Основы одновременного исполнения в Java 8

Средства Java 8 существенно упрощают программирование одновременного исполнения

Денис Сосноски  
Опубликовано 07.10.2014

**Серия контента:**

[Этот контент является частью серии:**Одновременное исполнение на платформе JVM**](javascript:void();)

**Об этом цикле статей**

Сегодня, когда многоядерные системы получили повсеместное распространение, программирование одновременного исполнения востребовано шире, чем когда-либо прежде. Однако корректная реализация одновременного исполнения может оказаться непростым делом, поэтому необходимы новые инструменты, помогающие использовать одновременное исполнение. Подобные инструменты разрабатываются для многих языков на базе JVM; особенно активен в этой сфере язык Scala. В данном цикле статей рассматриваются некоторые современные подходы к программированию одновременного исполнения на языках Scala и Java.

Ряд крупных улучшений в долгожданном релизе Java 8 имеют отношение к одновременному исполнению, в том числе добавленные классы в иерархии java.util.concurrent и новая мощная параллельная опция *streams* (потоки). Потоки предназначены для использования совместно с *лямбда-выражениями*. Это дополнение Java 8, которое также упрощает многие другие аспекты повседневного программирования (в [сопутствующей статье](http://www.ibm.com/developerworks/library/j-java8lambdas/index.html) по языковым расширениям в Java 8 изложена вводная информация по лямбда-выражениям и по соответствующим изменениям в interface).

В этой статье я сначала показываю, как новый класс CompletableFuture упрощает координацию асинхронных операций. Затем я показываю, как использовать параллельные потоки — мощное средство для поддержки одновременного исполнения в Java 8 — для параллельного исполнения операций с наборами значений. И, наконец, я показываю, как работают новые возможности Java 8. В частности, я произвожу сравнение с программным кодом из [первой статьи](http://www.ibm.com/developerworks/library/j-jvmc1/index.html) данного цикла (в разделе [Ресурсы](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#artrelatedtopics) приведена ссылка на полный учебный код для этой статьи).

Снова о Future

В [первой статье](http://www.ibm.com/developerworks/library/j-jvmc1/index.html) данного цикла была дана краткая вводная информация по Future в Java и в Scala. Java-версия future (в предварительном релизе Java 8) является слабой, она поддерживает лишь два типа использования: вы можете либо проверять, не завершился ли future-объект, либо ждать завершения future-объекта. Версия Scala является гораздо более гибкой: вы можете выполнить обратные вызовы, когда future-объект завершается, а аварийные завершения обрабатывать в форме Throwable .

В релизе Java 8 добавлен класс CompletableFuture<T>, который реализует новый интерфейс CompletionStage<T> и расширяет интерфейс Future<T> (все классы и интерфейсы для поддержки одновременного исполнения, рассматриваемые в этом разделе, находятся в пакете java.util.concurrent). CompletionStage представляет этап или шаг в потенциально асинхронном вычислении. Этот интерфейс определяет множество различных способов для связывания экземпляров CompletionStage в цепочку с другими экземплярами или с кодом, например, с методами, вызов которых осуществляется при завершении (в общей сложности 59 методов, для сравнения — у интерфейса Future этот показатель составляет 5 методов).

В листинге 1 показан класс ChunkDistanceChecker, основанный на коде сравнения расстояния редактирования из [первой статьи](http://www.ibm.com/developerworks/library/j-jvmc1/index.html).

Листинг 1. Класс ChunkDistanceChecker

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | public class ChunkDistanceChecker {      private final String[] knownWords;        public ChunkDistanceChecker(String[] knowns) {          knownWords = knowns;      }        /\*\*      \* Построение списка проверяющих элементов, охватывающего весь список слов.      \*      \* @param words      \* @param block      \* @return checkers      \*/      public static List<ChunkDistanceChecker> buildCheckers(String[] words, int block) {          List<ChunkDistanceChecker> checkers = new ArrayList<>();          for (int base = 0; base < words.length; base += block) {              int length = Math.min(block, words.length - base);              checkers.add(new ChunkDistanceChecker(Arrays.copyOfRange(words, base,  base + length)));          }          return checkers;          }      ...      /\*\*       \* Нахождение наилучшего расстояния от целевого слова до любого известного слова.       \*       \* @param target       \* @return best       \*/      public DistancePair bestDistance(String target) {          int[] v0 = new int[target.length() + 1];          int[] v1 = new int[target.length() + 1];          int bestIndex = -1;          int bestDistance = Integer.MAX\_VALUE;          boolean single = false;          for (int i = 0; i < knownWords.length; i++) {              int distance = editDistance(target, knownWords[i], v0, v1);              if (bestDistance > distance) {                  bestDistance = distance;                  bestIndex = i;                  single = true;              } else if (bestDistance == distance) {                  single = false;              }          }          return single ? new DistancePair(bestDistance, knownWords[bestIndex]) :              new DistancePair(bestDistance);      }  } |

