Конспект лекции
Потоки выполнения и синхронизация

Цель и задачи лекции

Цель – изучить механизмы выполнения потоков и синхронизацию между ними.

Задачи:

1. Изучить принципы работы многопоточных приложений
2. Изучить основные классы и интерфейсы работы с потоками, и их методы
3. Дать понимание синхронизации поток между друг другом

План занятия

1. Многопоточное программирование
2. Модель потоков исполнения в Java
3. Класс Thread и интерфейс Runnable
4. Использование интерфейса Callable и Future
5. Синхронизация потоков
6. Применение пакета Concurrent

Многопоточное программирование

В отличие от некоторых языков программирования, в Java предоставляется встроенная поддержка многопоточного программирования. Многопоточная программа содержит две или несколько частей, которые могут выполняться одновременно. Каждая часть такой программы называется потоком исполнения, причем каждый поток задает отдельный путь исполнения кода. Следовательно, многопоточность – это особая форма многозадачности. Процесс, по существу, является выполняющейся программой. Следовательно, многозадачность на основе процессов - это средство, которое позволяет одновременно выполнять две или несколько программ на компьютере. среде многозадачности на основе потакав наименьшей единицей диспетчеризируемого кода является поток исполнения. Это означает, что одна программа может выполнять две или несколько задач одновременно.

Многозадачные потоки исполнения требуют меньших издержек, чем многозадачные процессы. Процессы являются крупными задачами, каждой из которых требуется свое адресное пространство. Связь между процессами ограничена и обходится дорого. Переключение контекста с одного процесса на другой также обходится дорого. С другой стороны, потоки исполнения более просты. Они совместно используют одно и то же адресное пространство и один и тот же крупный процесс. Связь между потоками

исполнения обходится недорого, как, впрочем, и переключение контекста с одного потока исполнения на другой. Несмотря на то, что программы на java пользуются многозадачными средами на основе процессов, такая многозадачность в Java не контролируется, а вот многопоточная многозадачность контролируется.

Модель потоков исполнения в Java

Исполняющая система Java во многом зависит от потоков исполнения, и все библиотеки классов разработаны с учетом многопоточности. По существу, потоки исполнения используются в Java для того, чтобы обеспечить асинхронность работы всей исполняющей среды. Благодаря предотвращению бесполезной траты циклов ЦП удается повысить эффективность выполнения кода в целом.

Ценность многопоточной среды лучше понять в сравнении. В однопоточных системах применяется подход, называемый циклом ожидания событий с опросом. В этой модели единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий, чтобы принять решение, что делать дальше. Как только этот механизм опроса возвращает, скажем, сигнал, что сетевой файл готов к чтению, цикл ожидания событий передает управление соответствующему обработчику событий. И до тех пор, пока тот не возвратит управление, в программе ничего не может произойти. На это расходует драгоценное время ЦП. Это может также привести к тому, что одна часть программы будет господствовать над другими, не позволяя обрабатывать любые другие событие. Вообще говоря, когда в однопоточной среде поток исполнения блокируется (т.е. приостанавливается) по причине ожидая некоторого ресурса, то приостанавливается выполнение и всей программы.

Выгода от многопоточности состоит в том, что основной механизм циклического опроса исключается. Один поток может быть приостановлен без остановки других частей программы. Например, время ожидания при чтении данных из сети или вводе пользователем данных может быть выгодно использовано в любом другом месте программы. Многопоточность позволяет переводить циклы анимации в состояние ожидая на секунду в промежутках между соседними кадрами, не приостанавливая работу всей системы. Когда поток исполнения блокируется в программе на Java, приостанавливается только один заблокированный поток, а все остальные потоки продолжают выполняться.

За последние годы многоядерные системы стали вполне обычным явлением. Безусловно, одноядерные системы все еще широко распространены и применяются. Следует, однако, иметь в виду, что многопоточные средства Java вполне работоспособны в обоих типах систем. В одноядерной системе одновременно выполняющиеся потоки совместно используют ЦР, получая каждый в отдельности некоторый квант времени ЦП. Поэтому в одноядерной системе два или более потока фактически не выполняются одновременно, но ожидают своей очереди на использование времени ЦП. А в многоядерных системах два или более потока фактически могут выполняться одновременно. Как правило, это позволяет увеличить эффективность программы и повысить скорость выполнения некоторых операций.

Потоки исполнения находятся в нескольких состояниях. Поток может выполняться. Он может быть готовым к выполнению, как только получит время ЦП. Работающий поток может быть приостановлен, что приводит к временному прекращению его активности. Выполнение приостановленного потока может быть возобновлено, что позволяет

продолжить его выполнение с того места, где он был приостановлен. Поток может быть заблокирован на время ожидания какого-нибудь ресурса. В любой момент поток может быть прерван, что приводит к немедленной остановке его исполнения. Однажды прерванный поток исполнения не может быть возобновлен.

Приоритеты потоков

Java присваивает каждому потоку приоритет, который определяет поведение данного потока по отношению к другим. Приоритеты потоков задаются целыми числами, определяющими относительный приоритет одного потока по сравнению к другими. Значение приоритета само по себе никакого смысла не имеет — более высокоприоритетный поток не выполняется быстрее, чем низкоприоритетный, когда он является единственным исполняемым потоком в данный момент. Вместо этого приоритет потока используется для принятия решения при переключении от одного выполняющегося потока к другому. Это называется переключением контекста. Правила, которые определяют, когда должно происходить переключение контекста, достаточно просты.

- Поток может добровольно уступить управление. Это делается явным уступанием очереди выполнения, приостановкой или блокированием ожидания ввода-вывода. При таком сценарии все прочие потоки проверяются, и ресурсы процессора передаются потоку с максимальным приоритетом, который готов к выполнению.
- Поток может быть прерван другим, более приоритетным потоком. В этом случае низкоприоритетный поток, который не занимает процессор, просто приостанавливается высокоприоритетным потоком, независимо от того, что он делает. В основном, высокоприоритетный поток выполняется, как только он этого "захочет". Это называется вытесняющей многозадачностью (или многозадачностью с приоритетами).

В случае, когда два потока, имеющие одинаковый приоритет, претендуют на цикл процессора, ситуация усложняется. Для таких операционных систем, как Windows, потоки с одинаковым приоритетом разделяют время в циклическом режиме. Для операционных систем других типов потоки с одинаковым приоритетом должны принудительно передавать управление своим "родственникам". Если они этого не делают, другие потоки не запускаются.

Класс Thread и интерфейс Runnable

Многопоточная система в Java построена на основе класса Thread, его методах и дополняющем его интерфейсе Runnable. Класс Thread инкапсулирует поток исполнения. Обратится напрямую к нематериальному состоянию работающего потока исполнения нельзя, поэтому приходится иметь дело с его "заместителем" - экземпляром класса Thread, который и породил его. Чтобы создать новый поток исполнения, следует расширить класс Thread или же реализовать интерфейс Runnable.

В классе Thread определяется ряд методов, помогающих управлять потоками исполнения.

