МЕТОДЫ РАБОТЫ С ПОТОКАМИ В ЯЗЫКЕ JAVA

С.А. Опивалов, *старший разработчик программного обеспечения*
Кубанский государственный университет
(Россия, г. Краснодар)

DOI:10.24412/2500-1000-2023-4-3-93-99

Аннотация. *В статье рассмотрены наиболее распространенные методы работы с потоками в языке Java. Определено, что работа с потоками сегодня является одним из наиболее популярных методов повышения производительности программного обеспечения. Отмечено, что стремление к оптимизации и более высокой производительности программного обеспечения привело к развитию многопоточности – параллельного выполнения строк программного кода. Одним из основоположников многопоточного подхода к созданию программного обеспечения является язык Java. Сделан вывод о том, что рассмотренные отличия формируют общую картину использования методов повышения производительности программного обеспечения в языке Java, что позволит организовать более эффективную многопоточную обработку, и оптимизировать использование современных многоядерных процессоров.*

Ключевые слова: *создание программного обеспечения, Java разработка, многопоточность.*

Java содержит в себе возможность работы с потоками и формирования многопоточного выполнения программного кода. Ярким примером можно назвать работу многопроцессорной системы – здесь для каждого ядра на исполнение могут быть представлены собственные инструкции, что по итогу позволит решать задачи значительно быстрее. Однако и одноядерная система может организовать некое подобие одновременной работы над несколькими инструкциями, осуществляя быстрое переключение между ними, которое будет попросту незаметно для пользователя. Работа с отдельным набором инструкций будет организована виртуальным ядром. Подобное исполнение программного кода называют многопоточностью. Это один из важных моментов при изучении и работе с данным языком разработки, чем обусловлена актуальность выбранной темы статьи. В рамках исследования планируется рассмотрение основных способов и подходов организации работы с многопоточным кодом в языке программирования Java с целью формирования базовой картины работы с потоками [4].

Любой язык программирования в своей основе имеет принципы последовательного выполнения инструкций, то есть весь программный код выполняется последовательно, строка за строкой. Переход на следующую строку не может быть произведен до того, пока не будет выполнена предыдущая строка. Этот подход гарантирует выполнение всех требуемых строк кода в соответствии с алгоритмом работы, однако за счет этого иногда существенно возрастает время выполнения программного кода. Например, если в коде имеет обращение к удаленному серверу, то работа программы будет приостановлена до того момента, пока от сервера не придет необходимый ответ. Это один из недостатков последовательного выполнения программы.

Многопоточность является расширением идеи многозадачности в рамках функционирования программного обеспечения, так как позволяет разбить один программный продукт на несколько базовых задач и каждую задачу запускать на исполнение отдельным потоком. Работа потоков будет происходить в параллельном режиме, а распределением ресурсов между потоками будет руководить операционная система, по аналогии с распределением ресурсов между процессами. Многопоточность можно назвать многозадачностью в рамках одного программного продукта, управление которой осуществляется инструмента-

рием операционной системы. При реализации потоков в рамках процесса выполнения программного продукта для ОС каждый поток представлен в виде более легковесного подпроцесса, которому требуется собственный набор ресурсов, и который обладает собственным приоритетом исполнения [1].

Работа потоков в рамках многопоточной программы также может быть организована по-разному, например, они могут вообще не взаимодействовать, выполняться каждый сам по себе, и работать каждый со своим набором данных. Кроме того, потоки могут взаимодействовать друг с другом и обмениваться данными, либо работать независимо друг от друга, но по завершению их работы данные будут формировать единый результат. На рисунке 1 представлены стадии жизни потока.

