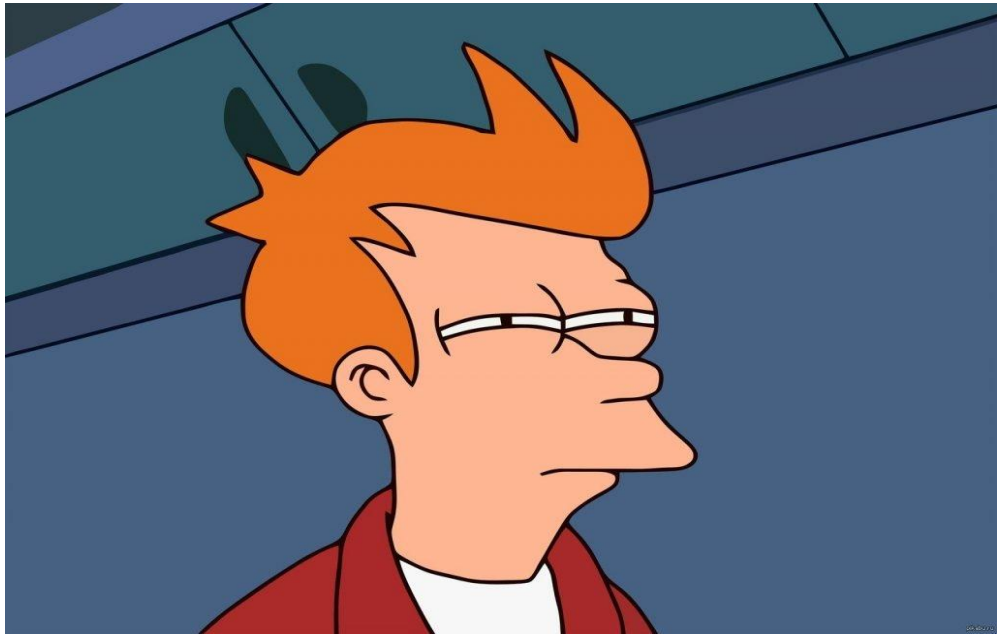
От Fork/Join к Stream API

- Лыткин Роман
- Javarules.ru
- https://habr.com/users/terran37/
- roman.lytkin@gmail.com

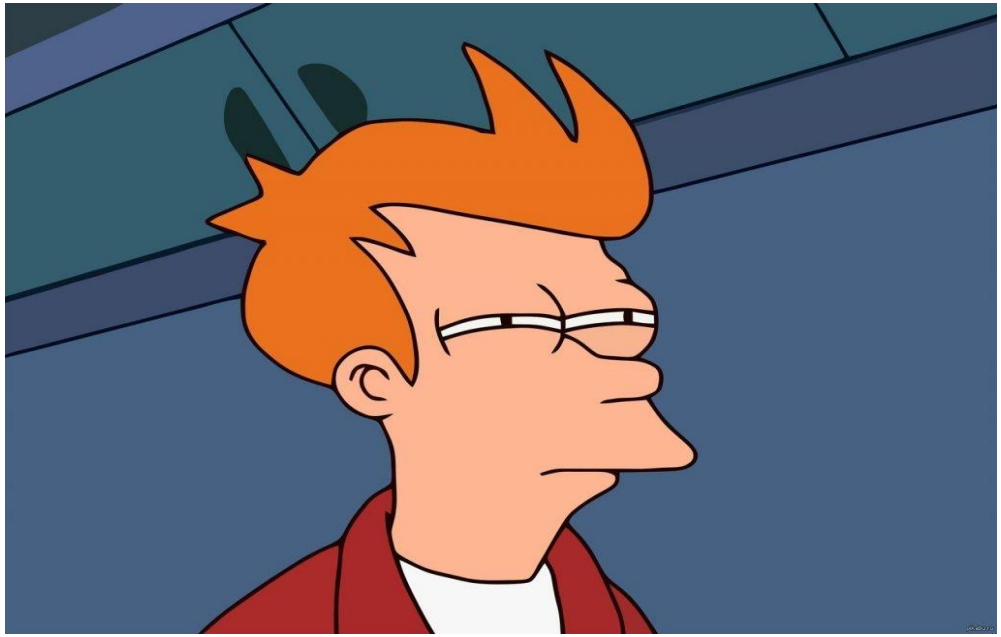
Этап размышления и поиск возможного пути оптимизации расчета (1)
Исходные данные

1. Таблицы с данными



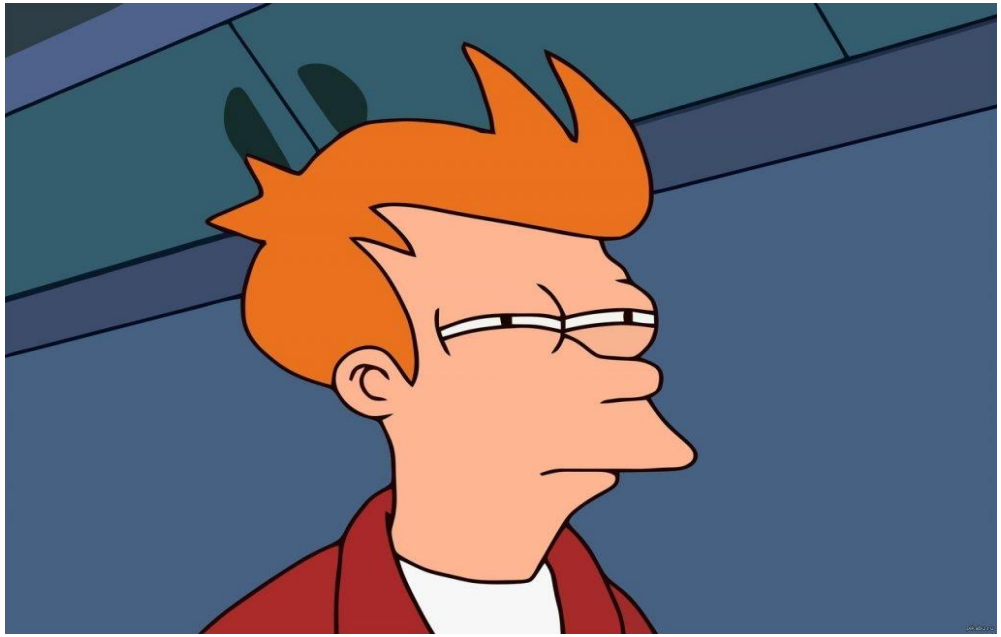
Этап размышления и поиск возможного пути оптимизации расчета (1)
Исходные данные

