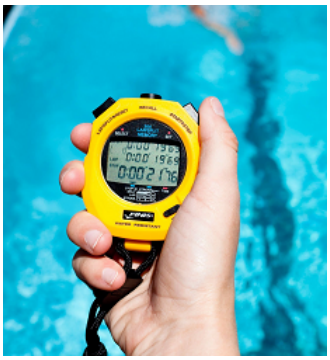
Luxoft
Компания
рейтинг
106,62

18 апреля в 20:22

Разработка → Шпаргалка Java программиста 7.2 Типовые задачи: Обход Map'ы, подсчет количества вхождений подстроки

Разработка веб-сайтов*, Программирование*, Java*, GitHub, Блог компании Luxoft



У меня есть хобби: я собираю различные решения типовых задач в Java, которые нахожу в инете, и пытаюсь выбрать наиболее оптимальное по размеру/производительности/элегантности. В первую очередь по производительности. Давайте рассмотрим такую типовые задачи, которые часто встречаются в программировании на Java как "обход Map'ы" и подсчет количества вхождений строк, разные варианты их решений (включая "красивые" и не очень) и их производительность.

Английские версии можно найти на Stackoverflow: по обходу map'ы и по подсчету вхождений подстрок.
Так же советую посмотреть мой opensource проект useful-java-links — возможно, наиболее полная коллекция полезных Java библиотек и фреймворков.

Общее оглавление 'Шпаргалок'

1. Обход Map'ы

Итак, если поискать в интернете найдется с десяток разных способов выполнить обход map'ы в общем случае, когда нам важны и ключи и значения (естественно, обход просто ключей или значений делается элементарно). Часть из них отличается лишь синтаксическим сахаром, часть принципиально разные. На самом деле, конечно, какие решения медленные и какие быстрые давно известно, но все-таки интересно сколько у нас получится в попугаях.

Для полноты картины рассмотрим не только обход обычных Map, но и их аналогов из **IterableMap** из `Apache Collections` и **MutableMap** of `Eclipse (CS) collections`.

Давайте посмотрим на варианты:

Условия задачи, пусть у нас будет Map'а чисел, будем искать сумму всех ключей и всех значений у этой Map'ы. Такая задача с одной стороны максимально упрощена, с другой гарантирует нам что оптимизатор Java не сможет выкинуть часть кода.

1. Используя **iterator**, **Map.Entry** и цикл while. Не самый красивый вариант, по сути аналог варианта (2), но посмотрим насколько они отличаются.

```
long i = 0;
Iterator<Map.Entry<Integer, Integer>> it = map.entrySet().iterator();
while (it.hasNext()) {
    Map.Entry<Integer, Integer> pair = it.next();
    i += pair.getKey() + pair.getValue();
}
```

2. Используя **Map.Entry** и цикл **foreach**. Классический вариант обхода map'ы.

```
long i = 0;
for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
    i += pair.getKey() + pair.getValue();
}
```

3. Используя **foreach** из `Java 8`. Посмотрим насколько производителен новый способ, появившийся в Java 8.

```
final long[] i = {0};
map.forEach((k, v) -> i[0] += k + v);
```

4. Используя **keySet** и **foreach**. Считается, что данный способ очень медленный, вопрос только на сколько.

```
long i = 0;
for (Integer key : map.keySet()) {
    i += key + map.get(key);
}
```

5. Используя **keySet** и **iterator**. По сути, вариант 4 только без синтаксического сахара.

```
long i = 0;
Iterator<Integer> itr2 = map.keySet().iterator();
while (itr2.hasNext()) {
    Integer key = itr2.next();
    i += key + map.get(key);
}
```

6. Используя **for** и **Map.Entry**. Аналог 1 и 2, только в "старом стиле"

```
long i = 0;
for (Iterator<Map.Entry<Integer, Integer>> entries = map.entrySet().iterator(); entries.hasNext(); ) {
    Map.Entry<Integer, Integer> entry = entries.next();
    i += entry.getKey() + entry.getValue();
}
```

7. Используя `Java 8` и **Stream Api**

```
final long[] i = {0};
map.entrySet().stream().forEach(e -> i[0] += e.getKey() + e.getValue());

// или другой вариант
```

7a. Используя `Java 8` и **Stream Api** с mapToLong и sum

```
map.entrySet().stream().mapToLong(e -> e.getKey() + e.getValue()).sum();
```

