Атомарные операции в Java

JSP & Servlets
19 уровень, 1 лекция

ОТКРЫТА

Предпосылки появления атомарных операций

Давай разберем данный пример, который поможет понять работу атомарных операций:

```
1  public class Counter {
2      int count;
3
4      public void increment() {
5          count++;
6      }
7  }
```

Когда у нас один поток, все работает классно, но если мы добавляем многопоточку, то получаем неправильные результаты, а все из-за того, что операция инкремента составляет не одну операцию, а три: запрос на получение текущего значения `count`, потом увелечение ее на 1 и запись снова в `count`.

И когда два потока захотят увеличить переменную, скорее всего, ты потеряешь данные. То есть оба потока получают 100, в результате оба запишут 101 вместо ожидаемого значения 102.

И как же это решить? Нужно использовать блокировки. Ключевое слово **synchronized** помогает решить данную проблему, использование этого слова дает вам гарантию, что один поток будет обращаться к методу единовременно.

```
1  public class SynchronizedCounterWithLock {
2      private volatile int count;
3
4      public synchronized void increment() {
5          count++;
6      }
7  }
```

Плюс надо добавлять ключевое слово **volatile**, которое обеспечивает корректную видимость ссылок среди потоков. Мы разбирали его работу выше.

Но все же есть минусы. Самый большой — это производительность, в тот момент времени, когда много потоков пытаются получить блокировку и один получает возможность для записи, остальные потоки будут или заблокированы, или приостановлены до момента освобождения потока.

Все эти процессы, блокировка, переход в другой статус — очень дороги для производительности системы.

Атомарные операции

Алгоритм использует низкоуровневые машинные инструкции, такие как сравнение и замена (CAS, compare-and-swap, что обеспечивает целостность данных и по ним уже существует большое количество исследований).

Типичная операция CAS работает с тремя операндами:

- Место в памяти для работы (M)
- Существующее ожидаемое значение (A) переменной

- Новое значение (B), которое необходимо установить

CAS атомарно обновляет M до B, но только если значение M совпадает с A, в противном случае никаких действий предприниматься не будет.

В первом и втором случае вернут значение M. Это позволяет объединить три шага, а именно — получение значения, сравнение значения и его обновление. И это все превращается в одну операцию на машинном уровне.

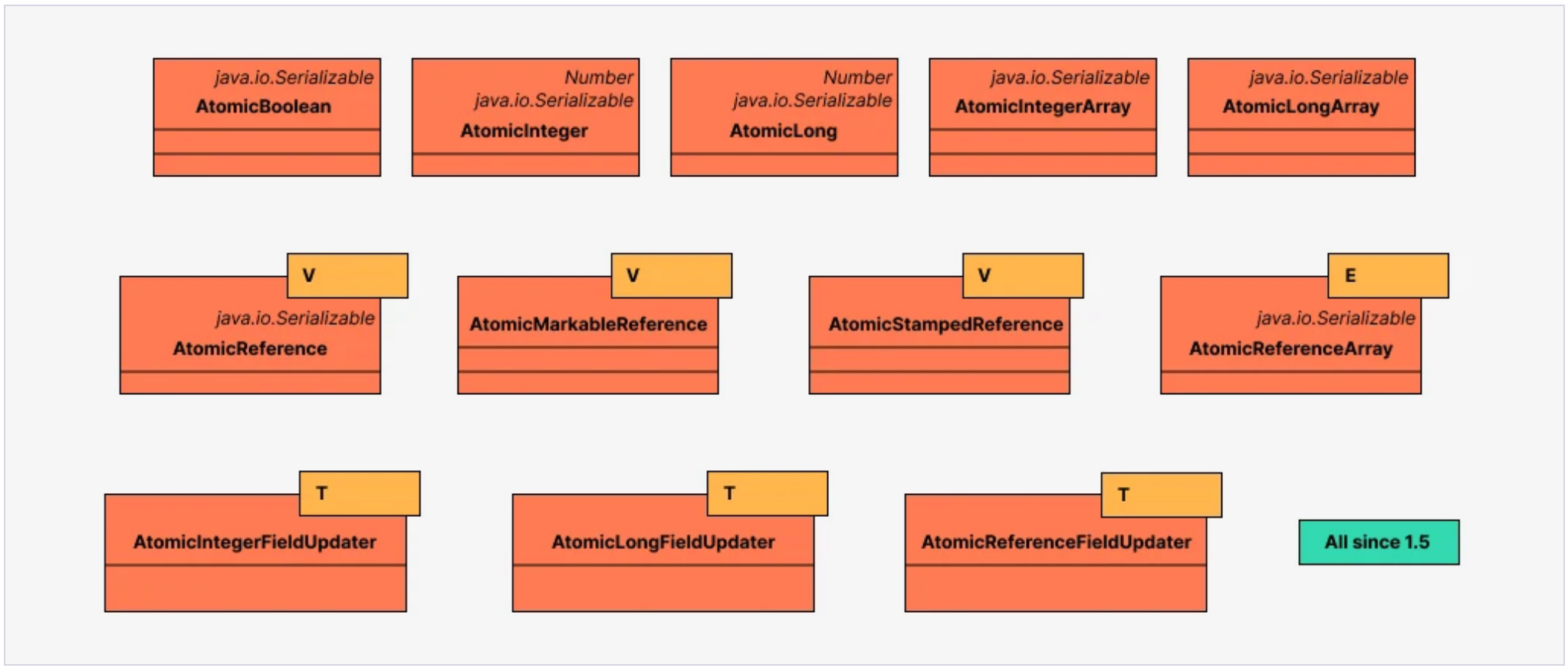
В тот момент времени, когда многопоточное приложение обращается к переменной и пытается обновить его и применяется CAS, то один из потоков получит его и сможет обновить его. Но в отличии от блокировок, другие потоки просто получат ошибки о том, что им не удалось обновить значение. Потом они перейдут к дальнейшей работе, а переключение полностью исключено при таком типе работе.

