链滴

Java 线程基础

作者：shuaibing90

原文链接：https://ld246.com/article/1605797959765

来源网站：链滴

线程基础内容

程序、进程与线程

- 程序：Program，是一个指令的集合
- 进程：Process，（正在执行中的程序）是一个静态的概念
  - 进程是程序的一次静态执行过程，占用特定的地址空间（资源）

- 进程是申请资源的一个最小单位
  - 每个进程都是独立的，由3部分组成CPU，data，code
  - 缺点：内存的浪费，CPU的负担
- 线程：是进程中一个"单一的连续的控制流程"（a single sThread,equential flow of control）/执路径
  - 线程又被称为 **轻量级的进程**（lightweight process）
  - Threads run at the same time,independently of one another
  - 一个进程可拥有多个并行的（concurrent）线程
  - 一个对象中的线程共享相同的内存单元/内存地址空间——>可以访问相同的变量和对象，而且们从同一堆中分配对象，而且它们从同一堆中分配对象-->通信、数据交换、同步操作
  - 由于线程间的通信是在同一地址空间上进行的，所以不需要额外的通信机制，这就使得通信更便而且信息传递的速度也更快

进程与线程的关系

- 一个进程中的线程共享代码和数据空间
- 一个进程中至少要包含一个线程，比如在Java虚拟机启动时候会有一个进程java.exe，该进程中至有一个线程在负责java程序的执行。而这个线程运行的代码存在于main方法中，该线程称之为主线程
- 线程结束，进程未必结束，但是进程结束，线程一定结束
- 进程中包含线程，线程是进程的一部分
-

区别	**进程**	**线程**
根本区别	作为资源分配的单位	度和执行的单位
开销	每个进程都有独立的代码和数据空间（进程上下文），进程的切换会有较大的开销	线程可以看成是轻量级的进程，同一类线程享代码和数据空间，每个线程都有独立的运行栈和程序计数器（PC），线程切换开销小
所处环境	在操作系统中能同时运行的多个任务（程序）	同一应用程序中有多个流同时执行
分配内存	系统在运行的时候会为每个进程分配不同的内存区域	了CPU之外，不会为线程分配内存（线程所使用的资源是它所属的进程的资源），线程组只能共享资源
包含关系	没有线程的进程是可以被看做是单线程的，如果一个进	内拥有多个线程则执行过程不是一条线的，而是多条线（线程）共同完成的 程是进程的一部分，所以线程有的时候会被称为是轻权进程或者轻量级进程

线程的创建与启动

- 在Java中负责线程的这个功能是Java.lang.Thread 这个类
- 可以通过创建Thread的实例来创建新的线程
- 每个线程都是通过某个特定的Thread的实例来创建新的线程
- 每个线程都是通过某个特定的Thread对象所对应的方法run() 来完成其操作的，方法run()称为线程体

- 通过调用Thread类的start方法来启动一个线程

创建线程的方式1

- 1.继承Thread类
- 2.重写run方法
- 3.创建对象，调用start方法启动线程

```
public class ThreadDemo extends Thread {
    @Override
    public void run() {
        for (int i = 0; i <10 ; i++) {
            System.out.println(Thread.currentThread().getName()+"---------"+i);
        }

    }
    public static void main(String[] args) {
        ThreadDemo threadDemo = new ThreadDemo();
        threadDemo.start();
        for (int i = 0; i <5 ; i++) {
            System.out.println(Thread.currentThread().getName()+"==========="+i);
        }
    }
}
```

注意：Thread类中的run方法是存储线程要运行的代码，主线程要运行的代码存放在main方法中

start方法是开启线程并执行该线程的run方法

继承Thread类方式的缺点：

如果我们的类已经从一个类继承（如小程序必须继承自Applet类），则无法再继承Thread类

如果我们的类已经从一个类继承（如小程序必须继承自Applet类），则无法再继承Thread类
如果我们的类已经从一个类继承（如小程序必须继承自Applet类），则无法再继承Thread类