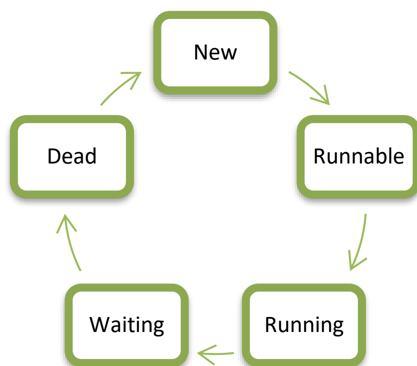

Рис. 1. Стадии жизни потока

На первой фазе происходит создание нового потока, и он остается в состоянии «New» до того момента, пока он не будет запущен программным продуктом. При вызове потока он переходит в состояние «Runnable», что дословно можно перевести как «Готовый к выполнению», и далее управление им переходит к планировщику. После того как будет начато выполнение потока, его состояние будет изменено на «Running» (Выполняемый). Планировщиком осуществляется выборка необходимого потока, после чего он выполняется в рамках работы программы. Переход в состояние «Waiting» (Ожидающий) формируется в тот момент времени, когда требуется синхронизация потоков, по причине чего поток должен приостановить выполнение, ожидая, когда будет завершено выполнение другого потока. После того, как поток будет остановлен, а процесс, в котором он выполнялся – завершен, поток перейдет в состояние «Dead», что можно дословно перевести как «Умерший», то есть его работа останавливается, и поток уничтожается, а ресурсы высвобождаются [5].

Работа потоков схожа с работой процессов, по этой причине они точно также используют ресурсы. С той разницей, что происходит использование ресурсов, выделенных основному процессу программы. Это является эффективной, но при этом проблематичной коммуникацией. В многопоточном приложении запускаемый первым поток носит название главного (основного) потока, который впоследствии может вызывать дополнительные потоки. Для каждого потока возможно планирование его выполнение на отдельном ядре процессора, либо исполнение на нескольких ядрах [2].

Многопоточность тем самым – отличный инструмент, позволяющий «распараллелить» выполнение программы и добиться более быстрой её работы. Данный подход к разработке программного обеспечения обладает как положительными сторонами, так и отрицательными. К числу преимуществ многопоточной реализации программного обеспечения следует отнести возможность разбиения работы программного продукта и получения в большей степени адаптивного ПО, оптимизация использования ресурсов, прирост в производительности программы. Недостатками многопоточной реализации кода является

необходимость обеспечения синхронизации и согласования потоков, по причине работе этих потоков с одним набором данных, расположенным в одной области памяти. При разработке многопоточного программного обеспечения его очень сложно проектировать, а также выполнять процедуры отладки при наличии ошибок. В рамках многопоточного приложения центральному процессору требуется выполнение операции переключения потока, что потребует от него существенных затрат процессорного времени. По этой причине важно грамотно организовать разбиение программы на потоки, так как при росте числа потоков может быть получена ситуация, когда производительность не возрастет, а наоборот упадет [6].

Реализация многопоточности в Java. Thread & Runnable

Язык Java уже в своих первых версиях содержал несколько инструментов организации работы с потоками. Это методы для работы с классом Thread, интерфейс Runnable, ключевое слово synchronized, а также набор методов, направленных на реализацию синхронизации в рамках класса Object. Далее, по мере развития, язык обзавелся пакетом java.util.concurrent, содержащим новые инструменты для работы с потоками, и классом CompletableFuture, позволяющим осуществлять построение цепочек асинхронных задач и комбинировать их.

При реализации многопоточного программного обеспечения в Java используется несколько различных методов:

- применение механизмов синхронизации, блокировки в совокупности с ключевым словом volatile;
- использование транзакционной памяти, что предоставляет своего рода рекурсивный параллелизм;
- применение модели «акторов», в рамках которой потоки выступают в роли объекта, взаимодействие между которыми происходит в формате обмена сообщениями [3].

Сегодня работа с потоками в рамках языка Java осуществляется с использованием двух подходов – класса Thread и интерфейса Runnable.

Thread

При работе с классом Thread необходимо создание нового объекта класса, который вызывается методом start для начала работы нового потока. Пример создания потока с использованием класса Thread:

```
class Work extends Thread {
public void run()
{ try {
System.out.println( «На текущий момент времени выполняется » + Thread.currentThread().getId()
+ » поток»);
}
catch (Exception e) {
System.out.println(«Произошло исключительное событие»); } } }
public class Multithread {
public static void main(String[] args)
{int i = 5;
for (int x = 0; x < i; x++) {
Work object
= new Work();
object.start(); } } }
```

Представленный пример является простым консольным приложением, при работе которого будет запущено 5 потоков, каждый из которых будет осуществлять вывод на экран своего номера в момент исполнения.

