

Erlang: Подводим

Андрей Ушаков завершает серию практикумов по многозадачности, выставляя функциям оценки за производительность.



Наш
эксперт

Андрей Ушаков
активно приближает тот день, когда функциональные языки станут мейнстримом.

Данной статьей мы заканчиваем наш практикум по многозадачности, посвященный созданию разнообразных многозадачных реализаций функций `map` и `reduce`. Мы проделали большую работу и создали несколько различных многозадачных реализаций функций `map` и `reduce`. Теперь пришла пора их сравнить. Сравнивать мы их в этот раз будем только по одному критерию: производительности.

Прежде чем сравнивать производительность разных реализаций функций `map` и `reduce`, необходимо разработать методику их сравнения. Для измерения времени выполнения некоторой функции мы будем использовать функцию `timer:tc/1` из модуля `timer`. Эта функция принимает в качестве аргумента некоторую функцию `Fun` и возвращает кортеж из двух элементов `{Time, Value}`, где `Time` — время выполнения функции `Fun` в микросекундах, а `Value` — значение, возвращаемое этой функцией. Но измерять время выполнения некоторой функции при помощи функции `timer:tc/1` не так уж и просто, как может это показаться. Давайте, например, измерим время выполнения функции `lists:seq(1,5)`, которая генерирует список чисел от 1 до 5. Для этого запустим среду времени выполнения Erlang и введем в ней следующее выражение: `timer:tc(fun() -> lists:seq(1,5) end)`. Результатом этого выражения будет следующее значение: `{0, [1,2,3,4,5]}`, откуда следует, что время выполнения функции `lists:seq(1,5)` равно 0. На первый взгляд может показаться, что функция `timer:tc/1` работает неправильно, так как в действительности генерация списка `[1,2,3,4,5]` занимает некоторое время, а не происходит мгновенно. Однако если вспомнить, что время на компьютере обновляется при помощи таймера, имеющего определенную разрешающую способность, то все становится на свои места. В действительности оказывается, что время выполнения функции `lists:seq(1,5)` меньше разрешающей способности таймера; именно поэтому мы получаем 0 в качестве значения времени выполнения этой функции с данными аргументами.

Возникает вполне логичный вопрос: как нам измерить время выполнения некоторой быстрой функции? Для понимания этого давайте ответим на другой вопрос: как измерить толщину обычного листа бумаги, причем достаточно точно? Если мы будем решать эту задачу в лоб (т.е. возьмем линейку или штангенциркуль и будем измерять толщину одного листа бумаги), то ответ будет очевиден: никак. Однако если мы возьмем стопку из 1000 листов бумаги, измерим ее толщину и разделим полученное значение

на количество листов бумаги в стопке (в нашем случае на 1000), то мы получим толщину одного листа бумаги; это значение будет достаточно точным. Аналогичным образом мы можем измерить и время некоторой быстрой функции: выполнить эту функцию **N** раз подряд, измерить время выполнения и разделить полученное значение на **N**. И мы получим время выполнения интересующей нас функции, но не так точно, как ожидается. Все дело в том, что параллельно с нами работает такой системный компонент, как сборщик мусора [garbage collector], причем периоды его работы для нас случайны. Пусть во время выполнения некоторой функции **N** раз подряд сборка мусора произошла **M** раз. Пусть время однократного выполнения интересующей нас функции — t_f , а время сборки мусора — t_{gc} (для простоты считаем, что все сеансы сборки мусора занимают одинаковое время). Тогда, используя приведенную выше методику, мы получим следующее значение времени: $(N*t_f + M*t_{gc})/N = t_f + (M/N)*t_{gc}$, т.е. верхнюю границу для времени выполнения интересующей нас функции. С одной стороны, чем больше значение **N** относительно **M**, тем ближе полученное нами значение к истинному времени выполнения интересующей нас функции. С другой стороны, мы не можем никак влиять на сборку мусора, поэтому вряд ли нам удастся сделать число **N** сильно больше числа **M**. Если быть более точным, то мы можем запустить внеплановую сборку мусора при помощи функции `garbage_collect/0,1`, но не можем отменить запланированную сборку мусора (или совсем прекратить ее на время). Более того, сборка мусора вносит свой вклад во время выполнения всегда: и во время наших измерений, и во время реального выполнения некоторой функции. Поэтому под временем выполнения мы будем понимать время выполнения вместе с возможной сборкой мусора.

