РАЗРАБОТКА ВЕБ-САЙТОВ*, JAVASCRIPT*, БЛОГ КОМПАНИИ RUVDS.COM

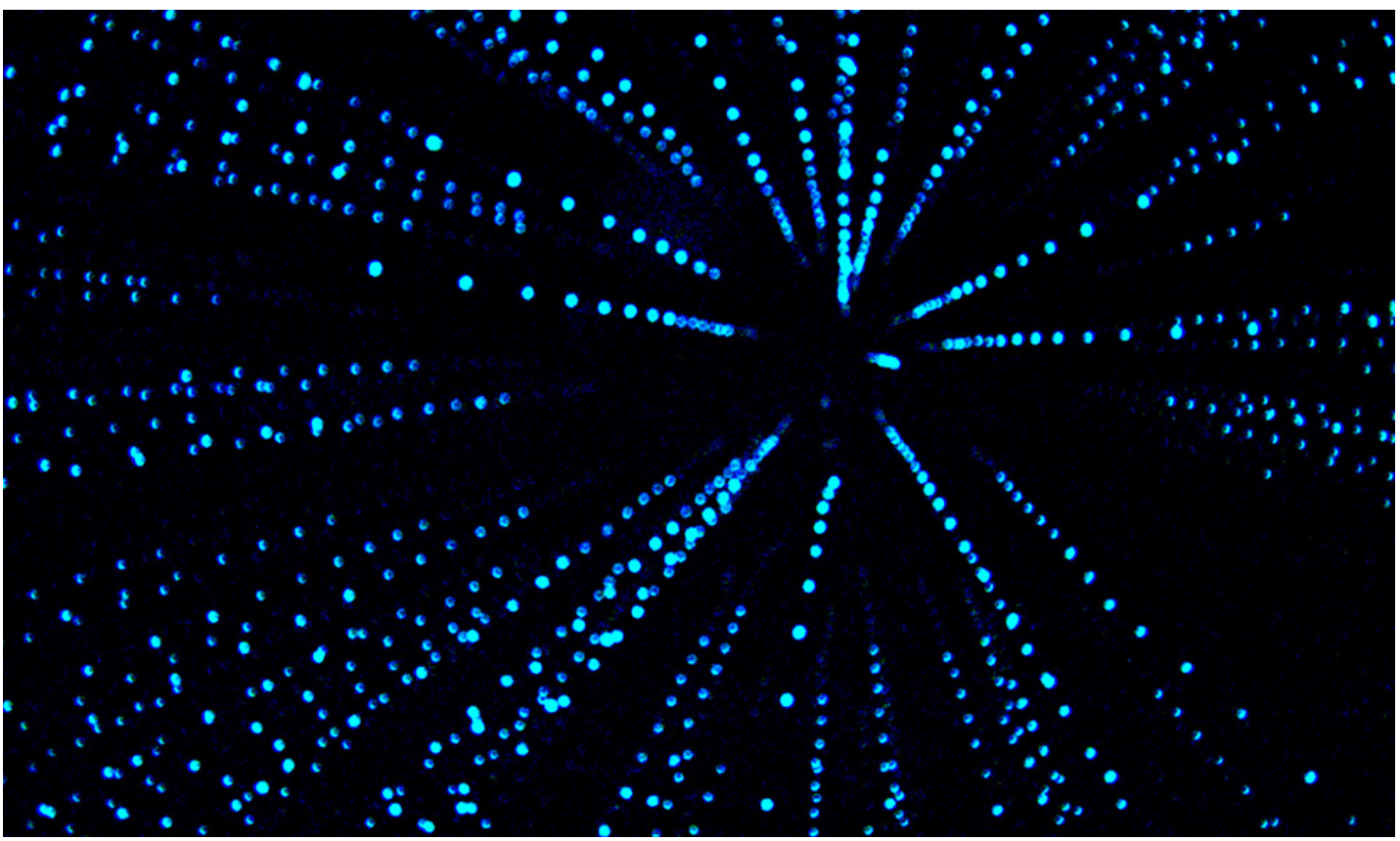
Путь к трансдьюсерам на чистом JavaScript

ПЕРЕВОД

ru_vds 26 мая 2017 в 14:08 14,6k

Оригинал: Reginald Braithwaite

Если вы наслышаны о так называемых «трансдьюсерах», но до сих пор не применяете их в JavaScript-разработке, сегодня у вас есть шанс найти ответы на вопросы: «Что такое трансдьюсеры?» и «Как ими пользоваться?». Это позволит вам понять, нужны ли они в ваших проектах, и, если нужны — поможет приступить к их использованию.



