Программирование на языке Java. Легковесные процессы и синхронизация

Картузов А.В.

Параллельное программирование, связанное с использованием легковесных процессов, или подпроцессов (multithreading, light-weight processes) — концептуальная парадигма, в которой вы разделяете свою программу на два или несколько процессов, которые могут исполняться одновременно.

Замечание:

Во многих средах параллельное выполнение заданий представлено в том виде, который в операционных системах называется многозадачностью. Это совсем не то же самое, что параллельное выполнение подпроцессов. В многозадачных операционных системах вы имеете дело с полновесными процессами, в системах с параллельным выполнением подпроцессов отдельные задания называются легковесными процессами (light-weight processes, threads).

1. Цикл обработки событий в случае единственного подпроцесса

В системах без параллельных подпроцессов используется подход, называемый циклом обработки событий. В этой модели единственный подпроцесс выполняет бесконечный цикл, проверяя и обрабатывая возникающие события. Синхронизация между различными частями программы происходит в единственном цикле обработки событий. Такие среды называют синхронными управляемыми событиями системами. Apple Macintosh, Microsoft Windows, X11/Motif — все эти среды построены на модели с циклом обработки событий.

Если вы можете разделить свою задачу на независимо выполняющиеся подпроцессы и

Copyright © 1999-2004 Javable.com. All rights reserved.

можете автоматически переключаться с одного подпроцесса, который ждет наступления события, на другой, которому есть чем заняться, за тот же промежуток времени вы выполните больше работы. Вероятность того, что больше чем одному из подпроцессов одновременно надолго потребуется процессор, мала.

2. Модель легковесных процессов в Java

Исполняющая система Java в многом зависит от использования подпроцессов, и все ее классовые библиотеки написаны с учетом особенностей программирования в условиях параллельного выполнения подпроцессов. Java использует подпроцессы для того, чтобы сделать среду программирования асинхронной. После того, как подпроцесс запущен, его выполнение можно временно приостановить (suspend). Если подпроцесс остановлен (stop), возобновить его выполнение невозможно.

3. Приоритеты подпроцессов

Приоритеты подпроцессов — это просто целые числа в диапазоне от 1 до 10 и имеет смысл только соотношения приоритетов различных подпроцессов. Приоритеты же используются для того, чтобы решить, когда нужно остановить один подпроцесс и начать выполнение другого. Это называется *переключением контекста.* Правила просты. Подпроцесс может добровольно отдать управление — с помощью явного системного вызова или при блокировании на операциях ввода-вывода, либо он может быть приостановлен принудительно. В первом случае проверяются все остальные подпроцессы, и управление передается тому из них, который готов к выполнению и имеет самый высокий приоритет. Во втором случае, низкоприоритетный подпроцесс, независимо от того, чем он занят, приостанавливается принудительно для того, чтобы начал выполняться подпроцесс с более высоким приоритетом.

4. Синхронизация

Поскольку подпроцессы вносят в ваши программы асинхронное поведение, должен существовать способ их синхронизации. Для этой цели в Java реализовано элегантное развитие старой модели синхронизации процессов с помощью *монитора.*

Copyright © 1999-2004 Javable.com. All rights reserved.

5. Сообщения

Коль скоро вы разделили свою программу на логические части—подпроцессы, вам нужно аккуратно определить, как эти части будут общаться друг с другом. Java предоставляет для этого удобное средство — два подпроцесса могут “общаться” друг с другом, используя методы wait и notify. Работать с параллельными подпроцессами в Java несложно. Язык предоставляет явный, тонко настраиваемый механизм управления созданием подпроцессов, переключения контекстов, приоритетов, синхронизации и обмена сообщениями между подпроцессами.