Каждый экземпляр класса ChunkDistanceChecker сравнивает целевое слово с массивом известных слов с целью нахождения наилучшего соответствия. Статический метод buildCheckers() создает список List<ChunkDistanceChecker> на основе всего массива известных слов и требуемого размера фрагмента. Данный класс ChunkDistanceChecker служит основой для нескольких описываемых в этой статье одновременно исполняемых реализаций класса для поиска по наилучшему соответствию. Первым из них является класс CompletableFutureDistance0 в листинге 2.

Листинг 2. Вычисление расстояния редактирования с помощью CompletableFuture

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class CompletableFutureDistance0 extends TimingTestBase {      private final List<ChunkDistanceChecker> chunkCheckers;        private final int blockSize;        public CompletableFutureDistance0(String[] words, int block) {          blockSize = block;          chunkCheckers = ChunkDistanceChecker.buildCheckers(words, block);      }      ...      public DistancePair bestMatch(String target) {          List<CompletableFuture<DistancePair>> futures = new ArrayList<>();          for (ChunkDistanceChecker checker: chunkCheckers) {              CompletableFuture<DistancePair> future =                  CompletableFuture.supplyAsync(() -> checker.bestDistance(target));              futures.add(future);          }          DistancePair best = DistancePair.worstMatch();          for (CompletableFuture<DistancePair> future: futures) {              best = DistancePair.best(best, future.join());          }          return best;      }  } |

**Лямбда-выражения**

Лямбда-выражение, которое я передаю методу supplyAsync(), относится к категории capturing, поскольку оно ссылается на значение параметра target. Для получения вводной информации по лямбда-выражениям и о различиях между лямбда-выражениями категорий capturing и noncapturing обратитесь к статье [Изменения в языке Java 8](http://www.ibm.com/developerworks/library/j-java8lambdas/index.html).

В [листинге 2](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing2) класс CompletableFutureDistance0 демонстрирует один из способов использования CompletableFuture для одновременных вычислений. Метод supplyAsync() принимает экземпляр Supplier<T> (функциональный интерфейс с методом, возвращающий значение типа T) и возвращает экземпляр CompletableFuture<T>, а также помещает Supplier в очередь для исполнения в асинхронном режиме. В первом цикле for я передаю лямбда-выражение в метод supplyAsync() с целью создания списка future-объектов, соответствующего массиву ChunkDistanceChecker. Второй цикл for ждет завершения каждого future-объекта (хотя большинство из них завершается до того, как этот цикл добирается до них, поскольку они исполняются асинхронно) и аккумулирует наилучшее соответствие из всех результатов.

Построение CompletableFuture

В [первой статье](http://www.ibm.com/developerworks/library/j-jvmc1/index.html) данного цикла было показано, что Future-объекты в Scala позволяют присоединять обработчики завершения и сочетать future-объекты различными способами. В языке Java 8 подобную гибкость обеспечивает класс CompletableFuture. В этом разделе вы изучите некоторые способы использования этих возможностей в контексте кода для проверки расстояния редактирования.

В листинге 3 показана иная версия метода bestMatch(), отличающаяся от показанной в [листинге 2](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing2). Эта версия использует обработчик завершения с классом CompletableFuture вместе с несколькими более старыми классами для одновременного исполнения.

Листинг 3. Класс CompletableFuture с обработчиком завершения

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public DistancePair bestMatch(String target) {      AtomicReference<DistancePair> best = new AtomicReference<>  (DistancePair.worstMatch());      CountDownLatch latch = new CountDownLatch(chunkCheckers.size());      for (ChunkDistanceChecker checker: chunkCheckers) {          CompletableFuture.supplyAsync(() -> checker.bestDistance(target))              .thenAccept(result -> {                  best.accumulateAndGet(result, DistancePair::best);                  latch.countDown();              });      }      try {          latch.await();      } catch (InterruptedException e) {          throw new RuntimeException("Interrupted during calculations", e);      }      return best.get();  } |

В [листинге 3](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing3) защелка (latch) CountDownLatch инициализируется по количеству future-объектов, созданных в программном коде. При создании каждого future-объекта я присоединяю к нему обработчик (в форме лямбда-экземпляра функционального интерфейса java.util.function.Consumer<T>) с помощью метода CompletableFuture.thenAccept(). Этот обработчик, который исполняется в случае нормального завершения future-объекта, использует метод AtomicReference.accumulateAndGet() (добавленный в релизе Java 8) для обновления найденного наилучшего значения, а затем декрементно уменьшает защелку. В это время основной поток исполнения вводит блок try-catch и ждет освобождения защелки. После завершения всех future-объектов основной поток продолжает исполнение, возвращая в итоге найденное наилучшее значение.

