Многопоточное программирование в Java 8. Часть третья. Атомарные переменные и конкурентные таблицы

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

Добро пожаловать в третью часть руководства по параллельному программированию в Java 8. В первой части (https://tproger.ru/translations/java8-concurrency-tutorial-1/) мы рассматривали, как выполнять код параллельно с помощью потоков, задач и сервисов исполнителей. Во второй (https://tproger.ru/translations/java8-concurrency-tutorial-2) разбирались с тем, как синхронизировать доступ к изменяемым объектам с помощью ключевого слова `synchronized`, блокировок и семафоров. Сегодня, в заключительной части, я расскажу о двух очень важных частях Concurrency API: об атомарных переменных и о конкурентных таблицах (*Concurrent Maps*).

AtomicInteger

Пакет `java.concurrent.atomic` содержит много полезных классов для выполнения атомарных операций. Операция называется атомарной тогда, когда её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни `synchronized`, как мы это делали в предыдущем уроке.

Внутри атомарные классы очень активно используют сравнение с обменом (https://ru.wikipedia.org/wiki/Сравнение_с_обменом) (*compare-and-swap*, *CAS*), атомарную инструкцию, которую поддерживает большинство современных процессоров. Эти инструкции работают гораздо быстрее, чем синхронизация с помощью блокировок. Поэтому, если вам просто нужно изменять одну переменную с помощью нескольких потоков, лучше выбирать атомарные классы.

Приведу несколько примеров с использованием `AtomicInteger`, одного из атомарных классов:

```
1  AtomicInteger atomicInt = new AtomicInteger(0);
2
3  ExecutorService executor = Executors.newFixedThreadPool(2);
4
5  IntStream.range(0, 1000)
6      .forEach(i -> executor.submit(atomicInt::incrementAndGet));
7
8  stop(executor);
9
10 System.out.println(atomicInt.get());    // => 1000
```

Как видите, использование `AtomicInteger` вместо обычного `Integer` позволило нам корректно увеличить число, распределив работу сразу по двум потокам. Мы можем не беспокоиться о безопасности, потому что `incrementAndGet()` является атомарной операцией.

Класс `AtomicInteger` поддерживает много разных атомарных операций. Метод `updateAndGet()` принимает в качестве аргумента лямбда-выражение и выполняет над числом заданные арифметические операции:

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.updateAndGet(n -> n + 2);
        executor.submit(task);
    });

stop(executor);

System.out.println(atomicInt.get());    // => 2000
```

Метод `accumulateAndGet()` принимает лямбда-выражения типа `IntBinaryOperator`. Вот как мы можем использовать его, чтобы просуммировать все числа от нуля до тысячи:

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.accumulateAndGet(i, (n, m) -> n + m);
        executor.submit(task);
    });

stop(executor);

System.out.println(atomicInt.get());    // => 499500
```

Среди других атомарных классов хочется упомянуть такие как AtomicBoolean (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html), AtomicLong (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html) и AtomicReference (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicReference.html).

LongAdder

Класс `LongAdder` может выступать в качестве альтернативы `AtomicLong` для последовательного сложения чисел.

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

IntStream.range(0, 1000)
    .forEach(i -> executor.submit(adder::increment));

stop(executor);

System.out.println(adder.sumThenReset());    // => 1000
```

Так же, как и у других атомарных чисел, у `LongAdder` есть методы `increment()` и `add()`. Но вместо того, чтобы складывать числа сразу, он просто хранит у себя набор слагаемых, чтобы уменьшить взаимодействие между потоками. Узнать результат можно с помощью вызова `sum()` или `sumThenReset()`. Этот класс используется в ситуациях, когда добавлять числа приходится гораздо чаще, чем запрашивать результат (часто это какие-то статистические исследование, например подсчёт количества запросов). Несложно догадаться, что, давая прирост в производительности, `LongAdder` требует гораздо большего количества памяти из-за того, что он хранит все слагаемые.

LongAccumulator

Класс `LongAccumulator` несколько расширяет возможности `LongAdder`. Вместо простого сложения он обрабатывает входящие значения с помощью лямбды типа `LongBinaryOperator`, которая передаётся при инициализации. Выглядит это так:

```
1  LongBinaryOperator op = (x, y) -> 2 * x + y;
2  LongAccumulator accumulator = new LongAccumulator(op, 1L);
3
4  ExecutorService executor = Executors.newFixedThreadPool(2);
5
6  IntStream.range(0, 10)
7      .forEach(i -> executor.submit(() -> accumulator.accumulate(i)));
8
9  stop(executor);
10
11 System.out.println(accumulator.getThenReset());      // => 2539
```

В этом примере при каждом вызове `accumulate()` значение аккумулятора увеличивается в два раза, и лишь затем суммируется с `i`. Так же, как и `LongAdder`, `LongAccumulator` хранит весь набор переданных значений в памяти.

прим. переводчика На самом деле, пример не совсем корректный; согласно документации (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAccumulator.html), LongAccumulator не гарантирует порядка выполнения операций. Корректной формулой была бы, например `x+2*y`, т.к. при любом порядке выполнения в конце будет получаться одно и то же значение.

ConcurrentMap

Интерфейс `ConcurrentMap` наследуется от обычного `Map` и предоставляет описание одной из самой полезной коллекции для конкурентного использования. Чтобы продемонстрировать новые методы интерфейса, мы будем использовать вот эту заготовку:

```
1  ConcurrentMap<String, String> map = new ConcurrentHashMap<>();
2  map.put("foo", "bar");
3  map.put("han", "solo");
4  map.put("r2", "d2");
5  map.put("c3", "p0");
```

Метод `forEach()` принимает лямбду типа `BiConsumer`. Этой лямбде будут передаваться в качестве аргументов все ключи и значения таблицы по очереди. Этот метод может использоваться как замена `for-each` циклам с итерацией по всем `Entry`. Итерация выполняется последовательно, в текущем потоке.

```
map.forEach((key, value) -> System.out.printf("%s = %s\n", key, value));
```

Метод `putIfAbsent()` помещает в таблицу значение, только если по данному ключу ещё нет другого значения. Этот метод является потокобезопасным (о крайней мере, в реализации `ConcurrentHashMap`), поэтому вам не нужно использовать `synchronized`, когда вы хотите использовать его в нескольких потоках (то же самое справедливо и для обычного `put()`):

```
String value = map.putIfAbsent("c3", "p1");
System.out.println(value);    // p0
```

Метод `getOrDefault()` работает так же, как и обычный `get()`, с той лишь разницей, что при отсутствии значения по данному ключу он вернёт значение по-умолчанию, передаваемое вторым аргументом:

```
String value = map.getOrDefault("hi", "there");
System.out.println(value);    // there
```

Метод `replaceAll()` принимает в качестве аргумента лямбда-выражение типа `BiFunction`. Этой лямбде по очереди передаются все комбинации ключ-значения из карты, а результат, который она возвращает, записывается соответствующему ключу в качестве значения:

```
map.replaceAll((key, value) -> "r2".equals(key) ? "d3" : value);
System.out.println(map.get("r2"));    // d3
```

Если же вам нужно изменить таким же образом только один ключ, это позволяет сделать метод `compute()`:

```
map.compute("foo", (key, value) -> value + value);
System.out.println(map.get("foo"));   // barbar
```

Кроме обычного `compute()`, существуют так же методы `computeIfAbsent()` и `computeIfPresent()`. Они изменяют значение только если значение по данному ключу отсутствует (или присутствует, соответственно).

И, наконец, метод `merge()`, который может быть использован для объединения существующего ключа с новым значением. В качестве аргумента он принимает ключ, новое значение и лямбду, которая определяет, как новое значение должно быть объединено со старым:

```
map.merge("foo", "boo", (oldVal, newVal) -> newVal + " was " + oldVal);
System.out.println(map.get("foo"));   // boo was bar
```

ConcurrentHashMap

Кроме методов, которые описаны в `ConcurrencyMap`, в `ConcurrentHashMap` было добавлено и ещё несколько своих. Так же, как и параллельные stream'ы, эти методы используют специальный `ForkJoinPool`, доступный через `ForkJoinPool.commonPool()` в Java 8. Этот пул использует свои настройки для количества потоков, основанные на количестве ядер. У меня их 4, а значит использоваться будет три потока:

```
System.out.println(ForkJoinPool.getCommonPoolParallelism());  // 3
```

Это значение может быть специально изменено с помощью параметра JVM:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

Мы рассмотрим три новых метода: `forEach`, `search` and `reduce`. У каждого из них есть первый аргумент, который называется `parallelismThreshold`, который определяет минимальное количество элементов в коллекции, при котором операция будет выполняться в нескольких потоках. Т.е. если в коллекции 499 элементов, а первый параметр выставлен равным пятистам, то операция будет выполняться в одном потоке последовательно. В наших примерах мы будем использовать первый параметр равным в единице, чтобы операции всегда выполнялись параллельно.

Для примеров ниже мы будем использовать всё ту же таблицу, что и выше (однако объявим её именем класса, а не интерфейса. чтобы нам были доступны все методы):

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
map.put("foo", "bar");
map.put("han", "solo");
map.put("r2", "d2");
map.put("c3", "p0");
```

ForEach

Работает метод так же, как и в `ConcurrentMap`. Для иллюстрации многопоточности мы будем выводить названия потоков (не забывайте, что их количество для меня ограничено тремя):

```
map.forEach(1, (key, value) ->
    System.out.printf("key: %s; value: %s; thread: %s\n",
        key, value, Thread.currentThread().getName()));

// key: r2; value: d2; thread: main
// key: foo; value: bar; thread: ForkJoinPool.commonPool-worker-1
// key: han; value: solo; thread: ForkJoinPool.commonPool-worker-2
// key: c3; value: p0; thread: main
```

Search

Метод `search()` принимает лямбда-выражение типа `BiFunction`, в которую передаются все пары ключ-значение по очереди. Функция должна возвращать `null`, если необходимое вхождение найдено. В случае, если функция вернёт не `null`, дальнейший поиск будет остановлен. Не забывайте, что данные в хэш-таблице хранятся неупорядоченно. Если вы будете полагаться на порядок, в котором вы добавляли данные в неё, вы можете не получить ожидаемого результата. Если условиям поиска удовлетворяют несколько вхождений, результат точно предсказать нельзя.

```
String result = map.search(1, (key, value) -> {
    System.out.println(Thread.currentThread().getName());
    if ("foo".equals(key)) {
        return value;
    }
    return null;
});
System.out.println("Result: " + result);

// ForkJoinPool.commonPool-worker-2
// main
// ForkJoinPool.commonPool-worker-3
// Result: bar
```

Или вот другой пример, который полагается только на значения:

```

```

```
String result = map.searchValues(1, value -> {
    System.out.println(Thread.currentThread().getName());
    if (value.length() > 3) {
        return value;
    }
    return null;
});

System.out.println("Result: " + result);

// ForkJoinPool.commonPool-worker-2
// main
// main
// ForkJoinPool.commonPool-worker-1
// Result: solo
```

Reduce

Метод `reduce()` вы могли уже встречать в Java 8 Streams. Он принимает две лямбды типа `BiFunction`. Первая функция преобразовывает пару ключ/значение в один объект (любого типа). Вторая функция совмещает все полученные значения в единый результат, игнорируя любые возможные `null`-значения.

```
tring result = map.reduce(1,
    (key, value) -> {
        System.out.println("Transform: " + Thread.currentThread().getName());
        return key + "=" + value;
    },
    (s1, s2) -> {
        System.out.println("Reduce: " + Thread.currentThread().getName());
        return s1 + ", " + s2;
    });

System.out.println("Result: " + result);

// Transform: ForkJoinPool.commonPool-worker-2
// Transform: main
// Transform: ForkJoinPool.commonPool-worker-3
// Reduce: ForkJoinPool.commonPool-worker-3
// Transform: main
// Reduce: main
// Reduce: main
// Result: r2=d2, c3=p0, han=solo, foo=bar
```