Многопоточное программирование в Java 8. Часть вторая. Синхронизация доступа к изменяемым объектам

Рассказывает Бенджамин Винтерберг (http://winterbe.com), Software Engineer

Добро пожаловать во вторую часть руководства по параллельному программированию (https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB% в Java 8. В предыдущей части (https://tproger.ru/translations/java8-concurrency-tutorial-1/) мы рассматривали, как выполнять код параллельно с помощью потоков, задач и сервисов исполнителей. Сегодня мы разберём, как синхронизировать доступ к изменяемым объектам с помощью ключевого слова `synchronized`, блокировок и семафоров.

Большинство принципов, которые описаны в этой статье, справедливы и для более старых версий Java. Тем не менее, я не задавался проблемами совместимости, и примеры содержат как лямбды, так и новые возможности многопоточности. Если вы не очень хорошо знакомы с лямбда-выражениями, то вам стоит сперва прочитать мой туториал по Java 8 (http://winterbe.com/posts/2014/03/16/java-8-tutorial/).

Для простоты примеров я использую в них два метода-помощника: `sleep(секунды)` и `stop(сервис-исполнитель)`. Их реализации я выложил на GitHub (https://github.com/winterbe/java8-tutorial/blob/master/src/com/winterbe/java8/samples/concurrent/ConcurrentUtils.java), если кому-то интересно.

Синхронизация

Мы уже узнали, как выполнять код параллельно с помощью сервисов-исполнителя (*ExecutorService*). Во время написания многопоточной программы нужно уделять особое внимание работе с общими для потоков изменяемыми объектами. Давайте представим, что мы хотим увеличить такую переменную на единицу.

Мы создаём поле `count` и метод `increment()`, который увеличивает `count` на единицу:

```
1  int count = 0;
2
3  void increment() {
4      count = count + 1;
5  }
```

Если мы будем вызывать этот метод одновременно из двух потоков, у нас возникнут серьёзные проблемы:

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10000)
    .forEach(i -> executor.submit(this::increment));

stop(executor);

System.out.println(count);  // 9965
```

Вместо ожидаемого постоянного результата 10000 мы будем каждый раз получать разные числа. Причина этого — использование изменяемой переменной несколькими потоками без синхронизации, что вызывает состояние гонки (https://ru.wikipedia.org/wiki/Состояние_гонки) (*race condition*).

Увеличение числа на единицу происходит в три шага: (1) считать значение переменной, (2) увеличить это значение на единицу и (3) записать назад новое значение. Если два потока будут одновременно выполнять эти шаги, то вполне вероятно, что они могут выполнить первый шаг одновременно, считав одно и то же значение. Затем они запишут в переменную одно и то же значение, и вместо увеличения на 2 получится увеличение на единицу. Поэтому конечное значение и получается меньше ожидаемого.

К счастью, Java поддерживает синхронизацию потоков с самых ранних версий, используя для этого ключевое слово `synchronized`. Вот как следовало бы переписать наш код:

```
synchronized void incrementSync() {
    count = count + 1;
}
```

Тогда, после выполнения метода 10000 раз, мы всегда будем получать значение 10000, и никакой гонки состояний возникать не будет:

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10000)
    .forEach(i -> executor.submit(this::incrementSync));

stop(executor);

System.out.println(count);  // 10000
```

Это ключевое слово можно применять не только к методам, но и к отдельным их блокам:

```
void incrementSync() {
    synchronized (this) {
        count = count + 1;
    }
}
```

Под капотом Java использует так называемый монитор (https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html) (*monitor lock*, *intrinsic lock*) для обеспечения синхронизации. Этот монитор привязан к объекту, поэтому синхронизированные методы используют один и тот же монитор соответствующего объекта. Все неявные мониторы устроены реентерабельно (https://ru.wikipedia.org/wiki/Реентерабельность) (*reentrant*), т.е. таким образом, что поток может без проблем вызывать

блокировку одного и того же объекта, исключая взаимную блокировку (https://ru.wikipedia.org/wiki/Взаимная_блокировка) (например, когда синхронизированный метод вызывает другой синхронизированный метод на том же объекте).

Блокировки

Кроме использования блокировок неявно (с помощью ключевого слова `synchronized`), Concrurrency API предлагает много способов их явного использования, определённых интерфейсом `Lock`. С помощью явных блокировок можно настроить работу программы гораздо тоньше и тем самым сделать её эффективнее.

Стандартный JDK предоставляет множество реализаций `Lock`, которые мы сейчас и рассмотрим.