Речь пойдёт о том, как писать код, который предназначен для построения хорошо подходящих для компоновки конвейеров преобразований данных, не потребляющий слишком много памяти. Для того, чтобы как следует разобраться в концепции трансдьюсеров, начнём мы с более простых механизмов, редьюсеров, или функций для свёртки данных.

Редьюсеры

Редьюсер — это функция, которая принимает на вход объект-накопитель и некий объект-элемент, после чего помещает этот элемент в накопитель. Например, вот редьюсер:

`(acc, val) => acc.concat([val])`. Если переданный ему накопитель — это массив `[1, 2, 3]`, а элемент — число `4`, он вернёт массив `[1, 2, 3, 4]`.

```
const acc = [1, 2, 3];
const val = 4;
const reducer = (acc, val) => acc.concat([val]);
reducer(acc, val)
  ///=> 1, 2, 3, 4
```

В нашем случае редьюсер возвращает результат конкатенации переданного ему списка элементов и единичного элемента.

Вот ещё один похожий редьюсер: `(acc, val) => acc.add(val)`. Он подходит для любого объекта, имеющего метод `.add()`, который, кроме прочего, возвращает этот объект (вроде Set.prototype.add()). Наш редьюсер добавляет переданный

ему элемент к накопителю, используя метод `.add()` накопителя.

```
const acc = new Set([1, 2, 3]);
const val = 4;
const reducer = (acc, val) => acc.add(val);
reducer(acc, val)
  ///=> Set{1, 2, 3, 4}
```

Вот — функция, которая создаёт массив из любого итерируемого объекта с использованием нашего конкатенирующего редьюсера.

```
const toArray = iterable => {
  const reducer = (acc, val) => acc.concat([val]);
  const seed = [];
  let accumulation = seed;

  for (value of iterable) {
    accumulation = reducer(accumulation, value);
  }

  return accumulation;
}

toArray([1, 2, 3])
  //=> [1, 2, 3]
```

Можно сделать переменные `reducer` и `seed` параметрами новой функции (принимающей, по аналогии с только что рассмотренной, и аргумент `iterable`), получив универсальную редуцирующую функцию.

```
const reduce = (iterable, reducer, seed) => {
  let accumulation = seed;

  for (const value of iterable) {
    accumulation = reducer(accumulation, value);
```

```
  return accumulation;
}

reduce([1, 2, 3], (acc, val) => acc.concat([val]), [])
  //=> [1, 2, 3]
```

JavaScript развивается в направлении соглашения о написании функций, вроде нашей `reduce`, первым параметром которых является редьюсер. Если переписать эту функцию в стиле JavaScript Allongé, получится следующее.

```
const reduceWith = (reducer, seed, iterable) => {
  let accumulation = seed;

  for (const value of iterable) {
    accumulation = reducer(accumulation, value);
  }

  return accumulation;
}

reduce([1, 2, 3], (acc, val) => acc.concat([val]), [])
  //=> [1, 2, 3]

// теперь вызов выглядит так:

reduceWith((acc, val) => acc.concat([val]), [], [1, 2, 3])
  //=> [1, 2, 3]
```

У массивов в JavaScript есть встроенный метод `.reduce`. Этот метод ведет себя точно так же, как вышеописанные функции `reduce` и `reduceWith`.

```
[1, 2, 3].reduce((acc, val) => acc.concat([val]), [])
  //=> [1, 2, 3]
```

Теперь функция `(acc, val) => acc.concat([val])` создаёт ненужную нагрузку на память, поэтому мы можем заменить её на такой редьюсер: `(acc, val) => { acc.push(val); return acc; }`.

Тут надо отметить, что запись вида `(acc, val) => (acc.push(val), acc)` с семантической точки зрения выглядит лучше, но оператор «запятая» может запутать тех, кто не знаком с особенностями его использования. Обычно в продакшн-коде такого лучше избегать.