Помимо сборки мусора, существует еще множество факторов, влияющих на измерение времени работы некоторой функции. Эти факторы связаны с работой компьютерного «железа», операционной системы и среды времени выполнения Erlang. Так, например, выполнение интересующей нас функции на процессоре (или на ядре процессора) может быть вытеснено более приоритетной задачей (при выполнении на операционной системе с вытесняющей многозадачностью). Все эти факторы влияют на измерение времени работы некоторой интересующей нас функции, причем (как и в случае со сборкой мусора) в сторону увеличения измеренного значения. Погрешности, вносимые этими факторами, носят как систематический (потому что происходят периодически), так и случайный (потому что невозможно предсказать, когда эти факторы в следующий раз повлияют на наши измерения) характер. Систематическую часть погрешности мы не будем трогать; она вносит одинаковый вклад как в процесс измерения, так и в реальную работу некоторой функции. Просто под временем выполнения мы будем понимать время выполнения вместе со всеми «паразитными» вкладами (от сборщика мусора, от планировщика заданий и т.д.). А вот значение случайной погрешности мы можем уменьшить. Для этого нам нужно провести измерение интересующей нас величины несколько раз и вычислить ее среднее значение. Чем больше будет количество измерений интересующей нас

Систематическая погрешность

Систематическая погрешность — это погрешность, изменяющаяся во времени по определенному закону (частным случаем является постоянная погрешность, не изменяющаяся с течением времени). Систематические погрешности могут быть связаны с ошибками приборов

(неправильная шкала, калибровка и т.п.), не учтенными экспериментатором.

Систематическую погрешность нельзя устранить повторными измерениями. Ее устраняют либо с помощью поправок, либо посредством «улучшения» эксперимента (повышения его чистоты).

ИТОГИ

величины, тем точнее будет эта величина и тем меньше будет значение случайной части погрешности.

Давайте подведем промежуточный итог по тому, как мы будем измерять время выполнения интересующей нас функции. Во-первых, мы будем измерять время выполнения «пакета», состоящего из нескольких вызовов интересующей нас функции. Время выполнения интересующей нас функции мы получим, разделив время выполнения «пакета» на количество вызовов в «пакете». Во-вторых, вычислять время выполнения интересующей нас функции мы будем несколько раз, для уменьшения случайной части погрешности вычисления. По полученному набору значений времени выполнения мы будем вычислять среднее значение и среднеквадратичное отклонение (СКО). Среднеквадратичное отклонение поможет оценить нам случайную часть погрешности. Давайте эту методику измерения времени выполнения интересующей нас функции мы реализуем. Все функции, относящиеся к измерению времени выполнения, мы расположим в отдельном модуле **performance_tester**. А начнем мы с «пакетов», состоящих из нескольких вызовов интересующей нас функции. Функция **time_test_body/2** (внутренняя относительно модуля **performance_tester**) служит для выполнения «пакета» из вызовов интересующей нас функции **Fun**:

```
time_test_body(_Fun, 0) -> true;
time_test_body(Fun, TestCount) ->
    Fun(),
    time_test_body(Fun, TestCount - 1).
```

Функция **time_test/2** (внутренняя относительно модуля **performance_tester**) служит для формирования «пакета» вызовов, его выполнения и вычисления времени однократного выполнения интересующей нас функции **Fun**:

```
time_test(Fun, TestCount) ->
    {Time, _Value} = timer:tc(fun() -> time_test_body(Fun,
TestCount end),
    Time / TestCount.
```

В этой функции для измерения времени выполнения «пакета» мы используем функцию **timer:tc/1**. Следующая функция, которую мы рассмотрим — функция **time_test/4** для получения нескольких значений времени выполнения интересующей нас функции:

```
time_test(_Fun, _TestCount, 0, Values) -> Values;
time_test(Fun, TestCount, Count, Values) ->
    Time = time_test(Fun, TestCount),
    erlang:garbage_collect(),
    time_test(Fun, TestCount, Count - 1, [Time] ++ Values).
```

