链滴

Java 多线程基础 & 线程间的协作与共享。

作者：DongXiaokai0819

原文链接：https://ld246.com/article/1571054789357

来源网站：链滴

Java并发编程基础

线程基础类

Java中创建一个线程有三种方式：

1. 通过继承Thread类。
2. 通过实现Runnable接口。
3. 通过Callable和Future创建线程。

```
public class NewThread {

    //runnable接口实现类
    private static class UseRunnable implements Runnable{
        @Override
        public void run() {//线程方法执行体
            System.out.println("I am a runnable interface!");
        }
    }
    //callable接口实现类
    private static class UseCallable implements Callable<String>{
        @Override
        public String call() {//线程方法执行体
            System.out.println("I am a callable interface!");
            return "callable";
        }
    }
    //Thread类子类
    private static class UseThread extends Thread{
        @Override
        public void run() {
            System.out.println("I am extends Thread!");
        }
    }

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        UseRunnable useRunnable = new UseRunnable();
        new Thread(useRunnable).start();//声明一个Thread实例，然后将runnable接口实例传入构
方法，调用start方法，即可执行线程

        UseCallable useCallable = new UseCallable();//同runnable接口一样，new一个实例对象
        FutureTask<String> stringFutureTask = new FutureTask<>(useCallable);//因为callable接
是有返回值的，所以用FutureTask对象包装一下callable对象
        new Thread(stringFutureTask).start();//然后将futuretask对象传入
        System.out.println(stringFutureTask.get());//get方法是阻塞的

        Thread useThread = new UseThread();
        useThread.start();
    }
}
```

原文链接：Java 多线程基础 & 线程间的协作与共享。

三种方法的使用步骤

通过继承Thread类的子类使用步骤

1. 直接继承Thread类，并重写run()方法，run()方法作为线程执行体。
2. 创建该子类的实例，并调用start()方法，便可启动子线程。

runnable接口的使用步骤

1. 创建runnable接口实现类，并实现run()方法，该run方法将作为线程执行体，无返回值。
2. 创建runnable实现类的实例，将其作为Thread对象的target创建，调用start方法启动新线程

callable接口使用步骤：

1. 创建callable接口实现类，并实现call()方法，该call方法将作为线程执行体，并且有返回值。
2. 创建callable实现类的实例，使用FutureTask类来包装callable对象，该FutureTask对象封装了call ble对象的call方法的返回值。
3. 使用FutureTask对象作为Thread对象的target创建并启动新线程。
4. 调用FutureTask对象的get()方法来获取子线程执行结束后的返回值。

Runnable 与 Callable 的区别?

Runnable和Callable在功能上基本是相似的，只是Callable接口中的call()方法是有返回值的，所以在用Callable的时候需要用到FutureTask对象再对Callable对象包装一下，然后再传给Thread对象，可通过FutureTask.get()方法来获得call()方法的返回值。

为什么java提供了Thread类又提供了俩个接口来实现多线程?

因为java是单继承的，但是可以多实现。

再对线程多了解一点

线程常用api

- void setDaemon(boolean on) 将此线程标记为daemon线程或用户线程。
- long getId(): 获取该线程的标识符
- String getName(): 获取该线程的名称
- void setName(Sting name): 设置该线程的名称
- boolean isAlive(): 测试线程是否处于活动状态
- void setPriority(int newPriority):设置该线程的优先级【优先级高的会大概率抢到CPU的时间片】
- void join()/join(long millis)等待指定的线程死亡/等待指定的线程死亡或者经过指定的毫秒数
- Thread.State getState() 获得线程这一时刻的状态
- static void yield() 线程让步

线程的优先级

在Java中每一个线程都有一个优先级。

默认情况下，一个线程继承它的父线程的优先级。

可用setPriority()方法设置线程的优先级，Java中线程的优先级有1-10，即MIN_PRIORITY (1)- MAX PRIORITY(10)。NORM_PRIORITY(5)。