1. Таблицы с данными
2. Сущность «договор», по которой необходимо собрать, обработать данные из таблиц пункта №1 и записать данные в новые таблицы («витрины»)



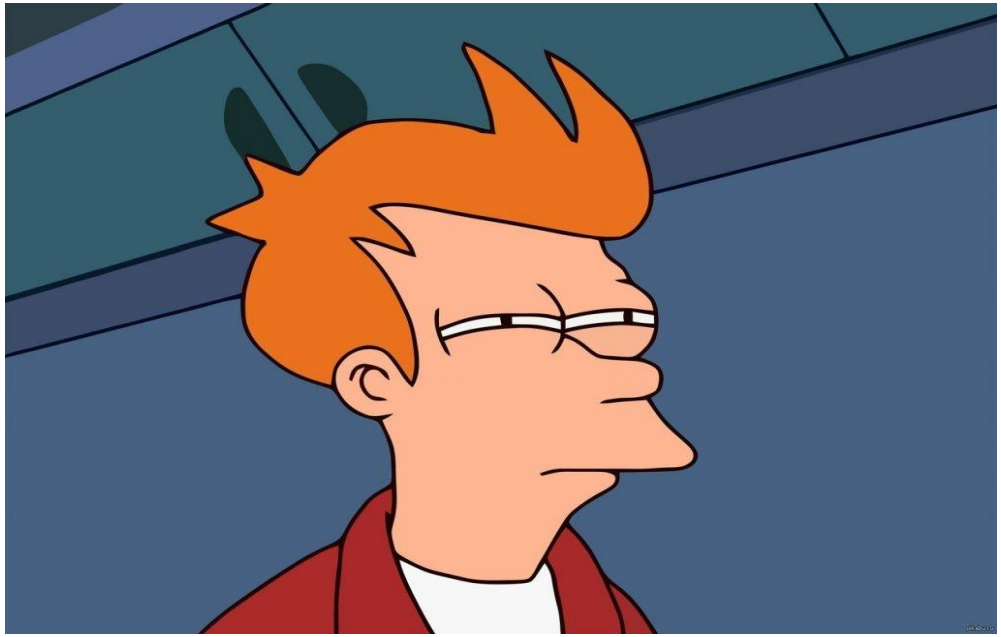
Этап размышления и поиск возможного пути оптимизации расчета (1)
Исходные данные

1. Таблицы с данными
2. Сущность «договор», по которой необходимо собрать, обработать данные из таблиц пункта №1 и записать данные в новые таблицы («витрины»)
3. Необходимо произвести расчеты, не используя уникальных возможностей баз данных



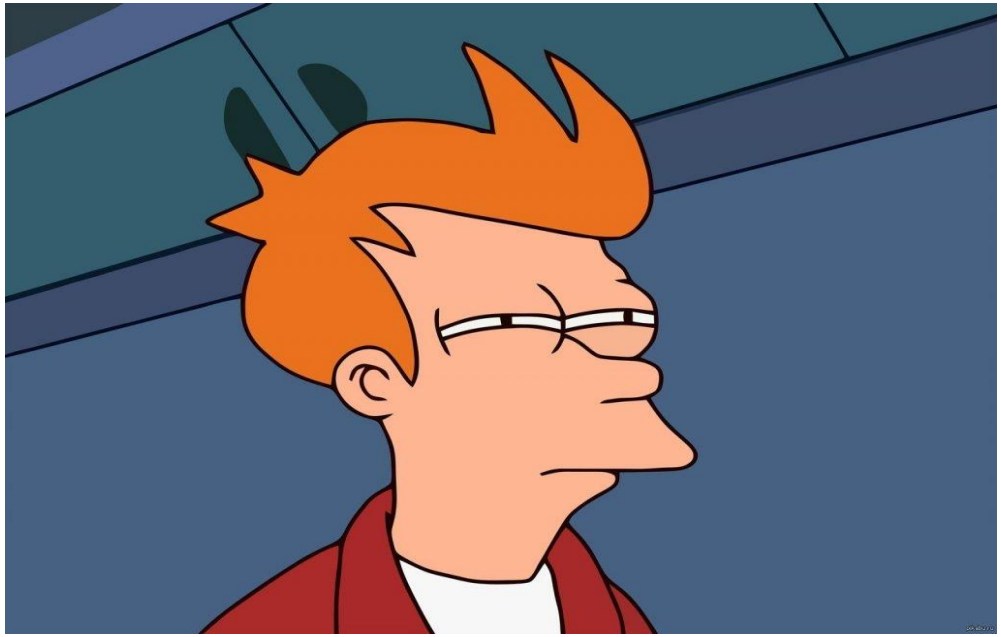
Этап размышления и поиск возможного пути оптимизации расчета (2) Решения на поверхности