В реализации этой функции интересно обратить внимание на следующее: на вызов функции **erlang:garbage_collect/0** между измерениями значений времени выполнения функции. Это нужно для того, чтобы убрать весь мусор, возникший после выполнения «пакета». После того, как мы получаем несколько значений времени выполнения, нам необходимо вычислить среднее значение этого времени и среднеквадратичное отклонение этого среднего. Для этого мы создаем две следующие функции — **calc_mean/1** и **calc_standard_deviation/2** (также внутренние функции модуля **performance_tester**):

Случайная погрешность

Случайная погрешность — это составляющая погрешности измерения, изменяющаяся случайным образом в серии повторных измерений одной и той же величины, проведенных в одинаковых условиях. В появлении таких погрешностей не наблюдается какой-либо закономерности, они обнаруживаются при повторных измерениях одной и той же величины в виде некоторого разброса получаемых результатов. Случайные погрешности неизбежны, неустранимы и всегда присутствуют в результате измерения, однако их влияние, как правило, можно снизить статистической обработкой. Описание случайных погрешностей возможно только на основе теории случайных процессов и математической статистики. Основным свойством случайной погрешности является возможность уменьшения искажения искомой величины путем усреднения данных. Уточнение оценки искомой величины при увеличении количества измерений (повторных экспериментов) означает, что среднее случайной погрешности при увеличении объема данных стремится к 0 (закон больших чисел).

```
calc_mean(Values) ->
    lists:sum(Values) / length(Values).
calc_standard_deviation(Mean, Values) ->
    Sum = lists:foldl(fun (Value, Acc) -> Acc + (Value - Mean) *
    (Value - Mean) end, 0, Values),
    math:sqrt(Sum / length(Values)).
```

Функция **calc_mean/1** вычисляет среднее значение по набору значений; функция **calc_standard_deviation/2** вычисляет среднеквадратичное отклонение по набору значений и вычисленному среднему значению для этого набора. И, наконец, функция **performance_tester:time_test/3** является точкой входа (и экспорт-функцией из модуля **performance_tester**) для измерения времени выполнения:

```
time_test(Fun, TestCount, Count) ->
    TimeValues = time_test(Fun, TestCount, Count, []),
    Mean = calc_mean(TimeValues),
    Deviation = calc_standard_deviation(Mean, TimeValues),
    erlang:garbage_collect(),
    {Mean, Deviation}.
```

В эту функцию мы передаем функцию **Fun**, время выполнения которой мы хотим измерить, количество вызовов функции **Fun** в «пакете» **TestCount** и количество значений в наборе **Count** (для вычисления среднего и среднеквадратичного отклонения). По полученному набору значений времени выполнения мы вычисляем среднее значение, среднеквадратичное отклонение и возвращаем полученные результаты в виде кортежа из двух значений.

Договоримся о методике сравнения производительности разных реализаций функций **map** и **reduce**. В качестве исходных данных для реализаций функций **map** и **reduce** мы будем использовать список из целых чисел. Для минимизации размера этого списка следует использовать целые числа минимального размера. Минимальный размер целых чисел — 1 слово; на 32-битных системах целое число занимает 1 слово при условии, что его значение лежит в диапазоне от **-134217729** до **134217728**. Кроме того, для большей чистоты получаемых результатов, мы будем использовать не предопределенный список целых чисел, а список чисел,

»

» Не хотите пропустить номер? Подпишитесь на [www.linuxformat.ru/subscribe/!](http://www.linuxformat.ru/subscribe/)

Таблица 1

N	1	2	3	4a	4б	5	6	7
1000	2020 (30)	1990 (20)	4040 (30)	1110 (20)	1120 (30)	1130 (20)	2810 (50)	1140 (20)
10000	26000 (3000)	26000 (3000)	42400 (500)	6900 (200)	6900 (200)	6900 (300)	8900 (400)	7000 (200)
100000	360000 (30000)	358000 (3000)	520000 (7000)	71000 (2000)	71000 (2000)	84000 (6000)	51000 (2000)	90000 (4000)
1000000	4200000 (400000)	4200000 (300000)	5500000 (200000)	870000 (30000)	870000 (30000)	850000 (40000)	580000 (10000)	900000 (10000)

генерируемых случайным образом. Для этого мы будем использовать функцию `data_generator:generate_int_data/2`, определенную в модуле `data_generator` (см. исходные коды на диске). При построении такой зависимости мы будем проводить измерения при следующих размерах списка исходных данных: 1000, 10000, 100000, 1000000. При измерении производительности разных реализаций функции `map` в качестве функции отображения мы будем использовать функцию `fun math:sqrt/1`. При измерении производительности разных реализаций функции `reduce` в качестве функции свертки мы будем использовать функцию `fun(Number, Sum) -> Sum + math:sqrt(Number) end;` в качестве начального значения операции свертки мы будем брать 0.