При этом логика становится труднее из-за того, что мы должны обработать ситуацию, когда операция CAS не отработала успешно. Мы просто смоделируем код таким образом, чтобы он не двигался дальше, пока операция не произойдет успешно.

Знакомство с атомарными типами

Ты столкнулся с ситуацией, когда тебе нужно настроить синхронизацию для самой простой переменной типа int?

Первый способ, который мы уже разобрали – это использование **volatile** + **synchronized**. Но есть еще специальные классы Atomic*.



Если у нас используется CAS, то операции работают быстрее по сравнению с первым способом. И в дополнение у нас есть специальные и очень удобные методы для добавления значения и операции инкремента и декремента.

`AtomicBoolean`, `AtomicInteger`, `AtomicLong`, `AtomicIntegerArray`, `AtomicLongArray` —классы в которых операции атомарны. Ниже мы разберем работу с ними.

AtomicInteger

Класс `AtomicInteger` предоставляет операции с значением int, которые могут быть прочитаны и записаны атомарно, в дополнение содержит расширенные атомарные операции.

У него есть методы `get` и `set`, которые работают, как чтение и запись по переменным.

То есть “происходит до (happens-before)” с любым последующим получением той же переменной, о которой мы говорили ранее. У атомарного метода `compareAndSet` также есть эти особенности согласованности памяти.

Все операции, которые возвращают новое значение, выполняются атомарно:

int addAndGet (int delta)	Добавляет определенное значение к текущему значению.

boolean compareAndSet (ожидаемое int, обновление int)	Устанавливает значение для данного обновленного значения, если текущее значение совпадает с ожидаемым значением.
int decmentAndGet ()	Уменьшает на единицу текущее значение.
int getAndAdd (int delta)	Добавляет данное значение к текущему значению.
int getAndDecrement ()	Уменьшает на единицу текущее значение.
int getAndIncrement ()	Увеличивает на единицу текущее значение.
int getAndSet (int newValue)	Устанавливает заданное значение и возвращает старое значение.
int incrementAndGet ()	Увеличивает на единицу текущее значение.
lazySet (int newValue)	В конце-концов устанавливается на заданное значение.
boolean weakCompareAndSet (ожидаемое, обновление int)	Устанавливает значение для данного обновленного значения, если текущее значение совпадает с ожидаемым значением.

Пример:

```
1  ExecutorService executor = Executors.newFixedThreadPool(5);
2  IntStream.range(0, 50).forEach(i -> executor.submit(atomicInteger::incrementAndGet));
3  executor.shutdown();
4  executor.awaitTermination(Long.MAX_VALUE, TimeUnit.HOURS);
5
6  System.out.println(atomicInteger.get()); // выведет 50
```

‹ Предыдущая лекция

Следующая лекция ›

Pavel Soros Уровень 34 — 6 октября 2022, 20:47

А вы последний пример запускали???
250 он не выведет никогда!!!
Вывод будет 50... и то если дать поспать главному потоку.

Ответить +5

Pugachev Ivan Уровень 35 — 4 июня 2022, 18:09

В строке

```
3   int decmentAndGet ()    Атомарно уменьшает на единицу текущее значение
```

опечатка (или недопечатка), должно быть

```
decrementAndGet()
```

Ответить +4

Mentor-02 Backend Developer в **JavaRush** MENTOR — 23 июня 2022, 10:00

Спасибо, исправлено

Ответить +2