

基本概念：程序、进程、线程


- 程序(program)是为完成特定任务、用某种语言编写的一组指令的集合。即指一段**静态的代码，静态对象**。
- 进程(process)是**程序的一次执行过程**，或是正在运行的一个程序。是一个动态的过程：有它自身的产生、存在和消亡的过程。——生命周期
 - 运行中的QQ，运行中的MP3播放器
 - 程序是静态的，进程是动态的
 - 进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域
- 线程(thread)，进程可进一步细化为线程，是一个**程序内部的一条执行路径**。
 - 若一个进程同一时间并行执行多个线程，就是支持多线程的
 - `main()`，就可以视为一个 thread
 - 线程作为调度和执行的单位，每个线程拥有独立的运行栈和程序计数器(pc)，线程切换的开销小
 - 线程各自享有独立的 Program Counter Register 和 VM Stack
 - 一个进程中的多个线程共享相同的内存单元/内存地址空间它们从同一堆中分配对象，可以访问相同的变量和对象。这就使得线程间通信更简便、高效。
 - 线程共享 Heap 和 Method Area
 - 因此，实现多个线程的通信比较方便
 - 但多个线程操作共享的系统资源可能会带来安全的隐患 – 如果多个线程抢着操作，那就崩了
- 单核和多核 CPU
 - 单核 CPU，其实是一种假的多线程，因为在一个时间单元内，也只能执行一个线程的任务
 - 但是单核 CPU 看起来实现多线程，其实是因为 CPU 分配每个线程一个时间单元进行运行，而且每个时间单元都很短 – 也就是挂起一个进程，然后运行别的
 - 所以实现了假的多线程
 - 如果是多核的话，才能更好的发挥多线程的效率。（现在的服务器都是多核的）
 - 一个 Java 应用程序 `java.exe`，其实至少有三个线程：`main()`主线程，`gc()`垃圾回收线程，异常处理线程。当然如果发生异常，会影响主线程。
- 并行和并发
 - 多个 CPU 同时执行多个任务，称为并行 parallel；十个人，各自有一球
 - 一个 CPU 采用时间片，同时执行多个任务，称为并发 concurrence；十个人抢一个球
- 多线程的优点
 - 以单核CPU为例，只使用单个线程先后完成多个任务（调用多个方法），肯定比用多个线程来完成用的时间更短，为何仍需多线程呢？
 - 因为单核 CPU 的切换的过程中，也需要时间；单线程反而更快
 - 但是多核 CPU，多个线程来完成比较好
 - 提高应用程序的响应。对图形化界面更有意义，可增强用户体验。

- 提高计算机系统 CPU 的利用率
- 改善程序结构。将既长又复杂的进程分为多个线程，独立运行，利于理解和修改
- 何时需要多线程
 - 程序需要同时执行两个或多个任务
 - 程序需要实现一些需要等待的任务时，如用户输入，文件读写操作，网络操作，搜索等
 - 需要一些后台运行的程序时 – 垃圾回收器
- 辨析

```
public class Sample {
    public static void main(String[] args) {
        Sample s = new Sample();
        s.method2("Hello");
    }
    public void method2(String str){
        method1(str);
    }
    public void method1(String str){
        System.out.println(str);
    }
}
```

- 上述代码并不实现多线程，他只是两个方法的来回顺次调用
- 我们只要发现一条路径可以画出程序的执行过程，那么就是单线程

线程的创建和使用

- 线程的调度 – 优先级
 - 调度策略
 - 时间片
 - CPU 没有偏好，轮流执行
 - 
 - 抢占式
 - 抢占式：高优先级的线程抢占 CPU
 - Java 调度方法
 - 同优先级线程组成先进先出队列（先到先服务），使用时间片策略
 - 对高优先级，使用优先调度的抢占式策略
 - 线程的优先级等级
 - MAX_PRIORITY：10
 - MIN_PRIORITY：1
 - NORM_PRIORITY：5
 - 涉及的方法
 - getPriority()：返回线程优先值
 - setPriority(int newPriority)：改变线程的优先级