В листинге 4 показана еще одна разновидность метода bestMatch() из [листинга 2](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing2).

Листинг 4. Объединение нескольких CompletableFuture

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public DistancePair bestMatch(String target) {      CompletableFuture<DistancePair> last =          CompletableFuture.supplyAsync(bestDistanceLambda(0, target));      for (int i = 1; i < chunkCheckers.size(); i++) {          last = CompletableFuture.supplyAsync(bestDistanceLambda(i, target))              .thenCombine(last, DistancePair::best);      }      return last.join();  }    private Supplier<DistancePair> bestDistanceLambda(int i, String target) {      return () -> chunkCheckers.get(i).bestDistance(target);  } |

В этом коде метод CompletableFuture.thenCombine () используется для слияния двух future-объектов посредством применения функции вида java.util.function.BiFunction (в данном случае метода DistancePair.best()) к двум результатам и возвращения future-объекта с результатом этой функции.

В листинге 4 представлена наиболее лаконичная и, возможно, самая "чистая" версия этого кода, однако у нее есть и недостаток: она создает дополнительный уровень объектов CompletableFuture для представления комбинации каждой "фрагментной" операции с предшествующими операциями. В исходном релизе Java 8 это может порождать исключение StackOverflowException, которое терялось внутри кода, в результате чего заключительный future-объект никогда не завершался. В настоящее время проводится работа над этой ошибкой; ошибка должна быть устранена в ближайшем релизе.

Класс CompletableFuture определяет множество вариаций на основе методов, используемых в этих примерах. Перед использованием класса CompletableFuture в своих приложениях изучите полный список методов завершения и методов объединения, чтобы найти метод, который наилучшим образом соответствует вашим потребностям.

Класс CompletableFuture лучше всего подходит для случаев, когда вы применяете различные типы операций и хотите координировать их результаты. Если вы осуществляете одно и то же вычисление со многими различными значениями данных, параллельные потоки проще и, весьма вероятно, выигрывают в производительности. Для примера с проверкой расстояния редактирования лучше подходит вариант на основе параллельных потоков.

Потоки

Потоки, важное новое средство Java 8, работают совместно с лямбда-выражениями. По существу потоки – это push-итераторы, применяемые к последовательности значений. Потоки можно связывать в цепочку через адаптеры для исполнения таких операций, как фильтрация и преобразование (примерно как последовательности в Scala). Кроме того, потоки имеют последовательные и параллельные разновидности, что также напоминает последовательности Scala (следует, однако, отметить, что в Scala имеется отдельная иерархия классов для параллельных последовательностей, а в Java 8 используется внутренний флаг для обозначения категории потока: последовательный или параллельный). Существуют разновидности потоков для примитивных типов int, long и double, а также потоков для типизированных объектов.

Новый API-интерфейс streams слишком сложен для полного рассмотрения в этой статье, поэтому я сосредоточусь на аспектах одновременного исполнения. Для получения более подробных сведений о потоках обратитесь в раздел [Ресурсы](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#artrelatedtopics).

В листинге 5 показан еще один вариант программного кода для отыскания наилучшего соответствия по расстоянию редактирования. Эта версия использует класс ChunkDistanceChecker из [листинга 1](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing1) для вычисления расстояния и класс CompletableFuture из примера в [листинге 2](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing2), однако на этот раз для нахождения наилучшего соответствия я использую потоки.

Листинг 5. Класс CompletableFuture с использованием потоков

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class CompletableFutureStreamDistance extends TimingTestBase {      private final List<ChunkDistanceChecker> chunkCheckers;        ...      public DistancePair bestMatch(String target) {          return chunkCheckers.stream()              .map(checker -> CompletableFuture.supplyAsync(()  -> checker.bestDistance(target)))              .collect(Collectors.toList())              .stream()              .map(future -> future.join())              .reduce(DistancePair.worstMatch(), (a, b) -> DistancePair.best(a, b));      }  } |

Всю работу делает состоящий из нескольких строк оператор в нижней части листинга 5, использующий API-интерфейс streams.

