
20     public static void main(String[] args) throws Exception {
21         //3.创建线程类对象
22         Thread t = new MyThread();
23         //4.调用start方法启动线程
24         t.start();
25     }
```

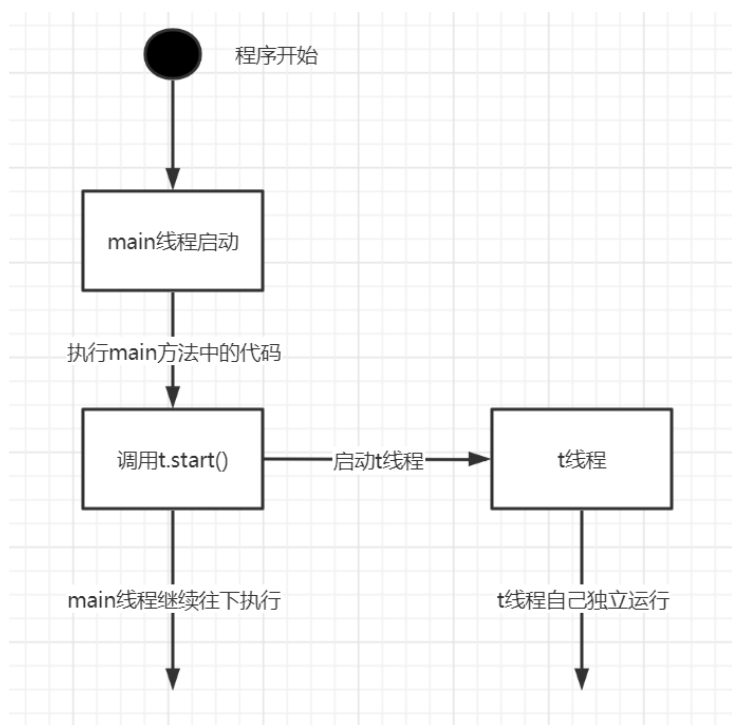


```

26         for (int i = 0; i < 10; i++) {
27             System.out.println("in main, hello");
28             //当前执行代码的线程睡眠500毫秒
29             Thread.sleep(500);
30         }
31     }
32 }

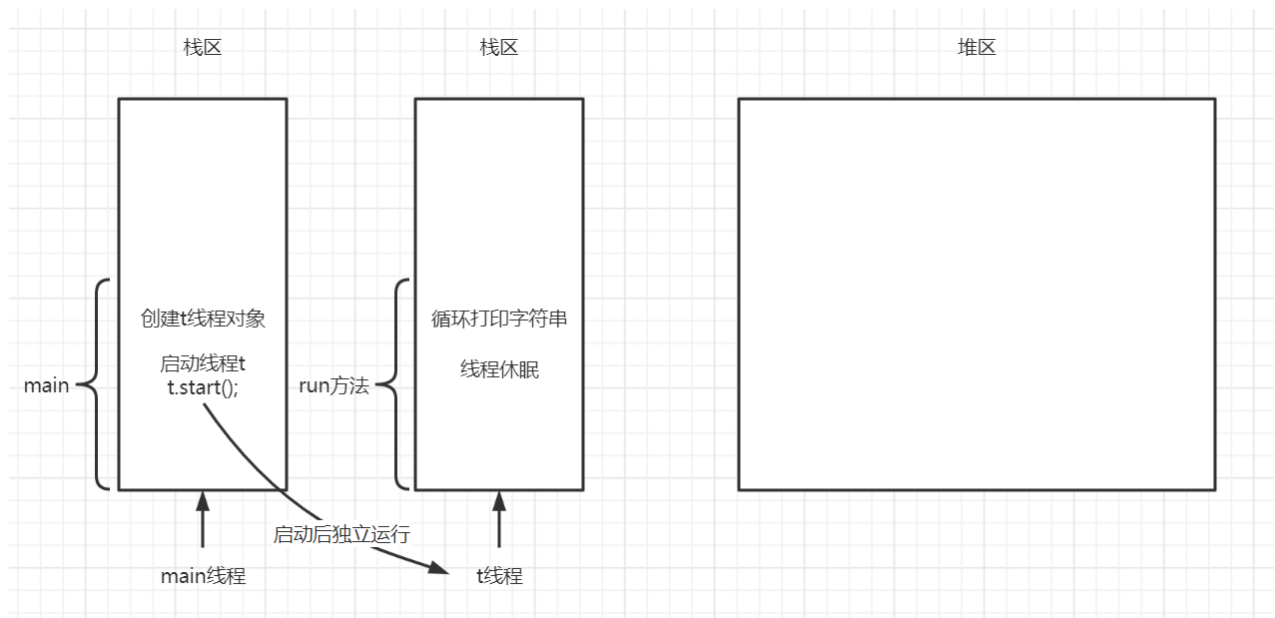
```

在此过程中，main线程和t线程之间的关系是：



可以看出，main线程在执行main方法的过程中，创建并启动了t线程，并且t线程启动后，和main线程就没有关系了，这时候main线程和t线程都是自己独立的运行，并且他们两个是要争夺CUP的时间片（使用权）的

以上代码在内存中的情况：



注意1，之前所提到的栈区，又被称为方法调用栈，是线程专门执行方法中代码的地方，并且每一个线程，都有自己独立的栈空间，和别的线程相互不影响

注意2，最先启动的线程是主线程（main线程），因为它要执行程序入口main方法，在主线程中，创建并且启动了t线程，启动之后main线程和t线程将各自独立运行，并且争夺CPU的时间片

注意3，线程启动之后（调用start方法），会开始争夺CPU的时间片，然后**自动执行run方法**，如果子类对象重写了，那么就调用到重写后的run方法

注意4，堆区是对所有线程共享的，每个线程中如果创建了对象，那么对象就会存放到堆区中

注意5，线程对象t被创建出来的时候，它还只是一个普通的对象，但是当调用了t.start()方法之后，线程对象t可以说才真正的“现出原形”：开辟了单独的栈空间，供线程t调用方法使用

思考，多线程比单线程的优势在哪里？我们一般会把什么样的代码交给多线程去执行处理？

5 匿名内部类

使用匿名内部类的形式来创建子线程：

```
1  public class Test05_Thread {
2      public static void main(String[] args) {
3          Thread t1 = new Thread() {
4              // 重写run方法
5              @Override
6              public void run() {
7                  System.out.println("in thread1 run...");
8
9                  //每隔1s输出一次
10                 for (int i = 0; i < 10; i++) {
11                     System.out.println("thread1 run ");
12
13                     //思考：为什么异常不能抛出？
14                     try {
15                         Thread.sleep(1000);
16                     } catch (InterruptedException e) {
17                         e.printStackTrace();
18                     }
19                 }
20             }
21         };
22
23         t1.start();
24     }
25 }
```

实际开发中，这样的方式更常见，书写简洁，**推荐使用**。

6 线程名称

默认线程名：

不管是主线程，还是我们创建的子线程，都是有名字的。默认情况下，主线程的名字为 `main`，main线程中创建出的子线程，它们名字命名规则如下：

```
1 //JavaAPI-Thread构造器源码
2 public Thread() {
3     init(null, null, "Thread-" + nextThreadNum(), 0);
4 }
```

其中，`"Thread-" + nextThreadNum()` 就是在拼接出这个线程默认的名字，比如第一个子线程Thread-0，第二个为Thread-1，第三个为Thread-2，以此类推。

获取当前线程对象：

```
public static native Thread currentThread();
```

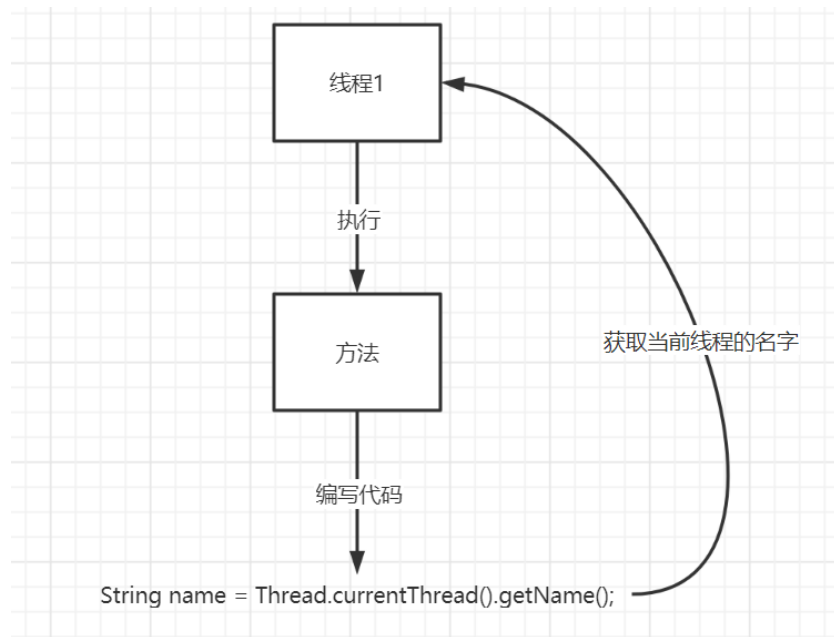
注意，这里说的当前线程，指的是**执行当前方法的线程**。

获取线程名：

```
public final String getName();
```

常见用法：

```
String name = Thread.currentThread().getName();
```



案例:

```
1 public static void main(String[] args) {
2     Thread thread = Thread.currentThread();
3     System.out.println("当前线程对象: " + thread);
4
5     String name = thread.getName();
6     System.out.println("线程名称: " + name);
7
8     Thread t = new Thread() {
9         @Override
10        public void run() {
11            Thread th = Thread.currentThread();
12            System.out.println("in run, 当前线程对象: " + th);
13
14            System.out.println("in run, 当前线程名称: " +
15                th.getName());
16        }
17    };
18    t.start();
```

```

19      // 思考：下一行输出结果是什么？
20      //t.run();
21  }
22
23  //运行结果为：
24  当前线程对象：Thread[main,5,main]
25  线程名称：main
26  in run, 当前线程对象：Thread[Thread-0,5,main]
27  in run, 当前线程名称：Thread-0

```

注意，一定要记得，start方法启动线程后，线程会自动执行run方法

千万不要直接调用run方法，这样就不是启动线程执行任务，而是普通的方法调用，和调用sayHello没区别

设置线程名：

- 通过线程对象设置线程名

```
public final synchronized void setName(String name);
```

- 创建对象时，设置线程名

```
public Thread(String name);
```

```
public Thread(Runnable target, String name);
```

```

1  public static void main(String[] args) {
2      Thread thread = Thread.currentThread();
3      //第一种设置方式
4      thread.setName("MAIN线程");
5      System.out.println("线程名称： " + thread.getName());
6
7      //第二种设置方式
8      Thread t = new Thread("子线程t") {

```

```

9         @Override
10        public void run() {
11            System.out.println("in run, 线程名称: " +
                Thread.currentThread().getName());
12        }
13    };
14    t.start();
15 }

```

7 main线程

使用 `Java` 命令来运行一个类的时候，首先会启动JVM（进程），JVM会创建一个名字叫做 `main` 的线程，来执行类中的程序入口（main方法）。

案例：

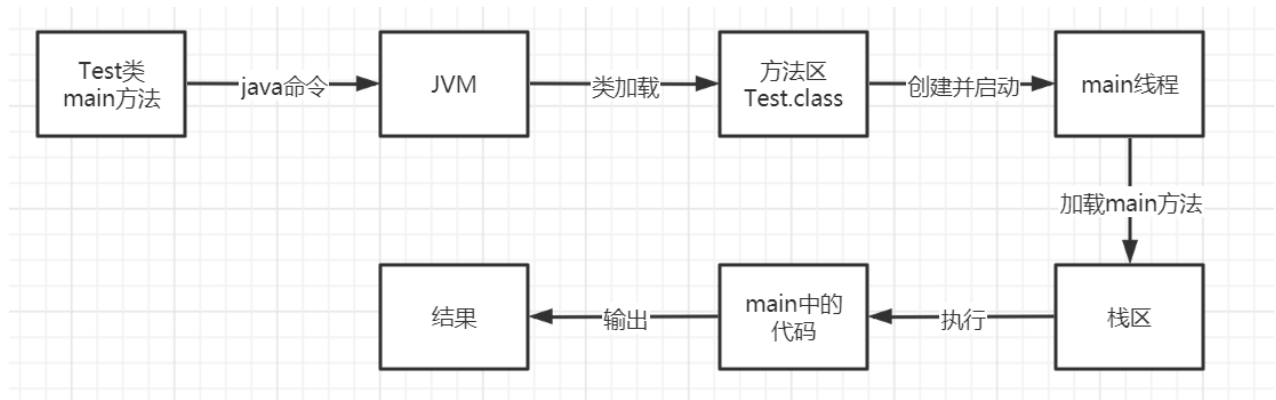
```