Runnable

Второй вариант работы с потоками в java – это работа с интерфейсом Runnable. Здесь по аналогии требуется создание нового класса с реализацией интерфейса java.lang.Runnable. Это позволяет реализовать новый объект Thread, запуск которого происходит на основании метода start:

```
class Work implements Runnable {
public void run()
{try { System.out.println(
« На текущий момент времени выполняется » + Thread.currentThread().getId()
+ » поток»); }
catch (Exception e) {
System.out.println(«Произошло исключительное событие»); } } }
class Multithread {
public static void main(String[] args)
{int i = 5; // Число потоков
for (int n = 0; n < i; n++) {
Thread object
```

= new Thread(new Work());
object.start(); } } }

Данный пример аналогичен предыдущем, с той разницей что используется интерфейс Runnable при создании нового потока [7].

Сравнение Thread и Runnable

Рассмотренные методы работы с потоками в Java обладают каждый своими особенностями. Например, при использовании метода Thread новый класс уже не может быть использован для расширения другого класса, так как в языке Java не существует множественного наследования. А вот реализация потока на основании интерфейса Runnable предоставляет гибкие возможности по расширению других классов. Далее – Thread предоставляет создаваемому классу базовый функционал в виде таких методов, как yield() и interrupt(), что недоступно в интерфейсе Runnable, однако этот интерфейс позволяет создавать общие для нескольких потоков объекты.

Применение интерфейса Runnable считается обоснованным для ситуаций, когда имеется наследование классом какого-либо родительского класса, в связи с чем нет возможности расширения класса Thread. Это обусловлено расширенным подходом в рамках Java, в рамках которого реализуются интерфейсы по причине возможности наследования только одного родительского класса. Ведь при наследовании класса Thread нельзя будет сформировать наследование какого-то другого класса. Использование же расширения Thread обосновано для ситуаций, когда в рамках другого класса требуется переопределение каких-либо других методов, помимо метода run.

Проблемы многопоточности в Java. Примитивы синхронизации

При работе потоков с кодом, имеющим доступ к одной и той же переменной, то результаты могут быть просто непредсказуемыми. Чтобы минимизировать риски ошибочных ситуаций в Java существует методика организации процесса синхронизации потоков.

Базовыми примитивами организации потоков в Java являются такие объекты, как «Монитор» и «Семафор». Монитор представляет собой объект, используемый для обеспечения взаимоисключающей блокировки. В ситуации, когда потоком происходит запрос блокировки ресурса, говорят, что данный поток осуществляет вход в монитор. Только один поток может работать с монитором в определенный момент времени, все остальные в это же время при попытке обращения к этому же монитору, будут приостановлены до момента освобождения монитора текущим потоком. Семафор можно назвать разновидностью монитора, обладающего существенным отличием – счетчиком с числом разрешений. Данный счетчик демонстрирует число потоков, которые могут одновременно работать с семафором. Для возможности использования семафора данный счетчик должен быть более единицы. При обращении к семафору счетчик уменьшается на единицу, а при завершении работы потока с семафором – увеличивается на единицу. Как только значение счетчика достигнет нуля, то установится блокировка семафора. Иными словами, в текущий момент времени с семафором будет работать максимально разрешенное число потоков. Работа с данными примитивами осуществляется с использованием пакета locks, в частности инструкций lock и unlock.

Более простым вариантом синхронизации потоков является использование ключевого слова synchronized и подхода, который называется концепцией «монитора». Эта концепция была реализована в языке Paskal, а в Java она была реализована в виде собственного «монитора» для каждого класса. В рамках данной концепции каждый монитор содержит 4 поля:

1. locked (boolean) – показывает, захвачен монитор или нет;
2. owner (Thread) сюда записывается поток, который захватил данный монитор;
3. blocked set – это подмножество, куда попадают потоки, которые не смогли захватить блокировку, или поток, который выходит из состояния wait;
4. wait set – в это множество попадают потоки, для которых был вызван метод wait.

Ни одно из полей монитора не может быть получено посредством рефлексии. А каждый из объектов обладает методами wait, notify и notify All, унаследованными от класса Object. При применении ключевого слова synchronized предоставляется гарантия в отношении того, что выполнение блоков кода в единицу времени будет реализовано только одним потоком.