8. Используя `Java 8` и параллельный **Stream Api**

```
final long[] i = {0};
map.entrySet().stream().parallel().forEach(e -> i[0] += e.getKey() + e.getValue()); // не совсем корректно, так как результат и
зменяется без синхронизации, может использовать только для замеров, см вариант 8a
```

8a. Используя `Java 8` и параллельный **Stream Api** с mapToLong и sum

```
map.entrySet().parallelStream().mapToLong(e -> e.getKey() + e.getValue()).sum();
```

9. Используя **IterableMap** от `Apache Collections`. Данная map'а отличается необычным способом обхода.

```
long i = 0;
MapIterator<Integer, Integer> it = iterableMap.mapIterator();
while (it.hasNext()) {
    i += it.next() + it.getValue();
}
```

10. Используя **MutableMap** от `Eclipse collections` (бывшие GS коллекции). Данная map'а получила свой forEach метод намного раньше чем появилась Java 8.

```
final long[] i = {0};
mutableMap.forEachKeyValue((key, value) -> {
    i[0] += key + value;
});
```

Тесты производительности

> **Типовое предупреждение**: Все результаты даны как есть, замеры производительности вещь сложная, в вашей системе все числа могут быть совсем другие, поэтому не стоит мне верить, исходные коды на github'e, всегда можно получить результаты самостоятельно. Если считаете, что я где-то ошибся при замерах производительности (а это всегда возможно), буду благодарен если напишите в личку или комментарии.

Режим = среднее время (AverageTime), система = Win 8.1 64-bit, Intel i7-4790 3.60GHz 3.60GHz, 16 GB, чем меньше score тем лучше)

1) Для небольших map (100 элементов), score 0.312 — лучшее значение

```
           Benchmark       Size Mode    Cnt     Score           Error  Units
3. ForEachAndJava8          100   avgt    100 0.312   ±   0.003   us/op
10. EclipseMap              100   avgt    100 0.354   ±   0.003   us/op
2. ForEachAndMapEntry       100   avgt    100 0.403   ±   0.005   us/op
1. WhileAndMapEntry         100   avgt    100 0.427   ±   0.006   us/op
6. ForAndIterator           100   avgt    100 0.427   ±   0.006   us/op
7a Java8StreamApi2              100  avgt    100    0.509 ±   0.036  us/op
7. Java8StreamApi           100   avgt    100 0.529   ±   0.004   us/op
9. ApacheIterableMap        100   avgt    100 0.585   ±   0.008   us/op
4. KeySetAndForEach         100   avgt    100 0.937   ±   0.011   us/op
5. KeySetAndIterator        100   avgt    100 0.94    ±   0.011   us/op
8. Java8StreamApiParallel 100   avgt    100 6.066   ±   0.051   us/op
8a. Java8StreamApiparallel2     100  avgt    5    9.468 ±   0.333  us/op
```

2) Для средних map с 10000 элементами, score 35.301 — лучшее значение

```
          Benchmark      Size   Mode   Cnt    Score          Error  Units
10. EclipseMap           10000  avgt   100 35.301  ±   0.697   us/op
3. ForEachAndJava8       10000  avgt   100 39.797  ±   0.309   us/op
2. ForEachAndMapEntry    10000  avgt   100 43.149  ±   0.313   us/op
1. WhileAndMapEntry      10000  avgt   100 43.295  ±   0.314   us/op
6. ForAndIterator        10000  avgt   100 44.009  ±   0.379   us/op
7. Java8StreamApi        10000  avgt   100 49.378  ±   0.415   us/op
5. KeySetAndIterator     10000  avgt   100 97.844  ±   0.896   us/op
4. KeySetAndForEach      10000  avgt   100 99.317  ±   0.862   us/op
8. Java8StreamApiParallel 10000  avgt   100 112.364 ±   0.167   us/op
9. ApacheIterableMap     10000  avgt   100 138.379 ±   1.387   us/op
```