1  public static void main(String[] args) {
2      //获取执行当前方法的线程对象
3      Thread currentThread = Thread.currentThread();
4      System.out.println("执行当前方法的线程名字
    为: "+currentThread.getName());
5  }
6
7  //运行结果：
8  执行当前方法的线程名字为: main

```

所以，我们写在main方法中的代码，其实都是由名字叫做main的线程去执行的

上面代码使用 `java` 命令运行的过程是：



1. 使用java命令运行Test类，会先启动JVM
2. 应用类加载器通过CLASSPATH环境变量配置的路径，找到Test.class文件，并加载到方法区。

注意：这里会同时生产一个Class类型对象，来代表这个Test类型，并且会优先处理类中的静态代码（静态属性、静态方法、静态代码块）
3. JVM创建并启动一个名字叫做main的线程
4. main线程将Test中的main方法加载到栈区中
5. 在栈里面，main线程就可以一行行的执行方法中的代码了
6. 如果在执行代码中，遇到了方法调用，那么线程会继续把被调用的方法，加载到栈中（压栈操作），然后执行**栈顶**这个最新添加进来的方法，栈顶方法执行完，就释放（出栈操作），然后在进行执行当前最新的栈顶方法（之前我们画过栈里面的方法调用图，例如在异常的学习过程中）
7. 代码执行过程输出执行结果
8. 当前是单线程程序，main线程结束了，JVM就停止了
9. 如果是多线程程序，那么JVM要等所有线程都结束了才会停止

8 Runnable

前面的课程中，我们通过Thread的子类创建线程。

现在我们学习第二种创建线程对象的方式：**借助Runnable接口的实现类完成。**

`java.lang.Runnable`，该接口中只有一个抽象方法 `run`

```
1  //JavaAPI-Runnable接口源码
2  public interface Runnable {
3      public abstract void run();
4  }
```

其实 `Thread` 类也是 `Runnable` 接口的实现类，其代码结构大致为：

```
1  //JavaAPI-Thread类源码分析
2  public class Thread implements Runnable {
3      /* What will be run. */
4      private Runnable target;
5
6      public Thread() {
7          //...
8      }
9
10     public Thread(Runnable target) {
11         this.target = target;
12         //..
13     }
14
15     @Override
16     public void run() {
17         if (target != null) {
18             target.run();
19         }
20     }
21 }
```

可以看出，子类重写Thread中的run方法，这个run方法其实也来自于Runnable接口

通过以上的代码结构可知，我们可以借助构造器 `public Thread(Runnable target)` 直接创建线程对象，该构造器需要传一个 `Runnable` 接口的实现类对象。

当线程对象创建成功后，调用线程对象 `run` 方法，默认会调用Runnable实现类重写的run方法！

案例：

```
1  //1.创建Runnable实现类
2  class MyRunnable implements Runnable {
3      //2.重写run方法
4      @Override
5      public void run() {
6          String name = Thread.currentThread().getName();
7          for(int i = 20; i <= 70; i++)
8              System.out.println("in thread: " + name + " i: " +
9          i);
10     }
11 }
12 public class Test08_Runnable {
13     public static void main(String[] args) {
14         //3.实例化对象
15         Runnable r = new MyRunnable();
16         //r.run();
17
18         //4.创建Thread对象
19         Thread th = new Thread(r);
```

```

20         th.setName("child-thread1");
21         //5.启动线程
22         th.start();
23
24         //匿名内部类方式 获取Runnable实现类对象
25         Runnable r2 = new Runnable() {
26             @Override
27             public void run() {
28                 String name =
Thread.currentThread().getName();
29                 for(int i = 80; i >= 30; i--)
30                     System.out.println("in thread: " + name +
" i: " + i);
31             }
32         };
33         Thread th2 = new Thread(r2, "子线程2");
34         th2.start();
35     }
36 }

```

两种线程实现方式对比:

- 继承Thread类

好处: 编程比较简单, 可以直接使用Thread类中的方法

缺点: 扩展性较差, 不能再继承其他的类

- 实现Runnable接口

好处: 扩展性强, 实现该接口的同时还可以继承其他的类

缺点: 编程相对复杂, 不能直接使用Thread类中的方法

9 守护线程

Java中，线程可以分为两类：

- 前台线程，又叫做执行线程、用户线程
- 后台线程，又叫做守护线程、精灵线程

前台线程：

这种线程专门用来执行用户编写的代码，地位比较高，JVM是否会停止运行，就是要看当前是否还有前台线程没有执行完，如果还剩下任意一个前台线程没有“死亡”，那么JVM就不能停止！

例如，执行程序入口的主线程（main），就是一个前台线程，在单线程程序中，main方法执行完，就代表main线程执行完了，这时候JVM就停止了

注意：我们在主线程创建并启动的新线程，默认情况下就是一个前台线程

后台线程：

这种线程是用来给前台线程服务的，给前台线程提供一个良好的运行环境，地位比较低，JVM是否停止运行，根本不关心后台线程的运行情况和状态。

例如，垃圾回收器，其实就一个后台线程，它一直在背后默默的执行着负责垃圾回收的代码，为我们前台线程在执行用户代码的时候，提供一个良好的内存环境。

思考，前台线程和后台线程的关系，在生活中是否也存在类似的例子呢？

设置后台（守护）线程：

```
1  //JavaAPI-Thread类守护线程设置源码分析
2  public class Thread implements Runnable {
3      //...省略
4
5      /* Whether or not the thread is a daemon thread. */
6      private boolean    daemon = false;
7
8      //...省略
9
10     public final void setDaemon(boolean on) {
11         checkAccess();
12         if (isAlive()) {
13             throw new IllegalStateException();
14         }
15         daemon = on;
16     }
17 }
```

案例需求：

- 创建子线程1：Thread子类 + 匿名内部类
从1循环输出到30，每次输出后停顿100ms
- 创建子线程2：Runnable实现类 + 匿名内部类
从100循环输出到50，每次输出后停顿200ms
- 运行测试
默认情况下，观察两个线程运行时效果
将线程2设置为守护线程，观察运行效果

具体实现:

```
1  package com.briup.chap10;
2
3  public class Test09_Daemon {
4      public static void main(String[] args) throws Exception {
5          // 1.创建子线程1: Thread子类 + 匿名内部类
6          Thread th1 = new Thread() {
7              // 重写: 权限修饰符不能变小
8              // 抛出异常范围不能变大 故而, 抛出异常是错误的
9              public void run()/* throws Exception */ {
10                  String name = this.getName();
11                  for (int i = 1; i <= 30; i++) {
12                      System.out.println("in thread: " + name +
13                          " i: " + i);
14
15                      // 在run方法, 必须 捕获异常进行处理, 而不能抛出
16                      try {
17                          Thread.sleep(100);
18                      } catch (InterruptedException e) {
19                          e.printStackTrace();
20                      }
21                  }
22              };
23          // 2.设置线程名字并启动
24          th1.setName("男神");
25          th1.start();
26
27          // 3.创建子线程2: Runnable实现类 + 匿名内部类
28          Thread th2 = new Thread(new Runnable() {
29              @Override
30              public void run() {
31                  String name =
32                      Thread.currentThread().getName();
33                  for (int i = 100; i >= 50; i--) {
```

```

33         System.out.println("in thread: " + name +
    " i: " + i);
34
35         try {
36             Thread.sleep(200);
37         } catch (InterruptedException e) {
38             e.printStackTrace();
39         }
40     }
41 }
42 });
43 // 4.设置线程名字并启动
44 th2.setName("备胎");
45 // 设置 th2 为守护线程
46 //th2.setDaemon(true);
47 th2.start();
48 }
49 }

```

注意，`t.setDaemon(true)`；这句代码，注释掉和不注释，观察代码的运行结果，看是否一样？

10 优先级

线程类Thread中，有一个属性，表示线程的优先级，取值1-10，默认为5。线程的优先级越高，越容易获得CPU时间片进而执行（建议）。

相关方法：

方法名	说明
final int getPriority()	返回线程的优先级
final void setPriority(int newPriority)	更改线程的优先级，其默认值5，范围是：1-10

源码分析：

```

1  //JavaAPI-Thread类线程优先级设置源码
2  public class Thread implements Runnable {
3      private int priority;
4
5      /**
6       * The minimum priority that a thread can have.
7       */
8      public final static int MIN_PRIORITY = 1;
9
10     /**
11      * The default priority that is assigned to a thread.
12      */
13     public final static int NORM_PRIORITY = 5;
14
15     /**
16      * The maximum priority that a thread can have.
17      */
18     public final static int MAX_PRIORITY = 10;
19
20     public final int getPriority() {
21         return priority;
22     }
23
24     public final void setPriority(int newPriority) {
25         ThreadGroup g;

```



```

26         checkAccess();
27         if (newPriority > MAX_PRIORITY || newPriority <
MIN_PRIORITY) {
28             throw new IllegalArgumentException();
29         }
30         if((g = getThreadGroup()) != null) {
31             if (newPriority > g.getMaxPriority()) {
32                 newPriority = g.getMaxPriority();
33             }
34             //核心代码，底层借助该方法实现
35             setPriority0(priority = newPriority);
36         }
37     }
38
39     private native void setPriority0(int newPriority);
40 }

```

可以看出，最终设置线程优先级的方法，是一个native方法，并不是Java语言实现的

多个线程争夺CPU时间片：

- 优先级相同，获得CPU使用权的概率相同
- 优先级不同，那么高优先级的线程有更高的概率获取到CPU的使用权

优先级是建议性的，而非强制，可能有效，也可能无效

案例：

t1和t2线程各自运行10000次循环，修改其优先级，观察结果

```

1 package com.briup.chap10;
2

```

```

3  //线程优先级：建议性
4  public class Test10_Priority {
5      public static void main(String[] args) {
6
7          Thread th1 = new Thread() {
8              @Override
9              public void run() {
10                 for(int i = 1; i <= 1000000; i++) {
11                     if(i % 5000 == 0)
12                         System.out.println("子线程1      i: "
+ i);
13                 }
14             }
15         };
16
17         Thread th2 = new Thread(new Runnable() {
18             @Override
19             public void run() {
20                 for(int i = 1; i <= 1000000; i++) {
21                     if(i % 5000 == 0)
22                         System.out.println("th2 i: " + i);
23                 }
24             }
25         });
26
27         //分别设置 最低 最高优先级
28         th1.setPriority(Thread.MAX_PRIORITY); //10
29         th2.setPriority(Thread.MIN_PRIORITY); // 1
30
31         th1.start();
32         th2.start();
33     }
34 }

```

11 线程组

Java中使用 `java.lang.ThreadGroup` 类来表示线程组，它可以对一批线程进行管理，对线程组进行操作，同时也会对线程组里面的这一批线程操作。

`java.lang.ThreadGroup` :

```
1  public class ThreadGroup{
2      public ThreadGroup(String name){
3          //..
4      }
5      public ThreadGroup(ThreadGroup parent, String name){
6          //..
7      }
8  }
```

创建线程组的时候，需要指定该线程组的名字。

也可以指定其父线程组，如果没有指定，那么这个新创建的线程组的父线程组就是当前线程组。

案例1:

```
1  package com.briup.chap10;
2
3  public class Test11_ThreadGroup {
4      public static void main(String[] args) {
5          //获取当前线程对象
6          Thread currentThread = Thread.currentThread();
7
8          //获取当前线程所属的线程组
9          ThreadGroup currentThreadGroup =
10             currentThread.getThreadGroup();
11 }
```

```

11         System.out.println(currentThreadGroup);
12     }
13 }
14
15 //运行结果:
16 java.lang.ThreadGroup[name=main,maxpri=10]

```

可以看出，当前线程组的名字为main，并且线程组中的线程最大优先级可以设置为10

案例2:

用户在主线程中创建的线程，属于默认线程组（名字叫"main"的线程组）

```

1  public static void main(String[] args) {
2      Thread t = new Thread();
3
4      ThreadGroup threadGroup = t.getThreadGroup();
5
6      System.out.println(threadGroup);
7  }
8
9
10 //运行结果:
11 java.lang.ThreadGroup[name=main,maxpri=10]

```

可以看出，主线程中，创建一个线程对象，它的线程组默认就是当前线程的线程组

案例3:

```

1  public static void main(String[] args) {
2      ThreadGroup group = new ThreadGroup("我的线程组");
3
4      //指定线程所属的线程组
5      Thread t = new Thread(group, "t线程");
6
7      ThreadGroup threadGroup = t.getThreadGroup();
8
9      System.out.println(threadGroup);
10 }
11
12 //运行结果:
13 java.lang.ThreadGroup[name=我的线程组,maxpri=10]

```

案例4:

```

1  public static void main(String[] args) {
2      ThreadGroup group = new ThreadGroup("我的线程组");
3
4      Runnable run = new Runnable() {
5          @Override
6          public void run() {
7              try {
8                  //让线程休眠一会，否则运行太快，死亡太快了
9                  Thread.sleep(10000);
10             } catch (InterruptedException e) {
11                 e.printStackTrace();
12             }
13         }
14     };
15
16     Thread t1 = new Thread(group, run, "t1线程");
17     Thread t2 = new Thread(group, run, "t2线程");
18     Thread t3 = new Thread(group, run, "t3线程");

```

```

19
20     //注意，启动后，三个线程都会进行休眠，等run方法运行完就“死亡”了
21     t1.start();
22     t2.start();
23     t3.start();
24
25     //返回当前线程组中还没有“死亡”的线程个数
26     System.out.println("线程组中还在存活的线程个数
    为: "+group.activeCount());
27
28     //准备好数组，保存线程组中还存活的线程
29     Thread[] arr = new Thread[group.activeCount()];
30
31     //将存活的线程集中存放到指定数组中，并返回本次存放到数组的存活线程
    个数
32     System.out.println("arr数组中存放的线程个数
    为: "+group.enumerate(arr));
33
34     //输出数组中的内容
35     System.out.println("arr数组中的内容
    为: "+Arrays.toString(arr));
36 }
37
38 //运行结果：
39 线程组中还在存活的线程个数为： 3
40 arr数组中存放的线程个数为： 3
41 arr数组中的内容为： [Thread[t1线程, 5, 我的线程组], Thread[t2线程, 5, 我
    的线程组], Thread[t3线程, 5, 我的线程组]]

```

注意，只有在创建线程对象的时候，才能指定其所在的线程组，线程运行中途不能改变它所属的线程组

12 线程状态

`java.lang.Thread.State` 枚举类型中（内部类形式），定义了线程的几种状态，其代码结果为：

```
1  public class Thread{
2      /* Java thread status for tools,
3       * initialized to indicate thread 'not yet started'
4       */
5      private volatile int threadStatus = 0;
6
7      public enum State {
8          //Thread state for a thread which has not yet started.
9          NEW,
10
11          RUNNABLE,
12
13          /**
14           * Thread state for a thread blocked waiting for a
15           * monitor lock.
16           * {@link Object#wait() Object.wait}.
17           */
18          BLOCKED,
19
20          /**
21           * Thread state for a waiting thread.
22           * For example, a thread that has called
23           * <tt>Object.wait()</tt>
24           * on an object is waiting for another thread to call
25           * <tt>Object.notify()</tt> or <tt>Object.notifyAll()
26           * </tt> on
27           * that object. A thread that has called
28           * <tt>Thread.join()</tt>
29           * is waiting for a specified thread to terminate.
30           */
31      }
```

```

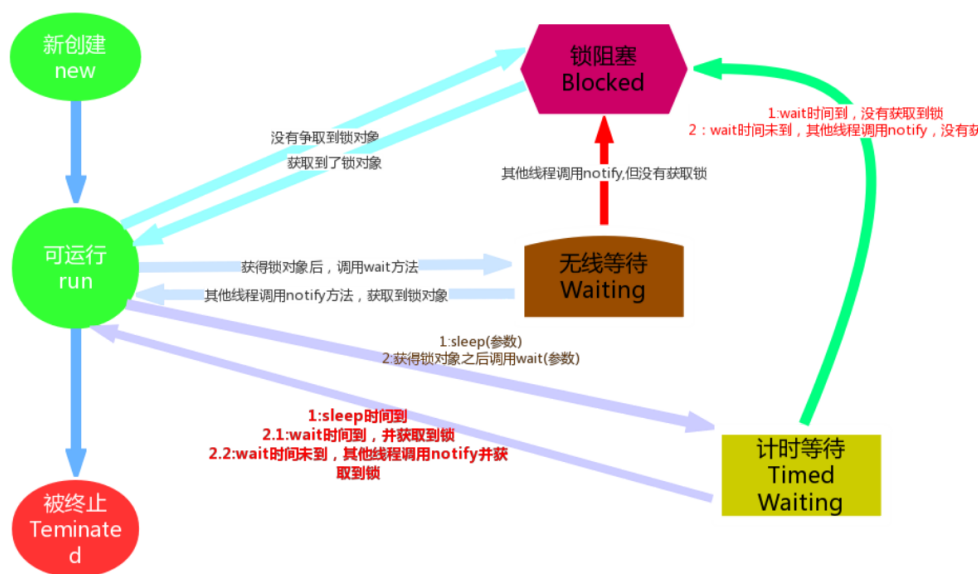
27         WAITING,
28
29         /**
30          * Thread state for a waiting thread with a specified
waiting time.
31          * A thread is in the timed waiting state due to
calling one of
32          * the following methods with a specified positive
waiting time:
33          */
34         TIMED_WAITING,
35
36         // Thread state for a terminated thread.
37         TERMINATED;
38     }
39
40     //返回线程当前所处的状态
41     public State getState() {
42         // get current thread state
43         return sun.misc.VM.toThreadState(threadStatus);
44     }
45 }

```

状态解释:

线程状态	导致状态发生条件
NEW(新建)	线程刚被创建，但是并未启动。还没调用start方法。
Runnable(可运行)	线程可以在java虚拟机中运行的状态，可能正在运行自己代码，也可能没有，这取决于操作系统处理器。
Blocked(锁阻塞)	当一个线程试图获取一个对象锁，而该对象锁被其他的线程持有，则该线程进入Blocked状态；当该线程持有锁时，该线程将变成Runnable状态。
Waiting(无限等待)	一个线程在等待另一个线程执行一个（唤醒）动作时，该线程进入Waiting状态。进入这个状态后是不能自动唤醒的，必须等待另一个线程调用notify或者notifyAll方法才能够唤醒。
Timed Waiting(计时等待)	同waiting状态，有几个方法有超时参数，调用他们将进入Timed Waiting状态。这一状态将一直保持到超时期满或者接收到唤醒通知。带有超时参数的常用方法有Thread.sleep、Object.wait。
Terminated(被终止)	因为run方法正常退出而死亡，或者因为没有捕获的异常终止了run方法而死亡。

线程状态变化的情况如下：



一个线程从创建到启动、到运行、到死亡，以及期间可能出现的情况都在上图中进行了描述。

注意：当前大家对线程状态有个大致印象即可，后续课程会详细讲解各种线程状态。

基础案例：

```
1  package com.briup.chap10;
2
3  public class Test13_State {
4      public static void main(String[] args) throws
5      InterruptedException {
6          Thread th1 = new Thread("t1子线程") {
7              @Override
8              public void run() {
9                  for(int i = 0; i < 6; i++) {
10                     try {
11                         if(i == 3)
12                             Thread.sleep(500);
13                     } catch (InterruptedException e) {
14                         e.printStackTrace();
15                     }
16                     System.out.println("i: " + i);
17                 }
18             }
19         };
20
21         //线程启动前，预计 NEW
22         System.out.println(th1.getState());
23         System.out.println(th1.getState());
24
25         //启动线程
26         th1.start();
27
28         // RUNNABLE
29         System.out.println(th1.getState());
30         System.out.println(th1.getState());
31         System.out.println(th1.getState());
```

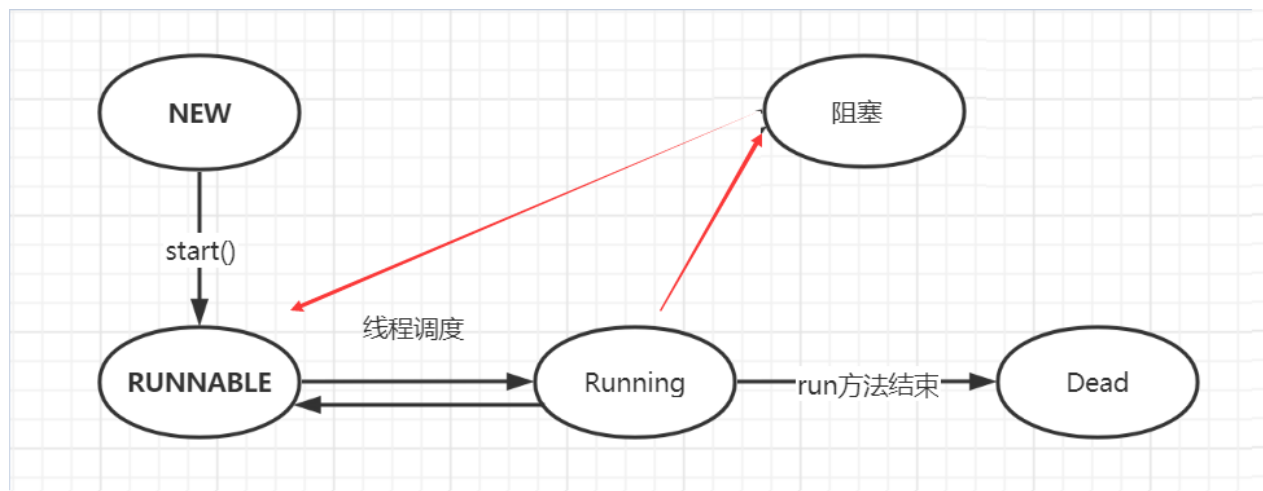
```
32         System.out.println(th1.getState());
33         System.out.println(th1.getState());
34         System.out.println(th1.getState());
35
36         //main线程sleep
37         Thread.sleep(300);
38
39         //此处：子线程依旧 sleep TIMED_WAITING
40         System.out.println(th1.getState());
41         System.out.println(th1.getState());
42         System.out.println(th1.getState());
43         System.out.println(th1.getState());
44
45         //main线程sleep
46         Thread.sleep(1000);
47
48         //预计输出：终止状态
49         System.out.println(th1.getState());
50         System.out.println(th1.getState());
51     }
52 }
53
54 //运行效果如下
55 NEW
56 NEW
57 RUNNABLE
58 RUNNABLE
59 RUNNABLE
60 RUNNABLE
61 RUNNABLE
62 RUNNABLE
63 i: 0
64 i: 1
65 i: 2
66 TIMED_WAITING
67 TIMED_WAITING
```

