链滴

030 多线程和并行程序设计

作者：pzs233
原文链接：https://ld246.com/article/1537288722185
来源网站：链滴

30.1 引言

Java 的重要功能之一就是内部支持多线程——在一个程序中允许同时运行多个任务。

30.2 线程的概念

一个程序可能包含多个可以同时运行的任务。线程是指一个任务从头到尾的执行流程。

在 Java 中，每个任务都是 Runnable 接口的一个实例，也称为 可运行对象（runnable object）。程本质上讲就是便于任务执行的对象。

30.3 创建任务和线程

一个任务必须实现 Runnable 接口。任务必须从线程运行。

- 定义一个实现 Runnable 接口的类。Runnable 接口只包含一个 run 方法。需要实现这个方法来告系统将如何运行。

```
//Custom task class
public class TaskClass implements Runnable {
    ...
    public TaskClass (...){

    }

    //Implement the run method in Runnable
    public void run() {
        //Tell system how to custom thread
    }
}
```

- 用它的构造方法创建任务对象
- 把任务对象放到线程中
- 调用 start() 方法告诉 Java 虚拟机 运行该线程
- Java 虚拟机通过调用任务的 run() 方法执行任务

任务中的 run() 方法指明如何完成任务。Java虚拟机会自动调用该方法，无需特意调用它。直接调用方法只是在同一个线程中执行该方法，而没有新线程被启动。

```
//Client class
public class Client {
    ...
    public void someMethod() {
        ...
        //Create an instance of TaskClass
        TaskClass task = new TaskClass(...);

        //Create a thread
        Thread thread = new Thread(task);
```

```
        //Start a thraad
        thread.start();
        ...
    }
    ...
}
```

30.4 Thread类

Thtread 类包含 为任务而创建的线程的构造方法，以及控制线程的方法。

```
+Thread()
+Thread(task: Runnable)
+start(): void
+isAlive(): boolean
+setPriority(p: int): void
+join(): void
+sleep(millis: long): void（静态方法）
+yield(): void（静态方法）
+interrupt(): void
```

由于 Thread 类实现了 Runnable，所以，可以定义一个 Thread 的拓展类，并且实现 run 方法。然在客户端程序创建这个类的一个对象，并且调用它的 start 方法来启动线程。但是，**不推荐使用**，因它**将任务和运行任务的机制混在一起**。将任务从线程中分离出来是比较好的设计。

可以使用 yield() 方法为其他线程临时让出 CPU 时间。eg：

```
public void run (){
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
        Thread.yield();
    }
}
```

方法 sleep(millis: long) 可以将该线程设置为休眠以确保其他线程的执行，休眠时间为指定的毫秒数 eg：

```
public void run() {
    try {
        for (int i = 1; i <= lastNum; i++){
            System.out.print(" " + i);
            if (i > 50) Thread.sleep(1);
        }
    }
    catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

sleep 方法可能抛出一个 InterruptedException，这是一个必检异常。当一个休眠线程的 interrupt() 方法被调用时，就会发生这样一个必检异常。但 interrupt() 方法极少在线程上被调用。

因为 Java 强制捕获必检异常，所以，必须将它放到 try-catch 块中。**如果在一个循环中调用了 sleep**

方法，那就应该将这个循环放到 try-catch 块中，eg：

```
public void run() {
    try {
        while (...) {
            ...
            Thread.sleep(1000);
        }
    }
    catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