Основные методы класса Thread:

- getName - Получает имя потока исполнения
- getPriority - Получает приоритет потока исполнения
- isAlive - Определяет, выполняется ли поток
- join - Ожидает завершения потока исполнения
- run - Задает точку входа в поток исполнения
- sleep - Приостанавливает выполнение потока на заданное время
- start - Запускает поток исполнения, вызывая его метода run ()

Главный поток исполнения

Когда программа на Java запускается на выполнение, сразу же начинает выполняться один поток. Он обычно называется главным потоком программы, потому что он запускается вместе с ней. Главный поток исполнения важен по двум причинам.

- От этого потока порождаются все дочерние потоки.
- Зачастую он должен быть последним потоком, завершающим выполнение программы, поскольку в нем производятся различные завершающие действия.

Несмотря на то что главный поток исполнения создается автоматически при запуске программы, им можно управлять через объект класса Thread. Для этого достаточно получить ссылку на него, вызвав метод currentThread(), который объявляется как открытый и статистический (public static) в классе Thread. Его общая форма выглядит следующим образом:

```
static Thread currentThread()
```

Этот метод возвращает ссылку на тот поток исполнения, из которого он был вызван. Получив ссылку на главный поток, можно управлять им таким же образом, как и любым другим потоком исполнения. Рассмотрим следующий пример программы:

```
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();

        System.out.println("Текущий поток исполнения: " + t);

        // изменить имя потока исполнения
        t.setName("My Thread");
        System.out.println("После изменения имени потока: " + t);

        try {
            for(int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван.");
        }
    }
}
```

В этом примере программы ссылка на текущий поток исполнения (в данном случае - главный поток) получается в результате вызова метода currentThread() и сохраняется в локальной переменной t. Затем выводятся сведения о потоке исполнения. Далее вызывается метод setName() для изменения внутреннего имени потока исполнения. После этого сведения о потоке исполнения выводятся заново. А в следующем далее цикле выводятся цифры в обратном порядке с задержкой на 1 секунду после каждой строки. Пауза организуется с помощью метода sleep(). Аргумент метода sleep() задает время задержки в миллисекундах. Обратите внимание на блок операторов try/catch, в котором находится цикл. Метод sleep() из класса Thread может сгенерировать исключение типа InterrupException, если в каком-нибудь другом потоке исполнения потребуется прервать ожидающий поток. В данном примере просто выводится сообщение, если поток исполнения прерывается, в а реальных программах подобную ситуацию придется обрабатывать иначе. Ниже приведен результат, выводимый данной программой.

```
Текущий поток исполнения: Thread[main,5,main]
После изменения имени потока: Thread[My Thread,5,main]
5
4
3
2
1
```

Обратите внимание на то, что вывод производится тогда, когда переменная t служит в качестве аргумента метода println(). Этот метода выводит по порядку имя потока исполнения, его приоритет и имя его группы. По умолчанию главный поток исполнения имеет имя main и приоритет, равный 5. Именем main обозначается также группа потоков исполнения, в которой относится данный поток. Группа потоков исполнения - это структура данных, которая управляет состоянием всего ряда потоков исполнения в целом. После изменения имени потока исполнения содержимое переменной t выводится снова - на этот раз новое имя потока исполнения.

Рассмотрим подробнее методы их класса Thread, используемые в приведенном выше примере программы. Метод sleep() вынуждает тот поток, из которого он вызывается, приостановить свое выполнение на указанное количество миллисекунд. Общая форма этого метода выглядит следующим образом:

```
static void sleep(long миллисекунд) throws InterrupException
```

Количество миллисекунд, на которое нужно приостановить выполнение, задает аргумент миллисекунд. Метод sleep() может сгенерировать исключение типа InterrupException. У него имеется и вторая, приведенная ниже форма, которая позволяет точнее задать время ожидания в милли- и наносекундах.

```
static void sleep(long миллисекунд, long наносекунд) throws InterrupException
```

Вторая форма данного метода может применяться только в тех средах, где предусматривается задание промежутков времени в наносекундах.

Как показано в предыдущем примере программы, установить имя потока исполнения можно с помощью метода setName(). А для того чтобы получить имя потока исполнения, достаточно вызвать метод getName(), хотя это в данном примере программы не показано.

Оба эти метода являются членами классами Thread и объявляются так, как показано ниже, где имя_потока обозначает имя конкретного потока исполнения.

```
final void setName(String имя_потока)

final String getName()
```

Создание потока исполнения

В наиболее общем смысле для создания потока исполнения следует получить экземпляр объекта типа Thread. В языке Java этой цели можно достичь следующими двумя способами:

- реализовав интерфейс Runnable;
- расширив класс Thread;
- использовать интерфейсы Callable и Future.

Использование интерфейса Runnable

Самый простой способ создать поток исполнения состоит в том, чтобы объявить класс, реализующий интерфейс Runnable. Этот интерфейс предоставляет абстракцию единицы исполняемого кода. Поток исполнения можно создать из объекта любого класса, реализующего интерфейс Runnable. Для реализации интерфейса Runnable в классе должен быть объявлен единственный метод run();

```
public void run()
```

В теле метода run() определяется код, который собственно, и составляет новый поток исполнения. Однако в методе run() можно вызывать другие методы, использовать другие классы, объявлять переменные таким же образом, как и в главном потоке исполнения. Единственное отличие заключается в том, что в методе run() устанавливается точка входа в другой, параллельный поток исполнения в программе. Этот поток исполнения завершится, когда метод run() возвратит управление.

После создания класса, реализующего интерфейс Runnable, в этом классе следует получить экземпляр объекта Thread. Для этой цели в классе Thread определен ряд конструкторов. Тот конструктор, который должен использоваться в данном случае, выглядит в общей форме следующим образом:

```
Thread(Runnable объект_потока, String имя_потока)
```

В этом конструкторе параметр объект_потока обозначает экземпляр класса, реализующего интерфейс Runnable. Этим определяется место, где начинается выполнение потока. Имя нового потока исполнения передается данному конструктору в качестве параметра имя_потока.

После того как новый поток исполнения будет создан, он не запускается до тех пор, пока не будет вызван метод start(), объявленный в классе Thread. По существу, в методе start() вызывается метод run(). Ниже показано, каким образом объявляется метод start().

```
void start()
```

Простой пример использования запуска потока:

```
class MyRunnable implements Runnable {
    public void run(){
        System.out.println("MyRunnable running");
    }
}

class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start();

        Runnable myRunnable = new Runnable(){
            public void run(){
                System.out.println("Runnable running");
            }
        };
    }
}
```

Рассмотрим следующий пример программы, демонстрирующий создание и запуск нового потока на выполнение:

```
class NewThread implements Runnable {
    Thread t;

    NewThread() {
    // создать новый, второй поток исполнения
        t = new Thread(this, "Демонстрационный поток");
        System.out.println("Дочерний поток создан: " + t);
        t.start(); // запустить поток исполнения
    }

    // Точка входа во второй поток исполнения
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Дочерний поток: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Дочерний поток прерван.");
        }
        System.out.println("Дочерний поток завершен.");
    }
}

class ThreadDemo {
    public static void main(String args[]) {
        new Thread(); // Создать новый поток

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Главный поток: " + i);
                Thread.sleep(1000);
            }
        } catch(InterruptedException e) {
            System.out.println("Главный поток прерван.");
        }

        System.out.println("Главный поток завершен.");
```

```
    }
}
```