```
68  TIMED_WAITING
69  TIMED_WAITING
70  i: 3
71  i: 4
72  i: 5
73  TERMINATED
74  TERMINATED
```

案例结果分析：

- 刚创建好的线程对象，就是出于NEW的状态
- 线程启动后，会出于RUNNABLE状态，其包含俩种情况
就绪状态，此时这个线程没有运行，因为没有抢到CPU的执行权
运行状态，此时这个线程正在运行中，因为抢到CPU的执行权
- JavaAPI中没有定义就绪状态和运行状态，而是统一叫做**RUNNABLE**（可运行状态），杰普课程中为了能更加清楚的描述问题，会用上就绪状态和运行状态
- 线程多次抢到CPU执行权，"断断续续"把run方法执行完之后，就变成了TERMINATED状态（死亡）
之所以"断断续续"的运行，是因为每次抢到CPU执行权的时候，只是运行很小的一个时间片，完了之后还要重新抢夺下一个时间片，并且中间还有可能抢不到的情况

状态转换图（对应上述案例）：



从就绪状态到运行状态，之间会经过多次反复的CPU执行权的争夺（线程调度）

这就是一个线程经历的最基本的状态变化。

其他的状态都是线程在Running的时候，线程中调用了某些方法，或者触发了某些条件，导致这个线程进入到了阻塞状态（上面介绍的三种阻塞情况），后面陆续讲解。

13 sleep方法

线程类Thread中的 `sleep` 方法：

```
1 //JavaAPI-Thread源码
2 public class Thread implements Runnable {
3     public static native void sleep(long millis) throws
4     InterruptedException;
5 }
```

该静态方法可以让当前执行的线程暂时休眠指定的毫秒数。

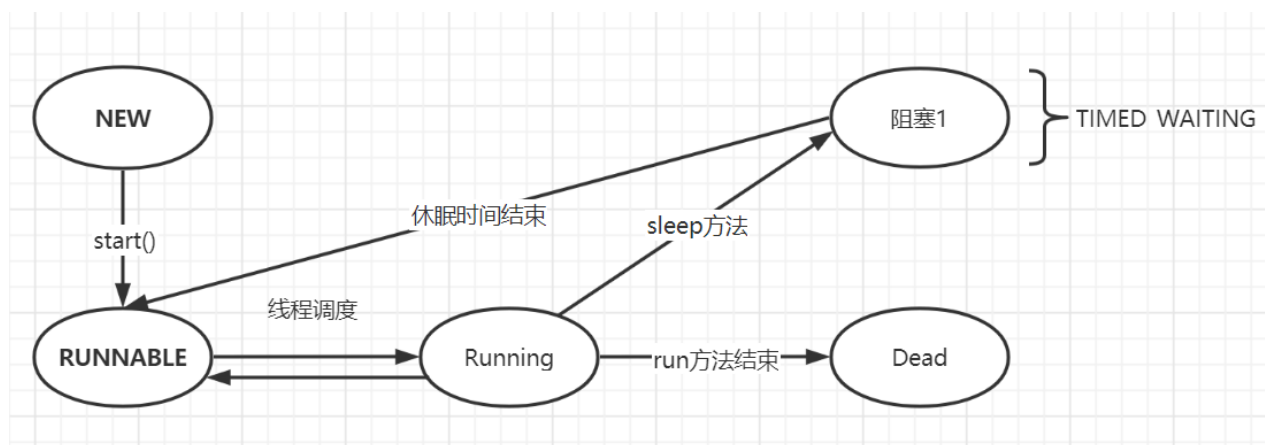
前面的案例已经大量使用，此处不再放案例。

注意事项:

线程执行了sleep方法后，会从RUNNABLE状态进入到TIMED_WAITING状态

TIMED_WAITING阻塞结束后，线程会自动回到RUNNABLE状态

sleep方法状态图:



14 join方法

Thread类中 `join` 方法:

```

1  //JavaAPI-Thread源码
2  public class Thread implements Runnable {
3      //
4      public final synchronized void join(long millis)throws
      InterruptedException{
5          //... 省略
6      }
7
8      //
9      public final void join() throws InterruptedException{
10         //... 省略
11     }
12 }

```

作用：

join方法，可以让当前线程阻塞，等另一个指定线程运行结束后，当前线程才可以继续运行

- join() 一直等待到线程结束，死等
- join(long millis) 只等待参数毫秒，时间到了后，继续运行

案例分析：

1. 创建子线程1，每隔10ms输出1-40
2. 创建子线程2，每隔10ms输出100-70
3. 当线程2输出到80时，执行th1.join()或th1.join(50)
4. 观察两种join的运行结果

完成上述功能后，再额外实现：主线程每隔7ms输出子线程2状态

案例实现:

```
1  package com.briup.chap10;
2
3  public class Test14_Join {
4      public static void main(String[] args) {
5          //1.创建子线程1, 每隔10ms输出1-40
6          Thread th1 = new Thread("th1") {
7              @Override
8              public void run() {
9                  String name =
10 Thread.currentThread().getName();
11                  for(int i = 1; i <= 40; i++) {
12                      System.out.println("in " + name + ", i: "
13 + i);
14                      try {
15                          Thread.sleep(10);
16                      } catch (InterruptedException e) {
17                          e.printStackTrace();
18                      }
19                  }
20              }
21          };
22
23          //2.创建子线程2, 每隔10ms输出100-70
24          Thread th2 = new Thread(new Runnable() {
25              @Override
26              public void run() {
27                  String name =
28 Thread.currentThread().getName();
29                  for(int i = 100; i >= 70; i--) {
30                      System.out.println("in " + name + ", i: "
31 + i);
32                      try {
```



```

29          //当i==80时，让th1线程join进来优先执行
30          if(i == 80) {
31              //a.th1线程插队，优先执行
32              //b.th1线程执行过程中，当前线程进入
WAITING状态
33              //c.th1线程执行结束，当前线程转入运行
34              th1.join();
35
36              //a.th1线程插队，优先执行100ms
37              //b.th1线程执行过程中，当前线程进入
TIMED_WAITING状态
38              //c.100ms后th1线程执行结束，当前线程转
入运行
39              //th1.join(100);
40          }
41
42          Thread.sleep(10);
43      } catch (InterruptedException e) {
44          e.printStackTrace();
45      }
46  }
47  }
48  }, "子线程2");
49
50  th1.start();
51  th2.start();
52
53  //主线程：每隔7ms输出子线程2状态
54  //      for(int i = 0; i < 50; i++) {
55  //          System.out.println("主线程, i: " + i + " th2状态: "
+ th2.getState());
56  //          try {
57  //              Thread.sleep(7);
58  //          } catch (InterruptedException e) {
59  //              e.printStackTrace();
60  //          }

```

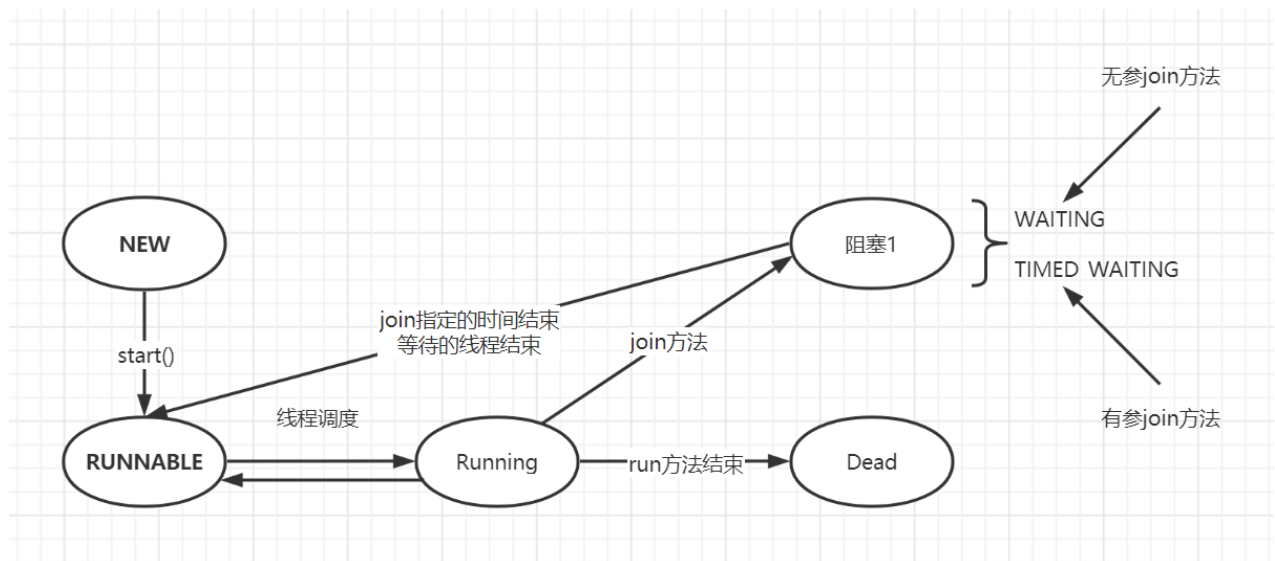
```
61    //    }
62    }
63 }
```

状态转换:

线程执行了join()方法后, 会从RUNNABLE状态进入到WAITING (无限期等待) 状态

线程执行了join(long million)方法后, 会从RUNNABLE进入到TIMED_WAITING (有限期等待) 状态

join方法状态图:



15 线程安全

问题描述:

如果有多个线程，它们在一段时间内，并发访问堆区中的同一个变量（含写入操作），那么最终可能会出现数据和预期结果不符的情况，这种情况就是线程安全问题。

我们经常会描述：这段代码是线程安全的，那段代码是非线程安全的。其实就是在说，这段代码在多线程并发访问的环境中，是否会出现上述情况。

案例演示：

模拟电影院售票业务，准备50张电影票，让多个窗口一起售卖。

```
1  package com.briup.chap10;
2
3  class TicketRunnable implements Runnable {
4      //待售票数量
5      private int num = 50;
6
7      @Override
8      public void run() {
9          while(true) {
10             //如果待售数量 小于0，跳出循环，线程结束
11             if(num <= 0)
12                 break;
13
14             //每隔50ms 销售 一张票
15             try {
16                 Thread.sleep(50);
17             } catch (InterruptedException e) {
18                 e.printStackTrace();
19             }
20
21             String name = Thread.currentThread().getName();
22             System.out.println(name + " 正在卖票，编号： " +
num);
```