但，如果循环在 try-catch 块外，即使线程被中断，它可能继续执行，eg：

```
public void run() {
    while(...) {
        try {
            ...
            Thread.sleep(1000);
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

可以使用 join() 方法使一个线程等待另一个线程结束。

调用 join() 方法的线程将暂停，等待 join() 方法所在线程先用运行结束再用运行。

如下面的程序中，thread3 线程到达50时，将等待thread4先运行，之后再继续运行。

Java 给每个线程指定一个优先级。默认情况下，线程继承生成它的线程的优先级。可以用 setPriority 方法提高或降低线程的优先级，还可用 getPriority 方法获取线程的优先级。优先级从 1 到 10 。Thread 类有 int 型常量 MIN_PRIORITY、NORM_PRIORITY 和 MAX_PRIORITY ，分别代表 1、5、10。线程的优先级是 Thread.NORM_PRIORITY。

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TaskThreadDemo {
    public static void main(String[] args) {

        ExecutorService executor = Executors.newFixedThreadPool(3);

        //Submit runnable tasks to the executor
        executor.execute(new PrintChar('a', 100));
        executor.execute(new PrintChar('b', 100));
        executor.execute(new PrintNum(100));

        //Shut down the executor
        executor.shutdown();
    }
```

```
}

class PrintChar implements Runnable {
    private char charToPrint;
    private int times;

    public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
    }

    @Override
    public void run () {
        for (int i = 0; i < times; i++){
            System.out.print(charToPrint);
            // Thread.yield();
        }
    }
}

class PrintNum implements Runnable {
    private int lastNum;

    public PrintNum(int n) {
        lastNum = n;
    }

    @Override
    public void run() {
        for (int i = 1; i <= lastNum; i++){
            System.out.print(" " + i);
        }
    }
    // public void run() {
    //     Thread thread4 = new Thread(new PrintChar('c', 400));
    //     thread4.start();
    //     try {
    //         for (int i = 1; i <= lastNum; i++) {
    //             System.out.print(" " + i);
    //             if (i == 50) thread4.join();
    //         }
    //     }
    //     catch (InterruptedException ex) {
    //         ex.printStackTrace();
    //     }
    // }
}
```

30.5 实例学习：闪烁文本

30.6 线程池

线程池是管理并发执行任务个数的理想方法。

Java 提供 Executor 接口来执行线程池中的任务，提供 ExecutorService 接口来管理和控制任务。 ExcutorService 是 Executor 的子接口。

```
<<interface>>java.util.concurrent.Executor
+execute(Runnable object): void

<<interface>>java.util.concurrent.ExecutorService
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
+isTerminated(): boolean

java.util.concurrent.Executors
+newFixedThreadPool(numberOFThreads: int): ExecutorService
+newCachedThreadPool(): ExecutorService
```

newFixedThreadPool(int) 方法在池中创建固定数目的线程。如果线程完成了任务的执行，它可以被新使用以执行另一个任务。如果线程池中所有的线程都不是处于空闲状态，而且还有任务在等待执行那么在关闭之前，如果由于一个错误终止了一个线程，就会创建一个新的线程来替代它。如果线程池所有的线程都不是空闲状态，而且有任务在等待执行，那么 newCachedThreadPool() 方法就会创建个新线程。如果缓冲池中的线程在 60 秒内都没有被使用就该终止它。

30.7 线程同步

线程同步 用于协调互相依赖的线程的执行。

如果一个共享资源被多个线程同时访问，可能会遭到破坏。

多线程中的多个任务以一种会引起冲突的方法访问一个公共资源。这是多线程程序中的一个普遍问题称为 **竞争状态**（race condition）。如果一个类的对象在多线程中没有竞争状态，则称这样的类为线安全的（thread-safe）。

30.7.1 synchronized 关键字

为避免竞争状态，应该防止多个线程同时进入程序的某一特定部分，程序中的这部分称为 临界区（critical region）。

可以使用 关键字 synchroinzed 来同步这个临界区，以便一次只有一个线程可以访问这个方法。

一个同步方法在执行之前需要加锁。锁是一种实现资源排他使用的机制。对于实例方法，要给调用该法的对象加锁。对于静态方法，要给这个类加锁。

30.7.2 同步语句

当执行方法中的某一个代码块时，同步语句不仅可用于对 this 对象加锁，而且可用于对任何对象加锁这个代码块称为 同步块（synchronized block）。同步语句的一般形式如下：

```
synchronized (expr) {
   statements;
}
```

表达式 expr 求值结果（括号内的）必须是一个对象的引用。如果对象已经被另一个线程锁定，则在锁之前，该线程将被阻塞。当获准对一个对象加锁时，该线程执行同步块中的语句，然后解除给对象

加的锁。

同步语句允许设置同步方法中的部分代码，而不必是整个方法。这大大增强了程序的并发能力。

30.8 利用加锁同步

可以显式地采用锁和状态来同步线程。

一个锁是一个 Lock 接口的实例，它定义了加锁和释放锁的方法。

```
<<interface>>java.util.concurrent.locks.Lock
+lock(): void
+unlock(): void
+newCondition(): Condition
```

