**Atomics**

**Atomics -** классы с поддержкой атомарных операций над примитивами и ссылками.

Операция называется атомарной тогда, когда её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок (volatile для операций set/get), ни synchronized.

Внутри атомарные классы используют [сравнение с обменом](https://ru.wikipedia.org/wiki/%D0%A1%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81_%D0%BE%D0%B1%D0%BC%D0%B5%D0%BD%D0%BE%D0%BC) (compare-and-swap, CAS), атомарную инструкцию, которая реализована внутри самих процессоров(т.е. это низкоуровневый код). Поэтому эти инструкции работают гораздо быстрее, чем синхронизация с помощью блокировок.

Atomic чаще всего используются для операции замены или инкремент/декремент.

Если вам просто нужно изменять одну переменную с помощью нескольких потоков, лучше выбирать атомарные классы. Хотя написание кода с использованием Atomic более громоздко, чем с простой блокировкой. Т.к. тут чтобы сделать любое действие с переменной/объектом (запиханым в объект класс Atomic) нужно каждый раз вызывать какой-то специальный метод:

***compareAndSet(oldValue, newValue)*** – для того, чтобы изменить значение переменной на новое;

***getAndDecrement()/getAndIncrement()***;

***get()*** – получить текущее значение;

***getAndSet(newValue)*** – без проверки устанавливает новое значение;

***addAndGet(Value)*** – атомарно добавляет данное значение к текущему.

Классы ***LongAdder*** и ***LongAccumulator*** используются вместо Atomic когда обновление переменной из нескольких потоков является более частым, чем чтение. Эти классы один раз берут переменную, операции по многократному добавлению, например 1, и потом все что должно быть добавлено для получения конечного результата хранят внутри себя. А для получения итогового значения вызываются методы ***sum(), sumThenReset(), getThenReset()***. Но как следствие они потребляют больше памяти.

**class** **synchronized** Counter

{

**private** **int** c = 0;

**public** **synchronized** **void** increment()

{

c++;

}

**public** **synchronized** **void** decrement()

{

c--;

}

**public** **synchronized** **int** value()

{

**return** c;

}

}

— Отличная работа. А вот, как бы он выглядел с использованием атомарных типов:

**class** **AtomicCounter**

{

**private** **AtomicInteger** c = **new** **AtomicInteger**(0);

**public** **void** **increment**()

{

c.**incrementAndGet**();

}

**public** **void** **decrement**()

{

c.**decrementAndGet**();

}

**public** **int** **value**()

{

**return** c.**get**();

}

}

Потокобезопасный счетчик, реализованный с помощью класса AtomicInteger  из пакета java.concurrent.atomic, показан в следующем примере:

public class SafeCounterWithoutLock {

private final AtomicInteger counter = new AtomicInteger(0);

public int getValue() {

return counter.get();

}

public void increment() {

while(true) {

int existingValue = getValue();

int newValue = existingValue + 1;

if(counter.compareAndSet(existingValue, newValue)) {

return;

}

}

}

}

Как видите, мы повторяем операцию compareAndSet и снова при неудаче, так как мы хотим гарантировать, что вызов метода increment всегда увеличивает значение на 1.

AtomicInteger

Пакет java.concurrent.atomic содержит много полезных классов для выполнения атомарных операций.

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

AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)

.forEach(i -> executor.submit(atomicInt::incrementAndGet));

stop(executor);

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" \t "_blank), [AtomicLong](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html" \t "_blank) и [AtomicReference](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicReference.html" \t "_blank).

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:

LongBinaryOperator op = (x, y) -> 2 \* x + y;

LongAccumulator accumulator = new LongAccumulator(op, 1L);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10)

.forEach(i -> executor.submit(() -> accumulator.accumulate(i)));

stop(executor);

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.