6. Подпроцесс

Класс Thread инкапсулирует все средства, которые могут вам потребоваться при работе с подпроцессами. При запуске Java-программы в ней уже есть один выполняющийся подпроцесс. Вы всегда можете выяснить, какой именно подпроцесс выполняется в данный момент, с помощью вызова статического метода Thread.currentThread. После того, как вы получите дескриптор подпроцесса, вы можете выполнять над этим подпроцессом различные операции даже в том случае, когда параллельные подпроцессы отсутствуют. В очередном нашем примере показано, как можно управлять выполняющимся в данный момент подпроцессом.

```
class CurrentThreadDemo {
        public static void main(String args[]) {
                Thread t = Thread.currentThread();
                t.setName("My Thread");
                System.out. println("current thread: " + t);
                try {
                        for (int n = 5; n > 0; n—) {
                                System.out.println(" " + n);
                                Thread.sleep(1000);
                        }
                }
                catch (InterruptedException e) {
                        System.out.println("interrupted");
                        }
                }
```


Copyright © 1999-2004 Javable.com. All rights reserved.

```
        }
```

В этом примере текущий подпроцесс хранится в локальной переменной t. Затем мы используем эту переменную для вызова метода setName, который изменяет внутреннее имя подпроцесса на “My Thread”, с тем, чтобы вывод программы был удобочитаемым. На следующем шаге мы входим в цикл, в котором ведется обратный отсчет от 5, причем на каждой итерации с помощью вызова метода Thread.sleep() делается пауза длительностью в 1 секунду. Аргументом для этого метода является значение временного интервала в миллисекундах, хотя системные часы на многих платформах не позволяют точно выдерживать интервалы короче 10 миллисекунд. Обратите внимание — цикл заключен в try/catch блок. Дело в том, что метод Thread.sleep() может возбуждать исключение InterruptedException. Это исключение возбуждается в том случае, если какому-либо другому подпроцессу понадобится прервать данный подпроцесс. В данном примере мы в такой ситуации просто выводим сообщение о перехвате исключения. Ниже приведен вывод этой программы:

```
C:\>; java CurrentThreadDemo
        current thread: Thread[My Thread,5,main]
        5
        4
        3
        2
        1
```

Обратите внимание на то, что в текстовом представлении объекта Thread содержится заданное нами имя легковесного процесса — My Thread. Число 5 — это приоритет подпроцесса, оно соответствует приоритету по умолчанию, “main” — имя группы подпроцессов, к которой принадлежит данный подпроцесс.

7. Runnable

Не очень интересно работать только с одним подпроцессом, а как можно создать еще один? Для этого нам понадобится другой экземпляр класса Thread. При создании нового объекта Thread ему нужно указать, какой программный код он должен выполнять. Вы можете запустить подпроцесс с помощью любого объекта, реализующего интерфейс Runnable. Для того, чтобы реализовать этот интерфейс, класс должен предоставить определение метода run. Ниже приведен пример, в котором

Copyright © 1999-2004 Javable.com. All rights reserved.

создается новый подпроцесс.

```
class ThreadDemo implements Runnable {
        ThreadDemo() {
                Thread ct = Thread.currentThread();
                System.out.println("currentThread: " + ct);
                Thread t = new Thread(this, "Demo Thread");
                System.out.println("Thread created: " + t);
                t.start();
                try {
                        Thread.sleep(3000);
                        }
                catch (InterruptedException e) {
                        System.out.println("interrupted");
                        }
                System.out.println("exiting main thread");
                }

        public void run() {
                try {
                        for (int i = 5; i > 0; i—) {
                                System.out.println("" + i);
                                Thread.sleep(1000);
                                }
                        }
                catch (InterruptedException e) {
                        System.out.println("child interrupted");
                        }
                System.out.println("exiting child thread");
        }
        public static void main(String args[]) {
                new ThreadDemo();
                }
        }
```

Обратите внимание на то, что цикл внутри метода run выглядит точно так же, как и в предыдущем примере, только на этот раз он выполняется в другом подпроцессе. Подпроцесс main с помощью оператора new Thread(this, "Demo Thread") создает новый объект класса Thread, причем первый параметр конструктора — this — указывает, что

Copyright © 1999-2004 Javable.com. All rights reserved.

