Урок 5

Многопоточность

Потоки в Java, способы создания и управления состояниями потоков, основные проблемы при работе с потоками и синхронизация.

Общие сведения

Различают две разновидности многозадачности: на основе процессов и на основе потоков. По сути, процесс представляет собой исполняемую программу, поэтому многозадачность на основе процессов — это средство, обеспечивающее возможность выполнения одновременно нескольких программ. При организации многозадачности на основе потоков в рамках одной программы могут выполняться одновременно несколько задач. Например, текстовый редактор позволяет редактировать текст во время автоматической проверки орфографии, при условии, что оба эти действия выполняются в двух отдельных потоках.
Многопоточность позволяет увеличить эффективность работы программы за счёт использования «свободных» периодов процессора. Как известно, многие устройства ввода-вывода работают медленнее, чем центральный процессор, поэтому большую часть своего времени программе приходится ожидать отправки или получения данных. Благодаря многопоточной обработке программа может решать какую-нибудь другую задачу во время вынужденного простоя процессора.
В одноядерной системе параллельно выполняющиеся потоки разделяют ресурсы одного ЦП, получая по очереди квант его времени. Поэтому в одноядерной системе два или более потока на самом деле не выполняются одновременно, а лишь используют время простоя ЦП. В многоядерных же системах несколько потоков могут выполняться действительно одновременно.

Создание потоков

В каждом процессе имеется как минимум один поток исполнения, который называется основным потоком. Он получает управление уже при запуске программы. От основного потока могут быть порождены другие, подчиненные, потоки. В основу системы многопоточной обработки в Java положены класс Thread, отвечающий за создание экземпляра потока, организацию доступа к нему и управления, и интерфейс Runnable. За выполняемую в отдельном потоке задачу отвечает метод run(), который создаёт точку входа в новый поток до тех пор, пока не произойдёт возврат из метода run(). Ниже приведён пример запуска потока через реализацию интерфейса Runnable:

```
public class MainClass {
    static class MyRunnableClass implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                try {
                    Thread.sleep(100);
                    System.out.println("new thread: " + i);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) {
        new Thread(new MyRunnableClass()).start();
        new Thread(new MyRunnableClass()).start();
    }
}
```

Класс MyRunnableClass реализует интерфейс Runnable, в теле метода run() прописан цикл, который

выводит в консоль числа от 0 до 9. Метод sleep() приостанавливает поток, из которого он был вызван, на указанное число миллисекунд, в нём может быть сгенерировано исключение InterruptedException, следовательно, его нужно вызывать в блоке try. В методе же main() создаётся два объекта типа Thread, конструктору которых в качестве аргумента передаются объекты класса MyRunnableClass, после чего новые потоки запускаются с помощью метода start().
Выполнение программы продолжается до тех пор, пока все потоки не завершат работу, при этом основной поток рекомендуется завершать последним.

Расширение класса Thread

Ту же задачу можно выполнить с использованием наследования от класса Thread. Для этого необходимо внести небольшие изменения в код, как показано ниже:

```
public class MainClass {
    static class MyThread extends Thread {
        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                try {
                    Thread.sleep(100);
                    System.out.println("new thread: " + i);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) {
        new MyThread().start();
    }
}
```

Какой из двух способов создания потоков предпочтительнее? В классе Thread имеется несколько методов, которые можно переопределить в порожденном классе. Из них обязательному переопределению подлежит только метод run(). Он же, безусловно, должен быть определён и при реализации интерфейса Runnable. В большинстве случаев создавать подкласс, порождённый от класса Thread, следует в случае, если требуется дополнить его новыми функциями. Так, если переопределять любые другие методы из класса Thread не нужно, то можно ограничиться только реализацией интерфейса Runnable. Кроме того, реализация интерфейса Runnable позволяет создаваемому потоку наследовать класс, отличающийся от Thread.

Приоритеты потоков

Каждый поток имеет определённый приоритет, от которого зависит доля процессорного времени, предоставляемого данному потоку. Однако, помимо приоритета, на частоту доступа потока к ЦП оказывают влияние и другие факторы, в том числе, как в операционной системе поддерживается многопоточность. То есть высокий приоритет потока лишь означает, что потенциально он может получить больше процессорного времени. Так, если высокоприоритетный поток ожидает доступа к некоторому ресурсу, он блокируется, и вместо него исполняется низкоприоритетный поток. Но когда высокоприоритетный поток получит доступ к ресурсам, он прервёт низкоприоритетный поток и возобновит работу.
При запуске порождённого потока его приоритет устанавливается равным приоритету родительского потока. По умолчанию, приоритет равен 5. Узнать или указать приоритет можно с помощью методов

setPriority() и getPriority() класса Thread. Значение параметра должно находиться в пределах от MIN_PRIORITY(1) до MAX_PRIORITY(10).

Синхронизация

При использовании нескольких потоков иногда возникает необходимость в координации их выполнения. Процесс, посредством которого это достигается, называют синхронизацией. Главным для синхронизации в Java является понятие монитора, контролирующего доступ к объекту. Монитор реализует принцип блокировки. Если объект заблокирован одним потоком, то он оказывается недоступным для других потоков. В какой-то момент объект разблокируется, благодаря чему другие потоки смогут получить к нему доступ.
У каждого объекта в Java имеется свой монитор, то есть синхронизировать можно любой объект. Для поддержки синхронизации предусмотрено ключевое слово synchronized и ряд методов, имеющихся у каждого объекта.