- 说明
 - 线程创建时继承父线程的优先级
 - 低优先级只是获得调度的概率低，并非一定是在高优先级线程之后才被调用
- Thread 类中常用方法
 - void start() 启动当前线程，调用当前线程的 run()
 - void run() 线程在被调度时执行的操作
 - currentThread() 静态方法，返回执行当前代码的线程
 - String getName() 返回线程的名字
 - String setName() 设置当前线程的名字
 - 改名改的是对象的名字，所以线程名是一个实例变量；而且，改名应在线程运行之前 – 线程运行中可以改名，不会报错
 - Thread.currentThread().setName("可爱");//这个可以改当前线程的名，如 main
 - 我们也可以通过重写线程的构造器来初始化实例变量
 - Thread(name) 提供了父类方法；子类可以必须创建自己的构造器，并使用 super 调用父类的有参构造器来创建自己的有参构造器
 - ```
public MyThread(String str){
 super(str);
}
```
  - static void yield()
    - 暂停当前正在执行的线程，把执行机会让给优先级相同或更高的线程 – 其实就是放弃这个线程在 CPU 中的执行权
    - 若队列中没有同优先级的线程，忽略此方法
  - join()
    - 当某个程序执行流中调用其他线程的 join() 方法时，调用线程其他线程的线程将被阻塞，直到 join() 方法加入的 join 线程执行完为止
      - 低优先级的线程也可以获得执行
      - 这个方法不是静态，所以 main 方法中不能直接调用；需要 t1.join()
      - 这个方法可能抛出异常，需要处理
  - static void sleep(long millis) (指定时间毫秒)
    - 让当前的进程暂停一段时间
    - 这个方法会抛出 InterruptedException 异常，所以需要 try-catch 处理异常；而且你不能通过抛出异常的方式来解决问题，因为 run() 方法在父类中没有抛出异常
    - 我们可以使用 sleep 方法来实现一个简单的计时器
  - stop() - Deprecated 强制结束当前线程
  - boolean isAlive()
    - 这个用于判断当前线程是否仍然存活
- JDK5.0<
  - 创建方法一

- 继承 `java.lang.Thread`，并重写 `public void run()` 方法
  - `run()` 方法中重写我们希望其执行的代码 – 也就是线程的功能
  - 经常把 `run()` 方法的主体称为线程体 - `ThreadBody`
- 然后创建子类对象，调用对象继承自父类的 `start()` 方法
  - 启动这个线程以及调用线程的 `run()`
  - 都是通过主线程 `main()` 调用方法的 `start()` 完成的
    - 但是**不能调用 `run()` 方法**，这样相当于就是普通的调用方法，不是多线程了
    - 我们可以通过 `Thread.currentThread().getName()` 的方法来得到运行线程的名字；在调用 `run()` 方法时，得到的名字就是 `main`
- 例子

```
class EvenNumber extends Thread {
 @Override
 public void run() {
 for (int i = 0; i <= 100; i++) {
 if (i % 2 == 0) {
 System.out.print(i + ",");
 }
 }
 }
}

public class ThreadTest {
 public static void main(String[] args) {
 EvenNumber t1 = new EvenNumber();
 t1.start();
 System.out.println("Main");//因为多线程，这个main输出的位置可能不一样
 }
}
```

- 创建两个线程
  - `java.lang.IllegalThreadStateException`
  - 如果当前线程运行过一次，那么 `threadStatus` 就变成了1；因此，如果调用两次 `start()` 试图得到两个线程，那么就会抛出异常
  - 所以，我们必须创建第二个 `Thread` 对象，并调用其 `start()` 方法
- 创建方法二
  - 过程
    - 定义子类，实现 `Runnable` 接口。
    - 子类中实现 `Runnable` 接口中的 `run` 方法 – 抽象方法
    - 创建实现类的对象
    - 将 `Runnable` 接口的子类对象作为实际参数传递给 `Thread` 类的构造器
      - `Thread t = new Thread();` – 这里面接受的参数为 `Runnable`，是多态的体现

