УДК 004.432 DOI: 10.18698/2541-8009-2017-3-77

СРАВНЕНИЕ МОДЕЛЕЙ МНОГОПОТОЧНОСТИ НА ЯЗЫКЕ SCALA

И.И. Семенченко dev.ivansem@gmail.com

МГТУ им. Н.Э. Баумана, Москва, Российская Федерация

Аннотация

Рассмотрены типичные проблемы многопоточности. Показано использование многопоточности на языке программирования Scala. Разобраны две разных модели многопоточности — на основе пула потоков и на основе футуров, приведена их сравнительная характеристика. Указано, каким образом использование этих моделей может помочь программисту, и какие сложности могут возникнуть. Описаны преимущества и недостатки каждой модели. Даны примеры кода, по которым наглядно видно, как решаются проблемы распараллеливания в каждом случае. Проведено тестирование эффективности каждой модели на примере конкретной задачи

Ключевые слова

Многопоточность, скала, пул потоков, футуры

Поступила в редакцию 28.03.2017

Каждый разработчик сталкивался с такими задачами, выполнение которых занимает много времени, из-за чего программа не может решать поставленные задачи с требуемой скоростью. Помочь в этом случае может применение многопоточности — возможности одновременно решать различные задачи (подзадачи) в разных потоках. Например, разбить одну большую задачу на несколько малых и решать их параллельно, а в конце собрать результат. Тогда скорость выполнения задачи будет равна скорости выполнения самой трудоемкой части.

Также часто случаются ситуации, когда требуется решить сразу несколько трудоемких задач, например, если на сервер одновременно приходят запросы от многих людей. В этом случае необходимо иметь несколько потоков выполнения и обслуживать клиентов одновременно.

Исходя из вышесказанного можно сделать вывод, что многопоточность необходима для того, чтобы программа могла успешно решать требуемые задачи. Однако у многопоточности существуют недостатки: проблемы с синхронизацией, сильное расходование памяти, нетипичные для последовательного кода ошибки (dead locks, race conditions). Поэтому было создано множество техник написания многопоточного кода, абстракций, благодаря которым можно сделать код более читаемым.

Проблемы многопоточности. В JVM (Java Virtual Machine) были введены потоки для более легкого написания параллельного кода, однако это не решает

все проблемы. Так как в многопоточной разработке несколько параллельных потоков могут быть использованы одни и те же ресурсы, возникает проблема их синхронизации — нужно, чтобы для всех потоков ресурс всегда имел одно и то же значение [1, 2].

Также часто возникает состояние гонки — ошибка, при которой результат выполнения зависит от того, в каком порядке разные потоки получили доступ к ресурсу. Так как нельзя точно угадать, какому потоку первому понадобится ресурс, эта ошибка приносит программистам много проблем.

Еще одна распространенная ошибка — взаимная блокировка или deadlock. Она возникает, когда первому потоку нужен ресурс, заблокированный вторым потоком, и наоборот. В этом случае потоки ждут друг друга и никогда.

В статье будет рассмотрено использование двух разных моделей многопоточности — на основе пула потоков и на основе футуров. У каждой модели многопоточности есть свои преимущества и недостатки. В объектно-ориентированных языках использование футуров неудобно из-за синтаксиса таких языков. Поэтому сравнение будет проводиться на Scala — функциональном языке программирования. Это очень лаконичный язык, поэтому относительно сложные конструкции на нем можно описать небольшим количеством кода.