3) Для map с 30000 элементами, score 122.277 — лучшее значение

```
          Benchmark      Size  Mode  Cnt        Score          Error  Units
8a. Java8StreamApiparallel2   30000  avgt    100   95.444 ±  13.706  us/op
10. EclipseMap           30000 avgt    100 122.277 ±   3.896   us/op
3. ForEachAndJava8       30000 avgt    100 136.906 ±   2.392   us/op
2. ForEachAndMapEntry    30000 avgt    100 145.845 ±   1.895   us/op
1. WhileAndMapEntry      30000 avgt    100 149.186 ±   2.621   us/op
6. ForAndIterator        30000 avgt    100 149.353 ±   2.427   us/op
7a. Java8StreamApi2          30000  avgt    100  180.803 ±  53.062  us/op
7. Java8StreamApi        30000 avgt    100 181.114 ±   3.272   us/op
8. Java8StreamApiParallel 30000 avgt    100 342.546 ±   1.206   us/op
5. KeySetAndIterator     30000 avgt    100 350.564 ±   8.662   us/op
4. KeySetAndForEach      30000 avgt    100 364.362 ±   9.416   us/op
9. ApacheIterableMap     30000 avgt    100 536.749 ±  25.819   us/op
```

Обратите внимание случай 8a (параллельный стрим с mapToInt и sum) показал хорошие результаты именно из-за работы с числовыми данными, в реальных задач он далеко не всегда применим.

График (тесты в зависимости от размера map'ы)

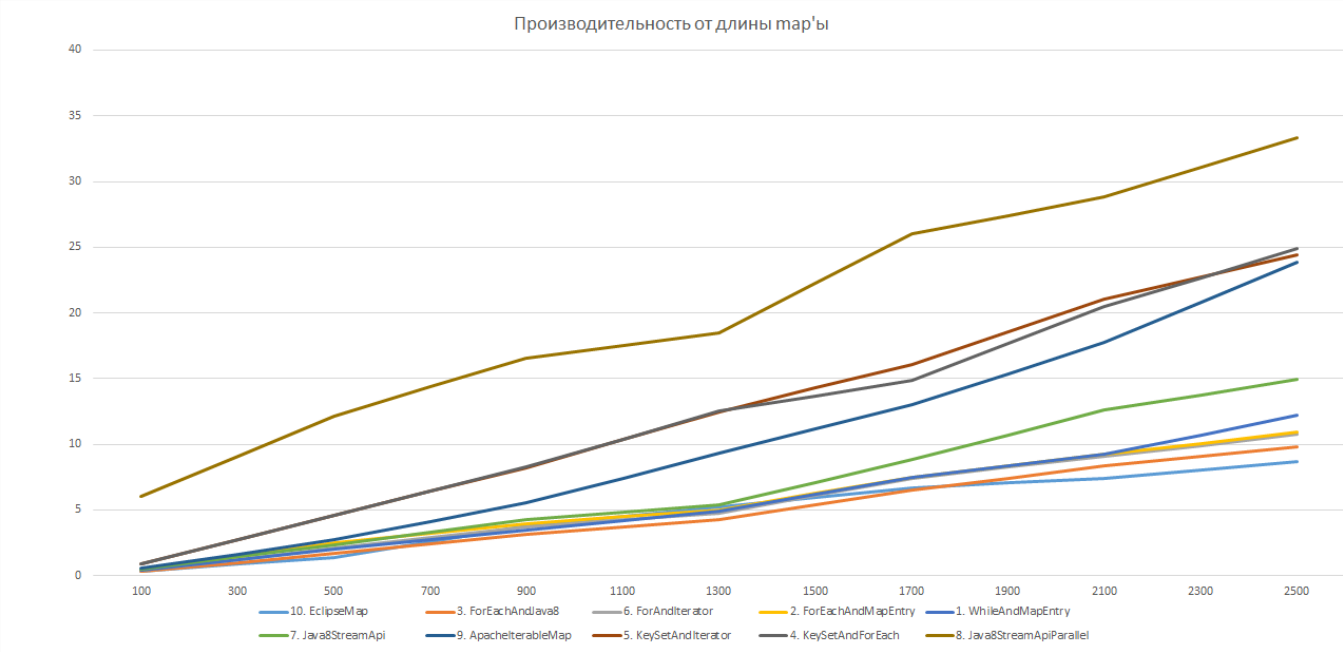


Таблица (тесты в зависимости от размера map'ы)

```
     Benchmark            100    500   900 1300    1700    2100    2500
10. EclipseMap           0.354  1.384  3.816   3.304   6.68    7.427   8.712
3. ForEachAndJava8       0.312  1.692  3.143   4.265   6.506   8.343   9.821
6. ForAndIterator        0.427  2.089  3.746   4.776   7.407   9.091   10.753
2. ForEachAndMapEntry    0.403 2.536    3.951  5.028   7.503   9.211   10.918
1. WhileAndMapEntry      0.427  2.026  3.4815  4.937   7.511   9.217   12.22
7. Java8StreamApi        0.529  2.343  4.264   5.399   8.826   12.633  12.918
```

```
9. ApacheIterableMap      0.585 2.725    5.574  9.292   13.01   17.719  23.882
5. KeySetAndIterator      0.94   4.592  8.24    12.496  16.077  21.012  24.389
4. KeySetAndForEach       0.937  4.572  8.251   12.522  14.831  20.502  24.881
8. Java8StreamApiParallel 6.066 12.152   16.563 18.512  25.987  28.813  33.336
```