创建线程的方式2（重点&&常用）

- 1.实现Runnable接口
- 2.重写run方法
- 创建对象，调用start()方法，启动线程

```
public class RunnableDemo implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i <10 ; i++) {
            System.out.println(Thread.currentThread().getName()+"---------------"+i);
        }

    }
```

```
    public static void main(String[] args) {
        RunnableDemo runnableDemo = new RunnableDemo();
        Thread thread = new Thread(runnableDemo);
        thread.start();
        for (int i = 0; i <5 ; i++) {
            System.out.println(Thread.currentThread().getName()+"====================
+i);
        }

    }
}
```

使用Runnable接口实现多线程优点：

可以实现继承。实现Runnale接口的方式要通用一些

可以实现继承。实现Runnale接口的方式要通用一些

可以实现继承。实现Runnale接口的方式要通用一些

- 1）避免单继承
- 2）方便共享资源，同一份资源 多个代理访问

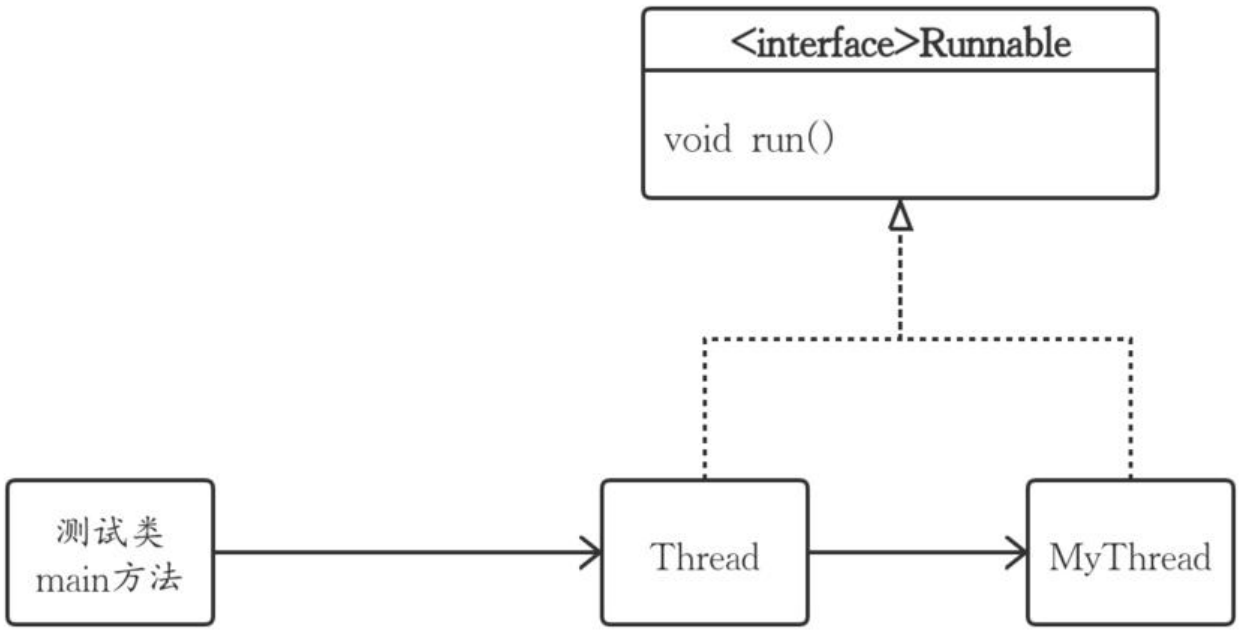
线程操作的相关方法

序号	方法名称	描述
1	public static Thread currentThread()	返回目前正在执行的线程
2	public final String getName()	返回线程的名称