Теперь давайте поговорим про получение результатов. Получать результаты производительности мы будем следующим образом: для каждой реализации функции `map` или `reduce` и для каждого размера исходного списка мы будем создавать новый экземпляр среды Erlang (или несколько, если необходимо). После получения результата мы будем завершать работу созданного экземпляра среды Erlang при помощи вызова функции (BIF) `halt/0`.

Последний шаг, который необходимо сделать перед тем, как переходить непосредственно к данным, это привести данные о конфигурации используемого оборудования. Для получения результатов использовались два компьютера со следующей конфигурацией: процессор Intel Core i5-2400 (с частотой 3,10 ГГц), память 8 ГБ ОЗУ, операционная система Ubuntu 13.04 64-бит, пропускная способность сети между компьютерами 100 Мбит/с.

Перейдем к полученным результатам. Таблица 1 содержит время выполнения разных реализаций функции `map` в зависимости от размера списка исходных данных.

Здесь столбец N содержит количество элементов (чисел) в списке исходных данных. Столбец 1 содержит время выполнения стандартной реализации: функции `lists:map/2`. Столбец 2 содержит время выполнения функции `parallel_map:usual_map/2`. Столбец 3 содержит время выполнения функции `parallel_map:simple_pmap/2`. Столбцы 4а и 4б содержат время выполнения функций `parallel_map:portion_pmap/3` и `parallel_map:portion_gen_pmap/3` соответственно. Столбец 5 содержит время выполнения функции `parallel_map:limited_pmap/4`. Столбец 6 содержит время выполнения функции `parallel_map:distributed_pmap/5`. И, наконец, столбец 7 содержит время выполнения функции `parallel_map:smartmsg_pmap/4`. Размеры порций данных (для данных из столбцов 4–7) устанавливаются следующим образом: 250 элементов при размере исходных данных 1000 чисел, 2500 элементов при размере исходных данных 10000 чисел и 10000 элементов во всех остальных случаях. При получении данных из столбцов 5 и 7 мы использовали 4 рабочих процесса. При получении данных из столбца 6 мы использовали 2 узла с 2 рабочими процессами на каждом из них.

В таблице 2 столбец N содержит количество элементов (чисел) в списке исходных данных. Столбец 1 содержит время выполнения стандартной реализации: функции `lists:foldl/3`. Столбец 2 содержит время выполнения функции `parallel_reduce:usual_reduce/2`. Столбцы 3а и 3б содержат время выполнения функций `parallel_reduce:portion_reduce/4` и `parallel_reduce:portion_gen_reduce/4` соответственно. Столбец 4 содержит время выполнения функции `parallel_reduce:limited_reduce/5`. Столбец 5 содержит время выполнения функции `parallel_reduce:distributed_reduce/6`. И, наконец, столбец 6 содержит время выполнения функции `parallel_reduce:smartmsg_reduce/5`. Размеры порций данных (для данных из столбцов 3–6) устанавливаются следующим образом: 250 элементов при размере исходных данных 1000 чисел, 2500 элементов при размере исходных данных 10000 чисел и 10000 элементов во всех остальных случаях. При получении данных из столбца 5 мы использовали 2 узла с 2 рабочими процессами на каждом из них. Все значения в обеих таблицах приведены в микросекундах.

Теперь мы можем перейти к цели этой статьи: к сравнению производительности разных реализаций функций `map` и `reduce` на основе приведенных выше данных. Из приведенных выше данных и условий тестирования производительности можно сделать следующие выводы:

1 Для всех измеренных данных о времени выполнения мы посчитали среднеквадратичное отклонение (СКО). Значение СКО приведено в скобках за соответствующим значением времени выполнения в той же ячейке таблицы. Для всех измеренных данных значение СКО не больше 10 % от соответствующего значения времени выполнения. Это означает, что случайная погрешность находится в допустимых рамках. Что же касается систематической погрешности, то факторы, которые влияют на значение систематической погрешности в наших измерениях, точно так же влияют и на время выполнения в реальной жизни. Поэтому мы можем считать, что полученное нами значение времени выполнения близко к своему значению при реальной работе.

