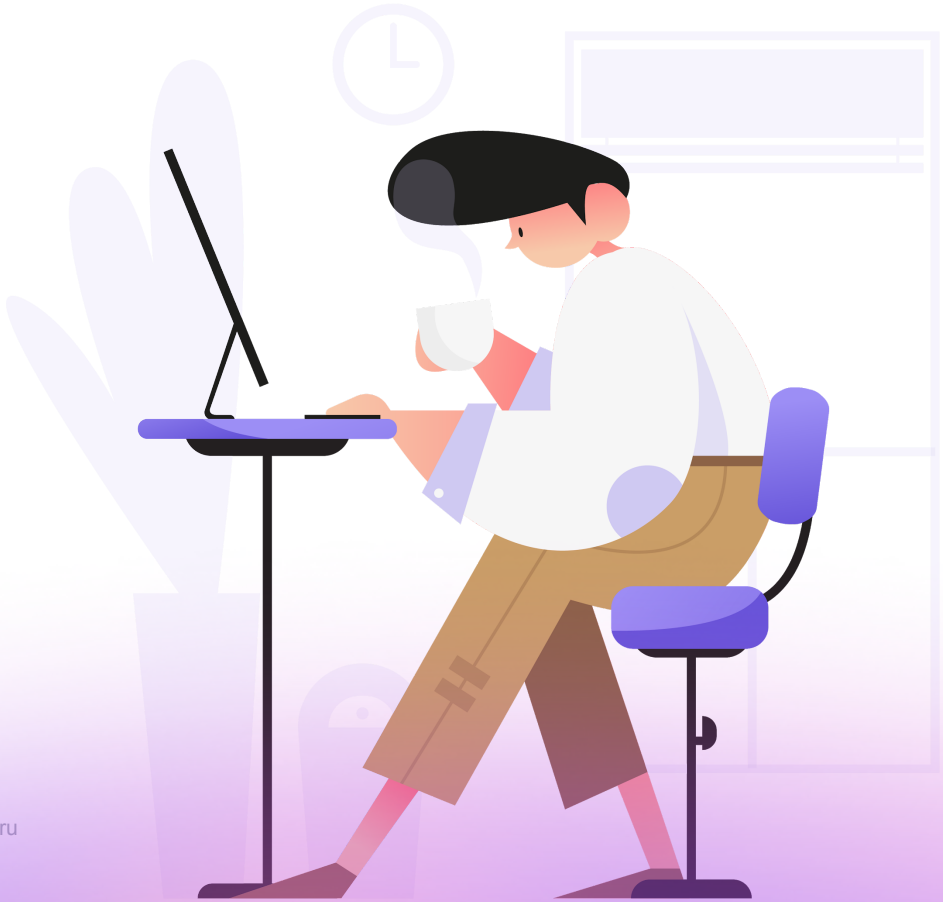
GeekBrains

Java Core для Android

Многопоточность. Часть I



На этом уроке

Многопоточность в Java. Разделяемая память, управление потоками и вопросы синхронизации.

Взаимодействие потоков исполнения, взаимная блокировка

Оглавление

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.

Главное преимущество многопоточной обработки заключается в том, что она позволяет писать программы, которые работают очень эффективно благодаря использованию «свободных» периодов процессора. Как известно, большинство устройств ввода-вывода работают много 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. С другой стороны, в многоядерных 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 создает точку входа в новый поток, до тех пор, пока не произойдет возврат из метода run(). Ниже приведен 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(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 нем может быть сгенерировано исключение InterruptedException. Следовательно, его нужно вызывать в блоке try. В методе же main() создается 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.

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(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(). Этот же метод, безусловно, должен быть определен и при реализации интерфейса Runnable. В большинстве случаев создавать подкласс, порожденный 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