ReentrantLock 是 Lock 的一个具体实现，用于创建互相排斥的锁。可以创建具有特定的 公平策略锁。公平策略值为 真，则确保 等待时间最长的线程首先获得锁。取值为 假 的公平策略将锁给任意一在等待的线程。

```
java.util.concurrent.locks.ReentrantLock
+ReentrantLock()
+ReentrantLock( fair: boolean )
```

在对 lock() 的调用之后紧随一个 try-catch 块并且在 finally 子句中释放这个锁是一个很好的编程习。

30.9 线程间协作

锁上的条件可以用于协调线程之前的交互。

一个线程可以指定在某种条件下该做什么。

条件是通过调用 Lock 对象的 newCondition() 方法而创建的对象。一旦创建了条件，就可以使用 await()、singnal() 和 signalAll() 方法来实现线程之间的互相通信。

```
<<interface>>java.util.concurrent.Condition
+await(): void
+signal(): void
+signalAll(): Condition
```

```
package com.zhisheng.ijp30;

import java.util.concurrent.*;

/**
 * @author zhisheng
 * @date 2018-08-15
 */

public class ThreadCooperation {
    private static Account account = new Account();

    public static void main(String[] args){
        /**
```

```
     * Create a thread pool with two threads
     */
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new DepositTask());
    executor.execute(new WithrawTask());
    executor.shutdown();

    System.out.println("Thread 1\t\tThread 2\t\t\tBalance");
}

public static class DepositTask implements Runnable {
    @Override
    public void run() {
        try {
            while (true) {
                account.deposit((int)(Math.random() * 10) + 1);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}

public static class WithrawTask implements Runnable {
    @Override
    public void run() {
        while (true) {
            account.withdraw((int)(Math.random() * 10) + 1);
        }
    }
}

private static class Account {
    private static Lock lock = new ReentrantLock();

    private  static Condition newDeposit = lock.newCondition();

    private int balance = 0;

    public int getBalance() {
        return balance;
    }

    public void withdraw(int amount) {
        lock.lock();
        try {
            while (balance < amount) {
                System.out.println("\t\t\t\tWait for a deposit");
                newDeposit.await();
            }

            balance -= amount;
```

```
            System.out.println("\t\t\t\tWithdraw " + amount + "\t\t\t\t" + getBalance());
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        finally {
            lock.unlock();
        }
    }

    public  void deposit(int amount) {
        lock.lock();
        try {
            balance += amount;
            System.out.println("Deposit " + amount + "\t\t\t\t\t\t\t" + getBalance());

            newDeposit.signalAll();
        }
        finally {
            lock.unlock();
        }
    }
  }
}
```

一旦线程调用条件上的 await()，线程就进入等待状态，等待恢复信号。如果忘记对状态调用 signal() signalAll(),那么线程就会永远等待下去。

条件由 Lock 对象创建。为了调用它的方法（如：await()、signal()和 signalAll()），必须首先拥有。如果没有获取锁就调用这些方法，会抛出 IllegalMonitorStateException 异常。

30.10 实例学习：生产者/消费者

演示线程的协调：

```
package chapter30;

import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ConsumerProducer {
  private static Buffer buffer = new Buffer();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }

  // A task for adding an int to the buffer
  private static class ProducerTask implements Runnable {
```

```
  public void run() {
    try {
      int i = 1;
      while (true) {
        System.out.println("Producer writes " + i);
        buffer.write(i++); // Add a value to the buffer
        // Put the thread into sleep
        Thread.sleep((int)(Math.random() * 10000));
      }
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}

// A task for reading and deleting an int from the buffer
private static class ConsumerTask implements Runnable {
  public void run() {
    try {
      while (true) {
        System.out.println("\t\t\tConsumer reads " + buffer.read());
        // Put the thread into sleep
        Thread.sleep((int)(Math.random() * 10000));
      }
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}

// An inner class for buffer
private static class Buffer {
  private static final int CAPACITY = 1; // buffer size
  private java.util.LinkedList<Integer> queue =
    new java.util.LinkedList<Integer>();

  // Create a new lock
  private static Lock lock = new ReentrantLock();

  // Create two conditions
  private static Condition notEmpty = lock.newCondition();
  private static Condition notFull = lock.newCondition();

  public void write(int value) {
    lock.lock(); // Acquire the lock
    try {
      while (queue.size() == CAPACITY) {
        System.out.println("Wait for notFull condition");
        notFull.await();
      }

      queue.offer(value);
      notEmpty.signal(); // Signal notEmpty condition
    } catch (InterruptedException ex) {
```

```
      ex.printStackTrace();
    } finally {
      lock.unlock(); // Release the lock
    }
  }

  public int read() {
    int value = 0;
    lock.lock(); // Acquire the lock
    try {
      while (queue.isEmpty()) {
        System.out.println("\t\t\tWait for notEmpty condition");
        notEmpty.await();
      }

      value = queue.remove();
      notFull.signal(); // Signal notFull condition
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    } finally {
      lock.unlock(); // Release the lock
      return value;
    }
  }
 }
}
```