```

23
24         //编号自减
25         num--;
26     }
27 }
28 }
29
30 public class Test15_ThreadSafe {
31     public static void main(String[] args) {
32         TicketRunnable r = new TicketRunnable();
33
34         Thread t1 = new Thread(r, "1号窗口");
35         Thread t2 = new Thread(r, "2号窗口");
36         Thread t3 = new Thread(r, "3号窗口");
37
38         t1.start();
39         t2.start();
40         t3.start();
41     }
42 }

```

运行效果:

1号窗口 正在卖票, 编号: 11
 2号窗口 正在卖票, 编号: 11
 3号窗口 正在卖票, 编号: 8
 2号窗口 正在卖票, 编号: 8
 1号窗口 正在卖票, 编号: 8
 3号窗口 正在卖票, 编号: 5
 2号窗口 正在卖票, 编号: 5
 1号窗口 正在卖票, 编号: 3
 3号窗口 正在卖票, 编号: 2
 2号窗口 正在卖票, 编号: 2
 1号窗口 正在卖票, 编号: 0
 3号窗口 正在卖票, 编号: -1

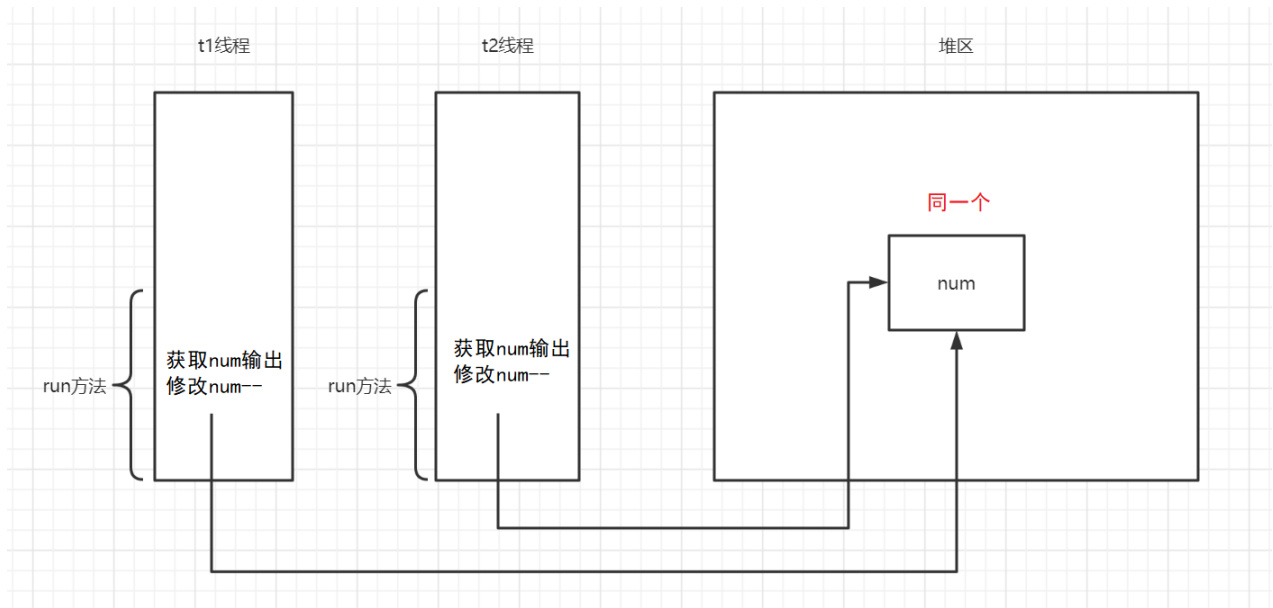


程序Bug:

- 相同的票被卖了多次，比如5号张票被销售2次
- 不存在的票被销售了，比如0和-1号票被销售

注意：程序的运行效果每次是不同的，但上述2个问题都会存在。

原理分析：



线程安全问题都是由全局变量及静态变量引起的

若每个线程中对全局、静态变量只有读操作，**而无写操作**，一般来说，线程是安全

若多个线程同时执行写操作，就很可能出现线程安全问题，此时需要**考虑线程同步技术**。

16 线程同步

Java中提供了线程同步的机制，来解决上述的线程安全问题。

Java中实现线程同步，主要借助 `synchronized` 关键字实现。

线程同步方式：

- 同步代码块
- 同步方法
- 锁机制

1) 同步代码块

格式：

```
1  //Object类及其子类对象都可以作为 线程同步锁对象 使用
2  synchronized(mutex锁对象) {
3      //需要同步操作的代码
4      //...
5  }
```

案例：

使用线程同步代码块，解决上述案例Bug

```
1  package com.briup.chap10;
2
3  class TicketRunnable2 implements Runnable {
4      //待售票数量
5      private int num = 50;
6
7      //准备锁对象【多个线程必须使用相同锁对象】
8      Object mutex = new Object();
9  }
```

```

10     @Override
11     public void run() {
12         while(true) {
13             //同步代码块：固定书写格式，需要使用同一把锁
14             //线程执行流程：
15             // 1.线程成功抢占到共享资源mutex(上锁成功)，才能进入代
码块执行
16             // 其他抢占资源失败的线程，则进入阻塞状态
17             // 2.同步代码执行完成，该线程自动释放共享资源(解锁)
18             // 其他线程由阻塞转入就绪状态，重新抢占资源(上锁)
19             synchronized (mutex) {
20                 //如果待售数量 小于0，跳出循环，线程结束
21                 if(num <= 0)
22                     break;
23
24                 //输出信息：模拟卖票
25                 String name =
Thread.currentThread().getName();
26                 System.out.println(name + " 正在卖票，编号：" +
num);
27
28                 //编号自减
29                 num--;
30
31                 //每隔50ms 销售 一张票【sleep放下面是为了更好的输出
效果】
32                 try {
33                     Thread.sleep(50);
34                 } catch (InterruptedException e) {
35                     e.printStackTrace();
36                 }
37             }
38         }
39     }
40 }
41

```

```

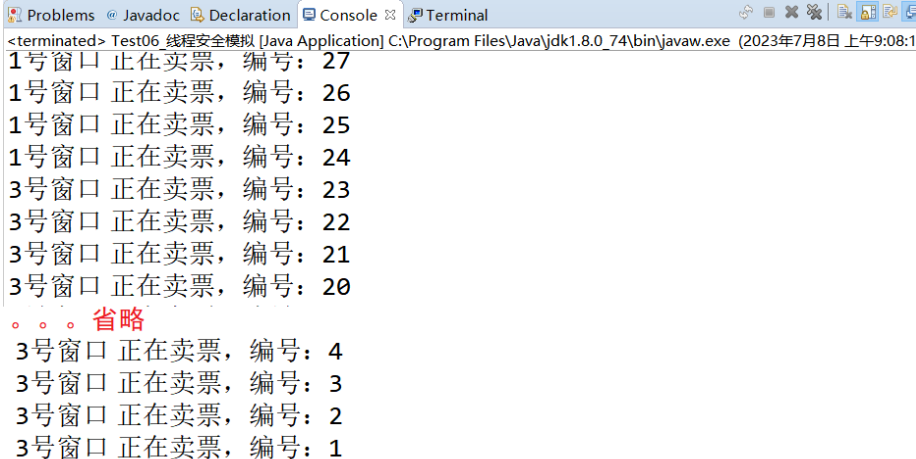
42 public class Test16_CodeBlock {
43     public static void main(String[] args) {
44         Runnable r = new TicketRunnable2();
45
46         Thread t1 = new Thread(r, "1号窗口");
47         Thread t2 = new Thread(r, "2号窗口");
48         Thread t3 = new Thread(r, "3号窗口");
49
50         t1.start();
51         t2.start();
52         t3.start();
53     }
54 }
55

```

注意，要实现线程同步，必须满足下面2个条件：

- 所有线程都需要参与线程同步
- 所有线程必须使用同一个锁对象

运行效果：



```

<terminated> Test06 线程安全模拟 [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月8日 上午9:08:11)
1号窗口 正在卖票，编号：27
1号窗口 正在卖票，编号：26
1号窗口 正在卖票，编号：25
1号窗口 正在卖票，编号：24
3号窗口 正在卖票，编号：23
3号窗口 正在卖票，编号：22
3号窗口 正在卖票，编号：21
3号窗口 正在卖票，编号：20
。。。省略
3号窗口 正在卖票，编号：4
3号窗口 正在卖票，编号：3
3号窗口 正在卖票，编号：2
3号窗口 正在卖票，编号：1

```

多次运行，观察效果可知，之前的2个问题都已经成功解决！

线程同步理解：

```
1 //线程只有竞争到锁，才能执行同步代码块中代码(上锁)
2 synchronized(mutex锁对象) {
3     需要同步操作的代码
4
5     //程序执行流程离开该代码块，则自动释放锁(解锁)
6     //离开含多种情况，正常出右大括号，break、return或遇到异常跳出
7 }
```

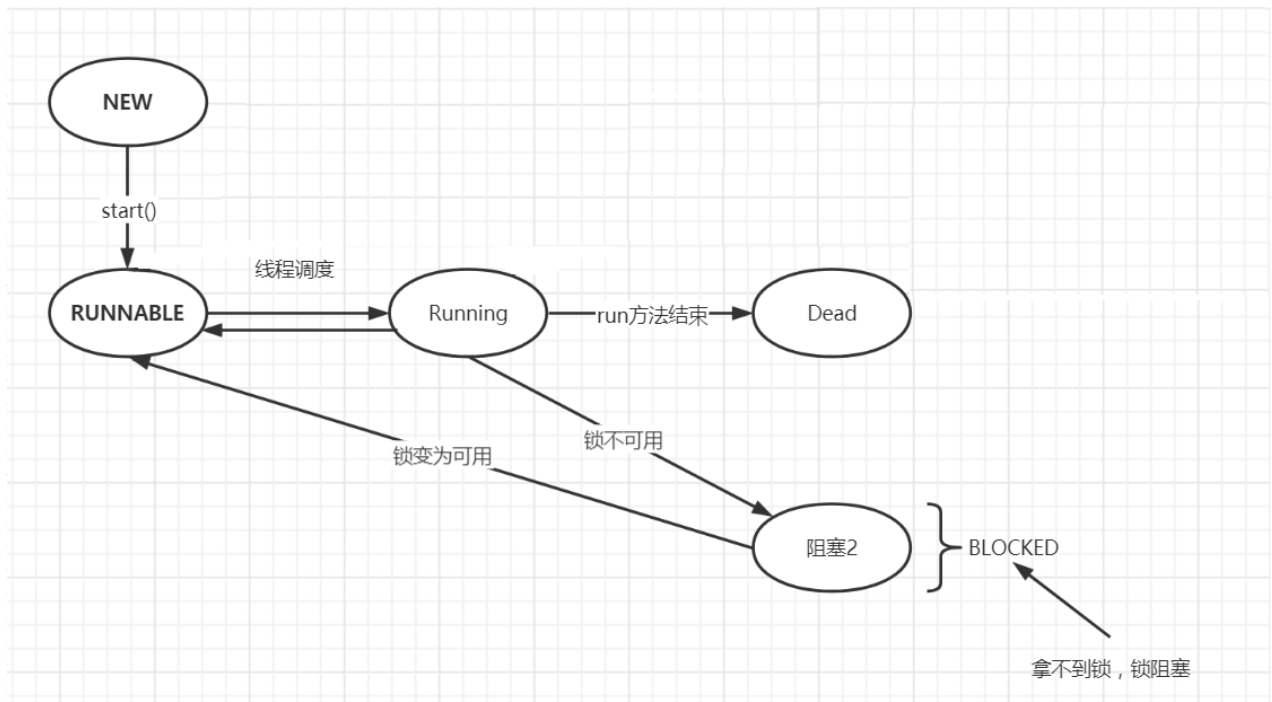
线程同步可理解成一个规则（进卫生间必须竞争到钥匙开门，离开卫生间必须锁门交出钥匙）

- 所有的线程都遵循这种规则，才能同步成功
- 如果个别线程不遵循规则（不用钥匙，翻窗进入），则无法实现同步

锁对象可以理解成保证线程同步的重要因素（打开卫生间门的唯一钥匙）

- 多个线程使用同一把锁（用唯一的钥匙开门），才能保证线程同步
- 如果使用不同的锁（不同的人用不同的钥匙去开门），则也无法实现线程同步效果

线程状态图为：



2) 同步方法

使用synchronized修饰的方法，就叫做同步方法，其固定格式如下：

```
1 public [static] synchronized 返回值类型 同步方法() {
2     可能会产生线程安全问题的代码
3 }
```

注意事项：

- 同步方法可以是普通成员方法，也可以是static静态方法
- 普通成员同步方法，默认锁对象为this，即当前方法的调用对象
- static静态同步方法，默认锁对象是当前类的字节码对象（一个类有且只有一个）

类的字节码对象：类名.class，固定用法（当前记着即可，后续反射章节会学习）

案例1：普通同步方法

创建子线程1，调用100次普通方法print1（逐字输出 "好好学习"）

创建子线程2，调用100次普通方法print2（逐字输出 "天天向上"）

要求，两个子线程在执行方法的过程中，不会被另一个线程打断。

```
1  class Printer {
2      //普通同步方法：锁对象默认为this
3      public synchronized void print1() {
4          System.out.print("天");
5          System.out.print("天");
6          System.out.print("向");
7          System.out.print("上");
8          System.out.println();
9      }
10
11     public void print2() {
12         //同步代码块，也使用this作为锁对象
13         //测试时，可以注释同步代码块，或使用其他锁对象，然后观察程序运
行效果
14         //synchronized (Printer.class) {
15         synchronized (this) {
16             System.out.print("努");
17             System.out.print("力");
18             System.out.print("学");
19             System.out.print("习");
20             System.out.println();
21         }
22     }
23 }
24
25 public class Test16_Funtion {
26     public static void main(String[] args) {
```