- 调用 `Thread` 类的 `start` 方法：开启线程，调用 `Runnable` 子类接口的 `run` 方法
- 原理
  - ```
public void run() {
    if (target != null) {
        target.run();
    }
}
```
 - 如上，在调用 `new Thread(Runnable Object)` 构造器时，我们实际上是给 `thread` 中的 `private Runnable Target` 赋值；因此，就可以运行 `implements Runnable interface` 的类中的 `run` 方法
- 创建两个线程
 - ```
Window w = Window.getInstance();
Thread t1 = new Thread(w);
Thread t2 = new Thread(w);
```
- 比较
  - 实现接口的方式更好
    - 接口可以实现多继承 – 也就是我们使用接口的方式，可以让其继承别的父类，而非必须 `Thread`
    - 接口实现可以默认实现共享数据 – 多个线程之间共享数据，而不需要加入 `static`
  - 共性
    - 其实 `Thread` 类就是一个实现了 `Runnable` 接口的类
- 线程的分类
  - Java中的线程分为两类：一种是守护线程，一种是用户线程。
    - 它们在几乎每个方面都是相同的，唯一的区别是判断JVM何时离开
      - 若 JVM 中都是守护线程，当前 JVM 将退出
    - 守护线程是用来服务用户线程的，通过在 `start()` 方法前调用
  - `thread.setDaemon(true)` 可以把一个用户线程变成一个守护线程。
  - Java 垃圾回收就是一个典型的守护线程。

## 线程的生命周期

- 定义
  - 生命周期，指的是线程从创建到死亡的几种状态
- 线程的状态 – 这些信息定义在 `java.lang.Thread.State` 中

- ```
public enum State {
    /**
     * Thread state for a thread which has not yet started.
     */
    NEW,
```

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

```
Thread t = new Thread()
```

```
o    /**
      * Thread state for a runnable thread.  A thread in the
      runnable
      * state is executing in the Java virtual machine but it may
      * be waiting for other resources from the operating system
      * such as processor. 没分配资源
      */
      RUNNABLE,
```

- 当一个新建的线程被启动，`t.start()`，但是可能还没有被分配到 CPU 时间片，他就是就绪 `Runnable` 状态

- 这个状态下包括线程的 就绪 和 运行 两个阶段

- CPU 主动分配资源，
- 就绪：处于新建状态的线程被 `start()` 后，将进入线程队列等待 CPU 时间片，此时它已具备了运行的条件，只是没分配到 CPU 资源
- 运行：当就绪的线程被调度并获得 CPU 资源时，便进入运行状态，`run()` 方法定义了线程的操作和功能；`yield()` 可能让他回到就绪状态

```
o    /**
      * Thread state for a thread blocked waiting for a monitor
      lock.
      * A thread in the blocked state is waiting for a monitor lock
      * to enter a synchronized block/method or
      * reenter a synchronized block/method after calling
      * {@link Object#wait() Object.wait}. 没进入同步结构
      */
      BLOCKED,
```

- `synchronized` 会阻塞线程，这时线程处于 `Blocked` 在状态，让出 CPU 并临时中
止自己的执行，进入阻塞状态

```
o    /**
      * Thread state for a waiting thread.
      * A thread is in the waiting state due to calling one of the
      * following methods:
      * <ul>
      *   <li>{@link Object#wait() Object.wait} with no timeout</li>
      *   <li>{@link #join() Thread.join} with no timeout</li>
      *   <li>{@link LockSupport#park() LockSupport.park}</li>
      * </ul>
      *
      * <p>A thread in the waiting state is waiting for another
      thread to
      * perform a particular action.
      *
      * For example, a thread that has called {@code Object.wait()}
      * on an object is waiting for another thread to call
```

```

    * {@code Object.notify()} or {@code Object.notifyAll()} on
    * that object. A thread that has called {@code Thread.join()}
    * is waiting for a specified thread to terminate. 线程等待其他线程干活 -- 被阻塞。因为自己调用了 join()
    */
    WAITING,

