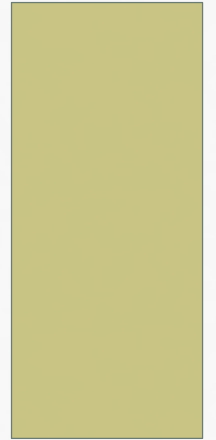


Java 8



Fork/Join

Fork/Join. Препосылки

Разделяй и властвуй - "Divide and conquer"

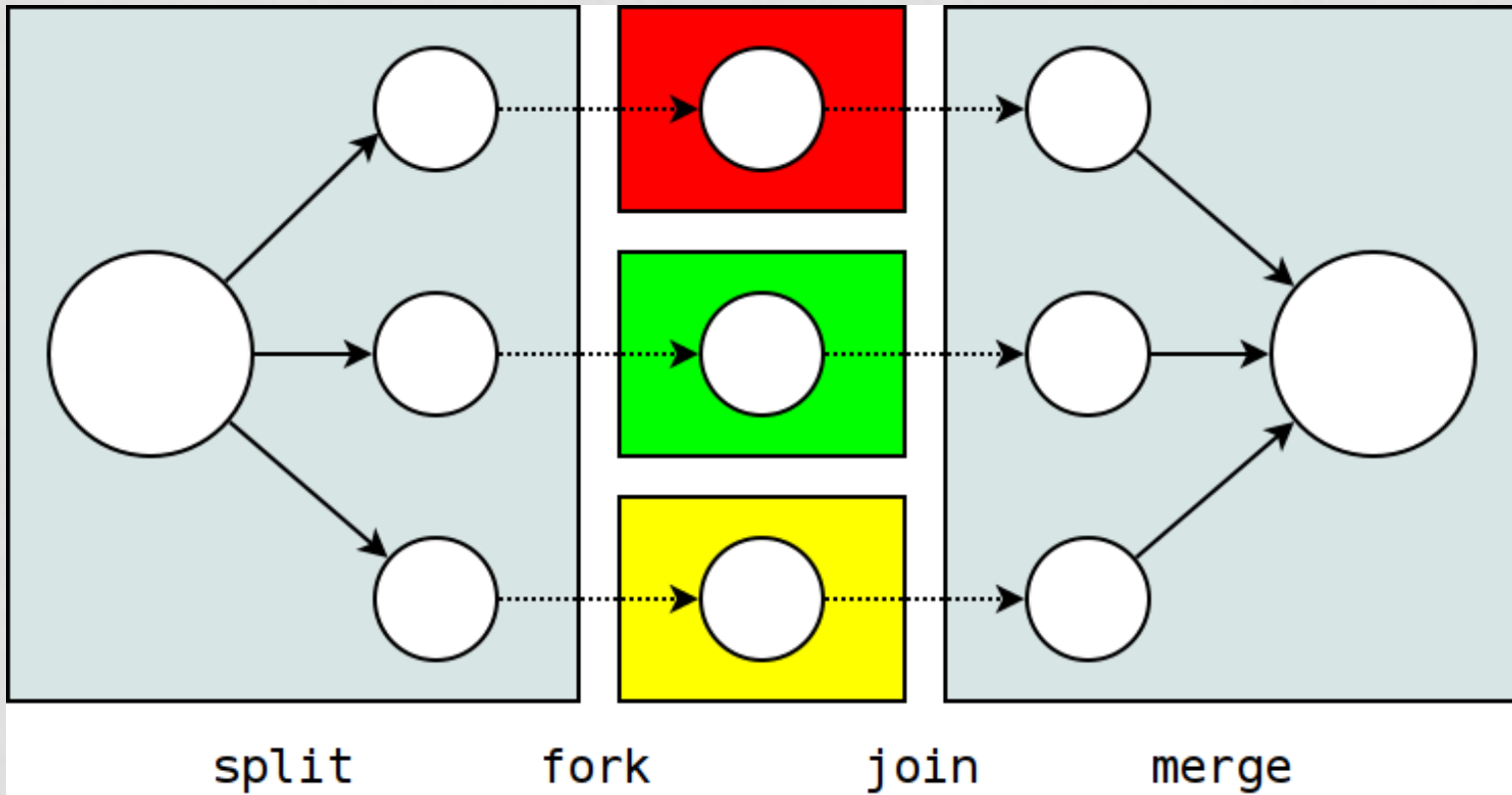
Идея: можно разбить большую задачу на подзадачи

- подзадачи можно будет выполнить одновременно
- часто задачи разбиваются естественным образом

Хорошо, если подзадачи независимы

- Иначе требуется коммуникация между подзадачами
- ...а это требует нетривиального планирования!