3	public final int getPriority()	返回线程的优先级
4	public final void setPriority(String name)	设定线程名称
5	public final boolean isAlive()	判断线程是否在活动，如果是，返回true,否则返回false
6	public final void join()	调用该方法的线程强制执行，其它线程处于**阻塞**状态，该线程执行完毕后，其它线程再执行
7	public static void sleep(long millis)	使用当前正在执行的线程休眠millis秒,线程处于**阻塞**状态
8	public static void yield()	当前正在执行的线程暂停一次，允许其他线程执行,**不阻塞**，线程进入**就绪状态** **当前线程就会马上恢复执行。**
9	public final void stop()	强迫线程停止执行。已过时。不推荐使用。

线程的代理设计模式

代理模式主要使用了 Java 的多态，干活的是被代理类，代理类主要是接活，你让我干活，好，我交幕后的类去干，你满意就成，那怎么知道被代理类能不能干呢？同根就成，大家知根知底，你能做啥我能做啥都清楚的很，同一个接口呗。

Thread与Runnable的子类都实现了Runnable接口，之后将Runnable的子类MyThread的子类MyThead放到Thread类之中，测试类调用是Thread类中的start 方法去启动多线程，实际上具体的执行者Runnable的子类MyThread中的run方法中的代码

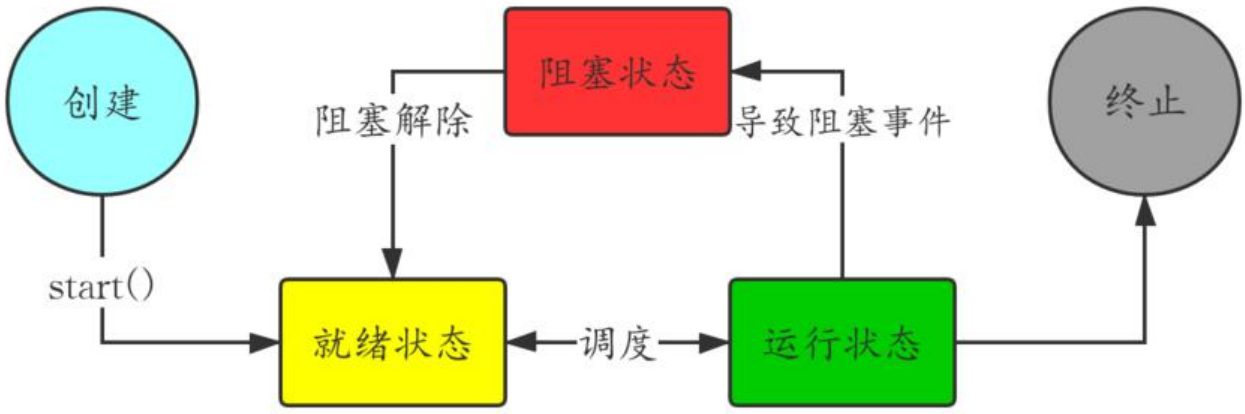


真实角色：MyThread

代理角色：Thread

实现共同接口：Runnable

线程的生命周期



- **新生状态**
  - 用new关键字建立一个线程后，该线程对象就处于新生状态
  - 处于新生状态的线程有自己的内存空间，调用start()方法进去就绪状态