В любом случае, у нас получится редьюсер, который собирает элементы в массив. Дадим ему имя и попробуем передать функции `reduceWith`.

```
const arrayOf = (acc, val) => { acc.push(val); return acc; };

reduceWith(arrayOf, [], [1, 2, 3])
  //=> [1, 2, 3]
```

Вот ещё один редьюсер.

```
const sumOf = (acc, val) => acc + val;

reduceWith(sumOf, 0, [1, 2, 3])
  //=> 6
```

Можно писать редьюсеры, которые сворачивают итерируемый объект одного типа (скажем, массив), в объект другого типа

(например — в число).

Декорирование редьюсеров

В JavaScript легко писать функции, которые возвращают другие функции. Вот, например, функция, которая позволяет создавать редьюсеры.

```
const joinedWith =
  separator =>
    (acc, val) =>
      acc == '' ? val : `${acc}${separator}${val}`;

reduceWith(joinedWith(', '), '', [1, 2, 3])
  //=> "1, 2, 3"

reduceWith(joinedWith('.'), '', [1, 2, 3])
  //=> "1.2.3"
```

Кроме того, в JS совершенно естественным является создание функций, которые принимают другие функции в качестве аргументов.

Декораторы — это функции, которые, принимая некую функцию в качестве аргумента, возвращают другую функцию, семантически связанную с аргументом. Например, эта функция принимает функцию с двумя аргументами, бинарную, если говорить языком функционального программирования, и декорирует её, добавляя единицу к её второму аргументу.

```
const incrementSecondArgument =
  binaryFn =>
    (x, y) => binaryFn(x, y + 1);
```

```
const power =
  (base, exponent) => base ** exponent;

const higherPower = incrementSecondArgument(power);

power(2, 3)
  //=> 8

higherPower(2, 3)
  //=> 16
```

В этом примере функция `higherPower` — это функция `power`, декорированная путём добавления единицы к её аргументу `exponent`. Таким образом, вызов `higherPower(2,3)` даёт тот же результат, что и `power(2,4)`. С подобными функциями мы уже работали, наши редьюсеры — тоже бинарные функции. Их можно декорировать.

```
reduceWith(incrementSecondArgument(arrayOf), [], [1, 2, 3])
  //=> [2, 3, 4]

const incremented =
  iterable =>
    reduceWith(incrementSecondArgument(arrayOf), [], iterable);

incremented([1, 2, 3])
  //=> [2, 3, 4]
```

Функции маппинга

Мы только что создали функцию для маппинга, которая, принимая итерируемый объект, возвращает результат обработки его значений путём увеличения каждого из них на единицу. Разрабатывая программы на JS, мы постоянно прибегаем к

маппингу, но, конечно, от функций, реализующих этот механизм, обычно ожидают несколько большего, нежели производство копий числовых массивов, элементы которых увеличены на единицу. Взглянем ещё раз на функцию `incrementSecondArgument`.

```
const incrementSecondArgument =
  binaryFn =>
    (x, y) => binaryFn(x, y + 1);
```

Так как мы используем её для декорирования редьюсеров, дадим ей более подходящее имя.

```
const incrementValue =
  reducer =>
(acc, val) => reducer(acc, val + 1);
```

Теперь при чтении кода сразу видно, что `incrementValue` принимает в качестве аргумента редьюсер и возвращает другой редьюсер, который, перед обработкой переданного ему элемента, прибавляет к нему единицу. Логику «инкрементации» можно вынести в параметр.

```
const map =
  fn =>
    reducer =>
      (acc, val) => reducer(acc, fn(val));

const incrementValue = map(x => x + 1);

reduceWith(incrementValue(arrayOf), [], [1, 2, 3])
  //=> [2, 3, 4]
```

Хотя всё это может выглядеть необычно для тех, кто не привык к функциям, которые принимают функции как аргументы и возвращают другие функции, которые, опять же, принимают функции как аргументы, мы можем поместить конструкцию `map(x => x + 1)` везде, где можно пользоваться `incrementValue`. Таким образом, можно написать следующее.

```
reduceWith(map(x => x + 1)(arrayOf), [], [1, 2, 3])
  //=> [2, 3, 4]
```

