第9课

多线程

SOFTCITS@2016

SoftCITS SOFT CULTURE IT SALON 软件园IT文化沙龙

线程概述

使用多线程的优点：

1. 进程之间不能共享内存，但线程之间可以共享内存

2. 系统创建进程需要重新分配系统资源，但创建线程的代价小得多，因为使用多线程实现多任务并发比多进程的效率高

3. Java相对于操作系统的多线程支持更加简单

Thread类

使用Thread类启动多线程的步骤：

1. 定义一个子类来继承Thread类，并重写run()方法，该run()方法内部就是线程需要完成的任务，因此run()方法被称为线程执行体
2. 创建该子类的实例，即创建了线程对象
3. 调用线程对象的start()方法来启动线程

使用继承Thread主类创建的线程，多线程之间无法共享实例变量！！！

示例： ThreadDemo

Runnable类

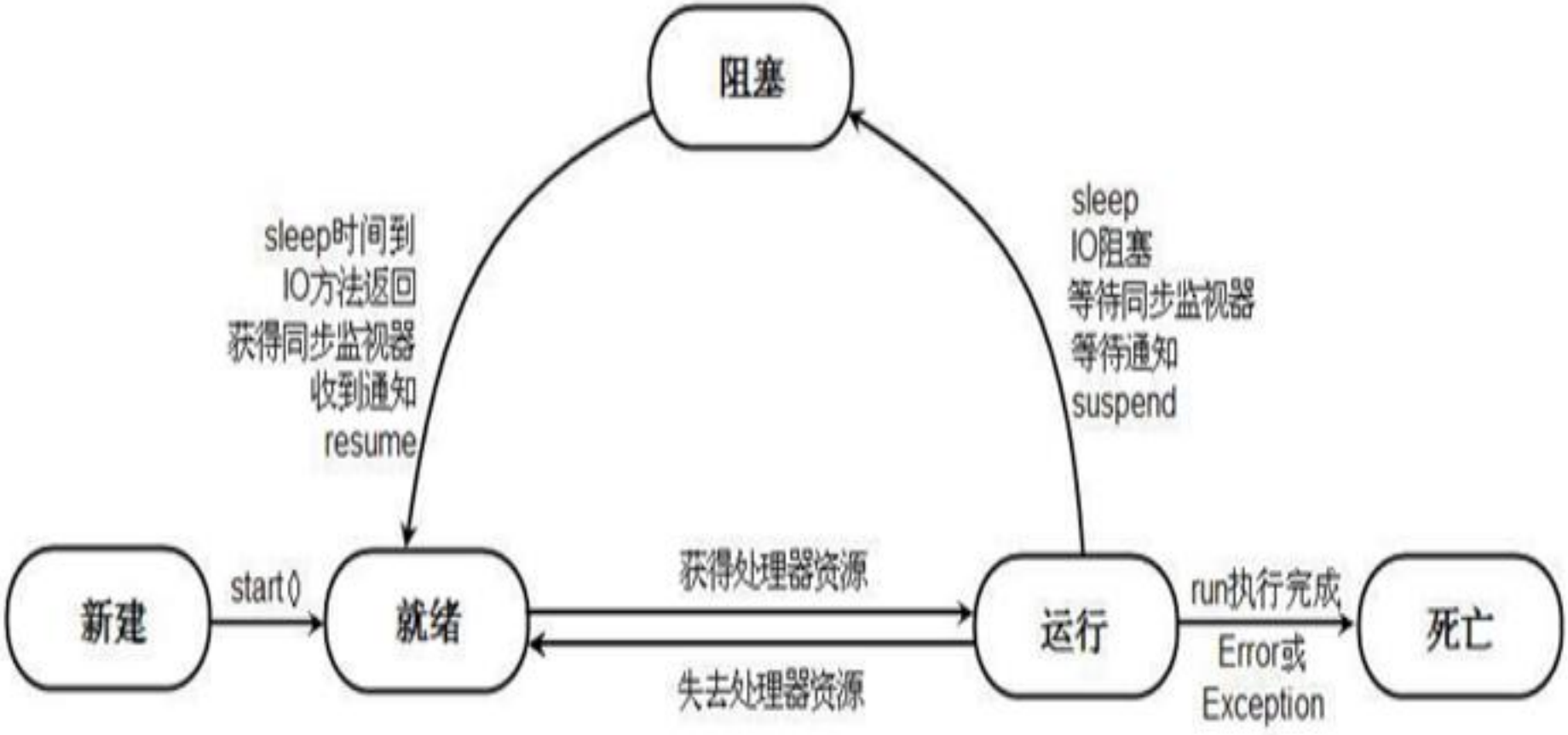
使用Runnable类的步骤：

1. 定义Runnable接口的实现类，并重写run()方法

2. 创建1中的实现类对象实例，并在此实例上创建多个多线程实例，从而实现共享该实例变量

代码： RunnableDemo

线程的状态和生命周期


阻塞
sleep时间到
IO方法返回
获得同步监视器
收到通知
resume
sleep
IO阻塞
等待同步监视器
等待通知
suspend
新建
start()
就绪
获得处理器资源
失去处理器资源
运行
run执行完成
Error或
Exception
死亡

join线程

Thread提供了让一个线程等待另一个线程完成的方法：join() 方法。当在某个程序执行流中调用其他线程的join()方法时，调用线程将被阻塞，直到被join方法加入的join线程完成为止。