1. На основе исходных данных создать «подготовленные» промежуточные таблицы



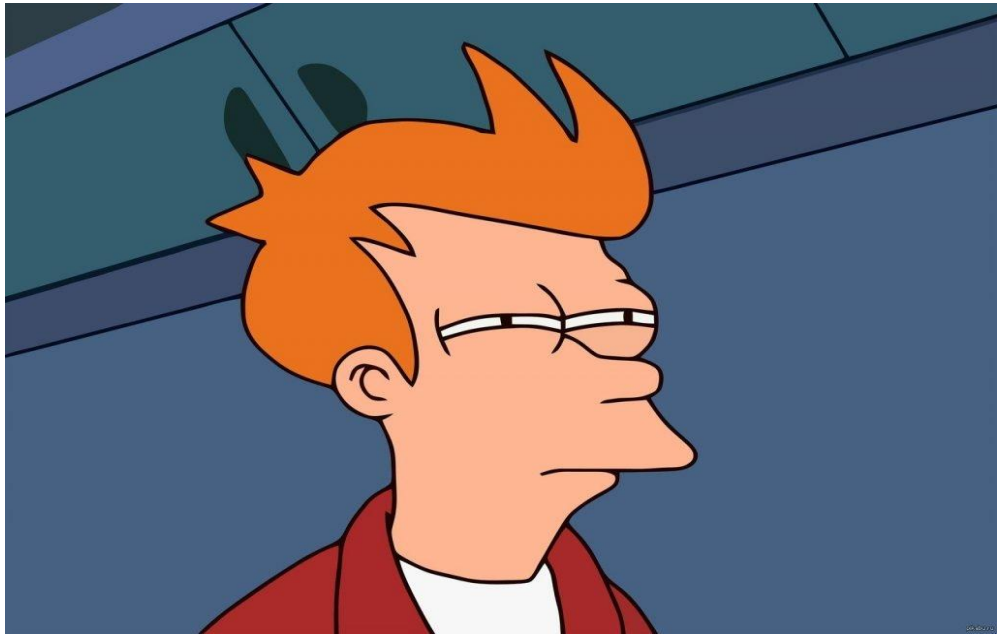
Этап размышления и поиск возможного пути оптимизации расчета (2)
Решения на поверхности

1. На основе исходных данных создать «подготовленные» промежуточные таблицы
2. Создать больше индексов, провести оптимизацию хранения, создать механизмы сжатия устаревших данных



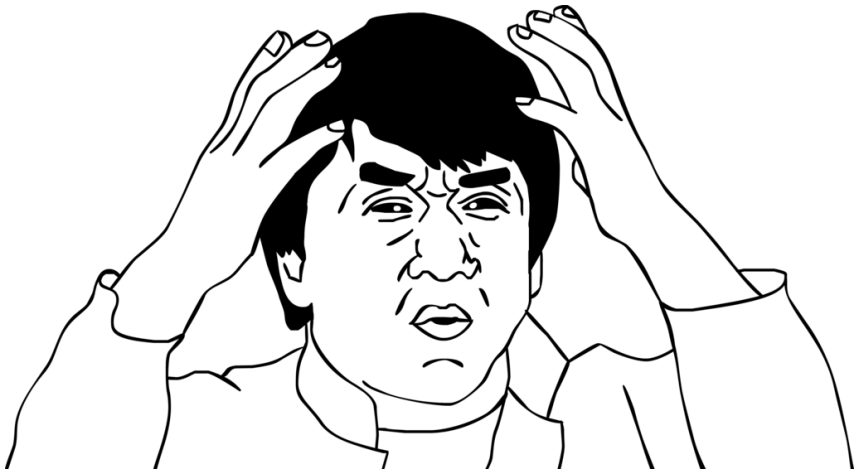
Этап размышления и поиск возможного пути оптимизации расчета (2)
Решения на поверхности

1. На основе исходных данных создать «подготовленные» промежуточные таблицы
2. Создать больше индексов, провести оптимизацию хранения, создать механизмы сжатия устаревших данных
3. Использовать механизмы параллельных расчетов



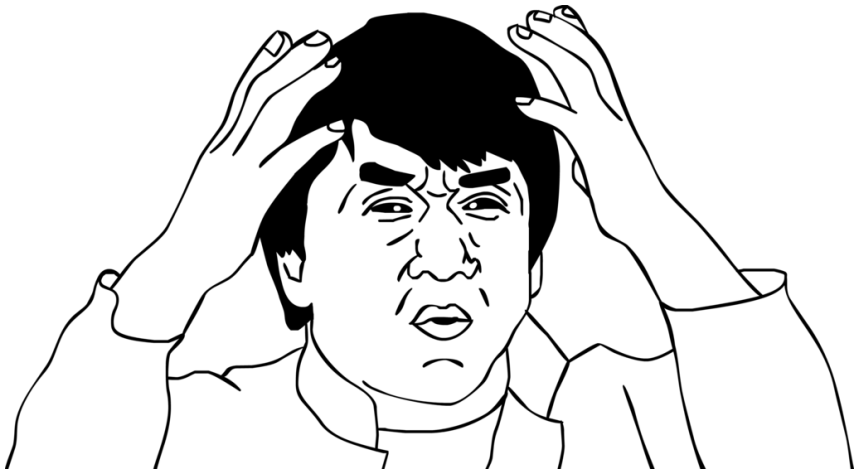
Был найден вариант работы с Fork/Join
Возникшие вопросы

- Подойдет ли концепция Fork/Join для вычислений?



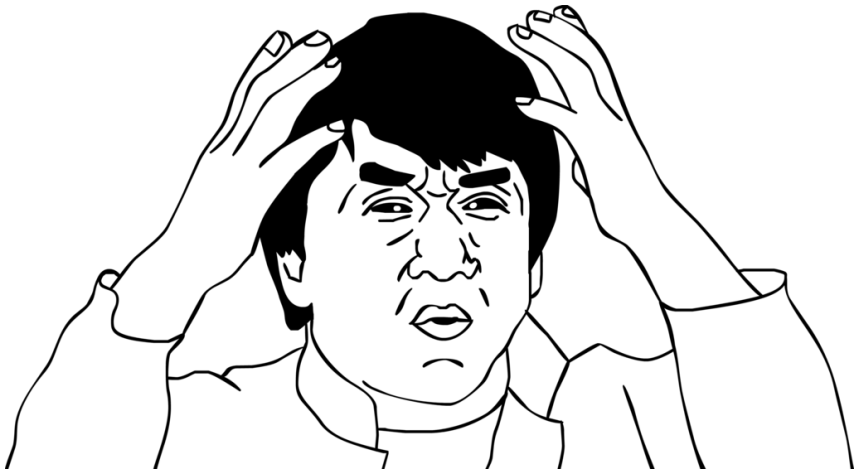
Был найден вариант работы с Fork/Join
Возникшие вопросы

- Подойдет ли концепция Fork/Join для вычислений?
- Насколько легко использовать данные конструкции?