Все тесты на github

Предупреждение: Обход HashMap, EclipseMap и ApacheIterableMap рассматривался только для типового случая когда коллизий нет или почти нет, в случае когда коллизий много — результаты могут быть совсем другими. Так же не рассмотрены случаи добавления элементов, случайного доступа и т.п., то есть только по данному тесту нельзя судить какая map'а быстрее и лучше в целом,

2. Подсчет количество вхождений подстроки в строку

Итак возьмем следующую подстроку и найдем все точки в ней:

```
    String testString = "a.b.c.d";
```

Хотел сразу предупредить: я взял для тестирования все варианты, предложенные в теме StackOverflow, здравый смысл подсказывает, что часть из них далеко не оптимальное забивание гвоздей любимым микроскопом, но мне было просто любопытно что получится в каждом случае. Поэтому, учитывайте, что **далеко не все варианты ниже стоит использовать в реальном приложении**!

1) Используя **Apache Commons**

```
int apache = StringUtils.countMatches(testString, ".");
System.out.println("apache = " + apache);
```

2) Используя **Spring Framework's**

```
int spring = org.springframework.util.StringUtils.countOccurrencesOf(testString, ".");
System.out.println("spring = " + spring);
```

3) Используя **replace**

```
int replace = testString.length() - testString.replace(".", "").length();
System.out.println("replace = " + replace);
```

4) Используя **replaceAll** (case 1)

```
int replaceAll = testString.replaceAll("[^.]", "").length();
System.out.println("replaceAll = " + replaceAll);
```

5) Используя **replaceAll** (case 2)

```
int replaceAllCase2 = testString.length() - testString.replaceAll("\\.", "").length();
System.out.println("replaceAll (second case) = " + replaceAllCase2);
```

6) Используя **split**

```
int split = testString.split("\\.",-1).length-1;
System.out.println("split = " + split);
```

7) Используя **Java8** (case 1). **Обратите внимание** в отличии от остальных вариантов тут возможен поиск только символов, не подстрок.

```
long java8 = testString.chars().filter(ch -> ch =='.').count();
System.out.println("java8 = " + java8);
```

8) Используя **Java8** (case 2), возможно несколько лучше в случае unicode строки чем case 1. **Обратите внимание** в отличии от остальных вариантов тут возможен поиск только символов, не подстрок.

```
long java8Case2 = testString.codePoints().filter(ch -> ch =='.').count();
System.out.println("java8 (second case) = " + java8Case2);
```

9) Используя **StringTokenizer**

```
int stringTokenizer = new StringTokenizer(" " +testString + " ", ".").countTokens()-1;
System.out.println("stringTokenizer = " + stringTokenizer);
```

Последний вариант, несколько некорректен, так как две точки подряд будут считаться за одну, то есть для a...b.c....d or ...a.b.c.d или a....b.....c.....d… несколько точек будут считаться за одну.

> **Типовое предупреждение**: Все результаты даны как есть, замеры производительности вещь сложная, в вашей системе все числа могут быть совсем другие, поэтому не стоит мне верить, исходные коды на github'e, всегда можно получить результаты самостоятельно. Если считаете, что я где-то ошибся при замерах производительности (а это всегда возможно), буду благодарен если напишите в личку или комментарии.