И, так как наш декоратор `map` может декорировать любой редьюсер, допустимо объединить результаты инкрементирования чисел, сформировав строку, или суммировать их.

```
reduceWith(map(x => x + 1)(joinedWith('.')), '', [1, 2, 3])
  //=> "2.3.4"

reduceWith(map(x => x + 1)(sumOf), 0, [1, 2, 3])
  //=> 9
```

Вооружившись вышеописанными приёмами, попытаемся найти сумму квадратов чисел от одного до десяти.

```
const squares = map(x => power(x, 2));
const one2ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

reduceWith(squares(sumOf), 0, one2ten)
  //=> 385
```

Как видите, нам это удалось. Теперь идём дальше — поговорим о

фильтрах.

Фильтры

Вернёмся к нашему первому редьюсеру.

```
const arrayOf = (acc, val) => { acc.push(val); return acc; };

reduceWith(arrayOf, 0, one2ten)
  //=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Что если нужно, чтобы на выход попали только числа, которые больше пяти? Сделать это несложно.

```
const bigUns = (acc, val) => {
  if (val > 5 ) {
    acc.push(val);
  }
  return acc;
};

reduceWith(bigUns, [], one2ten)
  //=> [6, 7, 8, 9, 10]
```

Естественно, мы можем скомбинировать всё то, с чем уже разобрались, для того, чтобы получить массив чисел, которые больше пяти, возведённых в квадрат.

```
reduceWith(squares(bigUns), [], one2ten)
  //=> [9, 16, 25, 36, 49, 64, 81, 100]
```

Однако, получилось тут совсем не то, что нужно. На выходе —

числа, квадраты которых больше пяти, а не числа, которые больше чем пять, возведённые в квадрат. Числа надо отбирать до возведения их в квадрат, а не после. Добиться такого поведения системы не так уж и сложно. Суть тут в том, что нам поможет декоратор, который отвечает за фильтрацию чисел, его мы можем использовать для декорирования редьюсера.

```
reduceWith(squares(arrayOf), [], one2ten)
  //=> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

const bigUnsOf =
  reducer =>
    (acc, val) =>
      (val > 5) ? reducer(acc, val) : acc;

reduceWith(bigUnsOf(squares(arrayOf)), [], one2ten)
  //=> [36, 49, 64, 81, 100]
```

Функция `bigUnsOf` довольно специфична. Поступим тут так же, как с `map`, а именно — извлечём функцию-предикат и сделаем её аргументом.

```
reduceWith(squares(arrayOf), [], one2ten)
  //=> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

const filter =
  fn =>
    reducer =>
      (acc, val) =>
        fn(val) ? reducer(acc, val) : acc;

reduceWith(filter(x => x > 5)(squares(arrayOf)), [], one2ten)
  //=> [36, 49, 64, 81, 100]
```

Фильтры, конечно, могут быть любыми. Им можно давать имена и

использовать многократно, можно обходиться и анонимными функциями.

```
reduceWith(filter(x => x % 2 === 1)(arrayOf), [], one2ten)
  //=> [1, 3, 5, 7, 9]
```

Воспользуемся фильтром для того, чтобы найти сумму квадратов нечётных чисел от одного до десяти.

```
reduceWith(filter(x => x % 2 === 1)(squares(sumOf)), 0, one2ten)
  //=> 165
```

Трансформеры и композиция

Термин «трансформер» пришёл в JavaScript из других языков программирования. Так называют функцию, принимающую некий аргумент и трансформирующую его во что-то другое. То, что мы выше называли «декоратором», является частным случаем трансформера. Таким образом, если вы встретите где-нибудь рассказ о функции-трансформере, которая делает из одного редьюсера другой, вам будет ясно, что речь идёт о такой же функции, которая «декорирует» редьюсер, добавляя к нему дополнительный функционал, вроде маппинга или фильтрации.

Функции маппинга и фильтры, о которых мы говорили — это тоже трансформеры. В контексте этого шаблона программирования, самой важной характеристикой трансформеров является то, что они применяют композицию для создания новых трансформеров. Вот, чтобы было понятнее, функция, которая выдаёт композицию

любых двух поданных ей на вход функций.