Новый объект класса Thread создается в следующем операторе из конструктора NewThread():

```
t = new Thread(this, "Демонстрационный поток");
```

Передача ссылки this на текущий объект в первом аргументе данного конструктора означает следующее: в новом потоке исполнения для текущего объекта по ссылке this следует вызвать метод run(). Далее в приведенном выше примере программы вызывается метод start(), в результате чего поток исполнения запускается, начиная с метода run(). Это, в свою очередь приводит к началу цикла for в дочернем потоке исполнения. После вызова метода start() конструктор NewThread() возвращает управление методу main(). Возобновляя свое исполнение, главный поток входит в свой цикл for. Далее потоки выполняются параллельно, совместно используя ресурсы процессора в одноядерной системе, вплоть до завершения своих циклов. Ниже приведен результат, выводимый данной программой (у вас он может оказаться иным в зависимости от конкретной исполняющей среды).

```
Дочерний поток: Thread[Демонстрационный поток,5,main]
Главный поток: 5
Дочерний поток: 5
Дочерний поток: 4
Главный поток: 4
Дочерний поток: 3
Дочерний поток: 2
Главный поток: 3
Дочерний поток: 1
Дочерний поток завершен.
Главный поток: 2
Главный поток: 1
Главный поток завершен.
```

Как упоминалось ранее, в многопоточной программе главный поток исполнения зачастую должен завершаться последним. На самом же деле, если главный поток исполнения завершается раньше дочерних потоков, то исполняющая система Java может "зависнуть", что характерно для некоторых старых виртуальных машин JVM. В приведенном выше примере программы гарантируется, что главный поток исполнения завершится последним, поскольку главный поток исполнения находится в состоянии ожидания в течение 1000 миллисекунд в промежутках между последовательными шагами цикла, а дочерний поток исполнения - только 500 миллисекунд. Это заставляет дочерний поток исполнения завершиться раньше главного потока.

Использование класса Thread

Еще один способ создать поток исполнения состоит в том, чтобы сначала объявить класс, расширяющий класс Thread, а затем получить экземпляр этого класса. В расширяющем классе должен быть непременно переопределен метод run(), который является точкой входа в новый поток исполнения. Кроме того, в этом классе должен быть вызван метод start() для запуска нового потока на исполнение. Ниже приведена версия программы из предыдущего примера, переделенная с учетом расширения класса Thread.

```
class MyThread extends Thread {
    public void run(){
        System.out.println("MyThread running");
    }
}


class Main {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start();

        Thread thread = new Thread(){
            public void run(){
                System.out.println("Thread Running");
            }
        };

        thread.start();
    }
}
```

Рассмотрим код, в котором создается второй поток исполнения, расширяя класс Thread.

```
class NewThread extends Thread {

    NewThread() {
    // создать новый поток исполнения
        super("Демонстрационный поток");
        System.out.println("Дочерний поток: " + this);
        start(); // запустить поток на исполнение
    }

    // Точка входа во второй поток исполнения
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Дочерний поток: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Дочерний поток прерван.");
        }
        System.out.println("Дочерний поток завершен.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // создать новый поток исполнненния

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Главный поток: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван.");
        }
        System.out.println("Главный поток завершен.");
    }
}
```

Эта версия программы выводит такой же результат, как и предыдущая версия ее версия. Как видите, дочерний поток исполнения создается при конструировании объекта класса NewThread, наследующего от класса Thread. Обратите внимание на метод super() в классе NewThread. Он вызывает конструктор Thread(), общая форма которого

приведена ниже, где параметр имя_потока обозначает имя порождаемого потока исполнения.

```
public Thread(String имя_потока)
```

Использование интерфейса Callable и Future

Callable подобен Runnable, но с возвратом значения. Интерфейс Callable является параметризованным типом, с единственным общедоступным методом call().

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Параметр представляет собой тип возвращаемого значения. Например, Callable представляет асинхронное вычисление, которое в конечном итоге возвращает объект Integer.

Future хранит результат асинхронного вычисления. Вы можете запустить вычисление, предоставив кому-либо объект Future, и забыть о нем. Владелец объекта Future может получить результат, когда он будет готов.

Интерфейс Future выглядит следующим образом:

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

Вызов первого метода get() устанавливает блокировку до тех пор, пока не завершится вычисление. Второй метод генерирует исключение TimeoutException, если истекает таймаут до завершения вычислений. Если прерывается поток, выполняющий вычисление, оба метода генерируют исключение InterruptedException. Если вычисление уже завершено, get() немедленно возвращает управление.

Метод isDone() возвращает false, если вычисление продолжается, и true — если оно завершено.

Вы можете прервать вычисление, вызвав метод Cancel(). Если вычисление еще не стартовало, оно отменяется и уже не будет запущено. Если же вычисление уже идет, оно прерывается в случае равенства true параметра mayInterrupt.

Класс-оболочка FutureTask представляет собой удобный механизм для превращения Callable одновременно в Future и Runnable, реализуя оба интерфейса.

Например:

```
Callable<Integer> myComputation = ...;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = ne Thread(task);  // это Runnable
t.start();
...
Integer result = task.get();  // это Future
```

Выбор способа создания потоков исполнения

В связи с изложенным выше могут возникнуть следующие вопросы: почему в Java предоставляются два способа для создания порождаемых потоков исполнения и какой из них лучше? Ответы на эти вопросы взаимосвязаны. В классе Thread определяется ряд методов, которые могут быть переопределены в производных классах. И только один из них должен быть непременно переопределен: метод run(). Безусловно, этот метод требуется и в том случае, когда реализуется интерфейс Runnable. Многие программирующие на Java считают, что классы следует расширять только в том случае, если они должны быть усовершенствованы или каким-то образом видоизменены. Следовательно, если ни один из других методов не переопределяется в классе Thread, то лучше и проще реализовывать интерфейс Runnable. Кроме того, при реализации интерфейса Runnable класс порождаемого потока исполнения не должен наследовать класс Thread, что освобождает его от наследования другого класса. В конечном счете выбор конкретного способа для создания потоков исполнения остается за вами. Тем не менее в примерах, приведенных далее в этой главе, потоки будут создаваться с помощью классов, реализующих интерфейс Runnable.

Применение методов isAlive() и join()

Как упоминалось ранее, нередко требуется, чтобы главный поток исполнения завершался последним. С этой целью метод sleep() вызывался в предыдущих примерах из метода main() с достаточной задержкой, чтобы все дочерние потоки исполнения завершились раньше главного. Но это неудовлетворительное решение, вызывающее следующий серьезный вопрос: откуда одному потоку исполнения известно, что другой поток завершился? Правда, в классе Thread предоставляется средство, позволяющее разрешить этот вопрос.