нам хочется вызвать метод run текущего объекта. Затем мы вызываем метод start, который запускает подпроцесс, выполняющий метод run. После этого основной подпроцесс (main) переводится в состояние ожидания на три секунды, затем выводит сообщение и завершает работу. Второй подпроцесс — "Demo Thread" — при этом по-прежнему выполняет итерации в цикле метода run до тех пор пока значение счетчика цикла не уменьшится до нуля. Ниже показано, как выглядит результат работы этой программы этой программы после того, как она отработает 5 секунд.

```
C:\> java ThreadDemo
        Thread created: Thread[Demo Thread,5,main]
        5
        4
        3
        exiting main thread
        2
        1
        exiting child thread
```

8. Приоритеты подпроцессов

Если вы хотите добиться от Java предсказуемого независимого от платформы поведения, вам следует проектировать свои подпроцессы таким образом, чтобы они по своей воле освобождали процессор. Ниже приведен пример с двумя подпроцессами с различными приоритетами, которые не ведут себя одинаково на различных платформах. Приоритет одного из подпроцессов с помощью вызова setPriority устанавливается на два уровня выше Thread. NORM_PRIORITY, то есть, умалчиваемого приоритета. У другого подпроцесса приоритет, наоборот, на два уровня ниже. Оба этих подпроцесса запускаются и работают в течение 10 секунд. Каждый из них выполняет цикл, в котором увеличивается значение переменной-счетчика. Через десять секунд после их запуска основной подпроцесс останавливает их работу, присваивая условию завершения цикла while значение true и выводит значения счетчиков, показывающих, сколько итераций цикла успел выполнить каждый из подпроцессов.

```
class Clicker implements Runnable {
        int click = 0;
```


Copyright © 1999-2004 Javable.com. All rights reserved.

```
        private Thread t;
        private boolean running = true;
        public clicker(int p) {
                t = new Thread(this);
                t.setPriority(p);
        }
        public void run() {
                while (running) {
                        click++;
                        }
                }
        public void stop() {
                running = false; }
                public void start() {
                t.start();
                }
        }

class HiLoPri {
        public static void main(String args[]) {
                Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
                clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
                clicker lo = new clicker(Thread.NORM_PRIORITY—2);
                lo.start();
                hi.start();
                try Thread.sleep(-10000) {
                        }
                catch (Exception e) {
                        }
                lo.stop();
                hi.stop();
                System.out.println(lo.click + " vs. " + hi.click);
                }
        }
```

По значениям, фигурирующим в распечатке, можно заключить, что подпроцессу с низким приоритетом достается меньше на 25 процентов времени процессора:

```
C:\>java HiLoPri
        304300 vs. 4066666
```


Copyright © 1999-2004 Javable.com. All rights reserved.

9. Синхронизация

Когда двум или более подпроцессам требуется параллельный доступ к одним и тем же данным (иначе говоря, к совместно используемому ресурсу), нужно позаботиться о том, чтобы в каждый конкретный момент времени доступ к этим данным предоставлялся только одному из подпроцессов. Java для такой синхронизации предоставляет уникальную, встроенную в язык программирования поддержку. В других системах с параллельными подпроцессами существует понятие монитора. Монитор—это объект, используемый как защелка. Только один из подпроцессов может в данный момент времени владеть монитором. Когда под-процесс получает эту защелку, говорят, что он вошел в монитор. Все остальные подпроцессы, пытающиеся войти в тот же монитор, будут заморожены до тех пор пока подпроцесс-владелец не выйдет из монитора.

У каждого Java-объекта есть связанный с ним неявный монитор, а для того, чтобы войти в него, надо вызвать метод этого объекта, отмеченный ключевым словом synchronized. Для того, чтобы выйти из монитора и тем самым передать управление объектом другому подпроцессу, владелец монитора должен всего лишь вернуться из синхронизованного метода.

```
class Callme {
        void call(String msg) {
        System.out.println("[" + msg);
        try Thread.sleep(-1000) {}
        catch(Exception e) {}
        System.out.println("]");
        }
}

class Caller implements Runnable {
        String msg;
        Callme target;
        public Caller(Callme t, String s) {
                target = t;
                msg = s;
                new Thread(this).start();
```