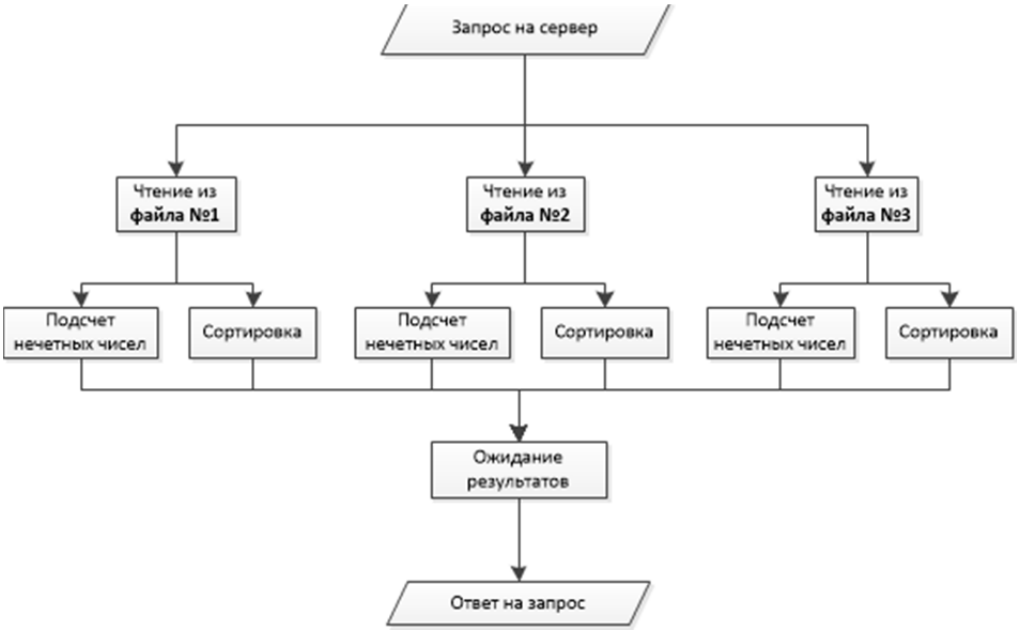
Пул потоков и футуры. Пул потоков — это заранее созданное множество потоков. Создание потока очень долгая по времени операция, поэтому часто выгодно один раз создать несколько потоков и далее их переиспользовать. Для управления пулом потоков используется паттерн Executor. Он заключается в том, что ему дают только саму задачу, а он сам решает, на каком потоке ее выполнить. Другими словами, это абстракция, которая еще больше упрощает управление потоками.

Помимо пула потоков есть более высокоуровневая абстракция — футуры. В этом случае программист может создавать цепочки параллельных вычислений, не имея прямого доступа к пулу потоков. После составления цепочки футуров можно дождаться завершения вычислений всей цепочки, остановив поток, либо указать действия, которые надо будет выполнить, после завершения цепочки вычислений. Такой код компактнее, позволяет придерживаться декларативного стиля, легче читается и поддерживается [3].

Постановка задачи. Чтобы оценить эффективность и удобство обеих моделей многопоточности, надо подобрать такую задачу, для решения которой нужно использовать как можно большее количество разных ресурсов компьютера (память, процессор, чтение с диска). Подходящая задача будет такой:

- есть 3 файла, в каждом из которых записано по 700 000 случайных чисел от 1 до 700 000;
- сначала нужно считать каждый файл в массив чисел (чтение с диска);
- для каждого массива подсчитать количество нечетных чисел (нагрузка на процессор);
- каждый массив отсортировать (нагрузка на память и процессор).

Самый эффективный способ выполнения этой задачи — решать подзадачи параллельно, как показано на рисунке.



Алгоритм параллельного решения поставленной задачи

Для удобства проведения тестирования эта задача обернута в сервер Spray. Таким образом, двумя запросами можно запустить решение задачи двумя методами:

localhost:8091/thread — решение с помощью потоков;

localhost:8091/future — решение с помощью футуров.

Реализация на основе пула потоков. Есть несколько способов построить параллельные вычисления посредством пула потоков. Для решения этой задачи будет использоваться объект класса CountDownLatch. Смысл в том, что с помощью этого объекта можно усыпить текущий поток до тех пор, пока не выполнятся *n* подзадач, где *n* — произвольное число. Фрагмент кода запуска трех параллельных потоков для каждого файла:

```
val latch: CountDownLatch = new CountDownLatch(3)
pool.execute(new Runnable {
 override def run() = {
   report1 = executeOnFileThread(PROJECT_DIR + "1.txt")
   latch.countDown()
 }
})
pool.execute(new Runnable {
 override def run() = {
   report2 = executeOnFileThread(PROJECT_DIR + "2.txt")
```

```
    latch.countDown()
  }
})
pool.execute(new Runnable {
 override def run() = {
    report3 = executeOnFileThread(PROJECT_DIR + "3.txt")
    latch.countDown()
  }
})
latch.await()
```