代码 JoinThread

后台线程

有一种线程，它是在后台运行的，它的任务是为其他的线程提供服务，这种线程被称为“后台线程（Daemon Thread）”，又称为“守护线程”。JVM的垃圾回收线程就是典型的后台线程。

后台线程有个特征：如果所有的前台线程都死亡，后台线程会自动死亡。

调用Thread对象setDaemon(true)方法可将指定线程设置成后台线程，并且必须在start()方法之前调用。

代码： DaemonThread

线程睡眠

•如果我们需要让当前正在执行的线程暂停一段时间，并进入阻塞状态，则可以通过调用Thread类的静态sleep方法，sleep方法：

- static void sleep(long millis)：让当前正在执行的线程暂停millis毫秒，并进入阻塞状态，该方法受到系统计时器和线程调度器的精度和准确度的影响。

代码： SleepTest

线程让步

yield()方法是一个和sleep方法有点相似的方法，它也是一个Thread类提供的一个静态方法，它也可以让当前正在执行的线程暂停，但它不会阻塞该线程。yield只是让当前线程暂停一下，让系统的线程调度器重新调度一次，完全可能的情况是：当某个线程调用了yield方法暂停之后，线程调度器又将其调度出来重新执行。

实际上，当某个线程调用了yield方法暂停之后，只有优先级与当前线程相同，或者优先级比当前线程更高的、就绪状态的线程才会获得执行的机会。

代码： YieldTest

线程睡眠和线程让步

- sleep方法暂停当前线程后，会给其他线程执行机会，不会理会其他线程的优先级。但yield方法只会给优先级相同，或优先级更高的线程执行机会。
- sleep方法会将线程转入阻塞状态，直到经过阻塞时间才会转入就绪状态。而yield不会将线程转入阻塞状态，它只是强制当前线程进入就绪状态。因此完全有可能某个线程调用yield方法暂停之后，立即再次获得处理器资源被执行。
- sleep方法声明抛出了InterruptedException异常，所以调用sleep方法时要么捕捉该异常，要么显式声明抛出该异常。而yield方法则没有声明抛出任何异常。
- sleep方法比yield方法有更好的可移植性，通常不要依靠yield来控制并发线程的执行。

线程安全问题

•多条线程并发修改共享资源就容易引发线程安全问题

代码： Account & DrawThread & DrawTest

同步代码块的引入：

Java的多线程支持引入了同步监视器来解决这个问题，使用同步监视器的通用方法就是同步代码块。

synchronized(obj)后括号里的obj就是线程开始执行同步代码块之前对共享资源锁定。

选择监视器的目的：阻止两条线程对同一个共享资源进行并发访问。因此通常推荐使用可能被并发访问的共享资源充当同步监视器。对于取钱模拟程序，我们应该考虑使用账户（account）作为同步监视器。

```
public void run(){
//任何进行进入同步代码块之前必须先锁定account,即“加锁-修改-释放锁”
synchronized(account){
        .....
        }
}
```

线程安全问题

同步方法的引入：对于synchronized修饰的方法（非static方法），无须显示指定同步监视器，同步方法的同步监视器是this，也就是调用该方法的对象

代码： 包synchronizedMethod

释放同步监视器

- 线程会在如下几种情况下释放对同步监视器的锁定：
  - 当前线程的同步方法、同步代码块执行结束，当前线程即释放同步监视器。
  - 当线程在同步代码块、同步方法中遇到break、return终止了该代码块、该方法的继续执行，当前线程将会释放同步监视器。
  - 当线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致了该代码块、该方法异常结束时将会释放同步监视器。
  - 当线程执行同步代码块或同步方法时，程序执行了同步监视器对象的wait()方法，则当前线程暂停，并释放同步监视器。
- 线程会在如下几种情况下不会释放对同步监视器的锁定：
  - 线程执行同步代码块或同步方法时，程序调用sleep(),yield()方法来暂停当前线程执行，当前线程不会释放同步监视器