30.11 阻塞队列

阻塞队列（blocking queue）在试图向一个满队列添加元素或从空队列中删除元素时会导致线程阻
。

```
<<interface>>java.util.concurrent.BlockingQueue<E>
+put( element: E ): void
+take(): E
```

三个具体阻塞队列:

- ArrayBlockingQueue <E>
- LinkedBlockingQueue <E>
- PriorityBlockingQueue <E>

```
ArrayBlockingQueue<E>
+ArrayBlockingQueue(capacity: int)
+ArrayBlockingQueue(capacity: int, fair: boolean)

LinkedBlockingQueue<E>
+LinkedBlockingQueue()
+LinkedBlockingQueue(capacity: int)

PriorityBlockingQueue<E>
+PriorityBlockingQueue()
+PriorityBlockingQueue(capacity: int)
```

实例，简化消费者生产者：

```
package chapter30;

import java.util.concurrent.*;

public class ConsumerProducerUsingBlockingQueue {
  private static ArrayBlockingQueue<Integer> buffer =
    new ArrayBlockingQueue<Integer>(2);

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }

  // A task for adding an int to the buffer
  private static class ProducerTask implements Runnable {
    public void run() {
      try {
        int i = 1;
        while (true) {
          System.out.println("Producer writes " + i);
          buffer.put(i++); // Add any value to the buffer, say, 1
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

  // A task for reading and deleting an int from the buffer
  private static class ConsumerTask implements Runnable {
    public void run() {
      try {
        while (true) {
          System.out.println("\t\t\tConsumer reads " + buffer.take());
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

上一个例子中，使用锁和条件同步生产者和消费者线程。在这个程序中，因为同步已经在 ArrayBlockingQueue 中实现，所以无需使用锁和条件。

30.12 信号量

可以使用 信号量 来限制访问一个共享资源的 线程数。

```
java.util.concurrent.Semaphore
+Semaphore(numberOfPermits: int)
+Semaphore(numberOfPermits: int, fair: boolean)
+acquire(): void
+release(): void
```

只有一个许可的信号量可以用来模拟一个互相排斥的锁。

30.13 避免死锁

使用一种称为 资源排序 的简单技术可以轻易的避免死锁的发生。

该技术是给每一个需要锁的对象制定一个顺序，确保每个线程都按这个顺序来获取锁。

30.14 线程状态

新建、就绪、运行、阻塞、结束。

30.15 同步合集

Java 合集框架为 线性表、集合 和 映射表。

Java 合集框架中的类不是线程安全的。可以通过锁定合集或同步合集来保护合集中的数据。

Collections 类提供 6 个静态方法来将 合集 转成 同步版本。使用这些方法创建的合集称为 同步包装 。

```
java.tuil.Collections（以下都为静态方法）
+synchronizedCollection(c: Collection): Collection
+synchronizedList(list: List): List
+synchronizedMap(m: Map): Map
+synchronizedSet(s: Set): Set
+synchronizedSortedMap(s: SortedMap): SortedMap
+synchronizedSortedSet(s: SortedSet): SortedSet
```

调用 synchronizedCollection(Collection c) 会返回一个新的 Collection 对象，在它里面所有访问和新原来的合集 c 的方法都被同步。这些方法使用 synchronized 关键字来实现，eg：