С каждым потоком ассоциируется определенный приоритет. В частности, от приоритета потока зависит относительная доля процессорного времени, предоставляемого данному потоку, по сравнению с остальными активными потоками.
Следует иметь в виду, что, помимо приоритета, на частоту доступа потока к ЦП оказывают влияние и другие факторы. Так, если высокоприоритетный поток ожидает доступа к некоторому ресурсу, например, для ввода с клавиатуры, он блокируется, и вместо него исполняется низкоприоритетный поток. Но когда высокоприоритетный поток получит доступ к ресурсам, он прервет низкоприоритетный поток и возобновит свое выполнение. На планирование работы потоков также влияет то, каким именно образом в операционной системе поддерживается многозадачность. Следовательно, если один поток имеет более высокий приоритет, чем другой, это еще не означает, что первый поток будет исполняться быстрее второго. Высокий приоритет потока лишь означает, что потенциально он может получить больше времени ЦП.
При запуске порожденного потока его приоритет устанавливается равным приоритету родительского потока. Изменить приоритет можно, вызвав метод setPriority() класса Thread. Значение параметра уровень должно находиться в пределах от MIN_PRIORITY до MAX_PRIORITY. В настоящее время этим константам соответствуют числовые значения от 1 до 10. Для того чтобы восстановить приоритет потока, заданный по умолчанию, следует указать значение 5. Получить текущий приоритет можно с помощью метода getPriority () класса Thread.

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. Этот механизм встроен в сам язык. Следовательно, синхронизировать можно любой объект. Для поддержки синхронизации в Java 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, определенных в классе данного объекта. Когда выполнение синхронизированного метода завершается, монитор разблокирует объект, что позволяет другому потоку использовать этот метод. Таким образом, для достижения синхронизации программисту не приходится прилагать каких-то особых усилий.
- 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 он вызывается. Когда синхронизированный метод завершается, объект, для которого он вызывался, разблокировался.
Ниже приведены три варианта синхронизации:

```
public class Example_SB_1 {
    public static void main(String[] args) {
        Example_SB_1 e1 = new Example_SB_1();
        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 приведенном выше примере два потока не смогут параллельно выполнять method1() и method2().
Второй способ синхронизации основан на использовании отдельных объектов типа java.lang.Object в качестве мониторов. Пример кода приведен ниже:

```
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("Старт main потока");
        new Thread(() -> e2.method1()).start();
        new Thread(() -> e2.method1()).start();
    }

    public void method1() {
        System.out.println("Метод запущен");
        System.out.println("Блок 1 начало");
        for (int i = 0; i < 3; i++) {
            System.out.println(i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Блок 1 конец");
        synchronized (lock1) {
            System.out.println("Начало синхронизированного блока");
            for (int i = 0; i < 10; i++) {
                System.out.println(i);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Конец синхронизированного блока");
        }
        System.out.println("Метод завершил свою работу");
    }
}
```

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 происходит захват монитора lock1.
При третьем способе синхронизации, в роли монитора может выступать сам класс. Пример кода

приведен ниже:

```
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() {
        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 сам класс.