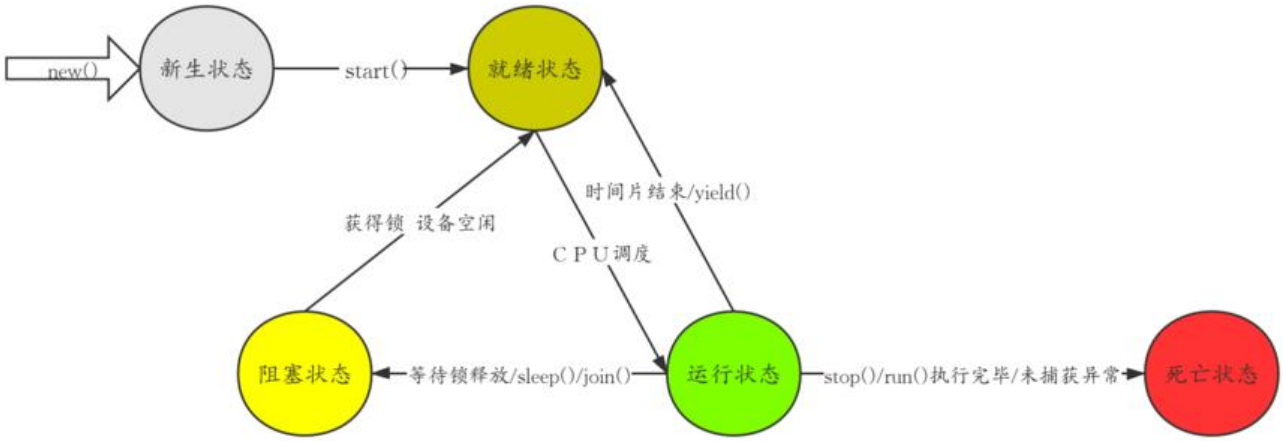
- **就绪状态**
  - 处于就绪状态线程具备了运行条件，但还没分配到CPU,处于线程就绪队列，等待系统为其分配CU
  - 当系统选定一个等待执行的线程后，它就回从就绪状态进入到执行状态，该动作被称为“CPU度“
  - java中代码实现进入就绪状态方法
    - yeild()
      - 让出cpu的使用权，从运行状态直接进入就绪状态。让CPU重新挑选哪一个线程进入运行态

- **运行状态**
  - 在运行状态的线程执行自己的run方法中的代码，直到等待某资源而完成而死亡
  - 如果在给定的时间片内没有执行结束，就会被系统给换下来回到等待执行状态

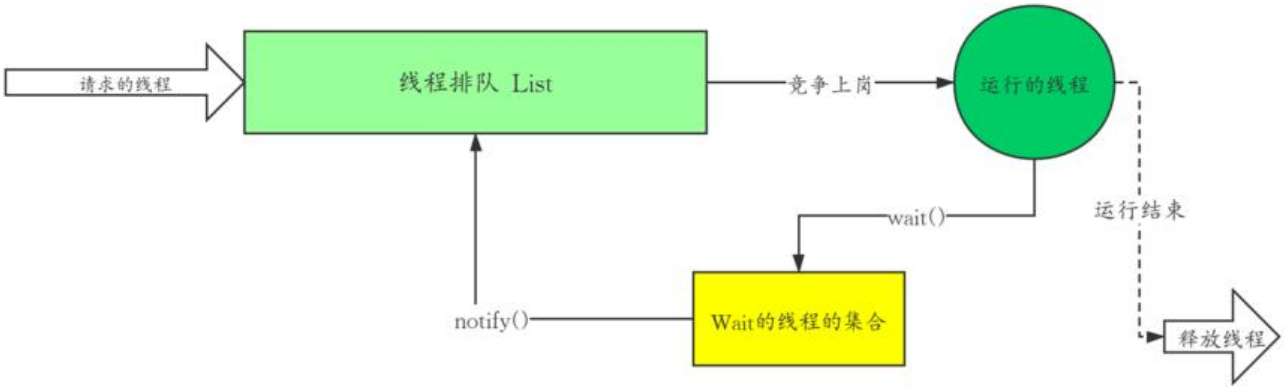
- **阻塞状态**
  - 处于运行状态的线程在某些情况下，如执行了sleep（睡眠）方法，或等待I/O设备等资源，将出CPU并暂时停止自己运行，进入阻塞状态
  - 在阻塞状态的进程不能直接进入就绪队列，只有当引起阻塞的原因消除时候，如睡眠时间已到或等待I/O设备空闲下来，线程便转入就绪状态，重新到就绪队列中排队等待，被系统选中后从原来停止状态开始继续执行
  - java中代码实现进入阻塞状态方法
    - sleep()
      - 不会释放锁，sleep时别的线程也不可以访问锁对象
    - join()
      - 当某个线程等待另一个线程执行结束后，才继续执行时，使调用该方法的线程在此之前执完毕，也就是等待调用该方法的线程执行完毕之后再往下继续执行