ReentrantLock

Класс `ReentrantLock` реализует то же поведение, что и обычные неявные блокировки. Давайте попробуем переписать наш пример с увеличением на единицу с помощью него:

```
1  ReentrantLock lock = new ReentrantLock();
2  int count = 0;
3
4  void increment() {
5      lock.lock();
6      try {
7          count++;
8      } finally {
9          lock.unlock();
10     }
11 }
```

Блокировка осуществляется с помощью метода `lock()`, а освобождаются ресурсы помощью метода `unlock()`. Очень важно оборачивать код в `try{}finally{}`, чтобы ресурсы освободились даже в случае выброса исключения. Код, представленный выше, так же потокобезопасен, как и его аналог с `synchronized`. Если один поток вызвал `lock()`, и другой поток пытается получить доступ к методу до вызова `unlock()`, то второй поток будет простаивать до тех пор, пока метод не освободится. Только один поток может удерживать блокировку в каждый момент времени.

Для большего контроля явные блокировки поддерживают множество специальных методов:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
ReentrantLock lock = new ReentrantLock();

executor.submit(() -> {
    lock.lock();
    try {
        sleep(1);
    } finally {
        lock.unlock();
    }
});

executor.submit(() -> {
    System.out.println("Locked: " + lock.isLocked());
    System.out.println("Held by me: " + lock.isHeldByCurrentThread());
    boolean locked = lock.tryLock();
    System.out.println("Lock acquired: " + locked);
});

stop(executor);
```

Пока первый поток удерживает блокировку, второй выведет следующую информацию:

```
Locked: true
Held by me: false
Lock acquired: false
```

Метод `tryLock()`, в отличие от обычного `lock()` не останавливает текущий поток в случае, если ресурс уже занят. Он возвращает булевый результат, который стоит проверить перед тем, как пытаться производить какие-то действия с общими объектами (истина обозначает, что контроль над ресурсами захватить удалось).

ReadWriteLock

Интерфейс `ReadWriteLock` предлагает другой тип блокировок — отдельную для чтения, и отдельную для записи. Этот интерфейс был добавлен из соображения, что считывать данные (любому количеству потоков) безопасно до тех пор, пока ни один из них не изменяет переменную. Таки образом, блокировку для чтения (*read-lock*) может удерживать любое количество потоков до тех пор, пока не удерживает блокировка для записи (*write-lock*). Такой подход может увеличить производительность в случае, когда чтение используется гораздо чаще, чем запись.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
ReadWriteLock lock = new ReentrantReadWriteLock();

executor.submit(() -> {
    lock.writeLock().lock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.writeLock().unlock();
    }
});
```

В примере выше мы можем видеть, как поток блокирует ресурсы для записи, после чего ждёт одну секунду, записывает данные в `HashMap` и освобождает ресурсы. Предположим, что в это же время были созданы ещё два потока, которые хотят получить из хэш-таблицы значение:

```
Runnable readTask = () -> {
    lock.readLock().lock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.readLock().unlock();
    }
};

executor.submit(readTask);
executor.submit(readTask);

stop(executor);
```

Если вы попробуете запустить этот пример, то заметите, что оба потока, созданные для чтения, будут простаивать секунду, ожидая завершения работы потока для записи. После снятия блокировки они выполнятся параллельно, и одновременно запишут результат в консоль. Им не нужно ждать завершения работы друг друга, потому что выполнять одновременное чтение вполне безопасно (до тех пор, пока ни один поток не работает параллельно на запись).

StampedLock

В Java 8 появился новый тип блокировок — `StampedLock`. Так же, как и в предыдущих примерах, он поддерживает разделение на `readLock()` и `writeLock()`. Однако, в отличие от `ReadWriteLock`, метод блокировки `StampedLock` возвращает «штамп» — значение типа `long`. Этот штамп может использоваться в дальнейшем как для высвобождения ресурсов, так и для проверки состояния блокировки. Вдобавок, у этого класса есть методы, для реализации «оптимистичной» блокировки. Но обо всём по порядку.