死锁

•两个线程互相等待对方释放同步监视器就会发生死锁，一旦出现死锁，程序不会发出任何异常，也不提供任何来自JVM的提示，只是处于阻塞状态，无法继续。

代码：DeadLock

线程通信

•以借助于Object类提供的wait()、notify()和notifyAll()三个方法，这三个方法并不属于Thread类，而是属于Object类。但这三个方法必须同步监视器对象调用。

•关于这三个方法的解释如下：

- wait()：导致当前线程等待，直到其他线程调用该同步监视器的notify()方法或notifyAll()方法来唤醒该线程。该wait()方法有三种形式：无时间参数的wait（一直等待，直到其他线程通知），带毫秒参数的wait和带毫秒、微秒参数的wait（这两种方法都是等待指定时间后自动苏醒）。调用wait()方法的当前线程会释放对该同步监视器的锁定。
- notify()：唤醒在此同步监视器上等待的单个线程。如果所有线程都在此同步监视器上等待，则会选择唤醒其中一个线程。选择是任意性的。只有当前线程放弃对该同步监视器的锁定后（使用wait()方法），才可以执行被唤醒的线程。
- notifyAll()：唤醒在此同步监视器上等待的所有线程。只有当前线程放弃对该同步监视器的锁定后，才可以执行被唤醒的线程。

•对于使用synchronized修饰的方法，同步监视器是this实例，所以可以直接调用上面的方法。

•对于使用synchronized修饰的代码块,同步监视器是括号里的实例，所以必须使用该实例调用上面的方法。

代码：包 threadCommunication

阻塞队列控制线程通信

•BlockingQueue接口是Queue的子接口，它的主要用途就是作为线程同步的工具，当生产者向队列放入元素时，如果队列已满，则阻塞该线程；当消费者从队列取出元素时，如果队列已空，则阻塞该线程

代码：BlockingQueueDemo

线程池

•系统启动一个新线程的成本是比较高的，因为它涉及到与操作系统交互。在这种情形下，使用线程池可以很好地提高性能，尤其是当程序中需要创建大量生存期很短暂的线程时，更应该考虑使用线程池。

•线程池在系统启动时即创建大量空闲的线程，程序将一个Runnable对象传给线程池，线程池就会启动一条线程来执行该对象的run方法，当run方法执行结束后，该线程并不会死亡，而是再次返回线程池中成为空闲状态，等待执行下一个Runnable对象的run方法。

•可以通过设置最大线程数控制系统中的线程数量，从而避免因线程过多导致JVM崩溃

步骤：

（1）调用Executors类的静态工厂方法创建一个ExecutorService对象或ScheduledExecutorService对象，其中前者代表简单的线程池，后者代表能以任务调度方式执行线程的线程池。

（2）创建Runnable实现类或Callable实现类的实例，作为线程执行任务。

（3）调用ExecutorService对象的submit方法来提交Runnable实例或Callable实例；或调用ScheduledExecutorService的schedule来执行线程。

（4）当不想提交任何任务时调用ExecutorService对象的shutdown方法来关闭线程池。

代码：ThreadPoolTest

ThreadLocal类

•ThreadLocal，是Thread Local Variable（线程局部变量）的意思，功用非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，使每一个线程都可以独立地改变自己的副本，而不会和其它线程的副本冲突。从线程的角度看，就好像每一个线程都完全拥有该变量。

- ThreadLocal类的用法非常简单，它只提供了如下三个public方法：
- T get()：返回此线程局部变量中当前线程副本中的值。
- void remove()：删除此线程局部变量中当前线程的值。
- void set(T value)：设置此线程局部变量中当前线程副本中的值。

代码：ThreadLocalTest

包装线程不安全集合

•如果程序有多条线程可能访问以上ArrayList、HashMap等集合，可以使用Collections提供的静态方法来把这些集合包装成线程安全的集合。Collections提供了如下几个静态方法：

- static <T> Collection<T> synchronizedCollection(Collection<T> c)：返回指定 collection 对应的线程安全的collection。
- static <T> List<T> synchronizedList(List<T> list)：返回指定List对应的线程安全的List对象。
- static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)：返回指定Map对象对应的线程安全的Map对象。
- static <T> Set<T> synchronizedSet(Set<T> s)：返回指定Set对应的线程安全的Set。

例如：HashMap m = Collections.synchronizedMap(new HashMap())

练习

1.编写两个线程，线程1打印1~9数字，线程2打印a~j字母，要求输出1a2b3c4d...

2.用三个线程，按顺序打印1~30，要求线程1打印1~3，然后线程2打印4~6，线程3打印7~9，线程1打印10~12，线程2打印13~15 。。。以此类推

3. 有三个线程1，2，3， 其中1打印A， 2打印B，2打印C，请循环打印10次ABCABC...

谢谢观看

SeeYou！

SoftCITS 软件园IT文化沙龙

SOFT CULTURE IT SALON