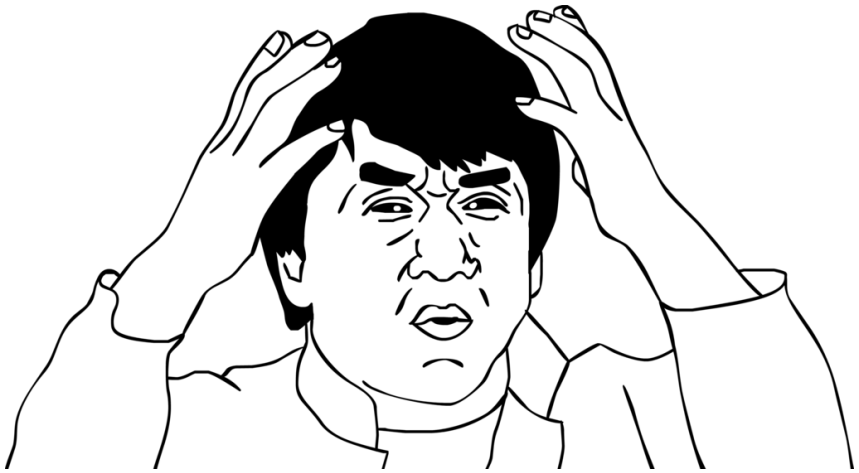
Был найден вариант работы с Fork/Join
Возникшие вопросы

- Подойдет ли концепция Fork/Join для вычислений?
- Насколько легко использовать данные конструкции?
- Ошибки будут?



Был найден вариант работы с Fork/Join
Возникшие вопросы

- Подойдет ли концепция Fork/Join для вычислений?
- Насколько легко использовать данные конструкции?
- Ошибки будут?
- Кто использовал Fork/Join до меня из знакомых?



Информация о Fork/Join (1)

- Fork/Join Framework появился в Java SE 7. С его помощью можно довольно просто использовать возможности процессоров, доступных в исполняемой среде

Информация о Fork/Join (1)

- Fork/Join Framework появился в Java SE 7. С его помощью можно довольно просто использовать возможности процессоров, доступных в исполняемой среде

- Класс RecursiveAction, поддерживает параллельное рекурсивное разложение для задач, не возвращающих результат
- Класс RecursiveTask делает то же самое для возвращающих результат задач

Информация о Fork/Join (1)

- Fork/Join Framework появился в Java SE 7. С его помощью можно довольно просто использовать возможности процессоров, доступных в исполняемой среде

- Класс RecursiveAction, поддерживает параллельное рекурсивное разложение для задач, не возвращающих результат
- Класс RecursiveTask делает то же самое для возвращающих результат задач

- “Общение” между задачами происходит через fork() и join()
- Схема с fork-join уменьшает состязание за очередь с помощью метода, называемого захват работы (work stealing)

Информация о Fork/Join (2)
Threshold

- Далее для понимания возможностей Fork/Join будем оперировать понятием «threshold»
- В переводе – это порог, предел, пороговый уровень

- В литературе понятие «threshold» связывают с некотором пределом, при достижении которого имеет смысл производить действия, направленные на деление задачи на несколько подзадач, при условии наличия аппаратных мощностей

- В случае с Fork/Join threshold ассоциируют с порогом перехода от последовательной обработки к параллельной

Примерный вид работы Fork/Join (в рамках RecursiveAction)

```
public class Example extends RecursiveAction {

    @Override
     protected void compute() {
          if (не достигли threshold) {
               ВЫПОЛНИМ ЗАДАЧУ!
          } else {
               РАЗДЕЛИМ НА МЕНЬШИЕ ПОДЗАДАЧИ!
          }
     }
}
```

Написание примера (1)

Запуск (1)

```
import java.util.concurrent.ForkJoinPool;

    public class Start {
        public static void main(String[] args) {

            // расчеты до  7 000 000
            int componentValue = 7 000 000;
            Long beginT = System.nanoTime();

            //создание пула
            ForkJoinPool fjp = new ForkJoinPool();
            Example example = new Example(0, componentValue);
            fjp.invoke(example);

             // вычисление времени работы
            Long endT  = System.nanoTime();
            Long timeStartEnd = endT - beginT;
            System.out.println("Time =" + timeStartEnd );

            boolean status = example.isCompletedNormally();

             // проверка корректности выполнения
            if (status) {
                System.out.println("NO ERROR");
            }
        }
    }
```

«Делим» (2)

```
import java.util.concurrent.RecursiveAction;

public class Example extends RecursiveAction {

    int cuntPr = Runtime.getRuntime().availableProcessors();

      // условный threshold (предел)
    int cntLimit = 3 000 000;
    int start;
    int end;

    Example(int startNumber, int endNumber) {
        start = startNumber;
        end = endNumber;
    }

    protected void compute() {
        if (end - start <= cntLimit) {
            for (int i = start; i <= end; i++) {
                new Calc().go(i);
            }
        } else {
            System.out.println("=split=");
            int middle = (start + end) / 2;
            invokeAll(new Example(start, middle),
                    new Example(middle + 1, end));
        }
    }
}
```