```

- 线程没有时间限制，长时间等待，包括但不限于 `wait()`, `join()`，让出 CPU 并临时中止自己的执行，进入阻塞状态

o

```

    /**
     * Thread state for a waiting thread with a specified waiting
     * time.
     * A thread is in the timed waiting state due to calling one of
     * the following methods with a specified positive waiting
     * time:
     * <ul>
     * <li>{@link #sleep Thread.sleep}</li>
     * <li>{@link Object#wait(long) Object.wait} with
     * timeout</li>
     * <li>{@link #join(long) Thread.join} with timeout</li>
     * <li>{@link LockSupport#parkNanos LockSupport.parkNanos}
     * </li>
     * <li>{@link LockSupport#parkUntil LockSupport.parkUntil}
     * </li>
     * </ul> 线程等待一段时间后执行
     */
    TIMED_WAITING,

```

- 线程等待一段时间后，继续运行，如 `sleep()`

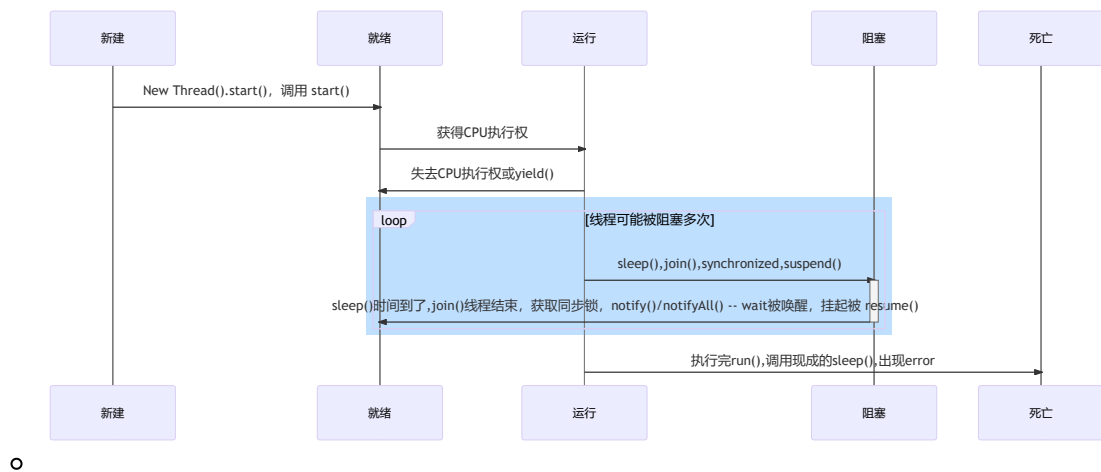
o

```

    /**
     * Thread state for a terminated thread.
     * The thread has completed execution. 线程结束了
     */
    TERMINATED;

```

- 线程死亡，线程完成了它的全部工作或线程被提前强制性地中止或出现异常导致结束
- 执行完 `run()`，或者被 `stop()`，或者异常没有处理
- 线程一定在 `Terminated` 状态结束，阻塞是不行的



线程的同步

- 问题
 - 多个线程执行的**不确定性**引起执行结果的不稳定
 - 多个线程对**账本的共享**，会造成操作的不完整性，会破坏数据
 - 例子
 - 两人同时取钱，取同一个账户的钱
 - 三线程卖票

```

public class WindowTest {
    public static void main(String[] args) {
        Window w = new Window();
        Thread t1 = new Thread(w);
        Thread t2 = new Thread(w);
        Thread t3 = new Thread(w);
        t1.start();
        t2.start();
        t3.start();
    }
}

class Window implements Runnable{
    private int ticket = 100;
    private static int UUID = 0;

    @Override
    public void run() {
        for(;;){
            if(ticket > 0){
                try {
                    Thread.sleep(100); // 加不加都可能出现0, -1;
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

我们sleep以后CPU同时在ticket=1时CPU把三个线程同时运行了一下的概率增加