Все тесты на github

Результаты замеров на короткой строке (используя JMH, mode = AverageTime, значение `0.010` лучше чем `0.351`):

```
Benchmark              Mode  Cnt  Score    Error  Units
1. countMatches        avgt    5  0.010 ±  0.001  us/op
2. countOccurrencesOf  avgt    5  0.010 ±  0.001  us/op
3. stringTokenizer     avgt    5  0.028 ±  0.002  us/op
4. java8_1             avgt    5  0.077 ±  0.005  us/op
5. java8_2             avgt    5  0.078 ±  0.003  us/op
6. split               avgt    5  0.137 ±  0.009  us/op
7. replaceAll_2        avgt    5  0.302 ±  0.047  us/op
8. replace             avgt    5  0.303 ±  0.034  us/op
9. replaceAll_1        avgt    5  0.351 ±  0.045  us/op
```

Результаты замеров на длинной строке длиной в 2142 символов (используя JMH, mode = AverageTime, значение `0.010` лучше чем `0.351`):

```
Benchmark              Mode  Cnt   Score   Error  Units
1. countMatches        avgt    5   2.392 ± 0.172  us/op
2. countOccurrencesOf  avgt    5   2.362 ± 0.060  us/op
3. stringTokenizer     avgt    5   5.931 ± 0.112  us/op
4. java8               avgt    5   9.626 ± 0.463  us/op
5. java8_1             avgt    5   8.586 ± 0.251  us/op
6. split               avgt    5  21.201 ± 1.037  us/op
7. replaceAll2         avgt    5  26.614 ± 1.026  us/op
8. replaceAll1         avgt    5  31.505 ± 1.046  us/op
9. replace             avgt    5  33.462 ± 2.329  us/op
```

P.S. Английские версии можно найти на Stackoverflow: по обходу map'ы и по подсчету вхождений подстрок, исходные коды в моем проекте useful-java-links. Буду благодарен плюсам в SO или github'e, если статья понравилась и считаете, что они заслужены.
P.P.S. Так же советую посмотреть мой opensource проект useful-java-links — возможно, наиболее полная коллекция полезных Java библиотек, фреймворков и русскоязычного обучающего видео. Так же есть аналогичная английская версия этого проекта и начинаю opensource подпроект Hello world по подготовке коллекции простых примеров для разных Java библиотек в одном maven проекте (буду благодарен за любую помощь).

Общее оглавление 'Шпаргалок'

java, map, string

+16 28,3k 319

Автор: @vedenin1980

рейтинг
Luxoft 106,62
Сайт Twitter

Похожие публикации

+29 Шпаргалка Java-программиста 5. Двести пятьдесят русскоязычных обучающих видео докладов и лекций о Java
96,8k 1205 26

+25 Шпаргалка Java программиста 4. Java Stream API
116k 727 17

+57 Шпаргалка Java программиста 3. Коллекции в Java (стандартные, guava, apache, trove, gs-collections и другие)
105k 1126 39

Комментарии (18)

lany 18 апреля 2016 в 22:46 +3

Как насчёт такого?

```
String testString = "a.b.c.d";
int count = 0;
for(int i=0; i<testString.length(); i++) if(testString.charAt(i) == '.') count++;
```

Вероятно, примерно это и написано внутри библиотечных методов Apache и Spring. Если этими библиотеками пользоваться нельзя, то будет полным идиотизмом использовать любой из вариантов 3-9. Я даже не представляю, как эти варианты могут прийти в голову. Надо просто написать трёхстрочный метод в своём утилитном классе и использовать.

vedenin1980 18 апреля 2016 в 23:50 0

> Как насчёт такого?

Речь шла не о поиске символов, а о поиске подстрок. Что несколько сложнее написать простым for'ом руками (хотя и возможно).

> о будет полным идиотизмом использовать любой из вариантов 3-9. Я даже не представляю, как эти варианты могут прийти в голову.

согласен, но все эти варианты были предложены и заплюсованы на SO. Впрочем, понятно что большая часть не столько реальные варианты, сколько способ забить гвозди любимым микроскопом. А мне было интересно проверить все варианты даже те где здравый смысл подсказывает что они очень неэффективные.

alex_kir 19 апреля 2016 в 13:20 0

> Речь шла не о поиске символов, а о поиске подстрок. Что несколько сложнее написать простым for'ом руками (хотя и возможно).

Особенно варианты 4, 7, 8. Их на работу со строками не переделаешь.

vedenin1980 19 апреля 2016 в 13:38 0

А почему вариант 4 не переделать? Который replaceAll? В 7 и 8 я же написал замечание что это только для поиска символов.

alex_kir 19 апреля 2016 в 16:58 +1

как например для строки «ааабббааабббб» найти количество вхождений «аб» используя конструкцию с крышечкой?

lany 18 апреля 2016 в 23:01 +3

Обход мэпки с параллельным стримом некорректен, так как небезопасно модифицируется разделённое состояние. Смысл измерять эту версию, если она выдаёт мусорный результат? Для параллельных стримов нужна редукция, а не `forEach`.