Вычисления. Возводим число в заданную степень (3)

```
public class Calc {
    public void go(int numberForCalc) {
        for(int i = 0; i <= numberForCalc; i++) {
            double pow = Math.pow(numberForCalc,45);
        }
    }
}
```

```
import java.util.concurrent.ForkJoinPool;

    public class Start {
        public static void main(String[] args) {

            // расчеты до  7 000 000
            int componentValue = 7 000 000;
            Long beginT = System.nanoTime();
            // создание пула
            ForkJoinPool fjp = new ForkJoinPool();
            Example example = new Example(0, componentValue);
            fjp.invoke(example);
                // вычисление времени работы
            Long endT  = System.nanoTime();
            Long timeStartEnd = endT - beginT;
            System.out.println("Time =" + timeStartEnd );
            boolean status = example.isCompletedNormally();

                // проверка корректности выполнения
            if (status) {
                System.out.println("NO ERROR");
            }
        }
    }
```

```
import java.util.concurrent.RecursiveAction;

  public class Example extends RecursiveAction {
    int cuntPr = Runtime.getRuntime().availableProcessors();
        // условный threshold (предел)
    int cntLimit = 3 000 000;
    int start;
    int end;

    Example(int startNumber, int endNumber) {
      start = startNumber;
      end = endNumber;
    }
    protected void compute() {
      if (end - start <= cntLimit) {
        for (int i = start; i <= end; i++) {
          new Calc().go(i);
        }
      } else {
        System.out.println("=split=");
        int middle = (start + end) / 2;
       invokeAll(new Example(start, middle),new Example(middle + 1, end));
      }
    }
  }
```

Какой будет результат в примере?

А)
NO ERROR
Time =……

Б)
=split=
Time = ……..
NO ERROR
=split=
Time = ……..
NO ERROR
=split=
Time = ……..
NO ERROR

В)
=split=
=split=
=split=
Time = ……..
NO ERROR

Г)
=split=
Time = ……..
NO ERROR

Особенности реализации

```
import java.util.concurrent.ForkJoinPool;

public class Start {
    public static void main(String[] args) {

    int componentValue = 7 000 000;

    ForkJoinPool fjp = new ForkJoinPool();

    Example example = new Example(0,
    componentValue);
    fjp.invoke(example);

    boolean status =
    example.isCompletedNormally();

     if (status) {
        System.out.println("NO ERROR");
     }
    }
}
```

Создание пула задач

ForkJoinPool fjp = new ForkJoinPool();

Создание пула задач начиная с JDK 8

ForkJoinPool fjp = ForkJoinPool.commonPool();

Уровень параллелизма

а) **поставить по умолчанию** (будет зависеть от количества процессоров);

Runtime.*getRuntime*().availableProcessors();

ВАЖНО: фактически это - **number of logical cores**

б) **указать кол-во в ForkJoinPool**

(int уровень_параллелизма);

Из документации

ForkJoinPool Creates a ForkJoinPool with parallelism equal to Runtime.availableProcessors(), using the default thread factory, no UncaughtExceptionHandler, and non-async LIFO processing mode

ForkJoinPool (int parallelism) Creates a ForkJoinPool with the indicated parallelism level, the default thread factory, no UncaughtExceptionHandler, and non-async LIFO processing mode

Выбор предела (1)

- Threshold можно определить по-разному в зависимости от вашей задачи
- Например, умножим N (количество элементов) на Q (стоимость на элемент), где Q - это количество операций, а затем проверяя, что N * Q больше или меньше threshold.
- Часто в презентациях добавлены дополнительные условия для определения корректного threshold

Выводы

- Если Q занимает много времени, то распараллеливать стоит уже при N = 2
- Если Q достаточно мало, то процесс деления на подзадачи может быть медленнее последовательного выполнения

Выбор предела (2)

- Threshold можно определить по-разному в зависимости от вашей задачи
- Например, умножим N (количество элементов) на Q (стоимость на элемент), где Q - это количество операций, а затем проверяя, что N * Q больше или меньше threshold.
- Часто в презентациях добавлены дополнительные условия для определения корректного threshold

Выводы

- Имеет смысл делать реализацию переключения в многопоточный режим через настройки
- Как вариант, создавать динамические настройки в зависимости от количества доступных процессоров и загруженности системы в данный момент. Возможны и другие варианты расчета

Результат

Вариант №1 (threshold = 3 000 000)

- значение = 7 000 000
- threshold = 3 000 000

Вариант №2 (threshold = 7 000 000)

- значение = 7 000 000
- threshold = 7 000 000

Время выполнения (оба варианта)

Вариант №1 Time = 194 712 300

Вариант №2 Time = 543 145 178

Вывод

Идет сокращение времени выполнения задачи, при корректно заданном threshold.

Пример «ускорения» при использовании Fork/Join (1)

An informal test was conducted on a Sun Fire T2000 server from Oracle where the number of cores to be available for a Java Virtual Machine could be specified. Both the fork/join and single thread variants of the above example were run to find the number of occurrences of import over the JDK source code files.

В примере демонстрируется подсчет вхождения слов в набор документов.

Number of Cores	Single-Thread Execution Time(ms)	Fork/Join Execution Time(ms)	Speedup
2	18 798	11 026	1.7
4	19 473	8 329	2.33
8	18 911	4 208	4.49
12	19 410	2 876	6.74

Пример «ускорения» при использовании Fork/Join (2)

Пример нахождения максимального элемента в массиве из 500 000 элементов на различных системах и с различными порогами минимальной размерности задачи для разбиения.

	500 000	50 000	5000	500	-50
Pentium-4 HT (2 потока)	1	1.07	1.02	0.82	0.2
Dual-Xeon HT (4 потока)	0.88	3.02	3.2	2.22	0.43
8-процессорный Opteron (8 потоков)	1	5.29	5.73	4.53	2.03
8-ядерный Niagara (32 потока)	0.98	10.46	17.21	15.34	6.49