Взаимодействие потоков исполнения

В предыдущих примерах другие потоки исполнения, безусловно, блокировались от асинхронного доступа к некоторым методам. Такое применение неявных мониторов объектов в Java оказывается довольно эффективным, но более точного управления можно добиться, организовав взаимодействие потоков исполнения.
В Java внедрен механизм взаимодействия потоков исполнения с помощью методов wait(), notify() и notifyAll(). Эти методы реализованы как завершенные в классе Object, поэтому они доступны всем классам. Все три метода могут быть вызваны только из синхронизированного контекста.
• Метод wait() вынуждает вызывающий поток исполнения уступить монитор и перейти в состояние ожидания до тех пор, пока какой-нибудь другой поток исполнения не войдет в тот же монитор и не вызовет метод notify().
• Метод notify() возобновляет исполнение потока, из которого был вызван метод wait() для того же самого объекта.
• Метод notifyAll() возобновляет исполнение всех потоков, из которых был вызван метод wait() для того же самого объекта. Одному из этих потоков предоставляется доступ.
Прежде чем рассматривать пример, демонстрирующий взаимодействие потоков исполнения, необходимо сделать одно важное замечание. Метод wait() обычно ожидает до тех пор, пока не будет вызван метод notify() или notifyAll(). Но вполне вероятно, хотя и в очень редких случаях, что ожидающий поток исполнения может быть возобновлен вследствие ложной активизации. При этом исполнение ожидающего потока возобновляется без вызова метода notify() или notifyAll(). Из-за этой маловероятной возможности в компании Oracle рекомендуют вызывать метод wait() в цикле, проверяющем условие, по которому поток ожидает возобновления.

```
public class WaitNotifyClass {
    private final Object mon = new Object();
    private volatile char currentLetter = 'A';
```

```
    public static void main(String[] args) {
        WaitNotifyClass w = new WaitNotifyClass();
        Thread t1 = new Thread(() -> {
            w.printA();
        });
        Thread t2 = new Thread(() -> {
            w.printB();
        });
        t1.start();
        t2.start();
    }

    public void printA() {
        synchronized (mon) {
            try {
                for (int i = 0; i < 3; i++) {
                    while (currentLetter != 'A') {
                        mon.wait();
                    }
                    System.out.print("A");
                    currentLetter = 'B';
                    mon.notify();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void printB() {
        synchronized (mon) {
            try {
                for (int i = 0; i < 3; i++) {
                    while (currentLetter != 'B') {
                        mon.wait();
                    }
                    System.out.print("B");
                    currentLetter = 'A';
                    mon.notify();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

В приведенном выше примере два потока синхронизируют свою работу с помощью методов wait() и notify(). Один поток отвечает за печать буквы A, второй – B. Решается задача последовательной печати ABABAB. Переменная `currentLetter` указывает на букву, которая должна быть отпечатана. При запуске невозможно предсказать какой из потоков начнет выполнение первым. Если первый запускается поток B, то он просто переходит в режим ожидания. После чего поток A производит печать буквы в консоль и будит поток B. Далее за счет механизма wait()/notify() эти потоки работают последовательно и печать гарантированно начинается с буквы A.

Взаимная блокировка

При работе с многопоточностью может возникнуть особый тип ошибок – взаимная блокировка(deadlock), которая происходит в том случае, когда потоки исполнения имеют циклическую зависимость от пары синхронизированных объектов.
Допустим, один поток исполнения входит в монитор объекта X, а другой - в монитор объекта Y. Если поток исполнения в объекте X попытается вызвать любой синхронизированный метод для объекта Y, он будет блокирован, как и предполагалось. Но если поток исполнения в объекте Y, в свою очередь, попытается вызвать любой синхронизированный метод для объекта X, то этот поток будет ожидать вечно, поскольку для получения доступа к объекту X он должен снять свою блокировку с объекта Y, чтобы первый поток исполнения мог завершиться. Взаимная блокировка является ошибкой, которую трудно отладить, по двум причинам:

- Взаимная блокировка возникает редко, когда исполнение двух потоков точно совпадает по времени;
- Взаимная блокировка может возникнуть, когда в ней участвует больше двух потоков исполнения.

Ниже приведен код, выполнение которого приводит к взаимной блокировки потоков:

```
public class ExampleDeadlock {
    private static final Object lock1 = new Object();
    private static final Object lock2 = new Object();

    public static void main(String[] args) {
        ThreadOne threadOne = new ThreadOne();
        ThreadTwo threadTwo = new ThreadTwo();
        threadOne.start();
        threadTwo.start();
    }

    private static class ThreadOne extends Thread {
        public void run() {
            synchronized (lock1) {
                System.out.println("Thread1 захватил Lock1");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Thread1 ожидает Lock2");
                synchronized (lock2) {
                    System.out.println("Thread1 захватил Lock1 и Lock2");
                }
            }
        }
    }

    private static class ThreadTwo extends Thread {
        public void run() {
            synchronized (lock2) {
                System.out.println("Thread2 захватил Lock2");
                try {
```