По окончании решения задачи для каждого из файлов будет выполнена функция latch.countDown(), и поток будет спать, пока эта функция не будет вызвана для каждого файла, т. е. до тех пор, пока все вычисления не завершатся. В итоге получатся три результата выполнения. Таким же образом распараллеливаются задачи сортировки и подсчета количества нечетных символов:

```
      def executeOnFileThread(filePath: String): String = {
  val localLatch: CountDownLatch = new CountDownLatch(2)
  var countOddNumbers: Int = 0
  var lastElement: Int = 0
       val fileName =
      filePath.substring(filePath.lastIndexOf('.') – 1,
      filePath.length)
  val intNumbers = readFileTaskThread(filePath)


  pool.execute(new Runnable {
    override def run() = {
      countOddNumbers = getOddNumbersThread(intNumbers, file-
Name)
      localLatch.countDown()
    }
  })
  pool.execute(new Runnable {
    override def run() = {
      lastElement = sortNumbersThread(intNumbers, fileName)
      localLatch.countDown()
    }
  })

  localLatch.await()
  generateReportThreads(countOddNumbers, lastElement)
}
```

Последняя строчка означает, что из функции вернется результат выполнения функции `generateReportThreads`. Слово return в данном случае необязательно, это особенность синтаксиса языка [4].

Преимущества вычисления посредством пула потоков:

• выполнение каждой подзадачи запускается вручную, сразу после передачи Runnable в пул потоков.

• Java-программист с минимальными знаниями в многопоточности легко может разобраться в данном коде.

Недостатки:

• необходимость усыплять поток выполнения, пока все подзадачи не будут решены;

• синхронизировать результаты выполнения каждой подзадачи приходится вручную;

• для того чтобы запустить выполнение, нужна ссылка на пул потоков;

• для декларативного стиля код слишком громоздкий.

Реализация на основе футуров. Футуры предназначены для более компактного и легкого написания параллельного кода. Java-программисту они могут показаться непонятными, но если поработать с ними некоторое время, то писать, используя футуры, станет легче, чем применяя пул потоков. Футур можно создать, обернув часть кода в специальную конструкцию:

```
val future = Future[Int] {
  val countOdd = getOddNumbers(intNumbers)
  countOdd
}
```

В этом случае текущий поток не заблокируется, и долгая по времени операция будет выполняться в другом потоке. Чтобы получить результат, можно дождаться завершения выполнения футура:

```
  val result = Await.result(future, Duration(10, dura-
tion.SECONDS))
```

Но если надо ждать завершения операции, возникает вопрос, зачем нужны футуры? Можно точно так же подождать, пока завершится выполнение потока.

Во-первых, ждать не обязательно, следует просто определить действия, которые надо будет выполнить при завершении футуры (onComplete, onSuccess, onFailure).

Во-вторых, один футур можно преобразовывать в другой. Например, есть футур, читающий последовательность чисел из файла и преобразовывающий ее в массив:

```
def readFileTaskFuture(filePath: String): Future[List[Int]] =
{
  val lines = readLines(filePath)
  Future[List[Int]] {
    lines.map(Integer.parseInt).toList
  }
}
```

Тогда можно выполнить преобразования, в результате которых из Future[List[Int]] получили сначала два Future[Int], а потом из них Future[String]:

```
readFileTaskFuture(filePath).flatMap { intNumbers =>
  val sortFuture: Future[Int] = sortNumbersFuture(intNumbers,
fileName)
  val oddNumbersFuture: Future[Int] = getOddNumbersFu-
ture(intNumbers, fileName)
  for { /*результат выполнения этой конструкции — Fu-
ture[String], где String — строка, возвращаемая функцией gener-
ateReportFutures(remainder, lastElement)*/
    lastElement <- sortFuture
    remainder <- oddNumbersFuture
  } yield generateReportFutures(remainder, lastElement)
```

В этом же примере видно, как легко запустить две параллельные задачи. Достаточно создать 2 футура и объединить их с помощью специальной конструкции. Запустить параллельные процессы для трех файлов также просто:

```
val finalFuture1 = executeOnFileFuture(filePath1)
val finalFuture2 = executeOnFileFuture(filePath2)
val finalFuture3 = executeOnFileFuture(filePath3)

val finalFuture = for {
  result1 <- finalFuture1
  result2 <- finalFuture2
  result3 <- finalFuture3
} yield result1 + result2 + result3
```

После чего нужно подождать завершения выполнения финального футура, который вернет результат:

```
val result: String = Await.result(finalFuture, Duration(10, du-
ration.SECONDS))
```

Стоит отметить, что сервер Spray поддерживает работу с футурами. Это означает, что можно вернуть серверу футур, и он сам вернет клиенту результат, когда вычисления завершатся, продолжая при этом обрабатывать другие запросы. Но так как сравниваются две модели многопоточности, пользоваться этой возможностью сервера в данном случае некорректно.