Итоги по использованию Fork/Join

1. Уменьшение времени расчета

Итоги по использованию Fork/Join

1. Уменьшение времени расчета
2. Расчет не зависит от БД

Итоги по использованию Fork/Join

1. Уменьшение времени расчета

2. Расчет не зависит от БД

3. В большинстве случаев не требуется закупка дополнительного оборудования

Итоги по использованию Fork/Join

1. Уменьшение времени расчета

2. Расчет не зависит от БД

3. В большинстве случаев не требуется закупка дополнительного оборудования

4. Возможность будущего роста скорости расчета

Итоги по использованию Fork/Join

1. Уменьшение времени расчета

2. Расчет не зависит от БД

3. В большинстве случаев не требуется закупка дополнительного оборудования

4. Возможность будущего роста скорости расчета

5. Реализованный механизм легко перенести/изменить/настроить

Итоги по использованию Fork/Join

1. Уменьшение времени расчета

2. Расчет не зависит от БД

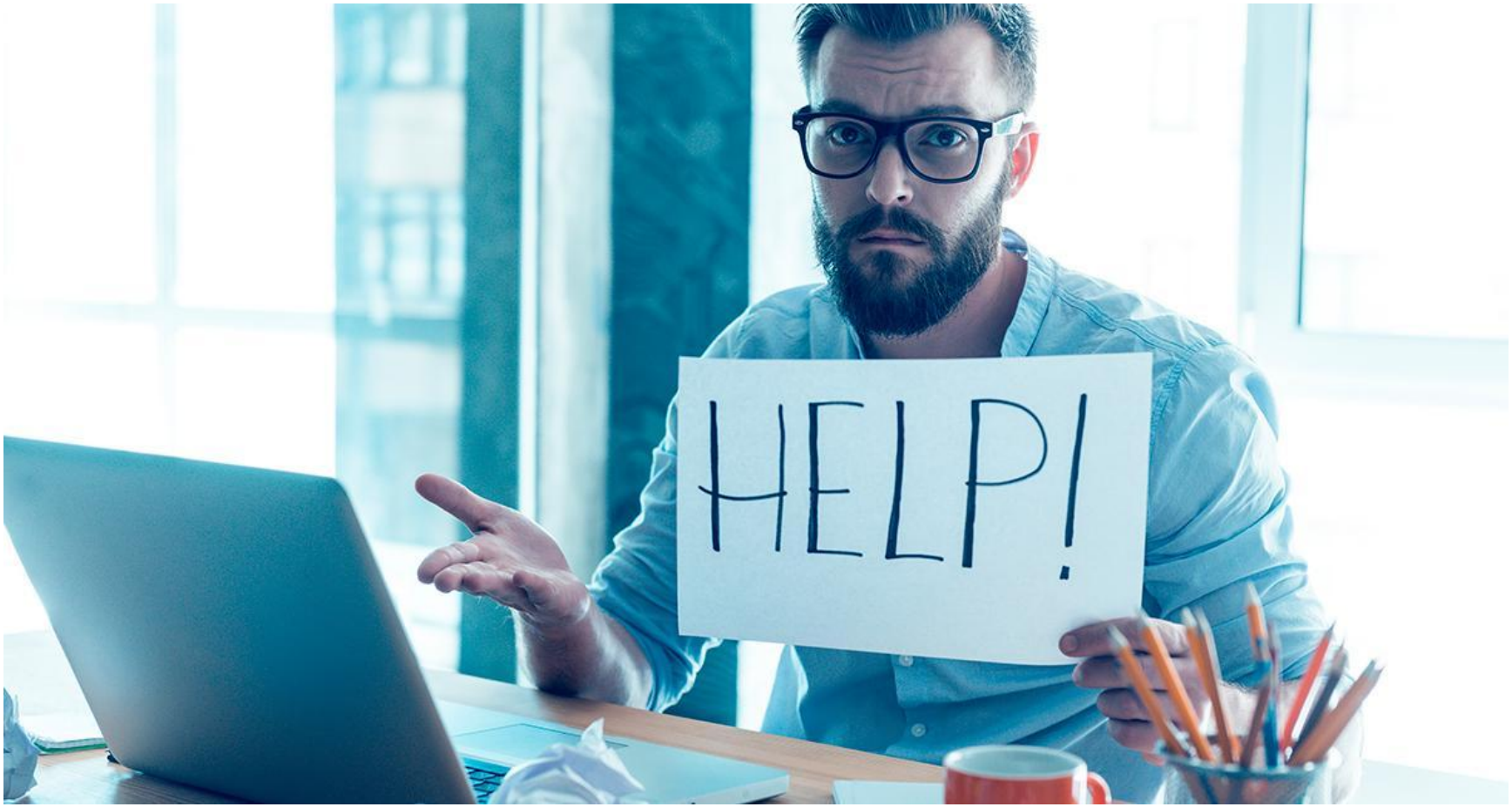
3. В большинстве случаев не требуется закупка дополнительного оборудования

4. Возможность будущего роста скорости расчета

5. Реализованный механизм легко перенести/изменить/настроить

6. Быстрый возврат к «стандартной» схеме работы

А были ли проблемы?



А были ли проблемы?

1. Проблема выбора или нахождения оптимального порогового значения (threshold)

Решается перебором возможных значений с учетом возможностей аппаратных средств. Возможно создание настроек для изменения применения параллельных вычислений и глубины деления на подзадачи.

А были ли проблемы?

1. Проблема выбора или нахождения оптимального порогового значения (threshold)
2. Проблема контроля загруженности ядер процессора и выбор оптимального времени запуска в течение дня

Если сервер используется для большого количества задач, то вариантом решения будет создание графика запусков/запуска в свободное от нагрузки время(механизма контроля запуска).

А были ли проблемы?

1. Проблема выбора или нахождения оптимального порогового значения (threshold)
2. Проблема контроля загруженности ядер процессора и выбор оптимального времени запуска в течение дня
3. Проблема контроля выполнения в текущий момент времени

При создании задачи особое внимание уделить журналированию.

А были ли проблемы?

1. Проблема выбора или нахождения оптимального порогового значения (threshold)
2. Проблема контроля загруженности ядер процессора и выбор оптимального времени запуска в течение дня
3. Проблема контроля выполнения в текущий момент времени
4. Проблема при получении ошибок в одном из «потоков»

Использование после окончания работы методов isCompletedNormally() и isCompletedAbnormally(). Дальнейшее изучение лог-файлов.

А были ли проблемы?