Определить, был ли поток исполнения завершен, можно двумя способами. Во-первых, для этого потока можно вызвать метод isAlive(), определенный в классе Thread. Ниже приведена общая форма этого метода.

```
final Boolean isAlive()
```

Метод isAlive() возвращает логическое значение true, если поток, для которого он вызван, еще исполняется. В противном случае он возвращает логическое значение false.

И во-вторых, в классе Thread имеется метод join(), который применяется чаще, чем метод isAlive(), чтобы дождаться завершения потока исполнения.

Ниже приведена общая форма этого метода.

```
final void join() throws InterruptedException
```

Этот метод ожидает завершения того потока исполнения, для которого он вызван. Его имя отражает следующий принцип: вызывающий потока ожидает, когда указанный поток присоединится к нему. Дополнительные формы метода join() позволяют указывать максимальный промежуток времени, в течение которого требуется ожидать завершения указанного потока исполнения.

Ниже приведена усовершенствованная версия программы из предыдущего примера, где с помощью метода join() гарантируется, что главный поток завершится последним. В данном примере демонстрируется также применение метода isAlive().

```
class NewThread implements Runnable {
    String name; // имя потока исполнения
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("Новый поток: " + t);
        t.start();
    }

    // Точка входа в поток исполнения
    public void run() {
        try {
            for(int i=0; i>0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch(InterruptedException e) {
            System.out.println(name + " прерван");
        }
        System.out.println(name + " завершен.");
    }
}

class DemoJoin {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("Один");
        NewThread ob2 = new NewThread("Два");
        NewThread ob3 = new NewThread("Три");

        System.out.println("Поток Один Запущен: " + ob1.t.isAlive());
        System.out.println("Поток Два Запущен: " + ob2.t.isAlive());
        System.out.println("Поток Три Запущен: " + ob3.t.isAlive());
        // Ожидать завершения потоков исполнения

        try {
            System.out.println("Ожидание завершения потоков.");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }

        System.out.println("Поток Один запущен: " + ob1.t.isAlive());
        System.out.println("Поток Два запущен: " + ob2.t.isAlive());
        System.out.println("Поток Три запущен: " + ob3.t.isAlive());
        System.out.println("Главный поток завершен.");
    }
}
```

Как можно увидеть, потоки прекращают исполнение после того, как вызовы метода join() возвращают управление.

Приоритеты потоков исполнения

Планировщик потоков использует приоритеты потоков исполнения, чтобы принять решение, когда разрешить исполнения каждому потоку. Теоретически высокоприоритетные потоки исполнения получают больше времени ЦП, чем низкоприоритетные. А на практике количество времени ЦП, которое получает поток исполнения, нередко зависит не только от его приоритета, но и от ряда других факторов. (Например, особенности реализации многозадачности в операционной системе могут оказывать влияние на относительную доступность времени ЦП.) Высокоприоритетный поток исполнения может также вытеснять низкоприоритетный. Например, когда низкоприоритетный поток исполняется, а высокоприоритетный собирается возобновить свое исполнение, прерванное в связи с приостановкой или ожиданием завершения операции ввода-вывода, то он вытесняет низкоприоритетный поток.

Теоретически потоки исполнения с одинаковым приоритетом должны получать равный доступ к ЦП. Но не следует забывать, что язык Java предназначен для применения в обширном ряде сред. В одних из этих сред многозадачность реализуется совершенно иначе, чем в других. В целях безопасности потоки исполнения с одинаковым приоритетом должны получать управление лишь время от времени. Этим гарантируется, что все потоки получат возможность выполнятся в среде операционной системы с невытесняющей многозадачностью. Но на практике даже в средах с невытесняющей многозадачностью большинство потоков все-тки имеют шанс для исполнения, поскольку во всех потоках неизбежно возникают ситуации блокировки, например, в связи с ожиданием ввода-вывода. Когда случается нечто подобное, исполнение заблокированного потока приостанавливается, а остальные потоки могут исполняться. Но если требуется добиться плавной работы многопоточой программы, то полагаться на случай лучше не стоит. К тому же в некоторых видах задач весьма интенсивно используется ЦП. Потоки, исполняющие такие задачи, стремятся захватить ЦП, поэтому передавать им управление следует изредка, чтобы дать возможность выполняться другим потокам. Чтобы установить приоритет потока исполнения, следует вызвать метода setPriority() из класса Thread. Его общая форма выглядит следующим образом:

```
final void setPriority(int уровень)
```

где аргумент уровень обозначает новый уровень приоритета для вызывающего потока исполнения. Значение аргумента уровень должно быть в пределах MIN_PRIORITY до MAX_PRIORITY. В настоящее время эти значения равны соответственно 1 и 10. Чтобы возвратить потоку исполнения приоритет по умолчанию, следует указать значение NORM_PRIORITY, которое в настоящее время равно 5. Эти приоритеты определены в классе Thread как статистические завершенные(static final) переменные. Для того чтобы получить текущее значение приоритета потока исполнения, достаточно вызвать метод getPriority() из класса Thread, как показано ниже.

```
final int getPriority()
```

Разные реализации Java могут вести себя совершенно иначе в отношении планирования потоков исполнения. Большинство несоответствий возникает при наличии потоков исполнения, опирающихся на вытесняющую многозадачность вместо совместного использования времени ЦП. Наиболее безопасный способ получить предсказуемое межплатформенное поведение многопоточных программ на Java состоит в том, чтобы использовать потоки исполнения, которые добровольно уступают управление ЦП.

Синхронизация потоков

Многопоточность дает возможность для асинхронного поведения прикладных программ, поэтому требуется каким-то образом обеспечить синхронизацию, когда в этом возникает потребность. Так, если требуется, чтобы два потока исполнения взаимодействовали и совместно использовали сложную структуру данных вроде связного списка, нужно найти способ предотвратить возможный конфликт между этими потоками. Это означает предотвратить запись данных в одном потоке исполнения, когда в другом потоке исполнения выполняется их чтение. Для этой цели в Java реализован изящный прием из старой модели межпроцессной синхронизации, называемый монитором. Монитор - это механизм управления, впервые определенный Чарльзом Энтони Хоаром. Монитор можно рассматривать как маленький ящик, одновременно хранящий только один поток исполнения. Как только поток исполнения войдет в монитор, все другие потоки исполнения должны ожидать до тех пор, пока тот не покинет монитор. Таким образом, монитор может служить для защиты общих ресурсов от одновременного использования более чем одним потоком исполнения.

Для монитора в Java отсутствует отдельный класс вроде Monitor. Вместо этого у каждого объекта имеется свой неявный монитор, вход в который осуществляется автоматически, когда для этого объекта вызывается синхронизированный метод. Когда поток исполнения находится в теле синхронизированного метода, ни один другой поток исполнения не может вызвать какой-нибудь другой синхронизированный метод для того же самого объекта. Это позволяет писать очень ясный и краткий многопоточный код, поскольку поддержка синхронизации встроена в язык.