- 每当线程调度器有机会选择新线程时，它首先会选择具有较高优先级的线程。
- 但是线程优先级高度依赖于系统，一般情况下不要依赖于线程优先级。

守护线程

Java中有俩类线程：User Thread（用户线程）、Daemon Thread（守护线程）

用个比较通俗的比如，任何一个守护线程都是整个JVM中所有非守护线程的保姆：

只要当前JVM实例中尚存在任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非护线程结束时，守护线程随着JVM一同结束工作。

Daemon的作用是为其他线程的运行提供便利服务，守护线程最典型的应用就是 GC (垃圾回收器)，就是一个很称职的守护者。

User和Daemon两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果 User Thread已经全退出运行了，只剩下Daemon Thread存在了，虚拟机也就退出了。 因为没有了被守护者，Daemon就没有工作可做了，也就没有继续运行程序的必要了。

- 可以通过 thread.setDaemon(Boolean) 方法设置线程为守护线程。
- 来通过thread.isDaemon()方法来判断线程是否为守护线程。

这里有几点需要注意：

1. thread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个IllegalThreadStateException异常。你不能把正在运行的常规线程设置为守护线程。
2. 在Daemon线程中产生的新线程也是Daemon的。
3. 不要认为所有的应用都可以分配给Daemon来进行服务，比如读写操作或者计算逻辑。
4. 优先级比较低，主要为系统中其它对象提供服务。

如何让线程停止工作

线程自然终止方式：

- 线程自然执行完成，执行完成后会自动释放资源。
- 线程执行过程中抛出未处理的异常，抛出异常后，线程自动停止，并释放资源。

Java已不建议使用的方式

- stop()方法 ，用于终止一个线程的执行，导致线程不会正确释放资源，通常是没有给予线程释放资的机会就将其杀死掉了。
- suspend()方法，用于暂停线程的执行，容易导致死锁，因为该线程在暂停的时候仍然占有该资源

这会导致其他需要该资源的线程与该线程产生环路等待，从而造成死锁。

● resume()方法，用于恢复线程的执行，需要和suspend()方法配对使用，因为被暂停的线程需要执恢复方法才能继续执行。

interrupt中断方式

● void interrupt()方法中断一个线程，并不是真的中断（关闭）这个线程，而只是将这个线程中的一属性（中断标志位）设为true，线程是否中断，由线程本身决定。

● boolean isInterrupted()方法判断当前线程是否处于中断状态。

● static boolean interrupted()方法，判断该线程是否处于中断状态，同时中断标志位改为false。

中断方式有一点小问题，就是在线程执行过程中可能会抛出InterruptedException，而抛出了这个异之后，线程的中断标志位会被复位成false，如果确实需要中断线程，要求我们自己在catch语句块里次调用interrupt()方法，将中断标志位设为true，现在来看一下这个例子。

```
    private static class TestInterruptException implements Runnable{
        @Override
        public void run() {
            String threadName = Thread.currentThread().getName();
            while (!Thread.currentThread().isInterrupted()){
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    System.out.println(threadName + "捕获到InterruptedException异常" + Thread.cur
entThread().isInterrupted());
                    e.printStackTrace();
//                    Thread.currentThread().interrupt();//中断请求
                }
                System.out.println("没有被中断!!!");
            }
            System.out.println("中断成功！！！");
        }
    }
    public static void main(String[] args) throws InterruptedException {
        TestInterruptException useRunnable = new TestInterruptException();
        Thread thread = new Thread(useRunnable);
        thread.start();//开启线程
        Thread.sleep(500);//制造InterruptedException异常
        thread.interrupt();//中断请求
    }
```

```
没有被中断!!!
Thread-0捕获到InterruptedException异常false
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at com.kk.线程基础.HasInterruptExcption$TestInterruptException.run(HasInterruptExcption.j
va:17)
    at java.lang.Thread.run(Thread.java:748)
没有被中断!!!
没有被中断!!!
没有被中断!!!
```

我们故意制造了一个InterruptedException异常，然后通过打印结果发现线程没有被中断，所以它会

直运行下去，直到下一次中断请求。

现在我们将catch里面的注释打开。然后再运行一遍。

```
没有被中断!!!
Thread-0捕获到InterruptedException异常false
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at com.kk.线程基础.HasInterruptExcption$TestInterruptException.run(HasInterruptExcption.j
va:17)
    at java.lang.Thread.run(Thread.java:748)
没有被中断!!!
中断成功！！！
```

从打印结果可以看到，中断请求成功，线程执行完成。

那么InterruptedException异常是如何产生的呢?