```

27         //准备一个对象
28         final Printer p = new Printer();
29
30         //创建子线程1，输出100次 "好好学习"
31         Thread th1 = new Thread() {
32             @Override
33             public void run() {
34                 for(int i = 0; i < 100; i++)
35                     p.print1();
36             }
37         };
38
39         //创建子线程2，输出100次 "天天向上"
40         Thread th2 = new Thread() {
41             @Override
42             public void run() {
43                 for(int i = 0; i < 100; i++)
44                     p.print2();
45             }
46         };
47
48         th1.start();
49         th2.start();
50     }
51 }

```

测试效果：

- print2方法不使用同步代码块，或不使用this作为锁对象，会出现输出混乱的情况，线程没有实现同步（上锁失败）

```
Problems Javadoc Declaration Console Terminal
<terminated> Test16_Funtion [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月8日 下午11:45:39 - 下午11:45:40)
努力学天天向上
天天向上
天天向上
天天向上
天天习
努力学习
```

两个线程执行方法的过程中，被另一个线程打断了，产生了输出乱入

- print2方法使用同步代码块，且用this作为锁对象，成功实现线程同步

```
Problems Javadoc Declaration Console Terminal
<terminated> Test16_Funtion [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月8日 下午11:48:23 - 下午11:48:23)
天天向上
努力学习
努力学习
努力学习
```

一切正常，成功实现 线程同步

案例2：静态同步方法

将上述案例中的普通同步方法，修改为静态同步方法，实现原有功能。

```
1 class Printer {
2     // ...省略print1() print2()
3
4     //static静态同步方法：锁对象默认为当前类字节码对象
5     public static synchronized void print3() {
6         System.out.print("天");
7         System.out.print("天");
8         System.out.print("向");
9         System.out.print("上");
10        System.out.println();
11    }
12
13    public void print4() {
14        //同步代码块，使用当前类字节码对象作为锁对象
15
16        //注释掉同步代码块，运行测试，观察效果
17        //不使用当前类字节码对象作为锁对象，运行测试，观察效果
18        //synchronized (this) {
```

```

19         synchronized (Printer.class) {
20             System.out.print("努");
21             System.out.print("力");
22             System.out.print("学");
23             System.out.print("习");
24             System.out.println();
25         }
26     }
27 }
28
29 public class Test16_Funtion {
30     public static void main(String[] args) {
31         //准备一个对象
32         final Printer p = new Printer();
33
34         //创建子线程1，输出100次 "好好学习"
35         Thread th1 = new Thread() {
36             @Override
37             public void run() {
38                 for(int i = 0; i < 100; i++)
39                     Printer.print3();
40             }
41         };
42
43         //创建子线程2，输出100次 "天天向上"
44         Thread th2 = new Thread() {
45             @Override
46             public void run() {
47                 for(int i = 0; i < 100; i++)
48                     p.print4();
49             }
50         };
51
52         th1.start();
53         th2.start();
54     }

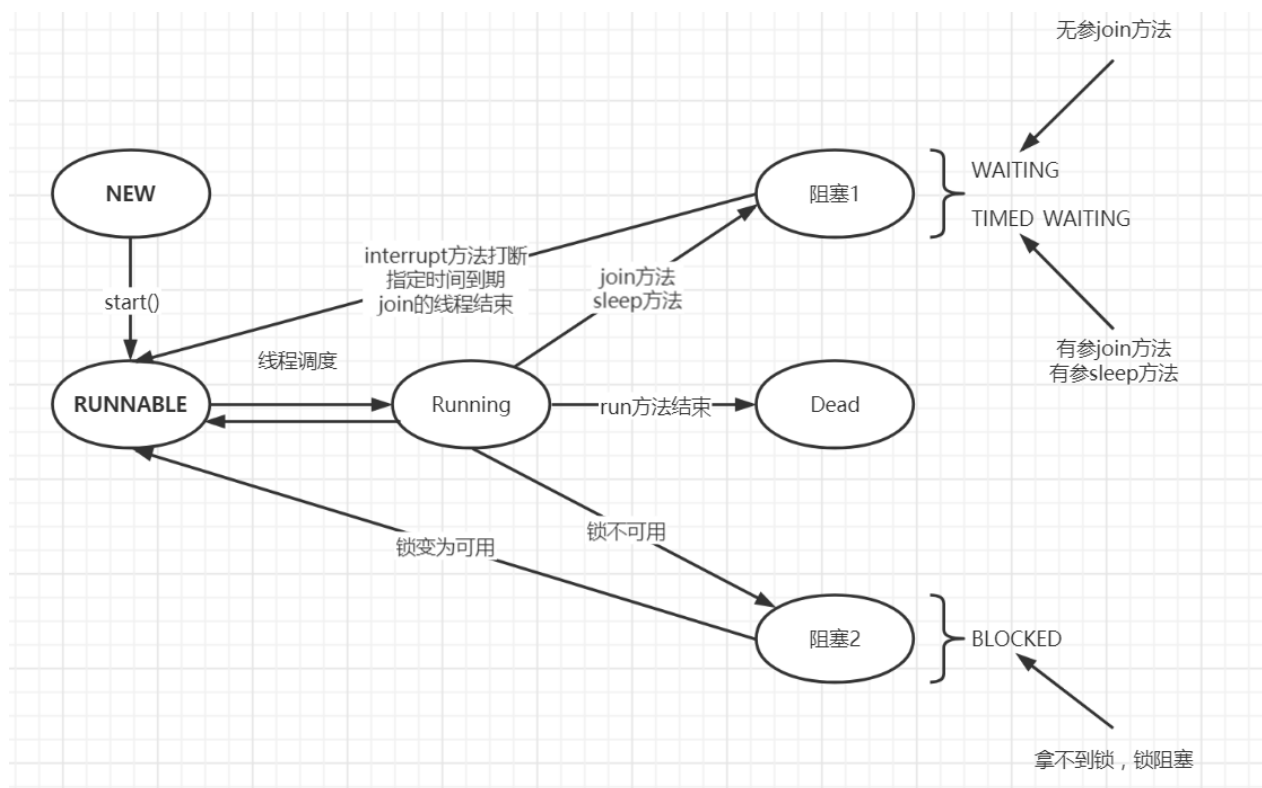
```

运行测试：

按照print4方法中描述进行测试，验证结论：

static静态同步方法: 锁对象默认为当前类字节码对象（类名.class）

线程状态图为：



17 线程通信

通过synchronized关键字，**可保证线程在运行的过程中不会被其他线程打断，但无法保证线程的执行次序**。比如上述案例中，我们可以保证t1线程输出"好好学习"时，不会被t2线程打断，但无法保证t1、t2两个线程执行的次序。如果想要控制线程输出次序，则需要学习线程通信技术。

1) 概念理解

线程间通信：

多个线程并发执行时，在默认情况下CPU是随机切换线程的，当我们需要多个线程来共同完成一件任务，并且我们希望它们有规律的执行，那么**多线程之间就需要一些协调通信**，以此来帮我们达到多线程共同操作一份数据。

等待唤醒机制：

多个线程在处理同一个资源，并且任务不同时，需要线程通信来帮助解决线程之间对同一个变量的使用或操作。就是多个线程在操作同一份数据时，避免对同一共享变量的争夺。也就是我们需要通过一定的手段使各个线程能有效的利用资源。而这种手段即——**等待唤醒机制**。

2) wait和notify

Object类中有三个方法：`wait()`、`notify()`、`notifyAll()`

当一个对象，在线程同步的代码中，充当锁对象的时候，在 `synchronized` 同步的代码块中，就可以调用这个锁对象的这三个方法了。

三个关键点：

- 任何对象中都一定有这三个方法
- 只有对象作为**锁对象**的时候，才可以调用

- 只有在**同步的代码块中**，才可以调用

其他情况下，调用一个对象的这三个方法，都会报错！

等待唤醒机制详解：

这是多个线程间的一种**协作**机制。谈到线程我们经常想到的是线程间的**竞争**（**race**），比如去争夺锁，但这并不是故事的全部，线程间也会有协作机制。就好比在公司里你和你的同事们，你们可能存在晋升时的竞争，但更多时候你们是一起合作以完成某些任务。

在一个线程进行了规定操作后，就进入等待状态（**wait()**），等待其他线程执行完指定代码过后再将其唤醒（**notify()**）；在有多个线程进行等待时，如果需要，可以使用 **notifyAll()**来唤醒所有的等待线程。

wait/notify 就是线程间的一种协作机制。

方法详解：

等待唤醒机制用于解决线程间通信的问题的，使用到的3个方法的含义如下：

1. **wait**：线程不再活动，不再参与调度，进入 **wait set** 中，因此不会浪费 CPU 资源，也不会去竞争锁了，这时的线程状态即是 **WAITING**。它还要等着别的线程执行一个**特别的动作**，也即是“**通知（notify）**”在这个对象上等待的线程从**wait set** 中释放出来，重新进入到调度队列（**ready queue**）中
2. **notify**：则选取所通知对象的 **wait set** 中的一个线程释放；例如，餐馆有空位置后，等候就餐最久的顾客最先入座。
3. **notifyAll**：则释放所通知对象的 **wait set** 上的全部线程。

注意：

哪怕只通知了一个等待的线程，被通知线程也不能立即恢复执行，因为它当初中断的地方是在同步块内，而此刻它已经不持有锁，所以**它需要再次尝试去获取锁**（很可能面临其它线程的竞争），成功后才能在当初调用 wait 方法之后的地方恢复执行。

总结：

- 如果能获取锁，线程就从 WAITING 状态变成 RUNNABLE 状态；
- 否则，从 wait set 出来，又进入 entry set，线程就从 WAITING 状态又变成 BLOCKED 状态

注意事项：

1. wait方法与notify方法必须要由同一个锁对象调用

因为：对应的锁对象可以通过notify唤醒使用同一个锁对象调用的wait方法后的线程。

2. wait方法与notify方法是属于Object类的方法的

因为：锁对象可以是任意对象，而任意对象的所属类都是继承了Object类的。

3. wait方法与notify方法必须要在同步代码块或者是同步方法中使用

因为：必须要通过锁对象调用这2个方法

3) 两线程通信

创建两个线程，一个是生产者线程，蒸包子，另一个是消费者线程，吃包子，要求两个线程轮流执行（先生产再消费）。

案例实现：

生产者线程类：

```
1  //包子类
2  class Bum {
3      //包子数量
4      int num = 0;
5
6      //包子存在标识
7      boolean flag = false;
8  }
9
10 //生产者
11 class Producer extends Thread {
12     private Bum bum;
13
14     public Producer(String name, Bum bum) {
15         super(name);
16         this.bum = bum;
17     }
18
19     @Override
20     public void run() {
21         for (int i = 1; i <= 100; i++) {
22             //同步
23             synchronized (bum){
24                 //根据flag判断包子是否存在，如果存在则 线程进行等待
25                 if(bum.flag){
26                     try {
27                         bum.wait();
28                     } catch (InterruptedException e) {
29                         e.printStackTrace();
30                     }
31                 }
32
33                 //生产包子
```

```

34         System.out.println("第" + i + "次，" +
    this.getName() + ": 开始生产包子...");
35         bum.num++;
36         System.out.println("生产完成，包子数量：" +
    bum.num + "，快来吃!");
37
38         //生产完成，修改flag存在标识为true
39         bum.flag = true;
40
41         //通知 消费者线程吃包子
42         bum.notify();
43     }
44 }
45 }
46 }

```

消费者线程类:

```

1  class Customer extends Thread {
2      private Bum bum;
3
4      public Customer(String name, Bum bum) {
5          super(name);
6          this.bum = bum;
7      }
8
9      @Override
10     public void run() {
11         for (int i = 1; i <= 100; i++) {
12             //同步
13             synchronized (bum){
14                 //根据flag判断包子是否存在，如果不存在则线程等待
15                 if(bum.flag == false){
16                     try {
17                         bum.wait();
18                     } catch (InterruptedException e) {

```

```

19             e.printStackTrace();
20         }
21     }
22
23     System.out.println("第" + i + "次, " +
this.getName() + " 开始吃包子...");
24     bum.num--;
25     System.out.println("消费完成, 包子数量: " +
bum.num + ", 快生产吧!");
26
27     //消费完成, 修改flag存在标识为false
28     bum.flag = false;
29
30     //通知 消费者线程吃包子
31     bum.notify();
32 }
33 }
34 }
35 }

```

测试类:

```

1  package com.briup.chap10;
2
3  //创建两个线程, 一个生产包子, 另一个消费包子, 要求线程按照
4  public class Test17_TwoCommunication {
5      public static void main(String[] args) {
6          //准备共享对象
7          Bum bum = new Bum();
8
9          //生产者线程
10         Thread th1 = new Producer("打工人", bum);
11         //消费者线程
12         Thread th2 = new Customer("吃货", bum);
13
14         //启动线程

```

```

15         th1.start();
16         th2.start();
17     }
18 }

```

注意事项:

- 2个线程通信主要借助wait()、notify()和flag标值完成


```

1  if(flag判断) {
2      执行wait()等待;
3  }

```

- wait()可以让线程进入等待状态
- notify()可以通知等待的某个线程，让其转入就绪状态

运行效果:



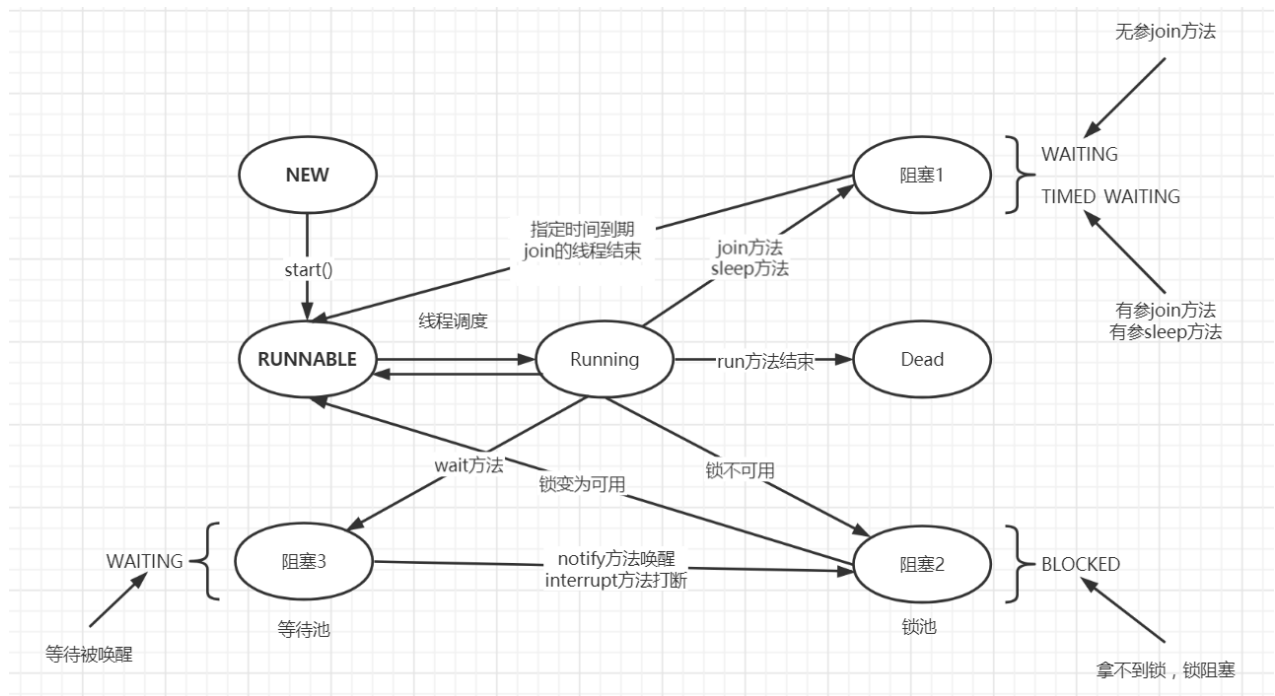
```

<terminated> Test17_Communication [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月9日 上午10:18:37 - 上午10:18:38)
第90次, 吃货 开始吃包子...
吃完了, 包子剩余数量: 0, 快生产吧!
第91次, 打工人: 开始生产包子...
包子蒸好了, 包子剩余数量: 1, 快来吃!
第91次, 吃货 开始吃包子...
吃完了, 包子剩余数量: 0, 快生产吧!
第92次, 打工人: 开始生产包子...
包子蒸好了, 包子剩余数量: 1, 快来吃!
第92次, 吃货 开始吃包子...
吃完了, 包子剩余数量: 0, 快生产吧!
第93次, 打工人: 开始生产包子...
包子蒸好了, 包子剩余数量: 1, 快来吃!
第93次, 吃货 开始吃包子...
吃完了, 包子剩余数量: 0, 快生产吧!

```

成功实现：生产-消费 轮流执行

线程状态图为:



4) 多线程通信

创建3个线程，第1个是生产者线程，每次蒸2只包子，第2个是消费者线程，吃1个包子，第3个也是消费者线程，吃1个包子，要求3个线程轮流执行（线程1生产，线程2消费，线程3消费）。

案例实现：

生产者线程类：

```

1  //包子类
2  class Bum2 {
3      // 包子数量
4      int num = 0;
5
6      // 线程执行标识：0表示线程1执行 1表示线程2执行 2表示线程3执行
7      int flag = 0;
8  }
9

```

```

10  //生产者
11  class Producer1 extends Thread {
12      private Bum2 bum;
13
14      public Producer1(String name, Bum2 bum) {
15          super(name);
16          this.bum = bum;
17      }
18
19      @Override
20      public void run() {
21          for (int i = 1; i <= 100; i++) {
22              // 同步
23              synchronized (bum) {
24                  // 根据flag判断包子是否存在，如果存在则 线程进行等待
25                  // 注意，此处必须改为while，用if无法实现功能
26                  while (bum.flag != 0) {
27                      try {
28                          bum.wait();
29                      } catch (InterruptedException e) {
30                          e.printStackTrace();
31                      }
32                  }
33
34                  // 生产包子
35                  System.out.println("第" + i + "次，" +
this.getName() + ": 开始生产包子...");
36                  // 每次生产2个包子
37                  bum.num += 2;
38                  System.out.println("生产完成，包子数量：" +
bum.num + "，快来吃!");
39
40                  // 生产完成，修改flag存在标识为true
41                  bum.flag = 1;
42
43                  // 通知 其他所有线程转入运行

```



```

44         bum.notifyAll();
45     }
46 }
47 }
48 }

```

2个消费者线程类：

```

1  class Customer2 extends Thread {
2      private Bum2 bum;
3
4      public Customer2(String name, Bum2 bum) {
5          super(name);
6          this.bum = bum;
7      }
8
9      @Override
10     public void run() {
11         for (int i = 1; i <= 100; i++) {
12             // 同步
13             synchronized (bum) {
14                 // 根据flag判断包子是否存在，如果不存在则线程等待
15                 // 注意，此处必须改为while，用if无法实现功能
16                 while (bum.flag != 1) {
17                     try {
18                         bum.wait();
19                     } catch (InterruptedException e) {
20                         e.printStackTrace();
21                     }
22                 }
23
24                 System.out.println(this.getName() + " 开始吃包子...");
25
26                 bum.num--;
27                 System.out.println("消费完成，包子剩余数量： " + bum.num);

```

```

27
28         // 消费完成，修改flag存在标识为false
29         bum.flag = 2;
30
31         // 通知 其他所有线程转入运行
32         bum.notifyAll();
33     }
34 }
35 }
36 }
37
38 class Customer3 extends Thread {
39     private Bum2 bum;
40
41     public Customer3(String name, Bum2 bum) {
42         super(name);
43         this.bum = bum;
44     }
45
46     @Override
47     public void run() {
48         for (int i = 1; i <= 100; i++) {
49             // 同步
50             synchronized (bum) {
51                 // 根据flag判断包子是否存在，如果不存在则线程等待
52                 // 注意，此处必须改为while，用if无法实现功能
53                 while (bum.flag != 2) {
54                     try {
55                         bum.wait();
56                     } catch (InterruptedException e) {
57                         e.printStackTrace();
58                     }
59                 }
60
61                 System.out.println(this.getName() + " 开始吃包子...");

```

```

62         bum.num--;
63         System.out.println("消费完成，包子剩余数量： " +
        bum.num);
64
65         // 消费完成，修改flag存在标识为false
66         bum.flag = 0;
67
68         // 通知 其他所有线程转入运行
69         bum.notifyAll();
70     }
71 }
72 }
73 }

```

测试类：

```

1  package com.briup.chap10;
2
3  public class Test17_MoreCommunication {
4      public static void main(String[] args) {
5          Bum2 bum = new Bum2();
6
7          // 生产者线程
8          Thread th1 = new Producer1("打工人", bum);
9          // 消费者线程
10         Thread th2 = new Customer2("1号吃货", bum);
11         Thread th3 = new Customer3("2号吃货", bum);
12
13         th1.start();
14         th2.start();
15         th3.start();
16     }
17 }

```

注意事项：

- 多个(3个及以上)线程通信主要借助wait()、notifyAll()和flag标识完成
- notifyAll()可以通知所有等待的某个线程，让其转入就绪状态
- flag的判断必须使用while，如果使用if则无法完成功能

```

1  //notifyAll()会唤醒所有wait线程
2  //但第一个醒来并上锁成功的那个线程，很可能不是我们想要的
3  //所以需要使用while再做一次状态判断
4  //从而保证，只有我们期望的线程 能够成功醒来并上锁成功，往下执行
5  while(flag判断) {
6      执行wait()等待；
7  }
8
9  //成功醒来并上锁成功，往下执行代码
10 doNext...

```

运行效果：

```

第99次，打工人：开始生产包子...
包子蒸好了，包子剩余数量： 2，快来吃！
1号吃货 开始吃包子...
吃完了，包子剩余数量： 1
2号吃货 开始吃包子...
吃完了，包子剩余数量： 0
第100次，打工人：开始生产包子...
包子蒸好了，包子剩余数量： 2，快来吃！
1号吃货 开始吃包子...
吃完了，包子剩余数量： 1
2号吃货 开始吃包子...
吃完了，包子剩余数量： 0

```

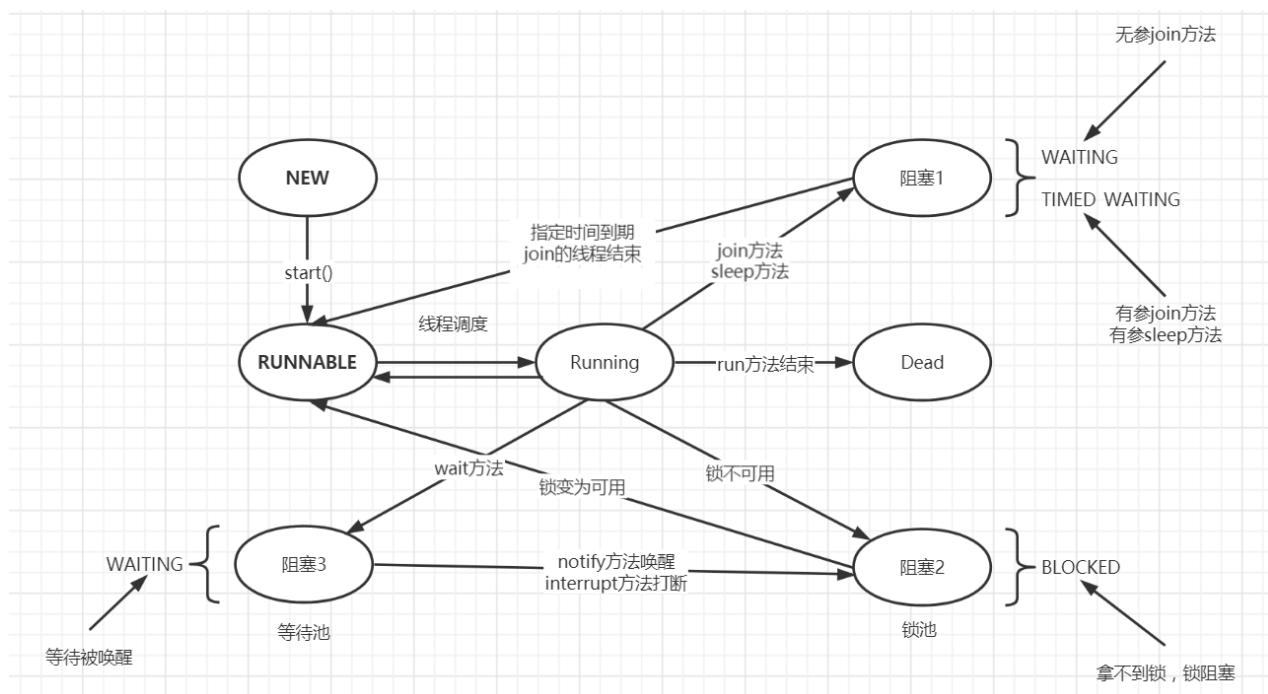
成功实现3个线程通信，线程执行次序为：
生产线程1 消费线程2 消费线程3

5) 状态总结

状态分类：

线程状态	导致状态发生条件
NEW(新建)	线程刚被创建，但是并未启动。还没调用start方法。
Runnable(可运行)	线程可以在java虚拟机中运行的状态，可能正在运行自己代码，也可能没有，这取决于操作系统处理器。
Blocked(锁阻塞)	当一个线程试图获取一个对象锁，而该对象锁被其他的线程持有，则该线程进入Blocked状态；当该线程持有锁时，该线程将变成Runnable状态。
Waiting(无限等待)	一个线程在等待另一个线程执行一个（唤醒）动作时，该线程进入Waiting状态。进入这个状态后是不能自动唤醒的，必须等待另一个线程调用notify或者notifyAll方法才能够唤醒。
Timed Waiting(计时等待)	同waiting状态，有几个方法有超时参数，调用他们将进入Timed Waiting状态。这一状态将一直保持到超时期满或者接收到唤醒通知。带有超时参数的常用方法有Thread.sleep、Object.wait。
Terminated(被终止)	因为run方法正常退出而死亡，或者因为没有捕获的异常终止了run方法而死亡。

线程状态变化情况：



18 死锁

在程序中要尽量避免出现死锁情况，一旦发生那么只能手动停止JVM的运行，然后查找并修改产生死锁的问题代码

简单的描述死锁就是：两个线程t1和t2，t1拿着t2需要等待的锁不释放，而t2又拿着t1需要等待的锁不释放，两个线程就这样一直僵持下去。

案例展示：