- **死亡状态**
  - 死亡状态是线程生命周期中的最后一个阶段。线程死亡的原因有三个
    - 正常运行的线程完成了它的全部工作
    - 线程被强制终止（如通过stop方法来终止一个线程-不推荐使用）
    - 线程抛出未捕获的异常

注意：在多线程时候，可以实现唤醒和等待的过程，但是唤醒和等待操作对应的类不是thread，而是们设置的共享对象或者共享变量（Object类中的方法）



线程同步



线程同步定义

（此处参照计算机操作系统一书中进程同步的概念,我把进程替换成了线程）

线程同步的主要任务是对多个相关线程在执行次序上进行协调，是并发执行的诸进程之间能按照一定规则（或时序）共享资源，并能很好的合作，从而使程序的执行具有可再现性。

线程同步是指多线程通过特定的设置（如互斥量，事件对象，临界区）来控制线程之间的执行顺序（所谓的同步）也可以说是在线程之间通过同步建立起执行顺序的关系。

线程同步其实实现的就是线程排队，防止线程同步访问共享资源造成冲突，变量需要同步，但是常量需要，因为常量存放于方法区。

只要这些线程的代码访问同一份可变的共享资源，这些进程之间就需要同步。

线程同步的必要性

如果没有线程同步操作，将会产生非常严重的后果。

举个生活中常见的例子：

小红和小绿是一对夫妻，小红每个月会给小绿的银行账户中存入1000元钱作为小绿的生活费，有一天红误操作存了2000进去，此时小绿正在查看账户余额忽然发现比平时钱多了一倍，喜出望外非常感动

马准备取钱出来准备去吃顿自助餐。与此同时小红也发现自己的操作失误，准备取出多出来的1000元。注意：此处小红和小绿**同时**取钱操作，是纳秒级别的并发操作。而由于银行系统没有进行线程同步作。此时会发生什么?

小绿成功的取出２０００块钱，小红成功的取出多转的１０００块钱。

明明卡里只有两千块钱，小红和小绿却取出了总金额３０００元。这么干下去，银行早倒闭了。

而此时银行的程序员小六立马发现了这个漏洞，开始考虑解决方案，都说程序员个个都智商绝顶（没冒犯的意思），很快想出了一个聪明的办法来解决这个八阿哥，既然是由于并发产生的问题，那么我它不并发不就好了。

当超过一个人同时进行取款操作时候，对这个账户余额上一把锁，同一时间（瞬时）只能让一个人进操作，其他人排队等待。当第一人操作完成之后释放锁，然后第二个人才能开始操作。

当小红和小绿发现这个财富密码之后，就又开始了薅资本主义羊毛的骚操作，这次还是同时取款，小和小绿发现这次操作时候自动取款机的程序好像变慢了一点（显示器上显示》》》》操作中请等待》》》》），这次是小绿先取出钱，然后小红的操作界面显示余额不足。此次小红和小绿的薅羊毛行动失了。

果然　排队　这个方法非常有效，再也没发生过这种事情，解决这个ｂｕｇ的代价只是让程序看起来了一丢丢而已，这对银行来说成本几乎可以忽略不计，于是银行的程序员小六很快就升职加薪并且找了同在一个银行上班的小红做自己的女朋友。

线程同步的实现(初级)