```
const plusFive = x => x + 5;
const divideByTwo = x => x / 2;

plusFive(3)
  //=> 8

divideByTow(8)
  //=> 4

const compose2 =
  (a, b) =>
    (...c) =>
      a(b(...c));

const plusFiveDividedByTwo = compose2(divideByTwo, plusFive);

plusFiveDividedByTwo(3)
  //=> 4
```

Трансформеры применяют композицию для создания новых трансформеров. Что это значит в применении к функции `compose2`? Это значит, что передав ей два любых трансформера, мы получим новый трансформер, который трансформирует редьюсер. Таким образом, получим следующее.

```
const squaresOfTheOddNumbers = compose2(
  filter(x => x % 2 === 1),
  squares
);

reduceWith(squaresOfTheOddNumbers(sumOf), 0, one2ten)
  //=> 165
```

То, что скрыто под именем `squaresOfTheOddNumbers` — это трансформер, который мы создали, применив функцию `compose2`

к фильтру и функции для маппинга.

Теперь, когда у нас есть возможность композиции декораторов, разобьём на части сложный код, отличающийся большой степенью связности, на небольшие узкоспециализированные блоки.

Композиция с помощью трансформеров

Зная, как работает функция `compose2`, которая позволяет получать композицию двух функций, подумаем о том, как быть, если нам нужна композиция произвольного количества функций. Ответ заключается в свёртке.

Перепишем `compose2`, сделав из неё трансформер `compositionOf`.

```
const compositionOf = (acc, val) => (...args) => val(acc(...args));
```

Теперь можно написать функцию `compose` для получения композиции произвольного количества функций как редукции её аргументов:

```
const compose = (...fns) =>
  reduceWith(compositionOf, x => x, fns);
```

Итак, мы подошли к самому интересному.

Трансдьюсеры

Рассмотрим следующую запись:

```
reduceWith(squaresOfTheOddNumbers(sumOf), 0, one2ten)
```

Здесь можно выделить четыре элемента. Трансформер для редьюсера (который может быть композицией трансформеров), начальное значение (накопитель) и итерируемый объект. Если вынести в отдельные параметры трансформер, редьюсер, накопитель и итерируемый объект, получим следующее.

```
const transduce = (transformer, reducer, seed, iterable) => {
  const transformedReducer = transformer(reducer);
  let accumulation = seed;

  for (const value of iterable) {
    accumulation = transformedReducer(accumulation, value);
  }

  return accumulation;
}

transduce(squaresOfTheOddNumbers, sumOf, 0, one2ten)
  //=> 165
```

Надо отметить, что в некоторых языках программирования имеется сильное стремление к сокращению длинных имён переменных или параметров. В результате довольно длинное имя `transformer` сокращается до `xform` или даже до `xf`. Не удивляйтесь, если увидите похожую конструкцию, запись которой выглядит как (xf, reduce, seed, coll), или `xf((val, acc) => acc) -> (val, acc) => acc`. Тут мы обойдёмся без сокращений, но в продакшн-коде имена вроде `xf` или `xform` вполне допустимы.

А теперь, собственно, то, ради чего всё это было написано. Редьюсер — это функция, которая передаётся методам вроде `.reduce` — она принимает объект-накопитель и входные данные, и возвращает накопитель, в который помещены новые данные. Трансформер — это функция, которая трансформирует редьюсер в другой редьюсер. А трансдьюсер (это название — результат совмещения терминов «трансформер» и «редьюсер», вот он ответ на вопрос: «Что такое редьюсеры?»), это функция, которая принимает трансформер, редьюсер, накопитель и итерируемый объект, после чего сворачивает итерируемый объект в некое значение.

Элегантность шаблона «трансдьюсер» заключается в том, что композиция трансформеров естественным образом ведёт к созданию новых трансформеров. В результате можно объединить в цепочку столько трансформеров, сколько нужно. Это очень важно, так как в результате получится один трансформированный редьюсер и по итерируемой коллекции нужно будет пройтись лишь один раз. Не нужно создавать промежуточные копии данных или выполнять несколько проходов по ним.

Трансдьюсеры пришли в JavaScript из языка Clojure, но, как вы можете видеть, они совершенно органично вписываются в JavaScript, для их реализации достаточно стандартных возможностей языка.

Итак, если кто-нибудь спросит нас о том, что же такое трансдьюсер, мы можем ответить так:

Примечание: этот фрагмент надо выделить.

Трансдьюсеры — это трансформеры, которые подходят для композиции, применённые к редьюсерам для свёртки итерируемых объектов.

Трансдьюсер в действии

Выше мы рассматривали фрагменты кода, ведущие к построению трансдьюсера. Вполне возможно, вы уже воспроизвели их в своём JS-редакторе и испытали, однако, вот, для удобства и наглядности, весь наш код, собранный в одном месте.