Не понимаю также смысла сравнения `HashMap` с `IterableMap` и `UnifiedMap`. Способ хэширования разный, способ разрешения коллизий разный. Общее количество коллизий в каждом случае неизвестно. Если уж проводить сравнение разных структур данных (а не разных способов обхода одной структуры), то надо серьёзнее подходить: посмотреть разные наборы входных данных, разные операции, не только итерацию, но и случайный доступ. Например, известно, что `HashMap` оптимизирует случайный доступ в случае большого числа коллизий, что, разумеется, добавляет небольшие накладные расходы на последовательный обход. `UnifiedMap` такой оптимизации не содержит и может не делать дополнительных проверок.

vedenin1980 18 апреля 2016 в 23:55 0

> Не понимаю также смысла сравнения HashMap с IterableMap и UnifiedMap.

Так сознательно не было никаких коллизий и случайный доступ не учитывался, только обход простой структуры без коллизий. Это не в коему случае не говорит что IterableMap и UnifiedMap хуже или лучше в принципе. Да, чуть позже попробую поиграться обходом при разных коллизиях.

terryP 19 апреля 2016 в 00:18 *(комментарий был изменён)* 0

дел

gena_glot 19 апреля 2016 в 01:07 0

По идее варианты 2 и 3 должны быть идентичны, не думаю что разработчики оракл настолько умны что добавив лямбды переписали все заново, придумав какой-то сим салавим. Если и не идентичны, то где-то рядом. В одном случае вижу массив long[], в другом просто long. Плюс непонятно как реализованы лямбды точно. Но по идее должны быть рядом.

vedenin1980 19 апреля 2016 в 01:36 *(комментарий был изменён)* +1

Если вы про обход мар, то так и есть, они рядом. Расхождения достаточно небольшие по большому счету, близкие к пределам статической погрешности.

lany 19 апреля 2016 в 10:00 +3

Не должны быть. В том-то и дело, что внутренняя итерация быстрее внешней: там есть, что оптимизировать. Не надо инкапсулировать состояние в итераторе, можно всё держать в локальных переменных. Не надо проверять modCount на каждой итерации. Разница может быть небольшая, но статистически значима.

Sirikid 19 апреля 2016 в 02:52 +2

Тестировать производительность потенциально некорректного кода? Пффф.
Почему такая приверженность к forEach, не все ли равно как в итоге Stream API обойдет эту мапу изнутри?

vedenin1980 19 апреля 2016 в 03:15 +1

> Тестировать производительность потенциально некорректного кода? Пффф.

Можете тесты параллельного стрима не обращать внимания.

> Почему такая приверженность к forEach, не все ли равно как в итоге Stream API обойдет эту мапу изнутри?

Завтра перепишу на использование sum() и числовых вместо forEach, но принципиальная разница тут вряд ли будет по сравнению с forEach.

vedenin1980 19 апреля 2016 в 04:13 +3

Исправил варианты с Stream API

Sirikid 19 апреля 2016 в 16:23 0

Спасибо. ЧТД, если данных много и обрабатывать их можно параллельно StreamAPI поможет это сделать в одну строчку.

afanasiy_nikitin 19 апреля 2016 в 16:30 0

в первом примере (с суммой) правильнее использовать коллектор, не?
int sum = map.keySet().stream().mapToInt(k -> k + map.get(k)).sum()

vedenin1980 19 апреля 2016 в 16:33 *(комментарий был изменён)* 0

keySet с последующим get(k), обычно работает медленнее entrySet (это видно в замерах выше). Сомневаюсь что в stream API что-то измениться

afanasiy_nikitin 23 апреля 2016 в 11:42 *(комментарий был изменён)* 0

я правильно понимаю, что с вариант с .sum() (8a) в итоге оказался самым быстрым? если да, то исправьте пожалуйста фразу «score 122.277 — лучшее значение» и поправьте графики

по поводу применимости на реальных задачах: если вам нужно посчитать сумму всех элементов — то это и есть «реальная задача», для этого и делался метод sum(); если вам кажется что ваши тесты слишком «синтетические», то придумайте другие тесты.

и еще, удалите пожалуйста из статьи примеры параллельной работы на внешней переменной — как вам уже указали выше, такой подход может вернуть мусорный результат