1. chunkCheckers.stream() создает поток из List<ChunkDistanceChecker>.
2. .map(checker -> ... применяет отображение к значениям в потоке (в этом случае с использованием того же подхода, что и в примере в [листинге 2](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing2)) с целью создания CompletableFuture для результата асинхронного исполнения метода ChunkDistanceChecker.bestDistance().
3. .collect(Collectors.toList()) собирает значения в список, который .stream() возвращает обратно в поток.
4. .map(future -> future.join()) ждет доступности результата каждого future, а .reduce(... находит наилучшее значение посредством многократного применения метода DistancePair.best() к предшествующему наилучшему результату и к самому последнему результату.

Возможно, это выглядит запутанно. Однако не прекращайте чтение статьи — уверяю вас, что следующий вариант кода будет "чище" и проще. Цель кода в [листинге 5](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing5) состоит в том, чтобы продемонстрировать возможность использования потоков в качестве альтернативы обычным циклам.

Код в листинге 5 был бы гораздо проще без многократных преобразований из потоков в списки и обратно в потоки. В данном случае преобразование необходимо, поскольку в противном случае код ждал бы лишь завершения метода CompletableFuture.join() сразу же после создания future-объекта.

Параллельные потоки

К счастью, существует более удобный способ реализации параллельных операций с потоками, чем громоздкий подход, показанный в [листинге 5](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing5). Последовательные потоки можно превратить в параллельные потоки, а параллельные потоки автоматически распределяют работу между несколькими потоками и позволяют собирать их результаты на более позднем этапе. В листинге 6 демонстрируется применение этого подхода для нахождения наилучшего соответствия в списке List<ChunkDistanceChecker>.

Листинг 6. Отыскание наилучшего соответствия с использованием параллельных потоков для фрагментов

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class ChunkedParallelDistance extends TimingTestBase {      private final List<ChunkDistanceChecker> chunkCheckers;      ...      public DistancePair bestMatch(String target) {          return chunkCheckers.parallelStream()              .map(checker -> checker.bestDistance(target))              .reduce(DistancePair.worstMatch(), (a, b) -> DistancePair.best(a, b));      }  } |

В этом случае всю работу также делает состоящий из нескольких строк оператор в нижней части листинга. Как и в [листинге 5](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing5), сначала из списка создается поток, однако в этой версии для получения потока используется метод parallelStream(), ориентированный на параллельную обработку (вы также можете преобразовать обычный поток для параллельной обработки, вызвав метод потока parallel()). Следующая часть кода, .map(checker -> checker.bestDistance(target)), находит наилучшее соответствие внутри фрагмента, состоящего из известных слов. Последняя часть кода, .reduce(..., аккумулирует наилучший результат для всех фрагментов, как это делается в [листинге 5](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing6).

Параллельные потоки исполняют определенные шаги, такие как операции map и filter, параллельно. Таким образом, "за кадром" код в [листинге 6](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing6) распространяет шаг map на несколько потоков, прежде чем консолидировать результаты на шаге reduce (это не обязательно делать в каком-либо определенном порядке, поскольку результаты поступают из операций, исполняемых параллельно).

Возможность разделения работы, подлежащей исполнению в потоке, обеспечивается новым интерфейсом java.util.Spliterator<T>, используемым в потоках. Класс Spliterator, как можно предположить по названию, подобен классу Iterator. Spliterator, как и Iterator, позволяет работать с элементами коллекции поочередно — хотя вместо получения элементов из Spliterator вы применяете действие к элементам с использованием метода tryAdvance() или метода forEachRemaining(). Однако Spliterator также предоставляет оценку количества содержащихся в нем элементов и потенциально может быть разделен на две части (подобно делению биологической клетки). Эти добавленные возможности облегчают коду параллельной обработки потоков работ (stream) распределение работы между доступными процессорными потоками (thread).

Код в [листинге 6](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing6) может показаться вам знакомым, поскольку он очень похож на пример параллельных коллекций в Scala из [первой статьи](http://www.ibm.com/developerworks/library/j-jvmc1/index.html)данного цикла:

|  |  |
| --- | --- |
| 1  2  3 | def bestMatch(target: String) =    matchers.par.map(m => m.bestMatch(target)).      foldLeft(DistancePair.worstMatch)((a, m) => DistancePair.best(a, m)) |

Можно увидеть некоторые различия в синтаксисе и в работе, однако по существу код параллельных потоков в Java 8 делает то же самое, что и код параллельных коллекций Scala, и таким же образом.