```
const arrayOf = (acc, val) => { acc.push(val); return acc; };
const sumOf = (acc, val) => acc + val;
const setOf = (acc, val) => acc.add(val);
const map =
  fn =>
    reducer =>
      (acc, val) => reducer(acc, fn(val));

const filter =
  fn =>
    reducer =>
      (acc, val) =>
        fn(val) ? reducer(acc, val) : acc;

const compose = (...fns) =>
  fns.reduce((acc, val) => (...args) => val(acc(...args)), x => x);

const transduce = (transformer, reducer, seed, iterable) => {
  const transformedReducer = transformer(reducer);
  let accumulation = seed;

  for (const value of iterable) {
    accumulation = transformedReducer(accumulation, value);
  }
```

```
  return accumulation;
}
```

Этот пример демонстрирует всё то, чем обычно пользуются, работая с массивами, а именно — это методы `.map`, `.filter`, `.reduce`, тут же имеются подходящие для композиции трансдьюсеры, которые не занимаются созданием множества копий обрабатываемого набора данных. На самом деле, трансдьюсеры, написанные для реальных проектов, предусматривают гораздо больше вариантов использования, например, воспроизводя функционал метода `.find`.

Надо отметить, что наша функция `transduse` рассчитана на то, что ей будет передана итерируемая коллекция, кроме того, мы должны предоставить ей начальное значение (накопитель) и редьюсер. В большинстве случаев и начальное значение, и редьюсер — это одни и те же функции для всех коллекций одного и того же типа. Это характерно и для соответствующих библиотек.

В объектно-ориентированном программировании эта проблема, конечно, решается через полиморфизм. У коллекций есть методы, поэтому, вызывая подходящий метод, мы получаем на выходе то, что нам нужно. Библиотеки, применимые для создания продакшн-кода, предоставляют интерфейсы для коллекций различных типов, позволяющие удобно пользоваться трансдьюсерами.

Полагаем, вышеизложенного достаточно для того, чтобы понять шаблон, лежащий в основе трансдьюсеров и оценить те полезные и удобные возможности, которые даёт наличие в языке функций

первого класса.

Трансдьюсеры: обработка списка пользователей

В этом материале показаны варианты решения следующей задачи: имеется набор пользователей и посещённых ими мест, и то и другое представлено в виде списка хэш-кодов. Первый код в каждой строке — пользователь, второй — посещённое им место, скажем, ресторан, или магазин. Порядок следования данных в списке имеет значение. Задача заключается в том, чтобы выяснить, какие переходы между местами наиболее популярны.

А именно, есть такой список:

```
1a2ddc2, 5f2b932
f1a543f, 5890595
3abe124, bd11537
f1a543f, 5f2b932
f1a543f, bd11537
f1a543f, 5890595
1a2ddc2, bd11537
1a2ddc2, 5890595
3abe124, 5f2b932
f1a543f, 5f2b932
f1a543f, bd11537
f1a543f, 5890595
1a2ddc2, 5f2b932
1a2ddc2, bd11537
1a2ddc2, 5890595
...
```

Внимательно просмотрев этот список, мы можем обнаружить, что пользователь `1a2ddc2` посетил места с кодами `5f2b932`, `bd11537`, `5890595`, `5f2b932`, `bd11537`, и `5890595`. В то же время,

пользователь `f1a543f` посетил места `5890595`, `5f2b932`, `bd11537`, `5890595`, `5f2b932`, `bd11537`, и `5890595`. И так далее.

Предположим, надо выяснить, куда обычно ходят люди, надо найти самые популярные переходы из «места А» в «место Б». Мы знаем, что история путешествия пользователя `1a2ddc2` выглядит следующим образом: `5f2b932`, `bd11537`, `5890595`, `5f2b932`, `bd11537`, `5890595`. Это значит, что для него можно построить такую схему переходов из места в место:

```
5f2b932 -> bd11537
bd11537 -> 5890595
5890595 -> 5f2b932
5f2b932 -> bd11537
bd11537 -> 5890595
```

Обратите внимание на то, что нам надо построить подобный список для каждого пользователя. Когда это будет сделано, нужно найти самые популярные переходы. Вот как может выглядеть подобный подсчёт:

Переход `5f2b932 -> bd11537` появляется в списке дважды.
Переход `bd11537 -> 5890595` также встречается дважды.
Переход `5890595 -> 5f2b932` встретился лишь один раз.

Теперь всё, что нужно сделать — это посчитать количество переходов по всем пользователям и найти наиболее популярные. Вот решение этой задачи с использованием трансдьюсеров.

```
const logContents = `1a2ddc2, 5f2b932
f1a543f, 5890595
3abe124, bd11537
f1a543f, 5f2b932
f1a543f, bd11537
f1a543f, 5890595
1a2ddc2, bd11537
1a2ddc2, 5890595
3abe124, 5f2b932
f1a543f, 5f2b932
f1a543f, bd11537
f1a543f, 5890595
1a2ddc2, 5f2b932
1a2ddc2, bd11537
1a2ddc2, 5890595`;
```

```
const asStream = function * (iterable) { yield * iterable; };
const lines = str => str.split('\n');
const streamOfLines = asStream(lines(logContents));
const datums = str => str.split(', ');
const datumize = map(datums);
const userKey = ([user, _]) => user;

const pairMaker = () => {
  let wip = [];

  return reducer =>
    (acc, val) => {
      wip.push(val);

      if (wip.length === 2) {
        const pair = wip;
        wip = wip.slice(1);
        return reducer(acc, pair);
      } else {
        return acc;
      }
  }
}

const sortedTransformation =
  (xfMaker, keyFn) => {
    const decoratedReducersByKey = new Map();

    return reducer =>
```

```
      (acc, val) => {
        const key = keyFn(val);
        let decoratedReducer;

        if (decoratedReducersByKey.has(key)) {
          decoratedReducer = decoratedReducersByKey.get(key);
        } else {
          decoratedReducer = xfMaker()(reducer);
          decoratedReducersByKey.set(key, decoratedReducer);
        }

        return decoratedReducer(acc, val);
      }
  }

const userTransitions = sortedTransformation(pairMaker, userKey);
const justLocations = map(([[u1, l1], [u2, l2]]) => [l1, l2]);
const stringify = map(transition => transition.join(' -> '));
const transitionKeys = compose(
  stringify, justLocations, userTransitions, datumize
);

const countsOf =
  (acc, val) => {
    if (acc.has(val)) {
      acc.set(val, 1 + acc.get(val));
    } else {
      acc.set(val, 1);
    }
    return acc;
  }

const greatestValue = inMap =>
  Array.from(inMap.entries()).reduce(
    ([wasKeys, wasCount], [transitionKey, count]) => {
      if (count < wasCount) {
        return [wasKeys, wasCount];
      } else if (count > wasCount) {
        return [new Set([transitionKey]), count];
      } else {
        wasKeys.add(transitionKey);
        return [wasKeys, wasCount];
      }
    }
    , [new Set(), 0]
  );

greatestValue(
```

```
  transduce(transitionKeys, countsOf, new Map(), streamOfLines)
)
  //=>
    [
      "5f2b932 -> bd11537",
      "bd11537 -> 5890595"
    ],
    4
```

Итоги

Надеемся, этот материал поможет всем желающим сделать трансдьюсеры своим постоянным инструментом. Если вы хотите углубиться в их изучение, вот хороший материал о трансдьюсерах, а вот — библиотека transdusers-js на GitHub.

Уважаемые читатели! Пользуетесь ли вы трансдьюсерами в своих JavaScript-проектах?

Может ли в JavaScript конструкция (a==1 && a==2 && a==3) оказаться равной true?

95

ПЕРЕВОД

ru_vds • 25 января в 16:08

JavaScript и ужасы мутаций

48

ПЕРЕВОД

ru_vds • 19 января в 11:42

Машины состояний и разработка веб-приложений

8

ПЕРЕВОД

ru_vds • 18 января в 12:02

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации 2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ 40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству? 8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32