● 当一个线程A通过调用 threadB.interrupt() 中断线程B时，如果那个线程B在执行一个低级可中断阻方法，例如 Thread.sleep()、 Thread.join() 或 Object.wait()，那么它将取消阻塞并抛出 Interrupted xception。

线程间的共享

synchronized内置锁

synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它饰的代码块或者方法在任意时刻只能有一个线程执行。

使用synchronized的三种方式

修饰实例方法：作用于当前对象实例，进入代码前要获得当前对象实例的锁。

```
public synchronized void test(){ // TODO }
```

修饰静态方法：也就是给当前类加锁，会作用于该类的所有实例对象。

```
public static synchronized void test(){ // TODO }
```

修饰代码块：指定加锁对象，进入代码块前要获得给定对象的锁

```
synchronized (Test.class) { // TODO }
```

对象锁：锁定的是当前对象，该对象只能给一个线程用，如果一个线程调用了这个对象的同步方法，么其它线程只能等待得到对象的线程将对象释放后才能调用该对象的其它同步方法。非同步方法不影。

- synchronized修饰实例方法
- synchronized代码块传入this

类锁：即对Class对象上锁，每个类的Class对象在虚拟机种只有一个，所有该类的实例对象都共享这个Class。所以类锁也只有一个。

- synchronized修饰静态方法
- synchronized代码块传入Class对象

volatile 关键字

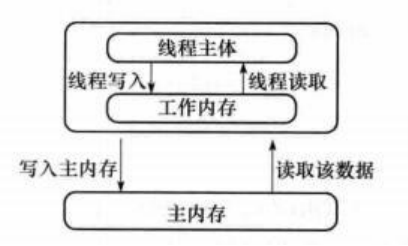
先介绍几个基本概念

1. 重排序：重排序是指为了提高指令运行的性能，在编译时或者运行时对指令执行顺序进行调整的机。重排序分为编译重排序和运行时重排序。编译重排序是指编译器在编译源代码的时候就对代码执行序进行分析，在遵循as-if-serial的原则前提下对源码的执行顺序进行调整。as-if-serial原则是指在单程环境下，无论怎么重排序，代码的执行结果都是确定的。运行时重排序是指为了提高执行的运行速，系统对机器的执行指令的执行顺序进行调整。

2. 可见性：内存的可见性是指线程之间的可见性，一个线程的修改状态对另外一个线程是可见的，用俗的话说，就是假如一个线程A修改一个共享变量flag之后，则线程B去读取，一定能读取到最新修改flag。

当前Java内存模型下的问题

在JDK1.2之前，Java的内存模型实现（即共享内存）读取变量是不需要特别注意的。而在当前的Java存模型下，每一个线程可以把变量copy一份保存在自己的工作内存中，而不是直接在主存中读写，从无法保证数据的可见性。



volatile关键字的用法及作用

用法：volatile只能用来修饰变量而无法修饰方法以及代码块。

作用：

- 用volatile修饰的变量，这也就指示JVM，这个变量是不稳定的，每次使用它经过主存。
- 保证变量的可见性，防止指令重排序。但不保证变量的原子性。

可以使用volatile的情况

- 该字段不遵循其他字段的不变式。
- 对字段的写操作不依赖于当前值。
- 没有线程违反预期的语义写入非法值。
- 读取操作不依赖于其它非volatile字段的值。