2 Стандартная реализация `lists:map/2` и обычная, не многозадачная реализация `parallel_map:usual_map/2` функции `map` являются наименее эффективными (за одним единственным исключением), что ожидаемо. Это означает, что их выполнение занимает наибольшее время по сравнению со всеми другими реализациями. При этом стандартная реализация `lists:map/2` и обычная, не многозадачная реализация `parallel_map:usual_map/2` функции `map` одинаково эффективны. То же самое справедливо и для реализаций `lists:foldl/3` и `parallel_reduce:usual_reduce/2` функции `reduce`.

3 Никогда, никогда, никогда не используйте реализацию `parallel_map:simple_pmap/2` функции `map`. Это приведет

Таблица 2

N	1	2	3а	3б	4	5	6
1000	3810 (40)	3720 (20)	1810 (30)	1800 (30)	1860 (20)	2440 (60)	1850 (20)
10000	38900 (100)	38300 (300)	11000 (100)	11000 (200)	11000 (100)	5600 (300)	10800 (200)
100000	383000 (2000)	385000 (3000)	104000 (3000)	106000 (2000)	122000 (3000)	28000 (1000)	117000 (2000)
1000000	3850000 (10000)	3890000 (30000)	1090000 (2000)	1090000 (4000)	1230000 (9000)	330000 (5000)	1040000 (20000)

» Пропустили номер? Узнайте на с. 108, как получить его прямо сейчас.

к преждевременной «пессимизации» производительности, т.к. время выполнения этой функции сильно больше, чем время выполнения обычных не многозадачных реализаций.

❸ Реализация на основе порций данных `parallel_map:portion_pmap/3` и обобщенная реализация на основе порций данных `parallel_map:portion_gen_pmap/3` функции `map` имеют одинаковую производительность. Это означает, что переход от обычной реализации на основе порций к обобщенной реализации не вносит никаких дополнительных затрат ко времени выполнения. Все наши дальнейшие реализации также основываются на обобщенном подходе. Поэтому мы можем быть уверены, что данный подход не вносит никаких дополнительных затрат ко времени выполнения ни для какой из реализаций. То же самое справедливо и для всех реализаций на основе порций функции `reduce`.

❹ Сравнение реализаций на основе порций `parallel_map:portion_gen_pmap/3` с реализацией с ограничениями на количество рабочих процессов `parallel_map:limited_pmap/4` функции `map` на самом деле не тривиально. Если посмотреть на приведенные данные, то может показаться, что реализация `parallel_map:portion_gen_pmap/3` эффективнее, чем `parallel_map:limited_pmap/4`. Но не стоит забывать, что при получении времени выполнения для реализации `parallel_map:limited_pmap/4` мы использовали 4 рабочих процесса (по числу ядер на компьютере). Подбор необходимого количества рабочих процессов для минимизации времени выполнения реализации `parallel_map:limited_pmap/4` следует для каждой задачи и конфигурации компьютера выполнять индивидуально. Так, например, для исходного списка из 1 000 000 элементов и размера порции в 100 элементов время выполнения реализации `parallel_map:portion_gen_pmap/3` будет 10,81 секунд. Время выполнения реализации `parallel_map:limited_pmap/4` с 4 рабочими процессами будет 10,97 секунд, а время выполнения реализации `parallel_map:limited_pmap/4` с 64 рабочими процессами будет 10,67 секунд. Все это справедливо и для реализаций `parallel_reduce:portion_gen_reduce/4` и `parallel_reduce:limited_reduce/5` функции `reduce`.

❺ Реализация функции `map` с созданием заданий по мере необходимости `parallel_map:smartmsg_pmap/4` менее эффективна, чем реализация с созданием всех заданий сразу `parallel_map:limited_pmap/4`. Однако разница в производительности достаточно незначительна (например, при размере исходного списка в 1 000 000 элементов разница в производительности будет менее 6%). Поэтому, если нет жестких требований к производительности, которым реализация `parallel_map:smartmsg_pmap/4` не удовлетворяет (а это означает, что реализация `parallel_map:limited_pmap/4` удовлетворяет этим требованиям впритык, без особого запаса), то лучше использовать ее. Причина этого в том, что реализация `parallel_map:smartmsg_pmap/4` не создает таких пиковых нагрузок на память и сеть, как реализация `parallel_map:limited_pmap/4`. Все это справедливо и для реализаций `parallel_reduce:smartmsg_reduce/5` и `parallel_reduce:limited_reduce/5` функции `reduce`.