Ключевое слово synchronized может быть использовано двумя различными вариантами:

- двумя потоками происходит выполнение программного кода, выполнение которого возможно в один момент времени только одним потоком;
- один из потоков ожидает возникновение некоторого события, которое обеспечивается методами wait, notify и notify All.

В примере представлена реализация использования ключевого слова synchronized:

```
public class MainClass  {
private static final Object LOCK = new Object();
public static void main(String [] args) throws InterruptedException {
synchronized(LOCK) {
LOCK.wait(); } } }
```

Помимо ключевого слова synchronized при работе с потоками в Java применяется ключевое слово volatile, которое позволяет организовать работу с какой-либо переменной непосредственно напрямую. При работе без данного ключевого слова переменная копируется в кэш процессора, и в случае использования многоядерной системы в различных ядрах одна и таже переменная может обладать различными значением. Это приведет к возникновению нескольких различающихся копий одной переменной. и. При использовании не volatile переменных нельзя знать наверняка, когда JVM читает значение переменной из главной памяти и когда записывается значение переменной в главную память. В примере представлена запись переменной счетчика с применением метода volatile:

```
public class SharedObject {
public volatile int counter = 0; }
```

Именно для устранения данной проблемы в рамках работы с потоками используется прямая работа с переменной, хранимой в оперативной памяти с использованием метода volatile.

Асинхронность. CompletableFuture

Асинхронное выполнение задач в рамках работы программного обеспечения представляет собой предоставление главному процессу возможности выполнить запуск какой-либо задачи, после чего не дожидаться её выполнения, подпроцесс сам уведомит главный поток о том, что задача была завершена.

Одним из первых вариантов реализации асинхронности в Java является использование интерфейса Runnable и класса для работы с потоками Thread. В качестве наиболее оптимального варианта для управления потоками существует возможность задействовать в программном коде исполнителей (Executor), которые используют в рамках своей рабочей деятельности разные пулы потоков. В том случае, если при выполнении потока потребуется получить какой-либо результат вычисления, можно использовать интерфейс Callable. Это приведет к тому, что от задачи будет незамедлительно получен ответ, достаточно после завершения вычислений вызвать метод get. Недостатком данного подхода является блокировка потока до того момента, пока не будет получен ответ. Для устранения данного недостатка может быть использован метод future.isDone, который в рамках своей работы будет выполнять постоянную проверку, завершено ли вычисление, и после того как данным методом будет возвращено значение true, методом get() будет передан результат вычислений.

Современные версии JDK содержат обновленный объект – CompletableFuture. Его отличительная особенность заключается в том, что помимо будущих объектов он позволяет реализовать этап завершения, называемый как CompletionStage. Это позволяет реализовать большое количество методов, упрощающих работу с результатами вычислений в различных потоках и на различных этапах.

Класс CompletableFuture представляет собой современный инструментарий организации информационного обмена между

параллельно выполняемыми потоками. Фактически это механизмы реализации блокирующей очереди, которая осуществляет передачу одного ссылочного значения, а также исключения в случае его возникновения при вычислении передаваемых значений.

Создание CompletableFuture осуществляется с использованием метода supplyAsync:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Привет");
```

Если требуется указать, где будет исполняться future, необходимо выполнить передачу Executor вторым параметром:

```
CompletableFuture<String> future = CompletableFuture
  .supplyAsync(() -> «Привет», Executors.newCachedThreadPool());
```

Существует еще один вариант создания CompletableFuture –с использованием runAsync. В приведенном ниже примере указано создание CompletableFuture с указанием Executor:

```
CompletableFuture<Void> future = CompletableFuture
  .runAsync(() -> System.out.println("Привет"), Executors.newCachedThreadPool());
```

Разница в двух рассмотренных подходах заключается том, что с помощью supplyAsync() можно вернуть результат, с runAsync() - нельзя.

Для того, чтобы получить результат с *CompletableFuture* необходимо вызвать метод get():

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep(500);
    } catch (InterruptedException e) {}
    return «Привет»;
});
System.out.println(future.get());
```

При подобной реализации вызова произойдет блокировка выполнения программы либо потока до того момента, пока CompletableFuture не вернет полученный результат.

