JAVA

Paralelizácia

Concurrency API

- JAVA 5
- 2004
- java.util.concurrent

Threads and Runnables

- Procesy bežia paralelne vedľa ostatných programov
- Využitie vlákien v rámci procesov na súbežné vykonávanie kódu
- Pred použitím vlákna je potrebné definovať úlohu pomocou rozhrania Runnable

```java
Runnable task = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
};

task.run();

Thread thread = new Thread(task);
thread.start();

System.out.println("Done!");
```

Executors

- Predstavený ExecutorService ako náhrada za priamu prácu s vláknami
- Schopné spúšťať asynchrónne úlohy
- Musia byť zastavené explicitne – shutdown() / shutdownNow()

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});

// => Hello pool-1-thread-1
```

Callables and Futures

- Callables sú rovnaké rozhrania ako Runnables, narozdiel od nich ale vracajú hodnotu
- Future je špeciálny typ ktorý poskytne výsledok neskôr

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

Timeouts

- Slúži na zabránenie prechodu aplikácie do neresponzívneho stavu

InvokeAll / InvokeAny

- InvokeAll – zoberie kolekciu Callables a vráti list obsahujúci Futures
- InvokeAny – blokuje pokiaľ neterminuje prvé Callable a následne vráti jeho hodnotu

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");

executor.invokeAll(callables)
    .stream()
    .map(future -> {
        try {
            return future.get();
        }
        catch (Exception e) {
            throw new IllegalStateException(e);
        }
    })
    .forEach(System.out::println);
```

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
    callable("task1", 2),
    callable("task2", 1),
    callable("task3", 3));

String result = executor.invokeAny(callables);
System.out.println(result);

// => task2
```

Scheduled Executors

- ScheduledExecutorService je schopný naplánovať úlohy s periodickým opakovaním

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());
ScheduledFuture<?> future = executor.schedule(task, 3, TimeUnit.SECONDS);

TimeUnit.MILLISECONDS.sleep(1337);

long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
System.out.printf("Remaining Delay: %sms", remainingDelay);
```

Synchronized

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

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

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

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

stop(executor);

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

Locks

- ReentrantLock - rovnaký ako keyword synchronized ale s rozšírenými možnosťami
- ReadWriteLock - zachováva dvojicu "lock-ov", na čítanie a zápis
- StampedLock - podobný ako ReadWriteLock, vracia "pečať" reprezentovanú hodnotou long

Semaphores

- Schopné udržiavať celé sety povolení
- Užitočné pri limitovaní počtu prístupov ku konkrétnym častiam aplikácie

AtomicInteger

- Balík java.concurrent.atomic - triedy pre atomické operácie
- Oveľa rýchlejšie ako synchronizácia

```
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
```

LongAdder

- Trieda môže byť použitá na postupné pridávanie hodnôt k číslu

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

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

stop(executor);

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

LongAccumulator

- Generalizovaná verzia LongAdder

```
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
```

ConcurrentMap / ConcurrentHashMap

- Tieto rozhrania rozširujú rozhranie map
- Jedny z najužitočnejších typov kolekcií
- Funkcionálne programovanie v JAVA 8 bolo predstavené pridaním funkcií do týchto rozhraní
- ConcurrentHashMap navyše obsahuje metódy pre paralelizáciu na mapách

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

ForEach / Search

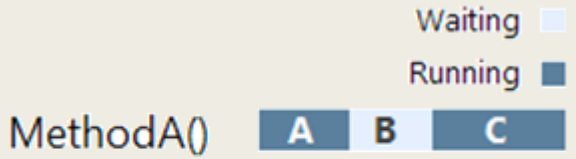
```
map.forEach(1, (key, value) ->
    System.out.printf("key: %s; value: %s; thread: %s\n",
        key, value, Thread.currentThread().getName()));

// 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
```

```
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);
```

Násobenie matíc

- Metódy merania :
  - *Real time meranie*
  - *Thread time meranie*



Real time = A + B + C
Thread time = A + C

Metóda využívajúca paralelne programovanie

```
try {
    for (row = 0; row < 3; row++) {
        for (col = 0; col < 3; col++) {
            // Creating thread for mul matrix
            thrd[threadCount] = new Thread(new WorkerTh(row, col, matrixTwo, matrixOne, result));
            thrd[threadCount].start(); //thread start

            thrd[threadCount].join(); // joining threads
            threadCount++;
        }
    }

} catch (InterruptedException ie) {
    //This should never happens
    log.println("RUN TIME EXCEPTION SHOULD NEVER BECOME");
    System.out.println("RUN TIME EXCEPTION SHOULD NEVER BECOME");
```