当只有一个线程可以修改字段的值，其它线程可以随意读取，那么把字段声明为volatile是合理的。

synchronized与volatile区别

1. volatile关键字是线程同步的轻量级实现，所以volatile性能肯定比synchronized关键字要好。但是olatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。synchronized关键字在

avaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以其它各种优化之后执行效率有了显著提升，实际开发中使用 synchronized 关键字的场景还是更多一。

2. 多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞

3. volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保。

4. volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个程之间访问资源的同步性。

ThreadLocal 类

ThreadLocal是一个本地线程副本变量工具类。主要用于将私有线程和该线程存放的副本对象做一个射，各个线程之间的变量互不干扰，在高并发场景下，可以实现无状态的调用，特别适用于各个线程赖不通的变量值完成操作的场景。

ThreadLocal api

- get()方法用于获取当前线程的副本变量值。
- set()方法用于保存当前线程的副本变量值。
- initialValue()为当前线程初始副本变量值。
- remove()方法移除当前前程的副本变量值。

下面来看一下一个例子，来演示一下ThreadLocal的使用

```
static ThreadLocal<Integer> threadLaocl = new ThreadLocal<Integer>(){
        @Override
        protected Integer initialValue() {
            return 1;
        }
    };

    /**
     *类说明：测试线程，线程的工作是将ThreadLocal变量的值变化，并写回，看看线程之间是否会相影响
     */
    public static class TestThread implements Runnable{
        int id;
        public TestThread(int id){
            this.id = id;
        }
        public void run() {
            System.out.println(Thread.currentThread().getName()+":start");
            Integer s = threadLaocl.get();//获得变量的值
            s = s+id;
            threadLaocl.set(s);
            System.out.println(Thread.currentThread().getName()+":"
            +threadLaocl.get());
            //threadLaocl.remove();
        }
    }
```

```
    public static void main(String[] args){
        //启动三个线程
        Thread[] runs = new Thread[3];
        for(int i=0;i<runs.length;i++){
            runs[i]=new Thread(new TestThread(i));//将i传入
        }
        for(int i=0;i<runs.length;i++){
            runs[i].start();
        }
    }
```