1. Проблема выбора или нахождения оптимального порогового значения (threshold)
2. Проблема контроля загруженности ядер процессора и выбор оптимального времени запуска в течение дня
3. Проблема контроля выполнения в текущий момент времени
4. Проблема при получении ошибок в одном из «потоков»
5. Проблема блокировок потоков при их взаимодействии внутри системы и блокировок в целом

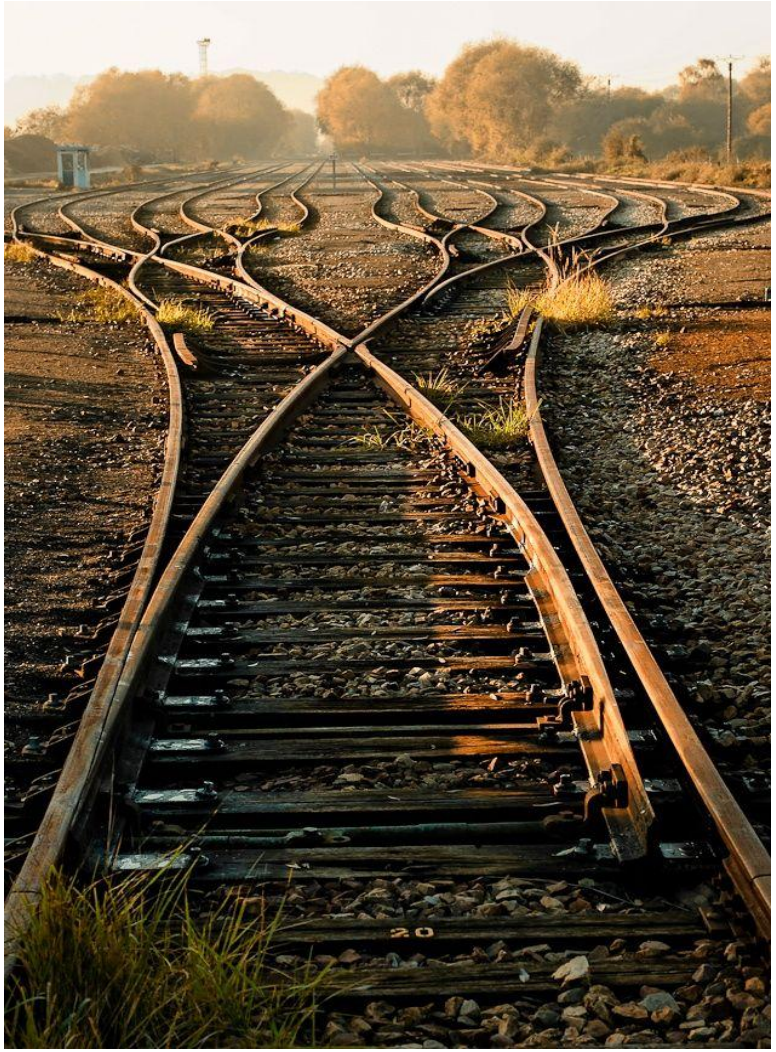
Использовать ForkJoinPool.ManagedBlocker, для обеспечения достаточного параллелизма.

А были ли проблемы?

1. Проблема выбора или нахождения оптимального порогового значения (threshold)
2. Проблема контроля загруженности ядер процессора и выбор оптимального времени запуска в течение дня
3. Проблема контроля выполнения в текущий момент времени
4. Проблема при получении ошибок в одном из «потоков»
5. Проблема блокировок потоков при их взаимодействии внутри системы и блокировок в целом
6. Проблема записи в базу данных при огромном количестве созданных записей

Проверка настроек «autocommit» на базе данных. В зависимости от этого возможны различные варианты записи данных.

Второй этап реализации – это совершенствование (1)



- Вынести повторяющийся код
- Повторно использовать полученные данные
- Получить часть данных до выполнения расчета
- Улучшить читаемость кода
- **Обработку части данных перевести на использование Stream API**

Второй этап реализации – это совершенствование (2)

- Вынести повторяющийся код
- Повторно использовать полученные данные
- Получить часть данных до выполнения расчета
- Улучшить читаемость кода
- **Обработку части данных перевести на использование Stream API**

Из документации

«Another implementation of the fork/join framework is used by methods in the java.util.streams package, which is part of Project Lambda scheduled for the Java SE 8 release. For more information, see the Lambda Expressions section.»

Stream API & ForkJoinPool
Рассмотрим параллельные «вычисления» в Stream API для Collection

- Для использования parallelStream() нужно понимать как он работает
- Перед использованием parallelStream() нужно понимание какие данные будут в обработке
- Перевод в parallelStream() не всегда даст выигрыш в производительности. Много влияющих факторов

Doug Lea (1)

Вопрос

The java.util.streams framework supports data-driven operations on collections and other sources. Most stream methods apply the same operation to each data element. When multiple cores are available, "data-driven" can become "data-parallel", by using the parallelStream() method of a collection. But when should you do this?

Ответ

Consider using S.parallelStream().operation(F) instead of S.stream().operation(F) when operations are independent, and either computationally expensive or applied to many elements of efficiently splittable data structures, or both. In more detail:F, the per-element function (usually a lambda) is independent: the computation for each element does not rely on or impact that of any other element. (See the stream package summary for further guidance about using stateless non-interfering functions.)S, the source collection is efficiently splittable. There are a few other readily parallelizable stream sources besides Collections, for example, java.util.SplittableRandom (for which you can use the stream.parallel() method to parallelize). But most sources based on IO are designed primarily for sequential use.The total time to execute the sequential version exceeds a minimum threshold. These days, the threshold is roughly (within a factor of ten of) 100 microseconds across most platforms. You don't need to measure this precisely though. You can estimate this well enough in practice by multiplying N (the number of elements) by Q (cost per element of F), in turn guestimating Q as the number of operations or lines of code, and then checking that N * Q is at least 10000. (If you are feeling cowardly, add another zero or two.) So when F is a tiny function like x -> x + 1, then it would require N >= 10000 elements for parallel execution to be worthwhile. And conversely, when F is a massive computation like finding the best next move in a chess game, the Q factor is so high that N doesn't matter so long as the collection is completely splittable. The streams framework does not (and cannot) enforce any of these. If the computation is not independent, then running it in parallel will not make any sense and might even be harmfully wrong.