```
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Thread2 ожидает Lock1");
                synchronized (lock1) {
                    System.out.println("Thread2 захватил Lock1 и Lock2");
                }
            }
        }
    }
}
```

Изменение состояний потоков исполнения

Иногда возникает потребность в приостановке исполнения потоков. Приостановку и возобновление исполнения потока довольно просто реализовать. Механизм временной или окончательной остановки потока исполнения, а также его возобновления отличался в ранних версиях Java. До версии Java 1.2 методы suspend() и resume(), определенные в классе Thread, использовались в программах для приостановки и возобновления потоков исполнения. Тем не менее пользоваться ими не рекомендуется по следующей причине: метод suspend() способен порождать серьезные системные сбои. Допустим, что поток исполнения получил блокировки для очень важных структур данных. Если в этот момент приостановить исполнение данного потока, блокировки не будут сняты. Другие потоки исполнения, ожидающие эти ресурсы, могут оказаться взаимно блокированными.

Метод resume() также не рекомендован к употреблению. И хотя его применение не вызовет особых осложнений, тем не менее им нельзя пользоваться без метода suspend(), который его дополняет.

Метод stop() также объявлен устаревшим, потому что он может послужить причиной серьезных системных сбоев. Допустим, поток выполняет запись в критически важную структуру данных и успел произвести лишь частичное ее обновление. Если его остановить в этот момент, структура данных может оказаться в поврежденном состоянии.

Дело в том, что метод stop() вызывает снятие любой блокировки, устанавливаемой вызывающим потоком исполнения. Следовательно, поврежденные данные могут быть использованы в другом потоке исполнения, ожидающем по той же самой блокировке.

Если методы suspend(), resume() или stop() нельзя использовать для управления потоками исполнения, то можно прийти к выводу, что теперь вообще нет никакого механизма для приостановки, возобновления или прерывания потока исполнения. Вместо этого код управления выполнением потока должен быть составлен таким образом, чтобы метод run() периодически проверял, должно ли исполнение потока быть приостановлено, возобновлено или прервано. Обычно для этой цели служит флаговая переменная, обозначающая состояние потока исполнения. До тех пор, пока эта флаговая переменная содержит признак "выполняется", метод run() должен продолжать выполнение. Если же эта переменная содержит признак "приостановить", поток исполнения должен быть приостановлен. А если флаговая переменная получает признак "остановить", то поток исполнения должен завершиться. Безусловно, имеются самые разные способы написать кода управления выполнением потока, но основной принцип остается неизменным для всех программ.

В приведенном ниже примере программы демонстрируется применение методов wait() и notify(), для управления выполнением потока. Класс RunnableClass содержит переменную suspended, используемую для управления выполнением потока. Метод run() содержит блок оператора synchronized, где проверяется состояние переменной suspended. Если она принимает логическое значение true, то вызывается метод wait() для приостановки выполнения потока. В методе mySuspend() устанавливается true переменной suspendFlag, а в методе myResume() - false и вызывается метод notify(), чтобы активизировать поток исполнения.

```
public class RunnableDemo {
    static class RunnableClass implements Runnable {
        boolean suspended = false;

        public void run() {
            System.out.println("Запуск потока");
            try {
                for (int i = 10; i > 0; i--) {
                    System.out.println(i);
                    Thread.sleep(300);
                    synchronized (this) {
                        while (suspended) {
                            wait();
                        }
                    }
                }
            } catch (InterruptedException e) {
                System.out.println("Поток прерван");
            }
            System.out.println("Завершение потока");
        }

        public void mySuspend() {
            suspended = true;
        }

        public synchronized void myResume() {
            suspended = false;
            notify();
        }
    }

    public static void main(String[] args) {
        RunnableClass rc = new RunnableClass();
        new Thread(rc).start();
        try {
            Thread.sleep(800);
            rc.mySuspend();
            Thread.sleep(3000);
            rc.myResume();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Если запустить эту программу на выполнение, то можно увидеть, как исполнение потока приостанавливается и возобновляется.

Получение состояния потока исполнения

Поток исполнения может находиться в нескольких состояниях. Для того чтобы получить текущее состояние, достаточно вызвать у потока метод getState(), который вернет значение типа Thread.State. Возможные состояния:

- BLOCKED – поток приостановил выполнение, поскольку ожидает получения блокировки;
- NEW – поток еще не начал выполнение;
- RUNNABLE – поток выполняется или начнет выполняться, когда получит доступ к ЦП;
- TERMINATED – поток завершил выполнение;
- TIMED WAITING – поток приостановил выполнение на определенное время (например, после вызова метода **sleep()**, **wait()** или **join()**);
- WAITING – поток приостановил выполнение, ожидая определенного действия (например, вызова версии метода **wait()** или **join()** без заданного времени ожидания).

Следует иметь в виду, что состояние потока исполнения может измениться сразу же после вызова метода getState(), поэтому он не предназначен для синхронизации потоков исполнения и служит для отладки или профилирования характеристик потока во время выполнения

ExecutorService

Интерфейс java.util.concurrent.ExecutorService представляет собой механизм асинхронного выполнения, который способен выполнять задачи в фоновом режиме. Фактически, реализация ExecutorService из пакета java.util.concurrent, представляет собой реализацию пула потоков.