首次接触，先了解一下，待以后再深入分析一下。

线程间的协作

线程的生命周期与状态

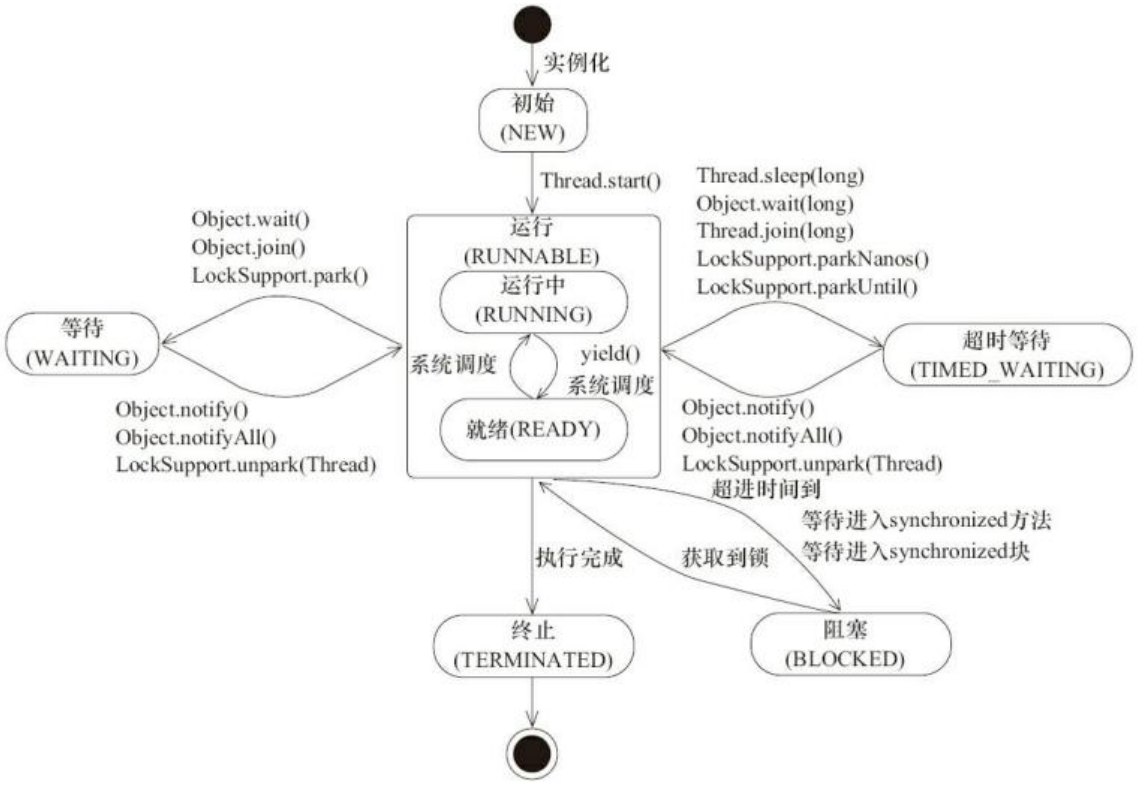
本段源自

- https://snailclimb.top/JavaGuide/#/docs/java/Multithread/JavaConcurrencyBasicsCommonInterviewQuestionsSummary
- https://blog.csdn.net/pange1991/article/details/53860651

Java线程在运行的生命周期中的指定时刻只可能处于下面6种不同状态的其中一种状态（图源《Java发编程艺术》4.1.4 节）

状态名称	说　明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状变迁如下图所示（图源《Java 并发编程艺术》4.1.4 节）：