Порядок применения потоков

До сих пор во всех примерах задача сравнения сохраняла разделенную на фрагменты структуру, представленную в [первой статье](http://www.ibm.com/developerworks/library/j-jvmc1/index.html)данного цикла. Это было необходимо для эффективного манипулирования параллельными задачами в более старых версиях Java. Механизм параллельных потоков Java 8 спроектирован для самостоятельного разделения работы, поэтому вы можете передать набор значений для обработки в виде потока, а встроенная обработка одновременного исполнения разделит этот набор с целью распределения работы между доступными процессорами.

При попытке применения этого подхода к задаче вычисления расстояния редактирования возникает несколько проблем. Если вы связываете шаги обработки в цепочку в виде *конвейера* (официальный термин для последовательности потоковых операций), то вы можете следующему этапу конвейера передать только один результат каждого шага. Если вам требуется несколько результатов (таких как значение наилучшего расстояния и соответствующее известное слово, использованное в задаче вычисления расстояния редактирования), их необходимо передавать как объект. Однако создание такого объекта для результата каждого отдельного сравнения снизило бы производительность "прямого" потокового подхода по сравнению с "фрагментированными" подходами. Что еще хуже, вычисление расстояния редактирования многократно использует два выделенных массива. Эти массивы не могут быть совместно использованы параллельными вычислениями, поэтому для каждого вычисления их необходимо выделять заново.

К счастью, API-интерфейс streams предоставляет эффективный способ обработки этой ситуации, хотя и ценой некоторых дополнительных усилий. В листинге 7 демонстрируется использование потока для обработки полного набора вычислений без создания промежуточных объектов или избыточных копий рабочих массивов.

Листинг 7. Потоковая обработка отдельных сравнений расстояния редактирования

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | public class NonchunkedParallelDistance extends TimingTestBase  {      private final String[] knownWords;      ...      private static int editDistance(String target, String known, int[] v0, int[] v1) {      ...      }        public DistancePair bestMatch(String target) {          int size = target.length() + 1;          Supplier<WordChecker> supplier = () -> new WordChecker(size);          ObjIntConsumer<WordChecker> accumulator = (t, value) -> t.checkWord(target,  knownWords[value]);          BiConsumer<WordChecker, WordChecker> combiner = (t, u) -> t.merge(u);          return IntStream.range(0, knownWords.length).parallel()              .collect(supplier, accumulator, combiner).result();      }        private static class WordChecker {          protected final int[] v0;          protected final int[] v1;          protected int bestDistance = Integer.MAX\_VALUE;          protected String bestKnown = null;            public WordChecker(int length) {              v0 = new int[length];              v1 = new int[length];          }            protected void checkWord(String target, String known) {              int distance = editDistance(target, known, v0, v1);              if (bestDistance > distance) {                  bestDistance = distance;                  bestKnown = known;              } else if (bestDistance == distance) {                  bestKnown = null;              }          }            protected void merge(WordChecker other) {              if (bestDistance > other.bestDistance) {                  bestDistance = other.bestDistance;                  bestKnown = other.bestKnown;              } else if (bestDistance == other.bestDistance) {                  bestKnown = null;              }          }            protected DistancePair result() {              return (bestKnown == null) ? new DistancePair(bestDistance) : new                  DistancePair(bestDistance, bestKnown);          }      }  } |

В листинге 7 для объединения результатов используется мутабельный контейнерный класс результатов (в данном случае — это класс WordChecker). Метод bestMatch() реализует сравнение с использованием трех "механизмов" в форме лямбда-выражений.

* Лямбда-выражение Supplier<WordChecker> supplier предоставляет экземпляры контейнера результатов.
* Лямбда-выражение ObjIntConsumer<WordChecker> accumulator помещает новое значение в контейнер результатов.
* Лямбда-выражение BiConsumer<WordChecker, WordChecker> combiner объединяет два контейнера результатов с целью получения объединенного значения.

После определения трех лямбда-выражений заключительный оператор метода bestMatch() создает параллельный поток int-значений для индексов в массиве известных слов и направляет этот поток в метод IntStream.collect(). Метод collect() вызывает эти три лямбда-выражения для фактического выполнения всей работы.