Copyright © 1999-2004 Javable.com. All rights reserved.

```
                }
        public void run() {
                target.call(msg);
                }
        }
        class Synch {
                public static void main(String args[]) {
                Callme target = new Callme();
                new Caller(target, "Hello.");
                new Caller(target, "Synchronized");
                new Caller(target, "World");
                }
        }
```

Вы можете видеть из приведенного ниже результата работы программы, что sleep в методе call приводит к переключению контекста между подпроцессами, так что вывод наших 3 строк-сообщений перемешивается:

```
        Hello.
        Synchronized
        World
```

Это происходит потому, что в нашем примере нет ничего, способного помешать разным подпроцессам вызывать одновременно один и тот же метод одного и того же объекта. Для такой ситуации есть даже специальный термин—race condition (состояние гонки), означающий, что различные подпроцессы пытаются опередить друг друга, чтобы завершить выполнение одного и того же метода. В этом примере для того, чтобы это состояние было очевидным и повторяемым, использован вызов sleep. В реальных же ситуациях это состояние, как правило, трудноуловимо, поскольку непонятно, где именно происходит переключение контекста, и этот эффект менее заметен и не всегда воспроизводятся от запуска к запуску программы. Так что если у вас есть метод (или целая группа методов), который манипулирует внутренним состоянием объекта, используемого в программе с параллельными подпроцессами, во избежание состояния гонки вам следует использовать в его заголовке ключевое слово synchronized.

10. Взаимодействие подпроцессов

Copyright © 1999-2004 Javable.com. All rights reserved.

В Java имеется элегантный механизм общения между подпроцессами, основанный на методах wait, notify и notifyAll. Эти методы реализованы, как final-методы класса Object, так что они имеются в любом Java-классе. Все эти методы должны вызываться только из синхронизованных методов. Правила использования этих методов очень просты:

- wait—приводит к тому, что текущий подпроцесс отдает управление и переходит в режим ожидания—до тех пор пока другой под-процесс не вызовет метод notify с тем же объектом.
- notify—выводит из состояния ожидания первый из подпроцессов, вызвавших wait с данным объектом.
- notifyAll—выводит из состояния ожидания все подпроцессы, вызвавшие wait с данным объектом.

Ниже приведен пример программы с наивной реализацией проблемы поставщик-потребитель. Эта программа состоит из четырех простых классов: класса Q, представляющего собой нашу реализацию очереди, доступ к которой мы пытаемся синхронизовать; поставщика (класс Producer), выполняющегося в отдельном подпроцессе и помещающего данные в очередь; потребителя (класс Consumer), тоже представляющего собой подпроцесс и извлекающего данные из очереди; и, наконец, крохотного класса PC, который создает по одному объекту каждого из перечисленных классов.

```
class Q {
        int n;
        synchronized int get() {
                System.out.println("Got: " + n);
                return n;
                }
        synchronized void put(int n) {
                this.n = n;
                System.out. println("Put: " + n);
                }
        }

class Producer implements Runnable {
        Q q;
```


Copyright © 1999-2004 Javable.com. All rights reserved.

```
        Producer(Q q) {
                this.q = q;
                new Thread(this, "Producer").start();
                }
        public void run() {
                int i = 0;
                while (true) {
                        q.put(i++);
                        }
                }
        }

class Consumer implements Runnable {
        Q q;
        Consumer(Q q) {
                this.q = q;
                new Thread(this, "Consumer").start();
                }
        public void run() {
                while (true) {
                        q.get();
                        }
                }
        }

class PC {
        public static void main(String args[]) {
                Q q = new Q();
                new Producer(q);
                new Consumer(q);
                }
        }
```

Хотя методы put и get класса Q синхронизованы, в нашем примере нет ничего, что бы могло помешать поставщику переписывать данные по того, как их получит потребитель, и наоборот, потребителю ничего не мешает многократно считывать одни и те же данные. Так что вывод программы содержит вовсе не ту последовательность сообщений, которую нам бы хотелось иметь:

```
C:\> java PC
```