Есть еще один важный момент. В вышеописанных примерах не указывалось, на каком пуле потоков будут проводиться вычисления. Существуют два варианта. Простой вариант — использовать глобальный контекст выполнения. Подключается он следующим образом:

```
import Scala.concurrent.ExecutionContext.Implicits.global
```

Второй вариант — создать свой пул потоков. Благодаря синтаксису языка Scala достаточно просто определить пул потоков, и он будет использоваться для вычислений:

```
implicit private val exContext = ExecutionContext.fromExecutor(
Executors.newFixedThreadPool(FUTURE_POOL_SIZE/*количество — 8*/))
```

Это происходит за счет слова **implicit** [5].

Преимущества футуров:

• вычисления описываются меньшим количеством кода, чем в случае пула потоков;

• не приходится явно использовать пул потоков;

• возможность сразу создать всю цепочку вычислений без промежуточных синхронизаций.

Недостатки:

• дополнительный слой абстракции затрудняет контроль над тем, что происходит на более низком уровне;

• синтаксис необычен и сложен для программиста, привыкшего использовать обычный пул потоков.

Тестирование. Тестирование каждой модели многопоточности проводилось посредством утилиты apache benchmark. В терминале выполнялась одна из команд:

```
ab -c 1 -n 50 -r http://localhost:8091/future
```

или

```
ab -c 1 -n 50 -r http://localhost:8091/thread
```

В этом случае на сервер отправлялись 50 запросов, а по окончании выводилась статистика, в которой среди прочих содержались следующие данные:

• время самого быстрого запроса;

• время самого долгого запроса;

• время выполнения всех запросов.

Помимо этого программой jvisualvm измерялся процент использования процессора (расход CPU).

Тестирование проводилось для пула потоков, футуров с глобальным пулом и футуров с созданным вручную пулом потоков. Для каждого случая A/B-тестирование будет проведено по 5 раз.

Результаты тестирования отображены в табл. 1.

Усредненные значения без минимального и максимального результатов, показаны в табл. 2.

Таблица 1

Результаты тестирования

Параметры	Потоки					Футуры									
						созданный пул потоков					глобальный пул потоков				
Общее время, с	39,74	35,09	37,162	39,092	39,385	37,83	39,172	39,858	42,171	37,436	44,879	43,099	38,78	37,914	40,046
Минимальное время, мс	760	654	706	741	746	714	767	765	824	738	791	829	742	707	778
Максимальное время, мс	1533	1223	1355	1180	1289	1235	1232	1201	1479	1166	3148	1456	1087	1505	1468
Расход CPU, %	78,3	77	76,1	77,2	73,7	76,6	77,8	76,4	75,9	73,6	74,8	73,4	75,6	76,5	77,6

Таблица 2

Усредненные значения

Параметры	Потоки	Футуры	
		созданный пул потоков	глобальный пул потоков
Среднее общее время, с	38,09	37,78	38,52
Среднее минимальное время, мс	721,4	712,2	734,8
Среднее максимальное время, мс	1316	1256,4	1258,2
Средний расход CPU, %	76,46	76,12	76,28

Из вышеописанных измерений видно, что:

• в среднем футуры с созданным пулом потоков быстрее справляются с поставленной задачей;

• самое быстрое время выполнения зафиксировано при решении задачи с использованием пула потоков;

• пул потоков больше всего использует CPU при меньшей скорости, значит расходует эффективность использования процессора у пула потоков меньше;

• худшие показатели у футуров с глобальным контекстом выполнения;

• разница в скорости выполнения между футурами и пулом потоков незначительная:

$$\Delta t = \frac{38{,}09 - 37{,}71}{38{,}09} \approx 0{,}01 \approx 1\ \%.$$

И так, были рассмотрены две разные модели многопоточности, использованные при решении задачи на языке Scala. В ходе написания кода и при проведении тестирования выяснилось, что у каждой модели есть свои преимущества и недостатки. С решением задачи обе модели справились практически одинаково, есть лишь небольшая разница в результатах.