Производительность одновременного исполнения в Java 8

На рис. 1 показано, как меняется измеряемая производительность в зависимости от размера фрагмента при исполнении тестового кода на моей четырехъядерной системе AMD с Oracle Java 8 под управлением 64-разрядной Linux®. Как и при определении времени исполнения в [первой статье](http://www.ibm.com/developerworks/library/j-jvmc1/index.html) данного цикла, каждое входящее слово поочередно сравнивается с 12564 известными словами, а каждая задача находит наилучшее соответствие внутри набора известных слов. Весь набор из 933 входящих слов с орфографическими ошибками обрабатывается многократно; с паузами между проходами для стабилизации состояния JVM. Наилучшее время после 10 проходов используется в графике, показанном на рис. 1. Заключительный размер блока 16384 превышает количество известных слов, таким образом, этот случай демонстрирует однопоточную производительность. В этот тест на время исполнения включены следующие реализации: четыре основных варианта из этой статьи и лучший вариант из первой статьи.

* CompFuture: CompletableFutureDistance0 из [листинга 2](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing2)
* CompFutStr: CompletableFutureStreamDistance из [листинга 5](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing5)
* ChunkPar: ChunkedParallelDistance из [листинга 6](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing6)
* ForkJoin: ForkJoinDistance из листинга 3 [первой статьи](http://www.ibm.com/developerworks/library/j-jvmc1/index.html)
* NchunkPar: NonchunkedParallelDistance из [листинга 7](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing7)

Рисунок 1. Производительность в Java 8

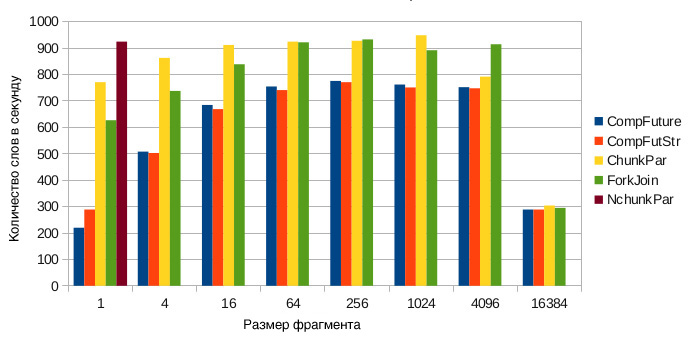


Рис. 1 демонстрирует впечатляющие результаты нового подхода Java 8 на основе параллельных потоков; особенно выделяется полностью оптимизированная реализация NchunkPar, показанная в [листинге 7](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing7). Оптимизация, избавляющая от необходимости создания объектов, дает для времени исполнения результат (на графике показано только одно значение, поскольку в данном подходе фрагменты не используются), соответствующий максимальной производительности среди всех других альтернативных вариантов. Подходы CompletableFuture немного слабее с точки зрения производительности, но это не является неожиданностью, поскольку в этом примере не используются сильные стороны этого класса. В [листинге 5](https://www.ibm.com/developerworks/ru/library/j-jvmc2/index.html#listing5) время исполнения для ChunkPar приблизительно соответствует результату для кода ForkJoin из [первой статьи](http://www.ibm.com/developerworks/library/j-jvmc1/index.html), хотя при меньшей чувствительности к размерам фрагмента. Все разновидности, которые поочередно проверяют фрагменты слов, при небольших размерах фрагмента демонстрируют снижение производительности относительно прогнозируемых показателей, поскольку растет роль относительных издержек на создание объектов в сравнении с фактической вычислительной работой.

Как и результаты из [первой статьи](http://www.ibm.com/developerworks/library/j-jvmc1/index.html), вышеприведенные результаты дают общее представление о производительности, которой можно ожидать в приложениях. Самый важный вывод состоит в том, что новые параллельные потоки Java 8 обеспечивают превосходную производительность при надлежащем применении. Объедините хорошую производительность с преимуществами функционального стиля кодирования потоков в разработке — и у вас на руках будет выигрышная комбинация на любой случай, когда вам потребуются вычисления над коллекциями значений.

Резюме одновременного исполнения в Java 8

Java 8 добавляет в инструментарий разработчика несколько новых важных возможностей. Если говорить об одновременном исполнении, реализация на основе параллельная потоков быстра и проста в использовании, особенно в сочетании с лямбда-выражениями для функционального стиля программирования, который ясно и кратко выражает намерения разработчика. Новый класс CompletableFuture также упрощает программирование одновременного исполнения, когда вы работаете с отдельными действиями, которые плохо подходят для потоковой модели.