По этой причине более приемлемым является вариант, при котором обработка полученных результатов работы будет происходить с использованием callback:

```
CompletableFuture <String> future = CompletableFuture.supplyAsync(() -> "Привет");
future.thenAccept(result -> System.out.println(result));
future.get();
```

Вызов get осуществляется с целью ожидания исполнения Future. При этом блокировки исполнения программы в данном случае не возникнет.

При работе с CompletableFuture существует возможность добавления нескольких callback, с использованием метода thenApply:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Привет");
future.thenApply(result -> {
    System.out.println(result + " всем");
    return result;
});
future.thenApply(result -> {
    System.out.println(result + ", мир!");
    return result;
});
future.get();
```

Важным фактом для данного примера является исполнение функции thenApply внутри того же потока, в котором она была вызвана.

Асинхронность vs. Многопоточность

Сравнивая работу асинхронности и многопоточности, необходимо отталкиваться от подходов их реализации. При многопоточной реализации программного продукта его исполняемый код выполняется в различных потоках. Например, происходит запуск главного потока, а вспомогательные вычисления производятся во вспомогательных потоках. Асинхронное программирование же подразумевает осуществление инициации определенной операции, о завершении выполнение которой основной поток узнает только по истечении определенного времени. Наиболее часто данный подход используется при организации работ с устройствами ввода-вывода. То есть асинхронное программирование подразумевает реализацию программного кода в рамках другого потока,

без осуществления блокировки текущего потока.

В заключение следует отметить, что многопоточность является одной из важных тем как при изучении программирования, так и при реализации программного обеспечения. Для любого современного программного продукта характерно наличие нескольких потоков, так как это позволяет добиться увеличения производительности и осуществлять одновременное решение нескольких задач.

Java содержит несколько методов работы с потоками, наиболее популярными из которых являются интерфейс Runnable и класс Thread. Данные методы схожи в плане реализации создания нового класса, однако принципы работы с ними в дальнейшем отличаются. Именно это является особенностью данных методов. Рассмотренные отличия этих методик формируют общую картину их использования, что позволит организовать более эффективную многопоточную обработку, и оптимизировать использование современных многоядерных процессоров.

Библиографический список

1. Болбот О.М. Классы в языке программирования Java: учебно-методическое пособие / О.М. Болбот, В.В. Сидорик; под редакцией В.В. Сидорика. – Минск: БНТУ, 2020. – 76 с.
2. Вязовик Н.А. Программирование на Java: учебное пособие. – М.: ИНТУИТ, 2016. – 603 с.
3. Гуськова О.И. Объектно ориентированное программирование в Java: учебное пособие. – М.: МПГУ, 2018.
4. Коузен К. Современный Java: рецепты программирования. – М.: ДМК Пресс, 2018. – 275 с.
5. Никитенкова С.П. Многопоточное программирование на языке JAVA: учебно-методическое пособие. – Нижний Новгород: ННГУ им. Н.И. Лобачевского, 2015. – 90 с.
6. Пономарчук Ю.В. Программирование на языке Java: учебное пособие / Ю. В. Пономарчук, И. В. Кузнецов. – Хабаровск: ДВГУПС, 2021. – 103 с.
7. Хабитуев Б.В. Программирование на языке Java: практикум: учебное пособие. – Улан-Удэ: БГУ, 2020. – 94 с.

JAVA THREADING TECHNIQUES

S.A. Opivalov, *Gradle Inc. Senior Software Engineer*
Kuban State University
(Russia, Krasnodar)

Abstract. *This article discusses the most common methods for working with threads in the Java language. It has been determined that working with threads today is one of the most popular methods for improving software performance. It is noted that the desire for optimization and higher software performance has led to the development of multithreading – parallel execution of lines of program code. One of the founders of the multi-threaded approach to software development is the Java language. It is concluded that the considered differences form a general picture of the use of methods to improve software performance in the Java language, which will allow organizing more efficient multi-threaded processing and optimizing the use of modern multicore processors.*

Keywords: *software development, Java development, multithreading.*