此处仅仅是线程基础内容，不会引出太多内容，不然这一个点挖出来的东西我也写不完（我还没学会）

- synchronized关键字修饰方法(同步方法)和synchronize代码块(同步代码块)
- 暂时就这两条吧，只是基础内容，更高阶的等我更新（你也可以继续Google一下）

```
/**
 * @Description TODO  同步方法
 * @Author Fedeline
 * @Date 11/19/20 12:22 PM
 */
public class TicketRunnable3 implements Runnable {
    private int ticket = 10;
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            sail();
        }
    }
    public static void main(String[] args) {
        TicketRunnable3 ticketRunnable = new TicketRunnable3();
        Thread t1 = new Thread(ticketRunnable,"A");
```

```
        Thread t2 = new Thread(ticketRunnable,"B");
        Thread t3 = new Thread(ticketRunnable,"C");
        Thread t4 = new Thread(ticketRunnable,"D");
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
    public synchronized void sail(){
        if (ticket>0){
            System.out.println(Thread.currentThread().getName()+"正在出售第"+(ticket--)+"张票");
        }
    }
}

/**
 * @Description TODO 同步代码块
 * @Author Fedeline
 * @Date 11/19/20 12:22 PM
 */
public class TicketRunnable2 implements Runnable {
    private int ticket = 10;
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (this){
                if (ticket>0){
                    System.out.println(Thread.currentThread().getName()+"正在出售第"+(ticket--)+"
票");

                }
            }


        }

    }

    public static void main(String[] args) {
        TicketRunnable2 ticketRunnable = new TicketRunnable2();
        Thread t1 = new Thread(ticketRunnable,"A");
        Thread t2 = new Thread(ticketRunnable,"B");
        Thread t3 = new Thread(ticketRunnable,"C");
        Thread t4 = new Thread(ticketRunnable,"D");
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
```

```
}
```

线程同步小结

- 同步监视器
  - synchronized(obj){}中的obj被称为同步监视器
  - 同步代码块中同步监视器可以是任何对象，但是推荐使用共享资源作为同步监视器
  - 同步方法中无需指定同步监视器，因为同步方法的监视器是this,也就是该对象本身
- 同步监视器的执行过程
  - 第一个线程访问，锁定同步监视器，执行其中代码
  - 第二个线程访问，发现同步监视器被锁定，无法访问
  - 第一个线程访问完毕，解锁同步监视器
  - 第二个线程访问，发现同步监视器未锁，锁定并访问

死锁

死锁的起因

死锁的起因，通常是源于多个线程对资源的整多，不仅对不可抢占资源进行争夺时会引起死锁，而且可消耗资源的进行争夺时，也会引起死锁。

在一组线程发生死锁的情况下，这组死锁进程中的每一个进程 ，都在等待另一个死锁进程所占用有资源。或者说每个线程所等待的事件是该组中其它线程释放所占有的资源。但是由于所有这些进程已无法运行，因此它们谁也不能释放资源，致使没有任何一个进程可被唤醒。这样这组进程只能无限期待下去。

死锁定义

同样参照计算机操作系统一书中的定义

如果每一组线程中的每个线程都在等待仅由该组线程中的其他线程才能引发的事件，那么该组进程是锁的（DeadLock）

产生死锁的必要条件

以下四个比必要条件必须同时具备才会形成死锁

- 互斥条件
  - 线程对所要访问的临界资源进行排他性使用，即在一段时间内，某一资源只能被一个线程占有如果此时还有其它线程请求该临界资源，则请求线程只能等待，直到占有该线程的资源用完临界资源释放
- 请求和保持条件
  - 线程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其他线程占有，此时求线程被阻塞，但对此时自己已经获得的资源保持不放

- 不可抢占条件
    - 线程已经获得的资源在未使用完之前不能被抢占，只能在线程使用完成完由自己释放
- 循环等待条件
    - 在发生死锁时，必然存在一个线程——资源的循环链，即进程集合｛A，B，C，D。。。F中的A正在等待B占用的资源，C正在等待D占用的一个资源，F正在等待已经被A占用的一个资源。

四个必要条件只要有一个被破坏就可以预防死锁