Copyright © 1999-2004 Javable.com. All rights reserved.

```
        Put: 1
        Got: 1
        Got: 1
        Got: 1
        Got: 1
        Got: 1
        Put: 2
        Put: 3
        Put: 4
        Put: 5
        Put: 6
        Put: 7
        Got: 7
```

Как видите, после того, как поставщик помещает в переменную n значение 1, потребитель начинает работать и извлекает это значение 5 раз подряд. Положение можно исправить, если поставщик будет при занесении нового значения устанавливать флаг, например, заносить в логическую переменную значение true, после чего будет в цикле проверять ее значение до тех пор пока поставщик не обработает данные и не сбросит флаг в false.

Правильным путем для получения того же результата в Java является использование вызовов wait и notify для передачи сигналов в обоих направлениях. Внутри метода get мы ждем (вызов wait), пока Producer не известит нас (notify), что для нас готова очередная порция данных. После того, как мы обработаем эти данные в методе get, мы извещаем объект класса Producer (снова вызов notify) о том, что он может передавать следующую порцию данных. Соответственно, внутри метода put, мы ждем (wait), пока Consumer не обработает данные, затем мы передаем новые данные и извещаем (notify) об этом объект-потребитель. Ниже приведен переписанный указанным образом класс Q.

```
class Q {
        int n;
        boolean valueSet = false;
        synchronized int get() {
                if (!valueSet)
                try wait();
                catch(InterruptedException e):
```


Copyright © 1999-2004 Javable.com. All rights reserved.

```
                System.out.println("Got: " + n);
                valueSet = false;
                notify();
                return n;
                }
        synchronized void put(int n) {
                if (valueSet)
                try wait(); catch(InterruptedException e);
                this.n = n;
                valueSet = true;
                System.out.println("Put: " + n);
                notify();
                }
        }
```

А вот и результат работы этой программы, ясно показывающий, что синхронизация достигнута.

```
C:\> java Pcsynch
        Put: 1
        Got: 1
        Put: 2
        Got: 2
        Put: 3
        Got: 3
        Put: 4
        Got: 4
        Put: 5
        Got: 5
```

11. Клинч (deadlock)

Клинч—редкая, но очень трудноуловимая ошибка, при которой между двумя легковесными процессами существует кольцевая зависимость от пары синхронизированных объектов. Например, если один подпроцесс получает управление объектом X, а другой—объектом Y, после чего X пытается вызвать любой синхронизированный метод Y, этот вызов, естественно блокируется. Если при этом и Y попытается вызвать синхронизированный метод X, то программа с такой структурой подпроцессов окажется заблокированной навсегда. В самом деле, ведь для того, чтобы

Copyright © 1999-2004 Javable.com. All rights reserved.

один из подпроцессов захватил нужный ему объект, ему нужно снять свою блокировку, чтобы второй подпроцесс мог завершить работу.

Сводка функций программного интерфейса легковесных процессов

Ниже приведена сводка всех методов класса Thread, обсуждавшихся в этой главе.

12. Методы класса

Методы класса—это статические методы, которые можно вызывать непосредственно с именем класса Thread.

12.1. currentThread

Статический метод currentThread возвращает объект Thread, выполняющийся в данный момент.

12.2. yield

Вызов метода yield приводит к тому, что исполняющая система переключает контекст с текущего на следующий доступный подпроцесс. Это один из способов гарантировать, что низкоприоритетные подпроцессы когда-нибудь получат управление.

```
        sleep(int n)
```

При вызове метода sleep исполняющая система блокирует текущий подпроцесс на n миллисекунд. После того, как этот интервал времени закончится, подпроцесс снова будет способен выполняться. В большинстве исполняющих систем Java системные часы не позволяют точно выдерживать паузы короче, чем 10 миллисекунд.

13. Методы объекта

13.1. start

Метод start говорит исполняющей системе Java, что необходимо создать системный

Copyright © 1999-2004 Javable.com. All rights reserved.