Ключом к синхронизации является понятие монитора. Монитор - это объект, используемый в качестве взаимоисключающей блокировки. Только один поток исполнения может в одно и то же время владеть монитором. Когда поток исполнения запрашивает блокировку, то говорят, что он входит в монитор. Все другие потоки исполнения, пытающиеся войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не выйдет из монитора. Обо всех прочих потоках говорят, что они ожидают монитор. Поток, владеющий монитором, может, если пожелает, повторно войти в него. Синхронизировать прикладной код можно двумя способами, предусматривающими использование ключевого слова synchronized. Оба эти способа будут рассмотрены далее по очереди.

Каждый объект в java имеет свой монитор. Для достижения эффектов взаимного исключения и синхронизации потоков используют следующие операции:

- monitorenter: захват монитора. В один момент времени монитором может владеть лишь один поток. Если на момент попытки захвата монитор занят, поток, пытающийся его захватить, будет ждать до тех пор, пока он не освободится. При этом, потоков в очереди может быть несколько.
- monitorexit: освобождение монитора
- wait: перемещение текущего потока в так называемый wait set монитора и ожидание того, как произойдёт notify. Выход из метода wait может оказаться и ложным. После того, как поток, владеющий монитором, сделал wait, монитором может завладеть любой другой поток.

- notify(all): пробуждается один (или все) потоки, которые сейчас находятся в wait set монитора. Чтобы получить управление, пробуждённый поток должен успешно захватить монитор (monitorenter)

Contention — ситуация, когда несколько сущностей одновременно пытаются владеть одним и тем же ресурсом, который предназначен для монопольного использования. От того, есть ли contention на владение монитором, зависит то, как производится его захват. Монитор может находиться в следующих состояниях:

- init: монитор только что создан, и пока никем не был захвачен
- biased: (умная оптимизация) Монитор «зарезервирован» под первый поток, который его захватил. В дальнейшем для захвата этому потоку не нужны дорогие операции, и захват происходит очень быстро. Когда захват пытается произвести другой поток, либо монитор пере-резервируется для него (rebias), либо монитор переходит в состояние thin (revoke bias). Также есть дополнительные оптимизации, которые действуют сразу на все экземпляры класса объекта, монитор которого пытаются захватить (bulk revoke/rebias)
- thin: монитор пытаются захватить несколько потоков, но contention нет (т.е. они захватывают его не одновременно, либо с очень маленьким нахлёстом). Если возникает contention, то монитор переходит в состояние inflated
- fat/inflated: синхронизация производится на уровне операционной системы. Поток паркуется и спит до тех пор, пока не настанет его очередь захватить монитор. Даже если забыть про стоимость смены контекста, то когда поток получит управление, зависит ещё и от системного шедулера, и потому времени может пройти существенно больше, чем хотелось бы. При исчезновении contention монитор может вернуться в состояние thin.

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 добавиться, организовав взаимодействие потоков исполнения. Как будет показано ниже, добиться такого взаимодействия особенно просто в Java.

Как обсуждалось ранее, многопоточность заменяет программирование циклов ожидания событий благодаря разделению задач на дискретные, логически обособленные единицы. Еще одно преимущество предоставляют потоки исполнения, исключая опрос. Как правило, опрос реализуется в виде цикла, организуемого для периодической проверки некоторого условия. Как только условие оказывается истинным, выполняется определенное действие. Но на это расходуется время ЦП. Рассмотрим в качестве примера классическую задачу организации очереди, когда некоторые данные поставляются в одном потоке исполнения, а в другом потоке они потребляются. Чтобы сделать эту задачу более интересной, допустим, что поставщик данных должен ожидать завершения работы потребителя, прежде чем сформировать новые данные. В системах с опросом потребитель данных тратит немало циклов ЦП на ожидание данных от поставщика. Как только поставщик завершит работу, он должен начать опрос, напрасно расходуя лишние циклы ЦП в ожидании завершения работы потребителя данных, и т.д. Ясно, что такая ситуация нежелательна.

Чтобы избежать опроса, в Java внедрен изящный 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. Правила применения этих методов достаточно просты, хотя с точки зрения вычислительной техники они принципиально прогрессивны. Эти правила состоят в следующем.

- 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.

Все эти методы объявлены в классе Object, как показано ниже. Существуют дополнительные формы методы wait(), позволяющие указать время ожидания.

- final void wait() throws InterrptedException
- final void notify()
- final void notifyAll()

Применение синхронизированных методов

Синхронизация достигается в Java просто, поскольку у объектов свои, неявно связанные с ними мониторы. Чтобы войти в монитор объекта, достаточно вызвать метод, объявленный с модификатором доступа synchronized. Когда поток исполнения оказывается в теле синхронизированного метода, все другие потоки исполнения или любые другие синхронизированные методы, пытающиеся вызвать его для того же самого экземпляра, вынуждены ожидать. Чтобы выйти из монитора и передать управление объектом другому ожидающему потоки исполнения, владелец монитора просто возвращает управление из синхронизированного метода.

Чтобы стала понятнее потребность в синхронизации, рассмотрим сначала простой пример, в котором синхронизация отсутствует, хотя и должна быть осуществлена. Приведенная ниже программа состоит из трех простых классов. Первый из них, Callme, содержит единственный метод call(). Этот метод принимает параметр msg типа String и пытается вывести символьную строку msg, он вызывает метод Thread.sleep(1000), который приостанавливает текущий поток исполнения на одну секунду.

Конструктор следующего класса, Caller, принимает ссылку на экземпляры классов Callme и String, которые сохраняются в переменных target и msg соответственно. В этом конструкторе создается также новый поток исполнения, в котором вызывается метод run() для данного объекта. Этот поток запускается немедленно. В методе run() из класса Caller вызывается метод call() для экземпляра target класса Callme, передавая ему символьную строку msg. И наконец, класс Synch начинается с создания единственного экземпляра класса Callme и трех экземпляров класса Caller с отдельными строками

сообщения. Один и тот же экземпляр класса Callme передается каждому конструктору Caller().

```
class CallMe {
    void call(String msg) {
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Прервано'");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable {
    String msg;
    CallMe target;
    Thread t;

    public Caller(CallMe targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    public void run() {
        target.call(msg);
    }
}

class Synch {
    public static void main(String args[]) {
        CallMe target = new CallMe();
        Caller ob1 = new Caller(target, "Добро пожаловать");
        Caller ob2 = new Caller(target, "в синхронизированный");
        Caller ob3 = new Caller(target, "мир!");


        // ожидать завершения исполнения
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Прервано");
        }
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Добро пожаловать [ в синхронизированный [ мир ! ]
]
]
```

Как видите, благодаря вызову метода sleep() из метод call() удается переключиться на исполнение другого потока. Это приводит к смешанному выводу трех строк сообщений. В данной программе отсутствует механизм, предотвращающий одновременный вызов в потоках исполнения одного и тоже же метода для того же самого объекта, или так называемое состояние гонок, поскольку три потока соперничают за окончание метода. В данном примере применяется метод sleep(), чтобы добиться повторяемости и

наглядности получаемого эффекта. Но, как правило, состояние гонок менее заметно и предсказуемо, поскольку трудно предугадать, когда именно произойдет переключение контекста. В итоге программа может быть выполнена один раз правильно, а другой раз - неправильно.