```
public boolean add(E o) {
    synchronized (this) {
        return c.add(0);
    }
}
```

同步合集可以很安全地被多线程并发访问和修改。

迭代器具有 快速失效的特性。这意味着当使用一个迭代器对一个合集进行遍历，而其依赖的合集被另个线程修改时，那么迭代器会抛出异常 java.util.ConcurrentModificationException 报错，该异常是 RuntimeException 的一个子类。为了避免这个错误，需要创建一个同步合集对象，并且做遍历它时

取对象上的锁，eg：

```
Set hashSet = Collections.synchronizedSet(new HashSet());
synchronized (hashSet) {
    Iterator iterator = hashSet.iterator();
    while (iterator.hasNext()) {
        Syetem.out.println(iterator.next());
    }
}
```

30.16 并行编程

Fork/Join 框架 用于在 Java 中实现并行编程。

框架是使用 ForkJoinTask 类定义一个任务，同时，在一个 ForkJoinPool 的实例中执行一个任务。

ForkJoinTask 是用于任务的抽象基类，是 Future 的子类。

```
<<interface>>java.util.concurrent.Future<V>
+cancel(interrupt: boolean): boolean
+get(): V
+isDone(): boolean

<<interface>>java.util.concurrent.ForkJoinTask<V>
+adapt(Runnable task): ForkJoinTask<V>
+fork(): ForkJoinTask<V>
+join(): V
+invoke(): V
+invokeAll(tasks ForkJoinTask<?>...): void
```

RecursiveAction 和 RecursiveTask 是 ForkJoinTask 的两个子类。**要定义具体的任务类，类应该继自 这两个子类。RecursiveAction 用于不返回值的任务。RecursiveTask 用于返回值的任务。任务应该重写 compute() 方法来指定任务是如何执行的。**

```
java.util.concurrent.RecursiveAction<V>
#compute(): void

java.util.concurrent.RecursiveTask<V>
#compute(): v
```

ForkJoinPool 继承接口 java.tuil.concurrent.ExecutorService。

```
java.util.concurrent.ForkJoinPool
+ForkJoinPool()
+ForkJoinPool(parallelism: int)
+invoke(ForkJoinTask<T>): T
```

用 Fork/Join 框架开发 归并排序：

```
package chapter30;

import java.util.concurrent.RecursiveAction;
import java.util.concurrent.ForkJoinPool;
import chapter23.MergeSort;
```

```
public class ParallelMergeSort {
  public static void main(String[] args) {
    final int SIZE = 7000000;
    int[] list1 = new int[SIZE];
    int[] list2 = new int[SIZE];

    for (int i = 0; i < list1.length; i++)
      list1[i] = list2[i] = (int)(Math.random() * 10000000);

    long startTime = System.currentTimeMillis();
    parallelMergeSort(list1); // Invoke parallel merge sort
    long endTime = System.currentTimeMillis();
    System.out.println("\nParallel time with "
      + Runtime.getRuntime().availableProcessors() +
      " processors is " + (endTime - startTime) + " milliseconds");

    startTime = System.currentTimeMillis();
    MergeSort.mergeSort(list2); // MergeSort is in Listing 24.5
    endTime = System.currentTimeMillis();
    System.out.println("\nSequential time is " +
      (endTime - startTime) + " milliseconds");
  }

  public static void parallelMergeSort(int[] list) {
    RecursiveAction mainTask = new SortTask(list);
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(mainTask);
  }

  private static class SortTask extends RecursiveAction {
    private final int THRESHOLD = 500;
    private int[] list;

    SortTask(int[] list) {
      this.list = list;
    }

    @Override
    protected void compute() {
      if (list.length < THRESHOLD)
        java.util.Arrays.sort(list);
      else {
        // Obtain the first half
        int[] firstHalf = new int[list.length / 2];
        System.arraycopy(list, 0, firstHalf, 0, list.length / 2);

        // Obtain the second half
        int secondHalfLength = list.length - list.length / 2;
        int[] secondHalf = new int[secondHalfLength];
        System.arraycopy(list, list.length / 2,
          secondHalf, 0, secondHalfLength);

        // Recursively sort the two halves
```

```
        invokeAll(new SortTask(firstHalf),
          new SortTask(secondHalf));

        // Merge firstHalf with secondHalf into list
        MergeSort.merge(firstHalf, secondHalf, list);
      }
    }
  }
}
```