Вот простой пример использования ExecutorService:

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
executorService.execute(new Runnable() {
   public void run() {
       System.out.println("Асинхронная задача");
   }
});
executorService.shutdown();
```

С помощью фабричного метода newFixedThreadPool() создается объект типа ExecutorService. Это пул потоков с 10 рабочими потоками, выполняющими задачи. В метод execute() передается реализация интерфейса Runnable, в которой прописана задача, передаваемая на выполнение одному из потоков ExecutorService.

Создание ExecutorService

Существует несколько типов ExecutorService, которые можно создать через класс Executors. newSingleThreadExecutor() создает пул в котором только один рабочий поток, то есть он может одновременно исполнять только одну задачу, но при каждом запуске не будет создавать новых потоков. newFixedThreadPool() создает пул с фиксированным количеством потоков, в примере ниже можно запустить одновременно выполнение не более 10 задач. newCachedThreadPool() создает пул, который может автоматически расширяться, если ему дать задачу и у него будут свободные потоки, пул отдаст задачу одному из таких потоков, если же в пуле свободных потоков нет, он создаст и запустит новый. Минусом такого подхода является то, что у cachedThreadPool нет верхней границы, и при высокой частоте появления новых задач он потенциально может создавать огромное количество потоков, пока не закончатся системные ресурсы. Ниже приведены примеры трех фабричных методов:

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();
ExecutorService executorService2 = Executors.newFixedThreadPool(10);
ExecutorService executorService3 = Executors.newCachedThreadPool();
```

Использование ExecutorService

Существует несколько различных способов делегирования задач для выполнения в ExecutorService:

- execute(Runnable)
- submit(Runnable)
- submit(Callable)
- invokeAny(...)
- invokeAll(...)

execute (Runnable)

Метод execute(Runnable) принимает объект java.lang.Runnable и выполняет его асинхронно. Пример:

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
executorService.execute(new Runnable() {
   public void run() {
       System.out.println("Асинхронная задача");
   }
});
executorService.shutdown();
```

Получить из потока результат выполнения не получится, для этого нужно использовать интерфейс Callable, о чем пойдет речь ниже.

submit (Runnable)

Метод submit(Runnable) также принимает реализацию Runnable, но возвращает объект типа Future, который можно использовать для проверки завершенности выполнения задачи. Пример:

```
public static void main(String[] args) throws Exception {
   ExecutorService executorService = Executors.newFixedThreadPool(2);
   Future future = executorService.submit(new Runnable() {
       public void run() {
           System.out.println("Асинхронная задача");
       }
   });
   future.get();  // вернет null если задача завершилась корректно
   executorService.shutdown();
}
```

submit (Callable)

Экземпляр Callable также позволяет дать потоку задачу, но в отличие от Runnable, его метод call() может возвращать результат. Результат Callable может быть получен через объект Future, возвращенный методом submit. Пример:

```
public static void main(String[] args) throws Exception {
   ExecutorService executorService = Executors.newFixedThreadPool(2);
   Future future = executorService.submit(new Callable(){
       public Object call() throws Exception {
           System.out.println("Асинхронный вызов");
           return "Результат из потока";
       }
   });
   System.out.println("future.get() = " + future.get());
   executorService.shutdown();
}
Результат:
Асинхронный вызов
future.get() = Результат из потока
```

shutdown() и shutdownNow()

Когда вы закончили использовать ExecutorService, его необходимо остановить, чтобы его свободные потоки прекратили свою работу. Например, если main поток завершил работу, а ExecutorService остался активным, активные потоки внутри ExecutorService не позволят JVM завершить работу приложения. Для завершения потоков внутри ExecutorService, необходимо вызвать метод shutdown(). ExecutorService не будет закрыт немедленно, но перестанет принимать новые задачи, и как только все потоки завершат текущие задачи, ExecutorService отключится.
Все задачи, отправленные в ExecutorService до shutdown(), выполняются. Если вы хотите немедленно закрыть ExecutorService, можно вызвать метод shutdownNow(), который попытается немедленно остановить все выполняемые задачи и пропустить все представленные, но не обработанные задачи.

Практическое задание

Необходимо написать два метода, которые делают следующее:
1) Создают одномерный длинный массив, например:

```
static final int SIZE = 10 000 000;
static final int HALF = 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;
2. Стив Макконнелл. Совершенный код;
3. Брюс Эккель. Философия Java;
4. Герберт Шилдт. Java 8: Полное руководство;
5. Герберт Шилдт. Java 8: Руководство для начинающих.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Герберт Шилдт. Java. Полное руководство // 8-е изд.: Пер. с англ. – М.: Вильямс, 2012. – 1 376 с.
2. Герберт Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 720 с.