上面那张可能看着比较混乱，下面这张是各个状态之间切换的简易版说明（图源《Java核心技术卷一 14.3.4节）

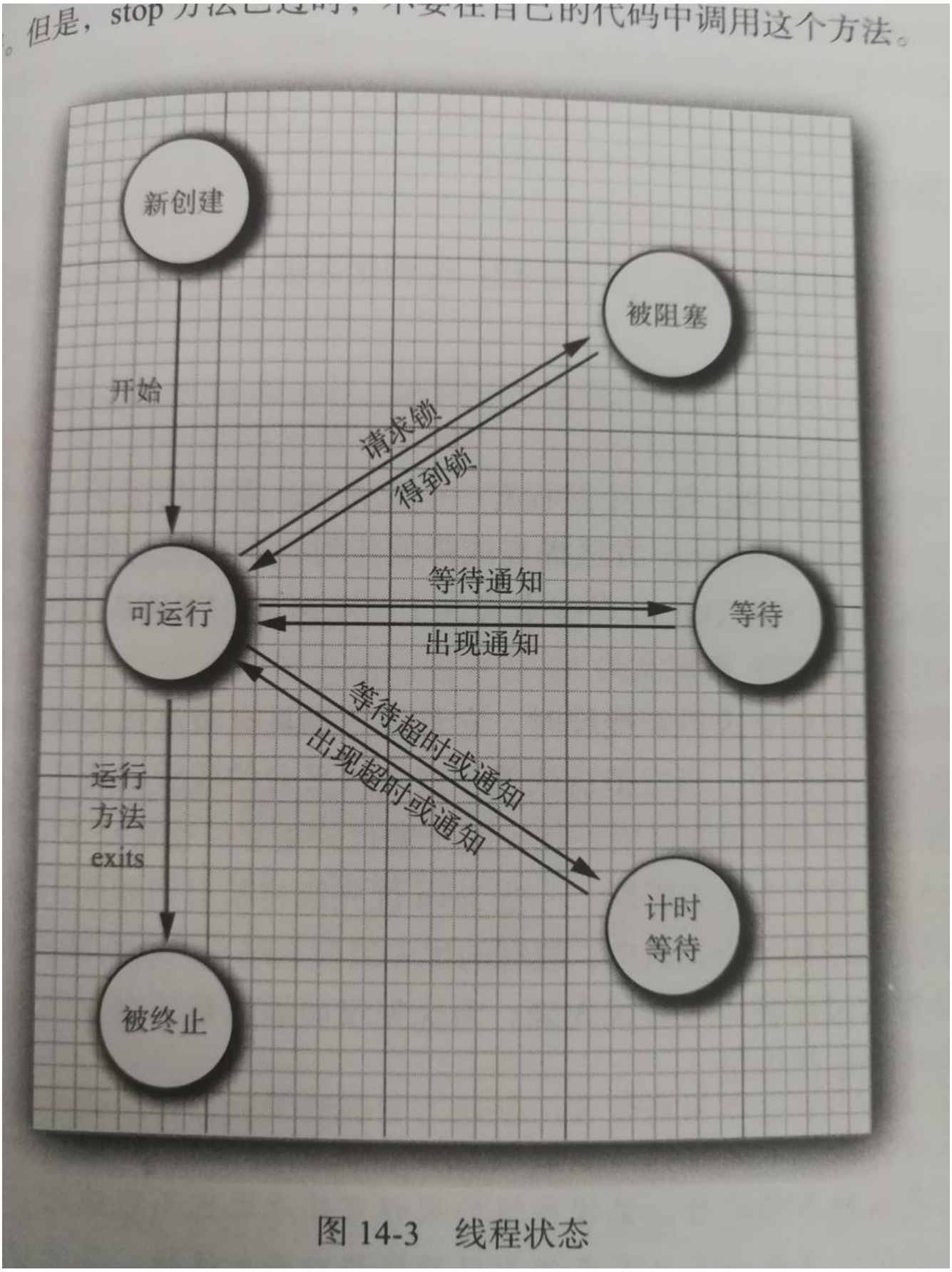


图 14-3　线程状态

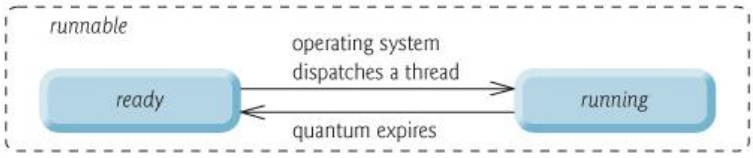
由上面可以看出：

● 线程创建之后它将处于 NEW（新建） 状态，调用 start() 方法后开始运行，线程这时候处于 READ

（可运行） 状态。

- 可运行状态的线程获得了 CPU 时间片（timeslice）后就处于 RUNNING（运行） 状态。

操作系统隐藏 Java 虚拟机（JVM）中的 READY和 RUNNING 状态，它只能看到 RUNNABLE 状态 图源：HowToDoInJava：Java Thread Life Cycle and Thread States），所以 Java 系统一般将这个状态统称为 RUNNABLE（运行中） 状态 。



- 当线程执行 wait()方法之后，线程进入 WAITING（等待） 状态。
- 进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 TIME_WAITING(超时等待 状态相当于在等待状态的基础上增加了超时限制，比如通过 sleep（long millis）方法或 wait（long millis）方法可以将 Java 线程置于 TIMED WAITING 状态。
- 当超时时间到达后 Java 线程将会返回到 RUNNABLE 状态。当线程调用同步方法时，在没有获取锁的情况下，线程将会进入到 BLOCKED（阻塞） 状态。线程在执行 Runnable 的run()方法之后将进入到 TERMINATED（终止） 状态。

下面我们来详细讲解一下，状态转换所需的方法。

线程的开始start()

线程创建之后它将处于 NEW（新建） 状态，调用 start() 方法后开始运行，线程这时候处于 READY 可运行） 状态。

但是Thread类还提供了一个run()方法，那么直接调用run方法和start方法有什么区别呢，我们通过一经典的案例来演示一下。

Thread 类 run()和start() 的区别

```
public static void main(String[] args) {
    Thread thread = new Thread(){
        public void run() {
            pong();
        }
    };
    thread.run();
   //        thread.start();
    System.out.println("Ping ");
   }
static void pong() {
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Pong ");
}
```