Чтобы исправить главный недостаток данной программы, следует упорядочить доступ к методу call(). Это означает, что доступ к этому методу из потоков исполнения следует разрешить только по очереди. Для этого достаточно предварить объявление метода call() ключевым словом synchronized, как показано ниже.

```
CallMe {

synchronized void call(String msg) {

...

}
```

Этим предотвращается доступ к методу call() из других потоков исполнения, когда он уже используется в одном потоке. После ввода модификатора доступ synchronized в объявление метода call() результат выполнение данной программы будет выглядеть следующим образом:

```
[Добро пожаловать]
[в синхронизированный]
[мир!]
```

Всякий раз, когда имеется метод или группа методов, манипулирующих внутренним состоянием объекта в многопоточной среде, следует употребить ключевое слово synchronized, чтобы исключить состояние гонок. Напомним, что как только поток исполнения входит в любой синхронизированный метод экземпляра, ни один другой поток исполнения не сможет войти в какой-нибудь другой синхронизированный метод того же экземпляра. Тем не менее несинхронизированные методы экземпляра по-прежнему остаются доступными для вызова.

Оператор synchronized для объекта

Несмотря на всю простоту и эффективность синхронизации, которую обеспечивает создание синхронизированных методов, такой способ оказывается пригодным далеко не всегда. Чтобы стало понятнее, почему так происходит, рассмотрим следующую ситуацию. Допустим, требуется синхронизировать доступ к объектам класса, не предназначенного для многопоточного доступа. Это означает, что в данном классе не используются синхронизированные методы. Более того, класс написан сторонним разработчиком, и его исходный код недоступен, а следовательно, в объявление соответствующих методов данного класса нельзя ввести модификатор доступа synchronized. Как же синхронизировать доступ к объектам такого класса? К счастью, существует довольно простое решение этого вопроса: заключить вызовы методов такого класса в блок оператора synchronized. Ниже приведена общая форма оператора synchronized.

```
synchronized(ссылка_на_объект) {
```

```
        // синхронизируемые операторы

}
```

Здесь ссылка_на_объект обозначает ссылку на синхронизируемый объект. Блок оператора synchronized гарантирует, что вызов метода, являющегося членом того же класса, что и синхронизируемый объект, на который делается ссылка_на_объект, произойдет только тогда, когда текущий поток исполнения успешно войдет в монитор данного объекта.

Ниже приведена альтернативная версия программы из предыдущего примера, где в теле метода run() используется синхронизированный блок.

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

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    // синхронизированные вызовы метода call()
    public void run() {
        synchronized(target) { // синхронизированный блок
            target.call(msg);
        }
    }
}

class Synch1 {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Добро пожаловать");
        Caller ob2 = new Caller(target, "в синхронизированный");
        Caller ob3 = new Caller(target, "мир!");


// ожидать завершения исполнения
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Прервано");
        }
    }
}
```

В данном примере метод call() объявлен без модификатора доступа synchronized. Вместо этого используется оператор synchronized в теле метода run() из класса Caller. Благодаря этому получается тот же правильный результат, что и в предыдущем примере, поскольку каждый поток исполнения ожидает завершения предыдущего потока.

Взаимная блокировка

Следует избегать особого типа ошибок, имеющего отношение к многозадачности и называемой взаимной блокировкой, 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 следующим причинам.

- В общем, взаимная блокировка возникает очень редко, когда исполнение двух потоков точно совпадает по времени.
- Взаимная блокировка может возникнуть, когда в ней участвует больше двух потоков исполнения и двух синхронизированных объектов. (Это означает, что взаимная блокировка может произойти в результате более сложной последовательности событий, чем в упомянутой выше ситуации.)

Чтобы полностью разобраться в этом явлении, его лучше рассмотреть в действии. В приведенном ниже примере программы создаются два класса, A и B, с методами foo() и bar() соответственно, которые приостанавливаются непосредственно перед попыткой вызова метода из другого класса. Сначала в главном классе Deadlock получаются экземпляры классов A и B, а затем запускается второй поток исполнения, в котором устанавливается состояние взаимной блокировки. В методах foo() и bar() используется метод sleep(), чтобы стимулировать появление взаимной блокировки.

```
class Z {
    synchronized void foo(R r) {
        String name = Thread.currentThread().getName();

        System.out.println(name + " вошел в метод Z.foo()");

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("Класс Z прерван");
        }
        System.out.println(name + " пытается вызвать метод R.last()");
        r.last();
    }

    synchronized void last() {
        System.out.println("В методе Z.last()");
    }
}
```

```
class R {
    synchronized void bar(Z z) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " вошел в метод R.bar()");

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("Класс R прерван");
        }

        System.out.println(name + " пытается вызвать метод Z.last()");
        z.last();
    }

    synchronized void last() {
        System.out.println("R методе Z.last()");
    }
}

class Deadlock implements Runnable {
    Z z = new Z();
    R r = new R();

    Deadlock() {
        Thread.currentThread().setName("Главный поток");
        Thread t = new Thread(this, "Соперничающий поток");
        t.start();

        z.foo(r); // получить блокировку для объекта a в потоке исполнения
        System.out.println("Назад в главный поток");
    }

    public void run() {
        r.bar(z); // получить блокировку для объекта b в другом потоке исполнения
        System.out.println("Назад в другой поток");
    }

    public static void main(String args[]) {
        new Deadlock();
    }
}
```

Приостановка, возобновление и остановка потоков исполнения

Иногда возникает потребность в приостановке исполнения потоков. Например, отдельные поток исполнения может служить для отображения времени дня. Если пользователю не требуется отображение текущего времени, этот поток исполнения можно приостановить. Но в любом случае приостановить исполнение потока совсем не трудно. Выполнение приостановленного потока может быть легко возобновлено.

Механизм временной или окончательной остановки потока исполнения, а также его возобновления отличался в ранних версиях Java, например, Java 1.0, от современных версий, начиная с Java 2. До версии Java 2 методы suspend() и resume(), определенные в классе Thread, использовались в программах для приостановки и возобновления потоков исполнения. На первый взгляд применение этих методов кажется вполне благоразумным и удобным подходом к управлению выполнением потоков. Тем не менее пользоваться ими в новых программах на Java не рекомендуется по следующей причине: метод suspend() из класса Thread несколько лет назад был объявлен не рекомендованным к употреблению, начиная с версии Java 2. Это было сделано потому, что иногда он 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() так же 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() из класса Thread также объявлен устаревшим с версии Java 2. Это было сделано потому, что он может иногда 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. К счастью, это не так. 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 написание кода управления выполнением потока, но основной принцип остается неизменным для всех программ.

В приведенном ниже примере программы демонстрируется применение методов wait() и notify(), унаследованных из класса Object, для управления выполнением потока. Рассмотрим подробнее работу этой программы. Класс NewThread содержит переменную экземпляра suspendFlag типа boolean, используемую для управления выполнением потока. В конструкторе этого класса они инициализируется логическим значением false. Метод run() содержит блок оператора synchronized, где проверяется состояние переменной suspendFlag. Если она принимает логическое значение true, то вызывается метод wait() для приостановки выполнения потока. В методе mysuspend() устанавливается логическое значение true переменной suspendFlag, а в методе myresime() - логическое значение false этой переменной и вызывается метод notify(), чтобы активизировать поток исполнения. И наконец, в методе main() вызывается оба метода - mysuspend() и myresume().