Fork/Join. Препосылки



Fork/Join. Препосылки

task1 и task2 – независимые подзадачи

- solveDirectly() выполняет задачу последовательно
- invokeAll() запускает подзадачи и ждёт их завершения
- split() разделяет задачу на две примерно равные
- merge()

```
Result solve(Problem problem)
{
    if (problem is small) {
        directly solve problem
    } else {
        split problem into independent parts
        fork new subtasks to solve each part
        join all subtasks
        compose result from subresults
    }
}
```

as proposed by Doug Lea

Fork/Join

- Проблемы – обычный пример с деревом.
- Решение - `join()` не должен занимать поток!
- В момент простоя можно заниматься полезной работой

ForkJoinTask

- Это абстрактный класс, который является в каком-то смысле легковесным аналогом потока.
- Суть в том, что благодаря ForkJoinPool, который будет рассмотрен позже, можно в небольшом количестве потоков выполнить существенно большее число задач.
- Это достигается путём так называемого work-stealing'a, когда спящая задача на самом деле не спит, а выполняет другие задачи.

RecursiveAction и RecursiveTask

ForkJoinTask имеет готовые реализации:

- RecursiveAction на случай, если никакого значения посчитать не нужно, а нужно лишь выполнить некоторое действие,
- RecursiveTask, когда нужно что-то вернуть.

Эти два класса аналогичны уже существующим Runnable и Callable.

ForkJoinPool

- В этом классе как раз и реализована логика по распределению нагрузки между реальными потоками.
- Снаружи он выглядит как обычный пул потоков, и особенностей в использовании нет.

```
void execute(ForkJoinTask<?> task)
```

```
T invoke(ForkJoinTask<T> task)
```

Балансировка задач

Хорошо решается только в динамике

- Задачи могут сильно отличаться по размеру
- Процессоры могут быть неравномерно заняты, etc.

Три базовых подхода:

- Арбитраж задач
 - Общая очередь задач
 - Общая очередь задач с маленькими thread-локальными буферами
- Work dealing
 - У каждого потока своя очередь
 - Перегруженные потоки отдают свои задачи на сторону
- Work stealing
 - У каждого потока своя очередь
 - Свободные потоки крадут задачи у перегруженных

Балансировка задач

Подход FJP – work stealing:

- Локальные очереди для каждого потока
- С головой очереди может работать только владелец
 - Это автоматически превращает очередь в стек (LIFO)
 - Даже без синхронизации
- Из хвоста могут брать задачи другие потоки
 - Если известно, что очередь не пуста, то можно не синхронизироваться
- Из хвоста же может брать и владелец
 - Асинхронный режим
 - Тогда очередь становится действительно очередью (FIFO)

Балансировка задач

Куда происходит `submit()` внешних задач?

- Загвоздка:
 - В голову очереди потока нельзя: требуется синхронизация
 - В хвост тоже особенно нельзя: порушим FIFO/LIFO
- Решение: отдельная очередь для внешних задач

Балансировка задач

Куда происходит `submit()` внешних задач?

- Загвоздка:
 - В голову очереди потока нельзя: требуется синхронизация
 - В хвост тоже особенно нельзя: порушим FIFO/LIFO
- Решение: отдельная очередь для внешних задач
- Решение №2: давайте расклеим очередь!
 - Каждому потоку по `submission queue`!
 - Клиенты случайным образом мультиплексируются на эти очереди

Fork/Join



ParallelStream in action

Parallel Streams

```
List<Integer> numbers = new ArrayList<>();  
for (int i = 0; i < 10_000_000; i++) {  
    numbers.add((int) Math.round(Math.random() * 100));  
}
```

Будем оставлять только четные числа и сортировать.

Parallel Streams

```
for (int i = 0; i < 100; i++) {  
    long start = System.currentTimeMillis();  
    List<Integer> even = numbers.stream()  
        .filter(n -> n % 2 == 0).sorted()  
        .collect(Collectors.toList());  
    System.out.printf(  
        "%d elements computed in %5d msec with %d  
threads\n",  
        even.size(), System.currentTimeMillis() - start,  
        Thread.activeCount());  
}
```

4999022 elements computed in 9793 msec with 1 threads

4999022 elements computed in 913 msec with 1 threads

4999022 elements computed in 4618 msec with 1 threads

4999022 elements computed in 877 msec with 1 threads

4999022 elements computed in 5335 msec with 1 threads

Parallel Streams

```
for (int i = 0; i < 100; i++) {  
    long start = System.currentTimeMillis();  
    List<Integer> even = numbers.parallelStream()  
        .filter(n -> n % 2 == 0).sorted()  
        .collect(Collectors.toList());  
    System.out.printf(  
        "%d elements computed in %5d msec with %d  
threads\n",  
        even.size(), System.currentTimeMillis() - start,  
        Thread.activeCount());  
}
```

5000749 elements computed in 663 msec with 4 threads

5000749 elements computed in 450 msec with 4 threads

5000749 elements computed in 715 msec with 4 threads

5000749 elements computed in 466 msec with 4 threads

Узнаем точнее про работающие потоки

```
Set<String> workerThreadNames = new Concurrent...Set<>();
for (int i = 0; i < 10; i++) {
    long start = System.currentTimeMillis();
    List<Integer> even = numbers.stream()
        .filter(n -> n % 2 == 0)
        .peek(n -> workerThreadNames.add(
            Thread.currentThread().getName()))
        .sorted()
        .collect(Collectors.toList());
    System.out.printf(
        "%d elements computed in %5d msec with %d threads\n",
        even.size(), System.currentTimeMillis() - start,
        workerThreadNames.size());
}
```

WTF

WTF???