Использование синхронизированных методов

Для того чтобы синхронизировать метод, в его объявлении следует указать ключевое слово synchronized. Когда такой метод получает управление, вызывающий поток активизирует монитор, что приводит к блокированию объекта. Если объект блокирован, он недоступен из другого потока, а кроме того, его нельзя вызвать из других синхронизированных методов, определённых в классе данного объекта. Когда выполнение синхронизированного метода завершается, монитор разблокирует объект, что позволяет другому потоку использовать этот метод.

- Синхронизированный метод создается путем указания ключевого слова synchronized в его объявлении. Как только синхронизированный метод любого объекта получает управление, объект блокируется и ни один синхронизированный метод этого объекта не может быть вызван другим потоком.
- Потоки, которым требуется синхронизированный метод, используемый другим потоком, ожидают до тех пор, пока не будет разблокирован объект, для которого они вызываются. Когда синхронизированный метод завершается, объект, для которого он вызывался, разблокируется.

Ниже приведены три варианта синхронизации:

```
public class Example_SB_1 {
    public static void main(String[] args) {
        Example_SB_1 e1 = new Example_SB_1();
        System.out.println("Start");
        new Thread(() -> e1.method1()).start();
        new Thread(() -> e1.method2()).start();
    }
    public synchronized void method1() {
        System.out.println("M1");
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("M2");
    }
    public synchronized void method2() {
        System.out.println("M1");
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("M2");
    }
}
```

При указании ключевого слова synchronized в объявлении метода в роли монитора выступает объект, у которого был вызван синхронизированный метод. То есть в приведённом выше примере два потока не могут параллельно выполнять method1() и method2().

```
public class Example_SB_2 {
    private Object lock1 = new Object();
    public static void main(String[] args) {
        Example_SB_2 e2 = new Example_SB_2();
        System.out.println("Start");
        new Thread(() -> e2.method1()).start();
        new Thread(() -> e2.method1()).start();
    }
    public void method1() {
        System.out.println("Block-1 begin");
        for (int i = 0; i < 3; i++) {
            System.out.println(i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Block-1 end");
        synchronized (lock1) {
            System.out.println("Synch block begin");
            for (int i = 0; i < 10; i++) {
                System.out.println(i);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Synch block end");
        }
        System.out.println("M2");
    }
}
```

В этом случае в роли монитора выступает объект lock1, соответственно, два потока смогут параллельно выполнять первую часть метода method1(), однако в блок синхронизации в единицу времени может зайти только один поток, так как захватывается монитор lock1.

```
public class Example_SB_3 {
    public static void main(String[] args) {
        System.out.println("Start");
        new Thread(() -> method()).start();
        new Thread(() -> method()).start();
    }
    public synchronized static void method() { // синхронизация по классу
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

При указании ключевого слова synchronized в объявлении статического метода в роли монитора выступает класс, метод которого был вызван потоком.

Домашнее задание

1. Необходимо написать два метода, которые делают следующее:

1) Создают одномерный длинный массив, например:

```
static final int size = 10 000 000;
static final int h = size / 2;
float[] arr = new float[size].
```

2) Заполняют этот массив единицами.
3) Засекают время выполнения: long a = System.currentTimeMillis().
4) Проходят по всему массиву и для каждой ячейки считают новое значение по формуле:

```
arr[i] = (float)(arr[i] * Math.sin(0.2f + i / 5) * Math.cos(0.2f + i / 5) *
Math.cos(0.4f + i / 2)).
```

5) Проверяется время окончания метода System.currentTimeMillis().
6) В консоль выводится время работы: System.out.println(System.currentTimeMillis() - a).

Отличие первого метода от второго:

- Первый просто бежит по массиву и вычисляет значения.
- Второй разбивает массив на два массива, в двух потоках высчитывает новые значения и потом склеивает эти массивы обратно в один.

Пример деления одного массива на два:

- System.arraycopy(arr, 0, a1, 0, h);
- System.arraycopy(arr, h, a2, 0, h).

Пример обратной склейки:

- System.arraycopy(a1, 0, arr, 0, h);
- System.arraycopy(a2, 0, arr, h, h).

Примечание:
System.arraycopy() — копирует данные из одного массива в другой:
System.arraycopy(массив-источник, откуда начинаем брать данные из массива-источника, массив-назначение, откуда начинаем записывать данные в массив-назначение, сколько ячеек копируем)
По замерам времени:
Для первого метода надо считать время только на цикл расчета:

```
for (int i = 0; i < size; i++) {
   arr[i] = (float)(arr[i] * Math.sin(0.2f + i / 5) * Math.cos(0.2f + i / 5) *
Math.cos(0.4f + i / 2));
}
```

Для второго метода замеряете время разбивки массива на 2, просчета каждого из двух массивов и склейки.

Дополнительные материалы

1 Кей С. Хорстманн, Гари Корнелл. Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2 Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3 Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.