执行程序后可以发现

调用run方法时，打印“ Pong Ping ”

调用start方法时，打印“Ping Pong”

这也就说明了，在调用run()方法时并没有开启线程，所以程序是顺序执行的，所以打印结果为“ Pon Ping ”。而调用start()方法，开启的子线程再休眠了100ms后才打印的“Pong”，所以打印结果为Ping Pong”。

- 直接调用run方法的话，会被当成普通方法调用，在主线程中执行。
- 调用 start() 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了 start() 会执行线程的相应准备工作，然后自动执行 run() 方法的内容。（只有调用start()法后，Java会将线程对象和操作系统中的实际对象进行映射，再来执行run()方法。）

Thread.yield() 线程让步

由上述可知Thread.yield() 调用后会将线程从RUNNING状态转变为READY状态

由于Java虚拟机将操作系统种的READY与RUNNING俩种状态统称为RUNNABLE状态，所以Thread.yield() 方法在Java层面来看是在RUNNABLE状态内完成的。

虽然该线程从RUNNING状态转变为了READY状态，但这并不意味着该线程就一定中断了（不执行了，而是该线程放弃了获得的时间片机会，cpu会重新从众多的可执行态里选择，也就是说，当前也就刚刚的那个线程还是有可能会被再次执行到的，并不是说一定会执行其他线程而该线程在下一次中不执行到了。

所以说yield翻译为线程让步不太准确，有可能让了步，人家还不情愿，然后自己又死皮赖脸的继续执了。

Thread.yield()，一定是当前线程调用此方法，当前线程放弃获取的CPU时间片，但不释放锁资源，运行状态变为就绪状态，让OS再次选择线程。作用：让相同优先级的线程轮流执行，但并不保证一会轮流执行。实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选。Thread.yield()不会导致阻塞。该方法与sleep()类似，只是不能由用户指定暂停多长时间。

join()

在ThreadA 的线程方法执行ThreadB.join()方法，ThreadA必须要等待ThreadB执行完之后，Thread才能继续自己的工作

thread.join()/thread.join(long millis)，当前线程里调用其它线程t的join方法，当前线程进入WAITIG/TIMED_WAITING状态，当前线程不会释放已经持有的对象锁。线程t执行完毕或者millis时间到，前线程一般情况下进入RUNNABLE状态，也有可能进入BLOCKED状态（因为join是基于wait实现的。

等待和通知

wait() And sleep()

wait()

obj.wait()，当前线程调用对象的wait()方法，当前线程释放对象锁，进入等待队列。依靠notify()/notiyAll()唤醒或者wait(long timeout) timeout时间到自动唤醒。

sleep()

Thread.sleep(long millis)，一定是当前线程调用此方法，当前线程进入TIMED_WAITING状态，但释放对象锁，millis后线程自动苏醒进入就绪状态。作用：给其它线程执行机会的最佳方式。

sleep()和wait()方法区别和共同点

- 两者最主要的区别在于：sleep 方法没有释放锁，而 wait 方法释放了锁 。
- 两者都可以暂停线程的执行。
- Wait 通常被用于线程间交互/通信，sleep 通常被用于暂停执行。
- wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。sleep() 方法执行完成后，线程会自动苏醒。或者可以使用wait(long timeout)超时后线程会动苏醒。

notify()/notifyAll()

obj.notify()唤醒在此对象监视器上等待的单个线程，选择是任意性的。notifyAll()唤醒在此对象监视上等待的所有线程。

notify()/notifyAll()应该用谁?