```
class NewThread implements Runnable {
    String name; // имя потока исполнения
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        suspendFlag = false;
        t.start(); // запустить поток исполнения
```

```
    }

    // Точка входа в поток исполнения
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " прерван");
        }

        System.out.println(name + " завершен.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}

class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("Один");
        NewThread ob2 = new NewThread("Два");

        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Приостановка потока Один");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Возобновление потока Один");
            ob2.mysuspend();
            System.out.println("Приостановка потока Два");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Возобновление потока Два");
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }

        // ожидать завершения потоков исполнения
        try {
            System.out.println("Ожидание завершения потоков.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }

        System.out.println("Главный поток завершен");
    }
}
```

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

Получение состояния потока исполнения

Как упоминалось ранее в этой главе, поток исполнения может находиться нескольких состояниях. Для того чтобы получить текущее состояние потока исполнения, достаточно вызвать метод getState(), определенный в классе Thread, следующим образом:

```
Thread.State getState()
```

Этот метод возвращает значение типа Thread.State, обозначающее состояние потока исполнения на момент вызова. Перечисление State определено в классе Thread.

Имея в своем распоряжении экземпляр класса Thread, можно вызвать метод getState(), чтобы получить состояние потока исполнения. Например, в следующем фрагменте кода определяется, находится ли поток исполнения thrd в состоянии RUNNABLE во время вызова метода getState():

```
Thread.State ts = thrd.getState();

if(ts == Thread.State.RUNNABLE) //...
```

Следует, однако, иметь в виду, что состояние потока исполнения может измениться после вызова метода getState(). Поэтому в зависимости от обстоятельств состояние, полученное при вызове метода getState(), мгновение спустя может уже не отражать фактическое состояние потока исполнения. По этой и другим причинам метод getState() не предназначен для синхронизации потоков исполнения. Он служит прежде всего для отладки или профилированных характеристик потока во время выполнения.

Применение пакета Concurrent

Параллельные коллекции облегчают разработку многопоточных программ, предоставляя потокобезопасные, удачно сделанные структуры данных. Однако в некоторых случаях разработчику нужно сделать еще один шаг вперед и подумать о регулировании и/или ограничении выполнения потоков. Так как пакет java.util.concurrent призван упростить многопоточное программирование. Впервые Concurrency API был представлен вместе с выходом Java 5 и с тех пор постоянно развивался с каждой новой версией Java.

Класс ReentrantLock

Среда Lock в java.util.concurrent.lock - абстракция для блокировки, допускающая осуществление блокировок, которые реализуются как классы Java, а не как возможность языка. Это дает простор разнообразным вариантам применения Lock, которые могут иметь различные алгоритмы планирования, рабочие характеристики, или семантику блокировки. Класс reentrantLock, который реализует Lock, имеет те же параллелизм и

семантику памяти, что и synchronized, но также имеет дополнительные возможности, такие как опрос о блокировании (lock polling), ожидание блокирования заданной длительности и прерываемое ожидание блокировки. Кроме того, он предлагает гораздо более высокую эффективность функционирования в условиях жесткой состязательности. (Другими словами, когда много потоков пытаются получить доступ к ресурсу совместного использования, JVM потребуется меньше времени на установление очередности потоков и больше времени на ее выполнение.)

Что мы понимаем под блокировкой с повторным входом (reentrant)? Просто то, что есть подсчет сбора данных, связанный с блокировкой, и если поток, который удерживает блокировку, снова ее получает, данные отражают увеличение, и тогда для реального разблокирования нужно два раза снять блокировку. Это аналогично семантике synchronized; если поток входит в синхронный блок, защищенный монитором, который уже принадлежит потоку, потоку будет разрешено дальнейшее функционирование, и блокировка не будет снята, когда поток выйдет из второго (или последующего) блока synchronized, она будет снята только когда он выйдет из первого блока synchronized, в который он вошел под защитой монитора.

Если посмотреть на пример кода ниже, сразу бросается в глаза различие между Lock и синхронизацией - блокировка должна быть снята в последнем блоке. Иначе, если бы защищенный код показал исключительное состояние (ошибку), блокировка не была бы снята! Эта особенность может показаться тривиальной, но, фактически, она очень важна. Если забыть снять блокировку в последнем блоке, это будет бомбой замедленного действия в вашей программе, причину которой будет очень трудно обнаружить, и в конце концов она взорвется. Используя синхронизацию, JVM гарантирует, что блокировка автоматически снимаются.

Защита блока кода reentrantLock:

```
Lock lock = new reentrantLock();

lock.lock();

try {

  // update object state

}

finally {

  lock.unlock();