Výsledky merania

Sekvenčne	Paralelne
850266 ns	5154481 ns
0.850266 ms	5.154481 ms

Parallel Merge Sort

```java
class ParallelMergeSort extends RecursiveAction {
    // Decides when to fork or compute directly:
    private static final int SORT_THRESHOLD = 128;

    private final int[] values;
    private final int from;
    private final int to;

    public ParallelMergeSort(int[] values) { this(values, from: 0, to: values.length - 1); }

    public ParallelMergeSort(int[] values, int from, int to) {
        this.values = values;
        this.from = from;
        this.to = to;
    }

    public void sort() { compute(); }

    @Override
    protected void compute() {
        if (from < to) {
            int size = to - from;
            if (size < SORT_THRESHOLD) {
                insertionSort();
            } else {
                int mid = from + Math.floorDiv(size, 2);
                invokeAll(
                        new ParallelMergeSort(values, from, mid),
                        new ParallelMergeSort(values, from: mid + 1, to));
                merge(mid);
            }
        }
    }
```

```java
private void insertionSort() {
    for (int i = from + 1; i <= to; ++i) {
        int current = values[i];
        int j = i - 1;
        while (from <= j && current < values[j]) {
            values[j + 1] = values[j--];
        }
        values[j + 1] = current;
    }
}

private void merge(int mid) {
    int[] left = Arrays.copyOfRange(values, from, to: mid + 1);
    int[] right = Arrays.copyOfRange(values, from: mid + 1, to: to + 1);
    int f = from;

    int li = 0, ri = 0;
    while (li < left.length && ri < right.length) {
        if (left[li] <= right[ri]) {
            values[f++] = left[li++];
        } else {
            values[f++] = right[ri++];
        }
    }

    while (li < left.length) {
        values[f++] = left[li++];
    }

    while (ri < right.length) {
        values[f++] = right[ri++];
    }
}
```

```
Parallel Merge Sort done in: 4126

Process finished with exit code 0
```

Najvačší spoločný deliteľ (Paralelná)

```java
class EuclidParallel {
    public static void main(String args[]) {
        long first = 1200000000001;
        long second = 750000000001;
        long startTime = System.nanoTime();
        new MyThread( thread: "PRIO-10",  prio: 10, first, second, startTime);
        new MyThread( thread: "PRIO-9",  prio: 9, first, second, startTime);
        new MyThread( thread: "PRIO-8",  prio: 8, first, second, startTime);
        new MyThread( thread: "PRIO-7",  prio: 7, first, second, startTime);
        new MyThread( thread: "PRIO-6",  prio: 6, first, second, startTime);
        new MyThread( thread: "PRIO-5",  prio: 5, first, second, startTime);
        new MyThread( thread: "PRIO-4",  prio: 4, first, second, startTime);
        new MyThread( thread: "PRIO-3",  prio: 3, first, second, startTime);
        new MyThread( thread: "PRIO-2",  prio: 2, first, second, startTime);
        new MyThread( thread: "PRIO-1",  prio: 1, first, second, startTime);
//        new MyThread("PRIO-9",9,2312515,215651511);
//        new MyThread("PRIO-8",8,1565165,78515852);
//        new MyThread("PRIO-7",7,1281218,1883154);
//        new MyThread("PRIO-6",6,2119832,1588);
//        new MyThread("PRIO-5",5,516588,12188);
//        new MyThread("PRIO-4",4,1891161,651651);
//        new MyThread("PRIO-3",3,5165165,1651656);
//        new MyThread("PRIO-2",2,1065050,54165);
//        new MyThread("PRIO-1",1,566532,516516);

        try {
            Thread.sleep( millis: 100);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

```java
class MyThread implements Runnable {
    String name;
    Thread t;
    long first;
    long second;
    long result;
    long startTime;

    //Thread Constructor setting name, new creating new thread , setting priority value 10 IS MAX and 1 is MIN
    MyThread(String thread, int prio, long firstNum, long secondNum, long startTime) {
        name = thread;
        t = new Thread( target: this, name);
//        System.out.println("Created new thread: " + t);
        t.setPriority(prio);
        first = firstNum;
        second = secondNum;
        this.startTime = startTime;
        t.start();
    }

    public void run() {
        try {
            result = gcd(second,  q: first % second);
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
        }
//        System.out.println("gcd for numbers " + first + " and " + second + " is = " + result);
        long stopTime = System.nanoTime();
        System.out.println("Execution time of thread " + name + " (nanosecond): " + (stopTime - startTime));
    }

    public static long gcd(long p, long q) {
        if (q == 0) return p;
        else {
            return gcd(q,  q: p % q);
        }
    }
}
```

```
Main thread exiting.
Execution time of thread PRIO-8 (nanosecond): 1001673705
Execution time of thread PRIO-6 (nanosecond): 1001673961
Execution time of thread PRIO-4 (nanosecond): 1001673705
Execution time of thread PRIO-3 (nanosecond): 1001682903
Execution time of thread PRIO-1 (nanosecond): 1001686480
Execution time of thread PRIO-2 (nanosecond): 1001690057
Execution time of thread PRIO-7 (nanosecond): 1001673961
Execution time of thread PRIO-10 (nanosecond): 1001673961
Execution time of thread PRIO-5 (nanosecond): 1001682903
Execution time of thread PRIO-9 (nanosecond): 1001682903

Process finished with exit code 0
```

Najvačší spoločný deliteľ (Sekvenčná)

```java
public class EuclidGCD {

    // recursive implementation of GCD Euclid algo
    public static long gcd(long p, long q) {
        if (q == 0) return p;
        else return gcd(q,  q: p % q);
    }


    public static void main(String[] args) {
        long startTime = System.nanoTime();
        long p = 1200000000001;
        long q = 750000000001;
        long d = gcd(p, q);
        System.out.println("gcd for numbers " + p + " and " + q + " is = " + d);
        long stopTime = System.nanoTime();
        System.out.println();
        System.out.println("Execution time (nanosecond): " + (stopTime - startTime));
    }
}
```

```
gcd for numbers 120000000000 and 75000000000 is = 15000000000

Execution time (nanosecond): 347720

Process finished with exit code 0
```

Zdroje

- https://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/
- https://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/
- https://winterbe.com/posts/2015/05/22/java8-concurrency-tutorial-atomic-concurrent-map-examples/