应该尽量使用notifyAll() 方法，因为notify()有可能发生信号丢失的情况。

等待和通知标准范式

等待方：

1. 获取对象的锁；
2. 循环里判断条件是否满足，不满足调用wait方法
3. 条件满足执行业务逻辑

通知方

1. 获取对象的锁
2. 改变条件
3. 通知所有等待在对象的线程

```
public class Express {
    public final static String CITY = "ShangHai";
    private int km;/*快递运输里程数*/
    private String site;/*快递到达地点*/

    public Express() {
    }

    public Express(int km, String site) {
        this.km = km;
        this.site = site;
```

```
    }

    /* 变化公里数，然后通知处于wait状态并需要处理公里数的线程进行业务处理*/
    //通知方
    public synchronized void changeKm(){//synchronized 关键字获取锁
        this.km = 101;//改变条件
        notifyAll();//通知所有在等待该对象的线程，这里可改为notify()测试一下俩者的区别。
        //其他的业务代码
    }

    /* 变化地点，然后通知处于wait状态并需要处理地点的线程进行业务处理*/
    public synchronized void changeSite(){
        this.site = "BeiJing";
        notify();
    }

    public synchronized void waitKm(){//synchronized 关键字获取锁
        while(this.km<=100) {//循环里判断条件是否满足，不满足调用wait方法
            try {
                wait();
                System.out.println("check km thread["+Thread.currentThread().getId()
                        +"] is be notifed.");
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        System.out.println("the km is"+this.km+",I will change db.");//条件满足执行业务逻辑

    }

    public synchronized void waitSite(){
        while(CITY.equals(this.site)) {
            try {
                wait();
                System.out.println("check site thread["+Thread.currentThread().getId()
                        +"] is be notifed.");
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        System.out.println("the site is"+this.site+",I will call user.");
    }
}

public class TestExpress {
    private static Express express = new Express(0,Express.CITY);

    /*检查里程数变化的线程,不满足条件，线程一直等待*/
    private static class CheckKm extends Thread{
        @Override
        public void run() {
            express.waitKm();
```

```
        }
    }

    /*检查地点变化的线程,不满足条件，线程一直等待*/
    private static class CheckSite extends Thread{
        @Override
        public void run() {
            express.waitSite();
        }
    }

 public static void main(String[] args) throws InterruptedException {
     for(int i=0;i<3;i++){//三个线程
         new CheckSite().start();
     }
     for(int i=0;i<3;i++){//里程数的变化
         new CheckKm().start();
     }

     Thread.sleep(1000);
     express.changeKm();//快递地点变化
 }
```

上述代码输出如下：

```
check km thread[17] is be notifed.
the km is101,I will change db.
check km thread[16] is be notifed.
the km is101,I will change db.
check km thread[15] is be notifed.
the km is101,I will change db.
check site thread[14] is be notifed.
check site thread[13] is be notifed.
check site thread[12] is be notifed.
```

用notifyAll()方法可唤醒在此对象监视器上等待的所有线程，代码正常运行。

如果用notify()方法的话，我们在main方法里改变了km的值，通知方调用notify()只唤醒了一个线程但是我们启动了六个线程，三个监听km的，三个监听site的，所以notify只唤醒一个线程的话，是可唤醒的线程就不是km的等待方，从而造成死锁。

等待超时模式

由于上述，经典的等待/通知标准范式，无法做到超时等待，也就是说，消费者（等待方）在获得锁，如果条件不满足，等待生产者改变条件之前会一直处于等待状态，在一些实际的应用中，这样会非浪费资源，降低运行效率。

所以可对上述模式的消费者（等待方）做一下小改动：

long overtime = now+T; long remain = T;//等待的持续时间 while(result不满足条件&& remain 0){ wait(remain); remain = overtime – now;//等待剩下的持续时间 } return result;

调用yield() 、sleep()、wait()、notify()等方法对锁有何影响?

- yield()： 线程执行yield()方法后，不释放锁

- sleep() ：线程执行sleep()方法后，不释放锁
- wait()： 调用wait()方法之前，线程必须要持有锁，调用时，锁会被释放，当wait()方法返回时，锁会释放
- notify()： 调用方法之前，必须要持有锁，调用notify()方法本身不会释放锁的

参考

- https://snailclimb.top/JavaGuide/#/?id=java
- https://blog.csdn.net/pange1991/article/details/53860651
- https://www.jianshu.com/p/98b68c97df9b
- 《Java核心技术卷一》
- 《Java编程思想》