链滴

Java 多线程、锁机制、lock 锁

作者：whoms

原文链接：https://ld246.com/article/1638272813653

来源网站：链滴

1.多线程

- 程序的并行

程序的并行指的是真正意义上的同时执行，CPU分配多个执行单元共同执行任务，效率高，但是依赖CPU的多核心硬件支持，单核处理器的CPU是不能并行处理多个任务的。

- 程序的并发

程序的并发指的是多个应用程序交替执行，CPU分配给每个应用程序一些“执行时间片”用于执行该用程序，由于CPU处理速度极快，并且“执行时间片”极短，给人造成视觉上的误差，让人认为是在同时执行”，其实是在交替执行

虽然是交替执行，但是程序的并发解决了多个程序之间不能“同时”执行的问题，并且程序的并发利CPU的空余时间，能将CPU的性能更好的发挥，另外并发不受CPU硬件的限制。

- 进程

是指内存中运行的应用程序，一个应用最少具备一个进程，也可能有多个进程，每个进程都有一个独的内存空间，进程是系统运行的基本单位

- 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，是一个程序内部的一条执行路径，一个程中至少有一个线程，一个进程中是可以有多个线程的。

- Java线程

Java使用java.lang.Thread类代表线程，所有的线程对象必须是Thread类或其子类的实例。

继承Thread类都将变为线程类，调用Thread类中的start()方法即开启线程；当线程开启后，将会执行hread类中的run方法，因此我们要做的就是重写Thread中的run方法，将线程要执行的任务自己定义。

- 线程执行流程

程序首先开始运行会开启main线程执行代码，执行start()方法时开启一条新的线程来执行任务，新的程与main线程争夺CPU的执行权，两个线程交替执行。

当开启一个新线程之后，JVM在栈内存开辟一块新的内存空间，每个线程都有自己独立的栈空间，进方法的弹栈和压栈。线程和线程之间的栈内存独立，堆内存和方法区共享，一个线程一个栈。

- Runnable接口

Thread执行的run方法实质就是执行Runnable接口中的run方法，所以可以传递一个Runnable对象给hread。

- 使用Runnable接口创建线程

1. 定义Runnable接口的实现类，并重写该接口的run方法。
2. 创建Runnable实现类的实例，并以此实例作为Thread的target来创建Thread对象，该Thead对象才是真正的线程对象。
3. 调用线程对象的start()方法启动线程。

- Thread和Runnable的区别

如果一个类继承Thread，则不适合资源共享，但是如果实现了Runnable接口的话，则很容易实现资共享

1. 适合多个相同的程序代码的线程去共享同一个资源
2. 可以避免Java中单继承的局限性
3. 增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和线程独立
4. 线程池只能放入实现Runnable或Callable类线程，不能直接放入继承Thread的类

扩充：在Java中，每次程序运行至少启动两个线程，一个是main线程，一个是垃圾回收线程。

- 线程的休眠

public static void sleep(long millis)：让当前线程睡眠指定的毫秒数

带锁休眠，当该线程在执行有锁的任务休眠，会带着锁一起休眠，其他线程都要等到该线程休眠结束行完成任务并释放锁之后才能执行。

- 线程的加入

多条线程时，当前指定线程调用join方法时，线程执行权交给该线程，并且当前线程必须等该线程执完毕之后才会执行，但有可能被其他线程抢走执行权

- public final void join()：让线程在当前线程优先执行,直至t线程执行完毕时,再执行当前线程
- public final void join(long millis)：让线程执行millis毫秒，然后将线程执行器抛出，给其线程争抢。

- 守护线程

当用户线程（非守护线程）执行完毕，守护线程也会停止执行，但由于CPU运行速度过快，当用户线执行完毕时，将信息传递给守护线程会有时间差，导致还会执行一点守护线程

不管开启多少个用户线程，守护线程总是随着第一个用户线程的停止而停止。

- public final void setDaemon(boolean on)：设置线程是否为守护线程

- 线程优先级

默认情况下，所有的线程优先级默认为5，最高为10，最低为1，优先级高的线程更容易抢到执行权。

- public final void setPriority(int newPriority)：设置线程的优先级。

- 线程礼让

在多线程执行时，线程礼让，告知当前线程可以将执行权礼让给其他线程，礼让给优先级相对高一点线程，但仅仅是一种告知，并不是强制将执行权转让给其他线程，当前线程将CPU执行权礼让出去后也有可能下次的执行权还在原线程这里；如果想让原线程强制让出执行权，可以使用join()方法。

- public static void yield()：将当前线程的CPU执行权礼让出来。

2.线程安全

当多条线程操作同一个资源时，就会产生线程安全问题。

- 线程同步

当使用多个线程访问同一资源时，且多个线程对资源有写的操作，就会出现线程安全问题，要解决安性问题，Java提供了同步机制（synchronize）

- 同步代码块

synchronized 关键字可以用于方法中的某个区块中，表示只对这个区块的资源实行互斥访问。

在任何时候，最多允许一个线程拥有同步锁,谁拿到锁就进入代码块,其他的线程只能在外等着(BLOCKD)。

只有把同步代码块中的代码执行完毕才会释放锁。

```
synchronized(同步锁){
  //代码块
}
```

同步锁：

- 锁对象可以是任意类型，除基本类型
- 多个线程对象要使用同一把锁

- 同步方法

使用synchronized修饰的方法，就叫做同步方法，保证线程执行该方法时，其他线程只能在方法外等。

同步方法也有锁对象，非静态方法的锁对象是this，静态方法的锁对象是当前类的字节码对象（.class）

```
public synchronized void method(){
  //代码块
}
```

- Lock锁

java.util.concurrent.locks.Lock机制提供了比synchronized代码块和synchronized方法更广泛的锁操作，同步代码块/同步方法具有的功能Lock都有，除此之外更强大，更体现面向对象。

Lock锁也称为同步锁，加锁和释放锁方法化了

- public void lock()：加同步锁
- public void unlock()：释放同步锁

- 死锁

多线程同步时，如果同步代码嵌套使用相同锁，就有可能出现死锁

```
public class Demo01 {
```

```
    public static void main(String[] args) {
        String s1 = "s1";
        String s2 = "s2";
        new Thread() {
            public void run() {
                while (true) {
                    synchronized (s1) {
                        System.out.println(this.getName() + "s1");
                        synchronized (s2) {
                            System.out.println(this.getName() + "s2");
                        }
                    }
                }
            }
        }.start();
        new Thread() {
            public void run() {
                while (true) {
                    /*
                    当线程0拿到s1执行s1中的打印语句时,如果线程切换到线程1
                    那么线程1拿到了s2锁对象,此时就造成了线程死锁
                    线程1想执行s1锁里面的代码执行不了,因为s1在线程0中还没有释放
                    那么此时线程1就会切换到线程0
                    线程0也不会执行s2锁里面的代码,因为此时s2已经被线程0中的锁拿去了
                    还没有释放,因此造成了线程的死锁
                    两个都没有释放  都卡住了  线程就卡住了
                     */
                    synchronized (s2) {
                        System.out.println(this.getName() + "s2");
                        synchronized (s1) {
                            System.out.println(this.getName() + "s1");
                        }
                    }
                }
            }
        }.start();
    }
}
```