```
1  package com.briup.chap10;
2
3  //结论：不要嵌套上锁(synchronized)
4  //容易造成死锁
5  public class Test18_DeadLock {
6      public static void main(String[] args) {
7          Object obj1 = new Object();
8          Object obj2 = new Object();
9
10         Thread th1 = new Thread() {
11             @Override
12             public void run() {
13                 for (int i = 0; i < 100; i++) {
14                     synchronized (obj1) {
15                         System.out.println("th1拿到左筷子");
16                         synchronized (obj2) {
17                             System.out.println("th1拿到右筷子");
18                             System.out.println("th1吃 水盆羊
19 肉");
19                     }
20                 }
21             }
22         };
23     };
24
25     Thread th2 = new Thread() {
```

```

26         @Override
27         public void run() {
28             for (int i = 0; i < 100; i++) {
29                 synchronized (obj2) {
30                     System.out.println("th2拿到右筷子");
31                     synchronized (obj1) {
32                         System.out.println("th2拿到左筷子");
33                         System.out.println("th2吃到了水盆羊
34                             肉");
35                     }
36                 }
37             }
38         };
39
40         th1.start();
41         th2.start();
42     }
43 }

```

19 线程池

1) 问题引入

之前的操作中，我们使用线程的时候就去创建一个线程，使用完成线程自动销毁，这样操作非常简便，但会产生问题：

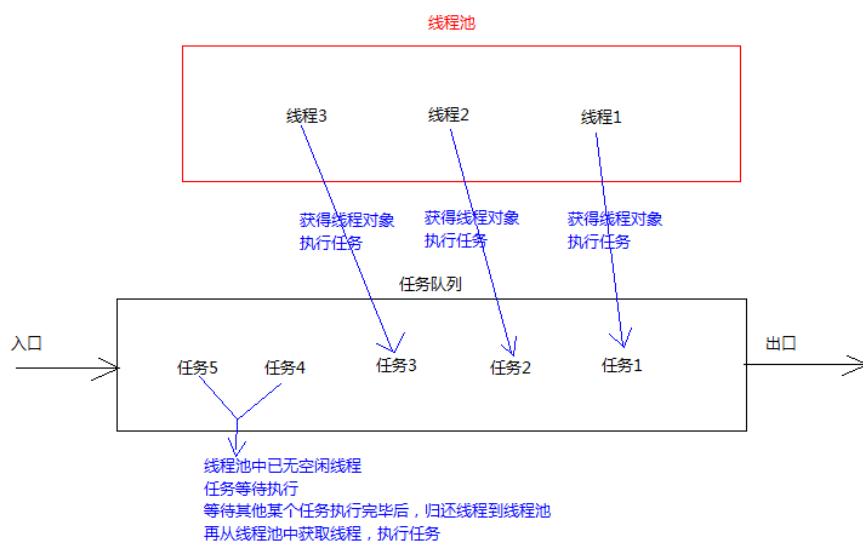
如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束，这样频繁创建线程就会大大降低系统的效率（频繁创建线程和销毁线程需要时间）。

那么有没有一种办法使得线程可以复用？即执行完一个任务，并不立即销毁，而是可以继续执行其他的任务。在Java中可以通过线程池来达到这样的效果。

2) 线程池概念

线程池其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。

线程池工作原理理解：



线程池优点：

1. 降低资源消耗

减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2. 提高响应速度

当任务到达时，任务可以不需要等到线程创建就能立即执行。

3. 提高线程的可管理性

可以根据系统的承受能力，调整线程池中工作线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

3) 线程池使用

Java里面线程池的顶级接口是 `java.util.concurrent.Executor`，但是严格意义上讲 `Executor` 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 `java.util.concurrent.ExecutorService`。

创建线程池的方法：

- `public static ExecutorService newFixedThreadPool(int nThreads)`
- 返回线程池对象。(创建的是有界线程池,也就是池中的线程个数可以指定最大数量)

使用线程池对象方法：

- `public Future<?> submit(Runnable task)`
- Future接口：用来记录线程任务执行完毕后产生的结果
- 获取线程池中的某一个线程对象，并执行

线程池操作步骤：

1. 创建线程池对象（ExecutorService类对象）
2. 创建Runnable接口子类对象
3. 提交Runnable接口子类对象（借助submit方法实现）

4. 关闭线程池(一般不做)

案例实现:

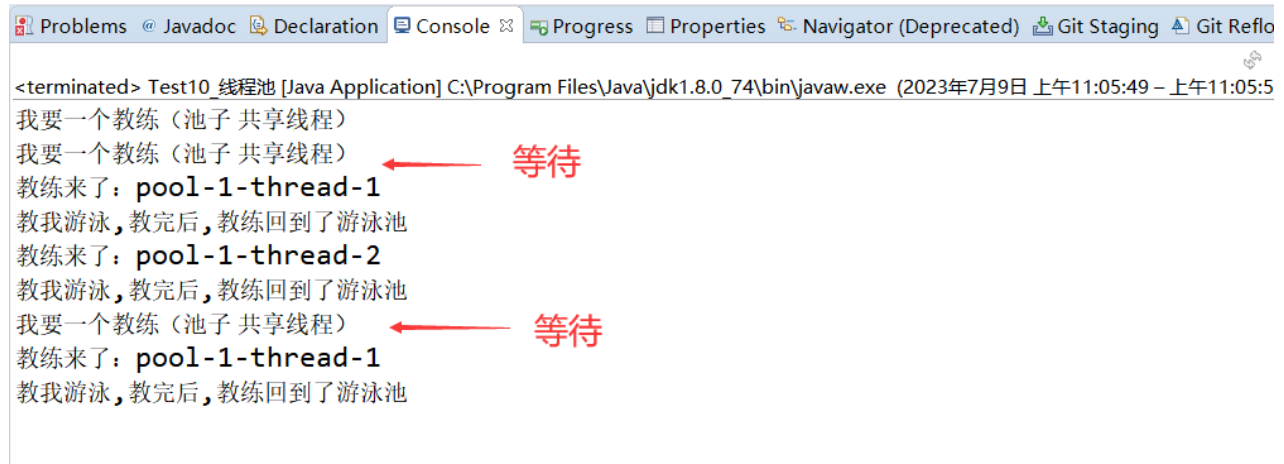
```
1  package com.briup.chap10;
2
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  class MyRunnable implements Runnable {
7      @Override
8      public void run() {
9          System.out.println("我要一个教练（池子 共享线程）");
10
11          try {
12              Thread.sleep(2000);
13          } catch (InterruptedException e) {
14              e.printStackTrace();
15          }
16
17          System.out.println("教练来
了: "+Thread.currentThread().getName());
18          System.out.println("教我游泳,教完后,教练回到了游泳池");
19      }
20  }
21
22  public class Test19_ThreadPool {
23      public static void main(String[] args) {
24          //关键点1: 获取线程池【含2个线程】
25          ExecutorService service =
26              Executors.newFixedThreadPool(2);
27
28          MyRunnable r = new MyRunnable();
29
30          //关键点2: 从线程池获取线程 跟 任务结合 运行
```

```

31         service.submit(r);
32         service.submit(r);
33         service.submit(r);
34
35         //释放线程池
36         service.shutdown();
37     }
38 }

```

运行效果：



20 Callable接口

补充内容：实现多线程的第三种方式: 实现Callable接口，该方式使用不多，大家**了解即可**。

- 相关方法

方法名	说明
V call()	计算结果，如果无法计算结果，则抛出一个异常
FutureTask(Callable callable)	创建一个 FutureTask，一旦运行就执行给定的 Callable
V get()	如有必要，等待计算完成，然后获取其结果

- 操作步骤
 - 定义一个类MyCallable实现Callable接口
 - 在MyCallable类中重写call()方法
 - 创建MyCallable类的对象
 - 创建Future的实现类FutureTask对象，把MyCallable对象作为构造方法的参数
 - 创建Thread类的对象，把FutureTask对象作为构造方法的参数
 - 启动线程
 - 再调用get方法，就可以获取线程结束之后的结果【可有可无】

- 代码演示

```
1 package com.briup.chap10;
2
3 import java.util.concurrent.Callable;
4 import java.util.concurrent.FutureTask;
5
6 //1.创建Callable的实现类
7 class MyCallable implements Callable<String> {
8     //2.重写call方法
```

```

9      @Override
10     public String call() throws Exception {
11         for (int i = 0; i < 100; i++) {
12             System.out.println("跟女孩表白" + i);
13         }
14
15         //返回值就表示线程运行完毕之后的结果
16         return "答应";
17     }
18 }
19
20 public class Test20_Callable {
21     public static void main(String[] args) throws Exception
22     {
23         //3.实例化Callable的实现类类对
24         MyCallable mc = new MyCallable();
25
26         //Thread t1 = new Thread(mc);
27
28         //4.创建Future的实现类FutureTask对象，把MyCallable对象
29         作为构造方法的参数
30         FutureTask<String> ft = new FutureTask<>(mc);
31
32         //5.创建Thread对象，并传递ft对象作为构造器参数
33         Thread t1 = new Thread(ft);
34
35         //String s = ft.get();
36
37         //6.开启线程
38         t1.start();
39
40         //7.获取线程方法执行后返回的结果
41         String s = ft.get();
42         System.out.println(s);
43     }
44 }

```

三种线程实现方式对比：

- 实现Runnable、Callable接口

好处：扩展性强，实现该接口的同时还可以继承其他的类。Callable接口可获取线程处理函数执行的结果

缺点：编程相对复杂，不能直接使用Thread类中的方法，

- 继承Thread类

好处：编程比较简单，可以直接使用Thread类中的方法

缺点：扩展性较差，不能再继承其他的类