❻ Распределенная реализация `parallel_map:distributed_pmap/5` функции `map` гораздо эффективнее всех остальных реализаций для списков исходных данных большого размера. Очевидно, в этом случае выгоды от распределения вычислений на нескольких компьютерах превышают затраты на передачу данных по сети. То же самое справедливо и для реализаций `parallel_reduce:distributed_reduce/6` функции `reduce`.

❼ Данные о производительности реализаций функции `map` на основе пула узлов `parallel_map:pool_pmap/5` и `parallel_map:pool_pmap_impl/4` не присутствуют в нашей таблице. Связано это с ошибкой реализации самого пула узлов в модуле `pool`: функция `pool:spawn_link/3` создание процесса и создание связи с этим процессом делает не атомарно. И проявляется эта ошибка при попытке использовать пул узлов с достаточно быстрыми задачами, как у нас. То же самое справедливо и для реализаций `parallel_`

Среднеквадратичное отклонение

Среднеквадратичное отклонение — в теории вероятностей и статистике наиболее распространенный показатель рассеивания значений случайной величины относительно ее математического ожидания. Измеряется в единицах измерения самой случайной величины. Равно корню квадратному из дисперсии случайной величины:

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - m)^2}$$

Здесь σ — среднеквадратичное отклонение, n — объем выборки, x_i — i -й элемент выборки, m — среднее арифметическое выборки.

`reduce:pool_reduce/6` и `parallel_reduce:pool_reduce_impl/5` функции `reduce`.

❽ И, наконец, давайте скажем пару слов о плате за повышение эффективности. Любой многозадачный алгоритм и любая многозадачная реализация обычно сложнее соответствующих однозадачных алгоритмов и реализаций. Кроме того, и объем кода многозадачных реализаций обычно больше. Так, например, однозадачная реализация `parallel_map:usual_map/2` содержит всего одно выражение, тогда как многозадачные реализации функции `map` существенно больше. Это означает, что сложность тестирования, сопровождения, дальнейшего развития многозадачных реализаций больше, чем соответствующих однозадачных реализаций.

Давайте скажем пару слов о потреблении памяти. Сделать это можно следующим образом: вычислить количество используемой памяти до и после вызова (например, при помощи функции `erlang:memory/1` с параметром `processes_used`), после чего получить их разность. Проблема в том, что полученное значение может быть любым и никак не связанным с реальным потреблением памяти. Причина этого в сборщике мусора. Поэтому более правильным было бы определение пикового потребления памяти в процессе выполнения функции, а также зависимости потребления памяти во времени. Получение подобных данных — это тема отдельной статьи; в будущем мы вернемся к этому вопросу.

Вот мы и закончили наш практикум по многозадачности. Но это не означает, что мы закончили вообще разговор про многозадачность: к этой теме мы еще не раз вернемся. Точно так же мы не раз еще вернемся и к этой задаче: к созданию разнообразных реализаций функций `map` и `reduce`, конечно, в рамках нескольких других вопросов. А в следующий раз мы начнем разговор о тестировании программного обеспечения самим разработчиком. [LXF](#)

Хронометраж для функций

Время выполнения интересующей нас функции в языке Erlang можно измерить одним из следующих способов:

» Использовать семейство функций `timer:tc/1,2,3` из модуля `timer` (что мы и делаем в нашей статье).

» Измерить время до и после выполнения интересующей нас функции при помощи функции (BIF) `now/0`, после чего посчитать разность полученных значений (например, при помощи функции `timer:now_diff/2` из модуля `timer`).

» Измерить время до и после выполнения интересующей нас функции при помощи функции `os:timestamp/0` из модуля `os`, после чего посчитать разность полученных значений (например, при помощи функции `timer:now_diff/2` из модуля `timer`).

» Измерить время до и после выполнения интересующей нас функции при помощи функции (BIF) `statistics/1`, после чего посчитать разность полученных значений.