}
```

Как дополнительное преимущество можно отметить, что реализация reentrantLock гораздо более масштабируемая в условиях состязательности, чем реализация synchronized. (Вероятно, в состязательном режиме работы синхронных средств будут дальнейшие усовершенствования в следующей версии JVM.) Это значит, что когда много потоков соперничают за право получения блокировки, общая пропускная способность обычно лучше у reentrantLock, чем у synchronized.

В то время как reentrantLock - очень внушительная разработка и имеет некоторые существенные преимущества над синхронизацией, я полагаю, что поспешный вывод о том, что синхронизация - не заслуживающая внимания возможность, - это серьезная ошибка. Классы блокировки в java.util.compatible.lock - это передовой инструментарий для продвинутых пользователей и сложных ситуаций. В общем, вам следует использовать синхронизацию, если нет особой необходимости в одной из продвинутых возможностей Lock, или если вы продемонстрировали доказательства (а не просто подозрение) того, что синхронизация в данной конкретной ситуации - это помеха масштабируемости.

Синхронизация все же имеет некоторые преимущества над классами блокировки в java.util.compatible.lock. Например, невозможно забыть снять блокировку при использовании синхронизации; JVM сделает это за вас, когда вы выйдете из блока synchronized. Легко забыть использовать блок finally для снятия блокировки, что связано с большим ущербом для вашей программы. Ваша программа пройдёт все тесты и зависнет во время эксплуатации, и будет очень трудно понять, почему (что само по себе является существенной причиной для того, чтобы совсем не позволять начинающим разработчикам использовать Lock.

Другая причина состоит в том, что когда JVM управляет синхронизмом с автоподстройкой (lock acquisition) и освобождением (release) с использованием синхронизации, JVM может включать информацию о блокировке при генерировании дампов потоков. Они могут быть бесценны при отладке, поскольку могут идентифицировать источник зависаний или других неожиданных проявлений. Классы Lock - это просто обычные классы, и JVM еще не знает, какими объектами Lock владеют отдельные потоки. Более того, синхронизация знакома почти каждому разработчику Java и работает на всех версиях JVM. До тех пор, пока JDK 5.0 не станет стандартом, что, вероятно, случится года через два, использование классов Lock будет означать использование возможностей, присутствующих не в каждой JVM и знакомых не каждому разработчику.

Среда Lock - совместимая замена синхронизации, которая предлагает многочисленные возможности, не предоставляемые synchronized, так же как применения, предлагающие более высокую производительность в условиях состязательности. Однако, существование этих очевидных преимуществ не является весомой причиной для постоянного использования reentrantLock вместо synchronized.

ConcurrentHashMap

Класс ConcurrentHashMap из util.concurrent (который также появится в пакете java.util.concurrent в JDK 1.5) - это потокобезопасная реализация Map, предоставляющая намного большую степень параллелизма, чем synchronizedMap. Сразу много операций чтения могут почти всегда выполняться параллельно, одновременные чтения и записи могут обычно выполняться параллельно, а сразу несколько одновременных записей могут зачастую выполняться параллельно. (Соответственный класс ConcurrentReaderHashMap предлагает аналогичный параллелизм для множественных операций чтений, но допускает лишь одну активную операцию записи.) ConcurrentHashMap спроектирован для оптимизации операций извлечения; на деле, успешные операции get() обычно успешно выполняются безо всяких блокировок. Достижение потокобезопасности без блокировок является сложным и требует глубокого понимания деталей Модели Памяти Java (Java Memory Model). Реализация ConcurrentHashMap и остальная часть util.concurrent были в значительной степени проанализированы экспертами по параллелизму на предмет

корректности и безопасности потоков. Мы рассмотрим детали реализации ConcurrentHashMap в статье в следующем месяце.

ConcurrentHashMap добивается более высокой степени параллелизма, слегка смягчая обещания, которые даются тем, кто её вызывает. Операция извлечения возвратит значение, вставленное самой последней завершившейся операцией вставки, а также может возвратить значение, добавленное операцией вставки, выполняемой в настоящее время (но она никогда не возвратит бессмыслицы). Итераторы, возвращаемые ConcurrentHashMap.iterator() возвратят каждый элемент не более одного раза и никогда не выкинут ConcurrentModificationException, но могут отображать или не отображать вставки или удаления, имевшие место со времени, когда итератор был сконструирован. Блокировки целой таблицы не требуются (да и невозможны) для обеспечения потокобезопасности при переборе коллекции. ConcurrentHashMap может использоваться для замены synchronizedMap или Hashtable в любом приложении, которое не основано на способности делать блокировку всей таблицы для предотвращения модификаций.

Данные компромиссы позволяют ConcurrentHashMap обеспечивать намного более высокую масштабируемость, чем Hashtable, не ставя под угрозу его эффективность для широкого множества распространённых случаев, таких как кэш-память с общим доступом.

CopyOnWriteArrayList

Класс CopyOnWriteArrayList предназначен на замену ArrayList в параллельных приложениях, где обходы значительно превосходят по количеству вставки и удаления. Это достаточно типично для случая, когда ArrayList используется для хранения списка подписчиков, как в приложениях AWT или Swing или вообще в классах JavaBean. (Похожие на них CopyOnWriteArraySetиспользуют CopyOnWriteArrayList для реализации интерфейса Set.)

Если вы используете обычный ArrayList для хранения списка подписчиков, то до тех пор, пока список допускает изменения и к нему могут обращаться много потоков, вы должны либо блокировать целый список во время перебора либо клонировать его перед перебором, оба варианта обходятся значительной ценой. CopyOnWriteArrayList вместо этого создаёт новую копию списка каждый раз, когда выполняется модифицирующая операция, и гарантируется, что её итераторы возвращают состояние списка на момент, когда итератор был сконструирован и не выкинут ConcurrentModificationException. Нет необходимости клонировать список до перебора или блокировать его во время перебора, потому что копия списка, которую видит итератор, не будет изменяться. Другими словами, CopyOnWriteArrayList содержит изменяемую ссылку на неизменяемый массив, поэтому до тех пор, пока эта ссылка остаётся фиксированной, вы получаете все преимущества потокобезопасности от неизменности без необходимости блокировок.

Синхронизированные классы коллекций Hashtable и Vector и синхронизированные классы обрамления Collections.synchronizedMap и Collections.synchronizedList предоставляют базовую условно потокобезопасную реализацию Map и List. Однако несколько факторов делают их непригодными для использования в приложениях с высоким уровнем параллелизма - используемая в них единственная блокировка на всю коллекцию является препятствием для масштабируемости, а зачастую бывает необходимо блокировать коллекцию на значительное время в момент перебора для предотвращения исключений ConcurrentModificationException. Реализации

ConcurrentHashMap и CopyOnWriteArrayList обеспечивают намного больший уровень параллелизма при сохранении безопасности потоков и при нескольких незначительных компромиссах в их обещаниях перед вызывающей стороной. ConcurrentHashMap и CopyOnWriteArrayList не обязательно полезны везде, где вы могли бы использовать HashMap или ArrayList, но они спроектированы для оптимизации в определённых распространённых ситуациях. Многие параллельные приложения будут в выигрыше от их использования.

Атомарные переменные в java.util.concurrent

Почти все классы в пакете java.util.concurrent используют атомарные переменные вместо синхронизации, либо прямо, либо косвенно. Такие классы, как ConcurrentLinkedQueue используют атомарные переменные для прямой реализации алгоритмов wait-free и такие классы, как ConcurrentHashMap используют ReentrantLock для блокировки, если это необходимо. В свою очередь, ReentrantLock использует атомарные переменные для соблюдения очередности потоков, ожидающих блокировки.

Эти классы невозможно было бы сконструировать без усовершенствования машины JVM в версии JDK 5.0, которая сделала явным (для библиотек классов, но не для библиотек пользователей) интерфейс доступа к базовым элементам синхронизации аппаратного уровня. Классы атомарных переменных и, в свою очередь, другие классы в java.util.concurrent экспонируют эти характеристики классам пользователей.

Литература и ссылки

1. Спецификация языка Java https://docs.oracle.com/javase/specs/jls/se8/html/index.html
2. Шилдт Г. Java 8. Полное руководство. 9-е издание. — М.: Вильямс, 2012. — 1377 с.
3. Эккель Б. Философия Java. 4-е полное изд. — СПб.: Питер, 2015. — 1168 с.
4. https://www.ibm.com/developerworks/ru/library/j-5things5/index.html
5. https://www.ibm.com/developerworks/ru/library/j-jtp10264/index.html
6. https://www.ibm.com/developerworks/ru/library/j-jtp07233/index.html
7. https://www.ibm.com/developerworks/ru/library/j-jtp02244/
8. https://www.ibm.com/developerworks/ru/library/j-jtp03304/index.html

Вопросы для самоконтроля

1. Перечислите основные способы запуска кода в потоке
2. Перечислите основные способы синхронизации
3. Что такое Deadlock?
4. Перечислите состав пакета java.util.concurrent