Doug Lea (2)

Вопрос читателя

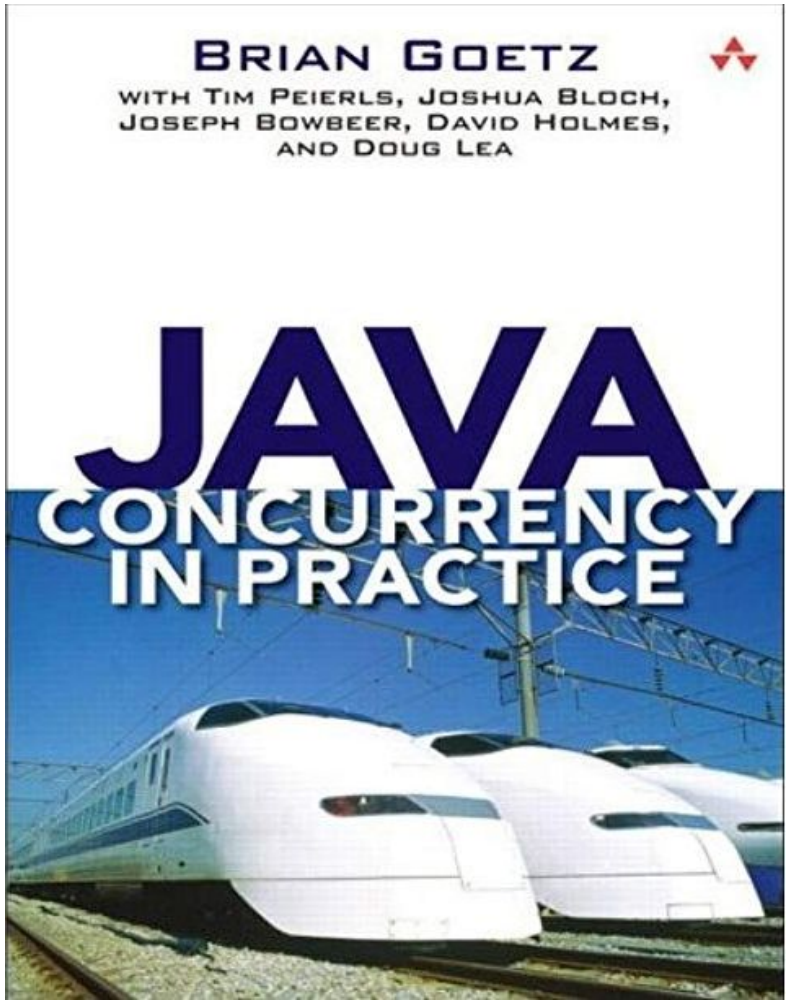
Основной смысл вопроса читателя сводится к желанию понять, когда можно безболезненно использовать parallelStream().

Ответ Doug Lea

Умножим N (количество элементов) на Q (стоимость на элемент F), где Q - это количество операций или строк кода, а затем проверим, что N * Q больше или меньше установленного предела.

Условность примера

Вы можете видеть число «10 000» в ответе. Это условное число, даже сам автор ответа (Doug Lea) говорит о том, что если вы опасаетесь (сомневаетесь), то добавьте к этому числу еще один ноль или даже два.



Java 8 Lambdas. Functional Programming for the Masses (1)

Ричард Уорбэртон в своей книге говорит о теме параллельности и пороговых значениях.

Данная книга доступна в переводе под названием «Лямбда-выражения в Java 8. Функциональное программирование – в массы».

Альбомы в примере – это музыкальные композиции.

В примерах вычисляется длительность звучания последовательности альбомов. Каждый альбом преобразуется в набор составляющих его произведений, после чего длительности произведений суммируются.

«При замере времени работы примеров 6.1 и 6.2 на 4-ядерной машине при 10 Альбомах последовательная версия оказывается в 8 раз быстрее. При 100 альбомах обе версии работают одинаково быстро, а при 10 000 альбомов параллельная версия опережает последовательную в 2,5 раза.

Все результаты измерений в этой главе приводятся только для сведения. На вашей машине они могут оказаться совершенно другими. Размер входного потока – не единственный фактор, определяющий, даст ли распараллеливание ускорение. Результаты могут также зависеть от способа написания кода и количества доступных ядер.»

Java 8 Lambdas. Functional Programming for the Masses (2)

Пример «6.1»

```
public int serialArraySum() {
    return albums.stream()
      .flatMap(Album::getTracks)
      .mapToInt(Track::getLength)
      .sum();
}
```

Пример «6.2»

```
public int parallelArraySum() {
    return albums.parallelStream()
                .flatMap(Album::getTracks)
                      .mapToInt(Track::getLength)
                 .sum();
}
```

Еще немного про Stream API

Темой Stream API активно занимается Тагир Валеев. В сети можно найти его статьи и видео по данной теме.

Статьи

- Stream API: универсальная промежуточная операция
- Используйте Stream API проще (или не используйте вообще)

Видео

- Странности Stream API
- Stream API: рекомендации лучших собаководов

«Ваша работа — это N*Q, где N — количество элементов, а Q - среднее время обработки элемента. Если Q очень велико, то распараллеливать имеет смысл уже при N =2. Если Q исключительно мало (например, сложение двух чисел), то параллелизм поможет только при больших N. Кроме того, многое зависит от самих операций в стриме. Если все операции stateless, вы можете получить хороший прирост.
Если имеются stateful-операции, можно сильно замедлиться даже для очень большого N. Переменных очень много.»

Спасибо
Время вопросов

Источники

- https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.ManagedBlocker.html
- https://www.oracle.com/technetwork/articles/java/fork-join-422606.html
- https://www.ibm.com/developerworks/java/library/j-jtp11137/index.html
- https://www.ibm.com/developerworks/ru/library/j-jtp11137/
- http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html
- https://habr.com/company/luxoft/blog/270383/
- https://habr.com/post/262139/
- https://habr.com/post/337350/
- https://youtu.be/t0dGLFtRR9c
- https://youtu.be/TPHMyVyktsw
- https://youtu.be/vxikpWnnnCU