контекст подпроцесса и запустить этот подпроцесс. После вызова этого метода в новом контексте будет вызван метод run вновь созданного подпроцесса. Вам нужно помнить о том, что метод start с данным объектом можно вызвать только один раз.

13.2. run

Метод run—это тело выполняющегося подпроцесса. Это—единственный метод интерфейса Runnable. Он вызывается из метода start после того, как исполняющая среда выполнит необходимые операции по инициализации нового подпроцесса. Если происходит возврат из метода run, текущий подпроцесс останавливается.

13.3. stop

Вызов метода stop приводит к немедленной остановке подпроцесса. Это—способ мгновенно прекратить выполнение текущего подпроцесса, особенно если метод выполняется в текущем подпроцессе. В таком случае строка, следующая за вызовом метода stop, никогда не выполняется, поскольку контекст подпроцесса "умирает" до того, как метод stop возвратит управление. Более аккуратный способ остановить выполнение подпроцесса—установить значение какой-либо переменной-флага, предусмотрев в методе run код, который, проверив состояние флага, завершил бы выполнение подпроцесса.

13.4. suspend

Метод suspend отличается от метода stop тем, что метод приостанавливает выполнение подпроцесса, не разрушая при этом его системный контекст. Если выполнение подпроцесса приостановлено вызовом suspend, вы можете снова активизировать этот подпроцесс, вызвав метод resume.

13.5. resume

Метод resume используется для активизации подпроцесса, приостановленного вызовом suspend. При этом не гарантируется, что после вызова resume подпроцесс немедленно начнет выполняться, поскольку в этот момент может выполняться другой более высокоприоритетный процесс. Вызов resume лишь делает подпроцесс способным

Copyright © 1999-2004 Javable.com. All rights reserved.

выполняться, а то, когда ему будет передано управление, решит планировщик.

13.6. setPriority

```
setPriority(int p)
```

Метод setPriority устанавливает приоритет подпроцесса, задаваемый целым значением передаваемого методу параметра. В классе Thread есть несколько предопределенных приоритетов-констант: MIN_PRIORITY, NORM_PRIORITY и MAX_PRIORITY, соответствующих соответственно значениям 1, 5 и 10. Большинство пользовательских приложений должно выполняться на уровне NORM_PRIORITY плюс-минус 1. Приоритет фоновых заданий, например, сетевого ввода-вывода или перерисовки экрана, следует устанавливать в MIN_PRIORITY. Запуск подпроцессов на уровне MAX_PRIORITY требует осторожности. Если в подпроцессах с таким уровнем приоритета отсутствуют вызовы sleep или yield, может оказаться, что вся исполняющая система Java перестанет реагировать на внешние раздражители.

13.7. getPriority

Этот метод возвращает текущий приоритет подпроцесса—целое значение в диапазоне от 1 до 10.

13.8. setName

```
setName(String name)
```

Метод setName присваивает подпроцессу указанное в параметре имя. Это помогает при отладке программ с параллельными подпроцессами. Присвоенное с помощью setName имя будет появляться во всех трассировках стека, которые выводятся при получении интерпретатором неперехваченного исключения.

13.9. getName

Метод getName возвращает строку с именем подпроцесса, установленным с помощью вызова setName.

Copyright © 1999-2004 Javable.com. All rights reserved.

Есть еще множество функций и несколько классов, например, ThreadGroup и SecurityManager, которые имеют отношение к подпроцессам, но эти области в Java проработаны еще не до конца. Скажем лишь, что при необходимости можно получить информацию об этих интерфейсах из документации по JDK API.

14. А дорога дальше вьется

Простые в использовании встроенные в исполняющую среду и в синтаксис Java легковесные процессы—одна из наиболее веских причин, по которым стоит изучать этот язык. Освоив однажды параллельное программирование, вы уже никогда не захотите возвращаться назад к программированию с помощью модели, управляемой событиями. После того, как вы освоились с основами программирования на Java, включая создание классов, пакетов, и модель легковесных процессов, для вас не составит труда разобраться в той коллекции Java-классов, к обсуждению которой мы сейчас приступим.

Copyright © 1999-2004 Javable.com. All rights reserved.