-Djava.util.concurrent.ForkJoinPool.common.parallelism=4

```
for (int i = 0; i < 10; i++) {  
    long start = System.currentTimeMillis();  
    List<Integer> even = numbers.stream()  
        .filter(n -> n % 2 == 0)  
        .peek(n -> workerThreadNames.add(  
            Thread.currentThread().getName()))  
        .sorted()  
        .collect(Collectors.toList());  
    System.out.printf(  
        "%d elements computed in %5d msec with %d threads\n",  
        even.size(), System.currentTimeMillis() - start,  
        workerThreadNames.size());  
}
```

WTF???



5000651 elements computed in 6123 msec with 5 threads



WTF again

WTF???



-Djava.util.concurrent.ForkJoinPool.common.parallelism=4

```
for (int i = 0; i < 10; i++) {
    long start = System.currentTimeMillis();
    List<Integer> even = numbers.stream()
        .filter(n -> n % 2 == 0)
        .peek(n -> workerThreadNames.add(
            Thread.currentThread().getName()))
        .sorted()
        .collect(Collectors.toList());
    System.out.printf(
        "%d elements computed in %5d msec with %d threads\n",
        even.size(), System.currentTimeMillis() - start,
        workerThreadNames.size());
    System.out.println(workerThreadNames.toString()); WTF???
}
```

- 5002528 elements computed in 563 msec with 5 threads
 - [ForkJoinPool.commonPool-worker-0, ForkJoinPool.commonPool-worker-1, ForkJoinPool.commonPool-worker-2, ForkJoinPool.commonPool-worker-3, **main**]
- 
- 

WTF???

Причина

- **ВСЕ** `ParallelStreams` используют общий `ForkJoinPool`
- Поток работающий с `ParallelStream` также используется в качестве `Worker`

Следствия:

- Работа с `ParallelStream` происходит синхронно для вызывающего потока
- Другие потоки и задачи, работающие с общим `ForkJoinPool` могут проседать в производительности

Грязный Гарри Хак

- `ParallelStream` можно заставить насильно использовать нужный нам `ForkJoinPool`

```
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
long start = System.currentTimeMillis();
ForkJoinTask<List<Integer>> task =
    forkJoinPool.submit(() -> {
        return numbers.parallelStream()
            .filter(n -> n % 2 == 0)
            .sorted()
            .collect(Collectors.toList());
    });
List<Integer> even = task.get();
```

Грязный Гарри Хак

Такой подход

- Не затрагивает другие `ParallelStreams`
- Не затрагивает других пользователей общего `ForkJoinPool`
- Уменьшает непредсказуемую задержку из-за нагрузки общего `ForkJoinPool` другими потоками
- Поток, вызывающий задачу не задействован как рабочий (асинхронный вызов задачи)

Проблемы общего ForkJoinPool

Blocking for IO

- Если URL зависнут на `ConnectionTimeOut`, то общая производительность сильно пострадает

```
Stream<String> urls =  
Files.Lines(Paths.get("urlsToCheck.txt"));  
  
List<String> errors = urls.parallel().filter(url -> {  
    //Connect to URL and wait for 200 response or timeout  
    return true;  
}).collect(toList());
```

Вложенные `parallelStream`

```
long start = System.currentTimeMillis();
IntStream.range(0, 10_000).parallel()
    .forEach(i -> {
        results[i][0] = (int)Math.round(Math.random()*100);
        IntStream.range(1, 9_999)
            .parallel().forEach((int j) ->
                results[i][j] =
                    (int)Math.round(Math.random()*1000));
    });
```

Process finalized in 22974 msec

Process finalized in 22575 msec

Вложенные `parallelStream`

```
long start = System.currentTimeMillis();
IntStream.range(0, 10_000).parallel()
    .forEach(i -> {
        results[i][0] = (int)Math.round(Math.random()*100);
        IntStream.range(1, 9_999)
            .sequential().forEach((int j) ->
                results[i][j] =
                    (int)Math.round(Math.random()*1000));
    });
```

Process finalized in 12491 msec

Process finalized in 12589 msec

Другие проблемы
производительности

Auto(un)boxing

- Boxing и unboxing в КАЖДОМ ВЫЗОВЕ filter

```
List<Integer> even = numbers.parallelStream()  
.filter(n -> n % 2 == 0)  
.sorted()  
.collect(Collectors.toList());
```

4999464 elements computed in 290 msec with 8 threads

4999464 elements computed in 276 msec with 8 threads

4999464 elements computed in 257 msec with 8 threads

4999464 elements computed in 265 msec with 8 threads

Auto(un)boxing

- Boxing и unboxing в КАЖДОМ ВЫЗОВЕ filter

```
List<Integer> even = numbers.parallelStream()  
.mapToInt(n -> n)  
.filter(n -> n % 2 == 0)  
.sorted()  
.boxed()  
.collect(Collectors.toList());
```

4999460 elements computed in 160 msec with 8 threads

4999460 elements computed in 243 msec with 8 threads

4999460 elements computed in 144 msec with 8 threads

4999460 elements computed in 140 msec with 8 threads