При распараллеливании задачи с помощью пула потоков код получился более громоздким, а CPU расходуется менее эффективно. Однако именно таким способом в лучшем случае задача решилась за наименьшее время.

При выполнении этой задачи с помощью футуров получился более короткий код, что типично для функционального подхода. Не потребовалось вручную синхронизировать результаты выполнения подзадач и усыплять процесс. Тестирование показало, что при созданном вручную пула потоков в среднем задача решается немного быстрее. С использованием глобального пула потоков были получены худшие результаты. Футуры являются абстракцией над пулом потоков, и чем выше слои абстракции, тем больше скрытых опасностей хранит в себе подход.

Таким образом, хоть модель футуров выглядит проще и понятнее, нельзя использовать футуры, не разобравшись, что происходит внутри. Новичку лучше сначала поработать с пулом потоков и лишь потом начать работать с футурами. А опытному программисту точно надо попробовать эту модель многопоточности и функциональный стиль программирования в целом. Ведь чем больше вариантов решения задачи знает программист, тем эффективнее он сможет выполнить ее, выбрав лучший вариант.

Проект расположен в открытом git-репозитории: https://bitbucket.org/BitSII/multithreading_variants

Литература

1. *Шилдт Г.* Java 8. Полное руководство. М.: Вильямс, 2015. 1376 с.

2. *Язык* программирования Java SE 8. Подробное описание / Дж. Гослинг, Б. Джой, Г.Л. Стил, Г. Брача, А. Бакли. М.: Вильямс, 2015. 672 с.

3. *Eriksen M.* Effective Scala // URL: http://twitter.github.io/effectivescala/ (дата обращения 19.12.2016).
4. *Хорстманн К.С.* Функциональное программирование. SCALA для нетерпеливых. М.: ДМК Пресс, 2013. 408 с.
5. *Scala* documentation // URL: https://www.Scala-lang.org/documentation/ (дата обращения 19.12.2016).

Семенченко Иван Игоревич — студент кафедры «Системы обработки информации и управления» МГТУ им. Н.Э. Баумана, Москва, Российская Федерация.

Научный руководитель — М.В. Чёрненький, канд. техн. наук, доцент кафедры «Системы обработки информации и управления», МГТУ им. Н.Э. Баумана, Москва, Российская Федерация.

COMPARISON OF MULTITHREADING MODELS IN SCALA PROGRAMMING LANGUAGE

I.I. Semenchenko dev.ivansem@gmail.com

Bauman Moscow State Technical University, Moscow, Russian Federation

Abstract

The article deals with typical problems of multithreading and shows the use of multithreading in Scala programming language. The study tested two different models of multithreading — the one based on the thread pool and the other based on the futuras. In our research we give their comparative characteristics and indicate how the use of these models can help the programmer, and what difficulties may arise. Moreover, we describe advantages and disadvantages of each model and give the examples of the code, which clearly show how the concurrency problems are solved in each case. The efficiency of each model was tested on the example of a specific task

Keywords

Multithreading, Scala, thread pool, futuras

References

[1] Shildt G. Java 8. Polnoe rukovodstvo [Java 8. Full guide]. Moscow, Vil'yams Publ., 2015. 1376 p. (in Russ.).

[2] Gosling Dzh., Dzhoy B., Stil G.L., Bracha G., Bakli A. Yazyk programmirovaniya Java SE 8 [Programming language Java SE 8. Detailed description]. Moscow, Vil'yams Publ., 2015. 672 p. (in Russ.).

[3] Eriksen M. Effective Scala. Twitter: website. URL: http://twitter.github.io/effectivescala/ (accessed 19.12.2016).

[4] Khorstmann K.S. Funktsional'noe programmirovanie. SCALA dlya neterpelivykh [Functional programming. SCALA for the impatient]. Moscow, DMK Press Publ., 2013. 408 p. (in Russ.).

[5] Scala documentation. Scala-lang: website. URL: https://www.Scala-lang.org/documentation/ (accessed 19.12.2016).

Semenchenko I.I. — student of the Department of Information Processing and Control Systems, Bauman Moscow State Technical University, Moscow, Russian Federation.

Scientific advisor — M.V. Chernen'kiy, Cand. Sc. (Eng.), Assoc. Professor of the Department of Information Processing and Control Systems, Bauman Moscow State Technical University, Moscow, Russian Federation.