```

        System.out.println(Thread.currentThread().getName()+":卖票"
+ ticket--);
    }else{
        break;
    }
}
}
}

```

- 有的时候会出现 0, -1, 1票
- 这种情况是因为三个线程同时在 ticket=1时得到了 CPU 运行权, 并且刚好在其他线程没有 -- 之前进入
- 线程的安全问题
 - 原因
 - 当多条语句在操作同一个线程共享数据时, 一个线程对多条语句只执行了一部分, 还没有执行完 if(ticket>0)到sysout, 另一个线程参与进来执行。导致共享数据的错误, E.g., ticket
 - 如果线程之间没有共享数据, 是不会有线程安全问题的
 - 解决方法
 - 当一个线程在操作共享数据的时候, 其他线程不能参与进来; 直到线程操作完共享数据, 其他线程才可以操纵共享数据, 即使线程被阻塞 (TIMED_WAITING)
 - 在 java 中, 我们通过同步机制, 来解决线程的安全问题
 - 同步代码块
 - 使用

```
synchronized(同步监视器){  
    // 需要被同步的代码 -- 也就是操作共享数据的代码  
}
```

- 操作共享数据的代码就是需要被同步的代码，共享数据就是多个线程同时处理的变量，比如 `ticket` 就是共享数据
- 同步监视器 - `Monitor Lock`
 - 监视器可以是任意对象，比如 `private Object obj = new Object()`
 - 要求
 - 多个线程**必须共用同一把锁**；如果存在多个锁，那么仍然不安全

```
class Window implements Runnable{
    private int ticket = 100;
    private Object obj = new Object();
    @Override
    public void run() {
        //Object obj = new Object();错误, 不保证唯一性
        for(;;){
            //synchronized(new object()),错误, 不保证
            唯一性
        }
    }
}
```

```

        synchronized(obj) {
            if (ticket > 0) {
                try {
                    sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                System.out.println(Thread.currentThread().getName(
) + ":卖票" + ticket--);
            } else {
                break;
            }
        }
    }
}

```

■ 常用

对象	Runnable	Thread
<pre>private obj = new Object()</pre>	可以	不可以，用 <pre>private static obj</pre>
this	可以，一个runnable 对象到处放	不可以，每个创建对象是有独立this
<pre><ClassName>.class</pre>	可以	可以

- 我们还可以直接写 `Windows.class` 作为同步监视器，因为类也是一个对象，并且类是唯一的 – 类只会加载一次

■ 优缺点

- 好处：解决了线程的安全问题
- 缺点：操作同步代码块时，只能由一个线程参与，其他线程等待，相当于是一个单线程的过程，效率较低 – 不过单线程的部分一般较小，可以忽略不记

■ 原理

- 只有持有同步监视器对象的线程才可以执行同步代码块；执行完成以后释放同步监视器对象
- 但是有可能同步监视器被同一个线程抢到多次

■ 继承方法的同步锁

- ```

public class WindowTest2 {
 public static void main(String[] args) {
 Windows t1 = new Windows();
 Windows t2 = new Windows();
 Windows t3 = new Windows();
 }
}

```

```

 t1.start();
 t2.start();
 t3.start();
 // 输出不是递减, 因为有可能一个程序先 --, 然后
System.out.println
 }
}

class Windows extends Thread{
 private static int Number = 1;
 private static int ticket = 100; // 有可能三个线程一起干
 活, 然后出来三个100
 private static Object obj = new Object(); // 不加
 static那么就有三个这样的变量
 public Windows(){
 super("窗口" + Number);
 Number++;
 }
 @Override
 public void run() {
 while(true){
 synchronized (obj) {
 if (ticket > 0) {
 try {
 sleep(100);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println(getName() + ":卖
 票, 票号为:" + ticket--);
 } else {
 break;
 }
 }
 }
 }
}

```

- 因为使用继承实现的多线程并不是默认共享数据, 我们必须把同步锁声明为 `static` 才可以
- 

## 线程的通信

- `notify()`
- `notifyAll()`
- `wait()`
- 上述三个方法定义在 `java.lang.Object` 类中, 而非 `Thread`

## JDK5.0新增线程创建方式