Вот таким образом следовало бы переписать наш предыдущий пример под использование `StampedLock`:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.unlockWrite(stamp);
    }
});

Runnable readTask = () -> {
    long stamp = lock.readLock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.unlockRead(stamp);
    }
};

executor.submit(readTask);
executor.submit(readTask);

stop(executor);
```

Работать этот код будет точно так же, как и его брат-близнец с `ReadWriteLock`. Тут, правда, стоит упомянуть, что в `StampedLock` *не* реализована реентерантность. Поэтому особое внимание нужно уделять тому, чтобы не попасть в ситуацию взаимной блокировки (*deadlock*).

В следующем примере демонстрируется «оптимистичная блокировка»:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.tryOptimisticRead();
    try {
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
        sleep(1);
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
        sleep(2);
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
    } finally {
        lock.unlock(stamp);
    }
});

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        System.out.println("Write Lock acquired");
        sleep(2);
    } finally {
        lock.unlock(stamp);
        System.out.println("Write done");
    }
});

stop(executor);
```

Оптимистичная блокировка для чтения, вызываемая с помощью метода `tryOptimisticRead()`, отличается тем, что она всегда будет возвращать «штамп» не блокируя текущий поток, вне зависимости от того, занят ли ресурс, к которому она обратилась. В случае, если ресурс был заблокирован блокировкой для записи, возвращённый штамп будет равняться нулю. В любой момент можно проверить, является ли блокировка валидной с помощью `lock.validate(stamp)`. Для приведённого выше кода результат будет таким:

```
Optimistic Lock Valid: true
Write Lock acquired
Optimistic Lock Valid: false
Write done
Optimistic Lock Valid: false
```

Оптимистичная блокировка является валидной с того момента, как ей удалось захватить ресурс. В отличии от обычных блокировок для чтения, оптимистичная не запрещает другим потокам блокировать ресурс для записи. Что же происходит в коде выше? После захвата ресурса блокировка является валидной и оптимистичный поток отправляется спать. В это время другой поток блокирует ресурсы для записи, не дожидаясь окончания работы чтения. Начиная с этого момента, оптимистичная блокировка перестаёт быть валидной (даже после окончания записи).

Таким образом, при использовании оптимистичных блокировок вам нужно постоянно следить за их валидностью (проверять её нужно уже после того, как выполнены все необходимые операции).

Иногда может быть полезным преобразовать блокировку для чтения в блокировку для записи не высвобождая ресурсы. В `StampedLock` это можно сделать с помощью метода `tryConvertToWriteLock()`, как в этом примере:

```
1  ExecutorService executor = Executors.newFixedThreadPool(2);
2  StampedLock lock = new StampedLock();
3
4  executor.submit(() -> {
5      long stamp = lock.readLock();
6      try {
7          if (count == 0) {
8              stamp = lock.tryConvertToWriteLock(stamp);
9              if (stamp == 0L) {
10                 System.out.println("Could not convert to write lock");
11                 stamp = lock.writeLock();
12             }
13             count = 23;
14         }
15         System.out.println(count);
16     } finally {
17         lock.unlock(stamp);
18     }
19 });
20
21 stop(executor);
```

В этом примере мы хотим прочитать значение переменной `count` и вывести его в консоль. Однако, если значение равно нулю, мы хотим изменить его на 23. Для этого нужно выполнить преобразования из readLock во writeLock, чтобы не помешать другим потокам обрабатывать переменную. В случае, если вы вызвали `tryConvertToWriteLock()` в тот момент, когда ресурс занят для записи другим потоком, текущий поток остановлен не будет, однако метод вернёт нулевое значение. В таком случае можно вызвать `writeLock()` вручную.

Семафоры

Семафоры — отличный способ ограничить количество потоков, которые одновременно работают над одним и тем же ресурсом:

```
ExecutorService executor = Executors.newFixedThreadPool(10);

Semaphore semaphore = new Semaphore(5);

Runnable longRunningTask = () -> {
    boolean permit = false;
    try {
        permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
        if (permit) {
            System.out.println("Semaphore acquired");
            sleep(5);
        } else {
            System.out.println("Could not acquire semaphore");
        }
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    } finally {
        if (permit) {
            semaphore.release();
        }
    }
}

IntStream.range(0, 10)
    .forEach(i -> executor.submit(longRunningTask));

stop(executor);
```

В этом примере сервис-исполнитель может потенциально запустить все 10 вызываемых потоков, однако мы создали семафор, который ограничивает количество одновременно выполняемых потоков до пяти. Снова напомню, что важно освобождать ресурсы именно в блоке `finally{}` на случай выброса исключений. Для приведённого выше кода вывод будет следующим:

```
Semaphore acquired
Semaphore acquired
Semaphore acquired
Semaphore acquired
Semaphore acquired
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
```

Это была вторая часть серии статей про многопоточное программирование. Настоятельно рекомендую разобрать вышеприведенные примеры самостоятельно. Все они, как обычно, доступны на GitHub (https://github.com/winterbe/java8-tutorial). Можете смело форкать репозиторий и добавлять его в избранное.