REACTJS*, JAVASCRIPT*, БЛОГ КОМПАНИИ DEVEXPRESS

Иммутабельность в JavaScript

MrCheater 30 мая 2016 в 06:18 44,1k

Что такое иммутабельность

Неизменяемым (англ. immutable) называется объект, состояние которого не может быть изменено после создания. Результатом любой модификации такого объекта всегда будет новый объект, при этом старый объект не изменится.

```
var mutableArr = [1, 2, 3, 4];
arr.push(5);
console.log(mutableArr); // [1, 2, 3, 4, 5]

//Use seamless-immutable.js
var immutableArr = Immutable([1, 2, 3, 4]);
var newImmutableArr = immutableArr.concat([5]);
console.log(immutableArr); //[1, 2, 3, 4];
console.log(newImmutableArr); //[1, 2, 3, 4, 5];
```

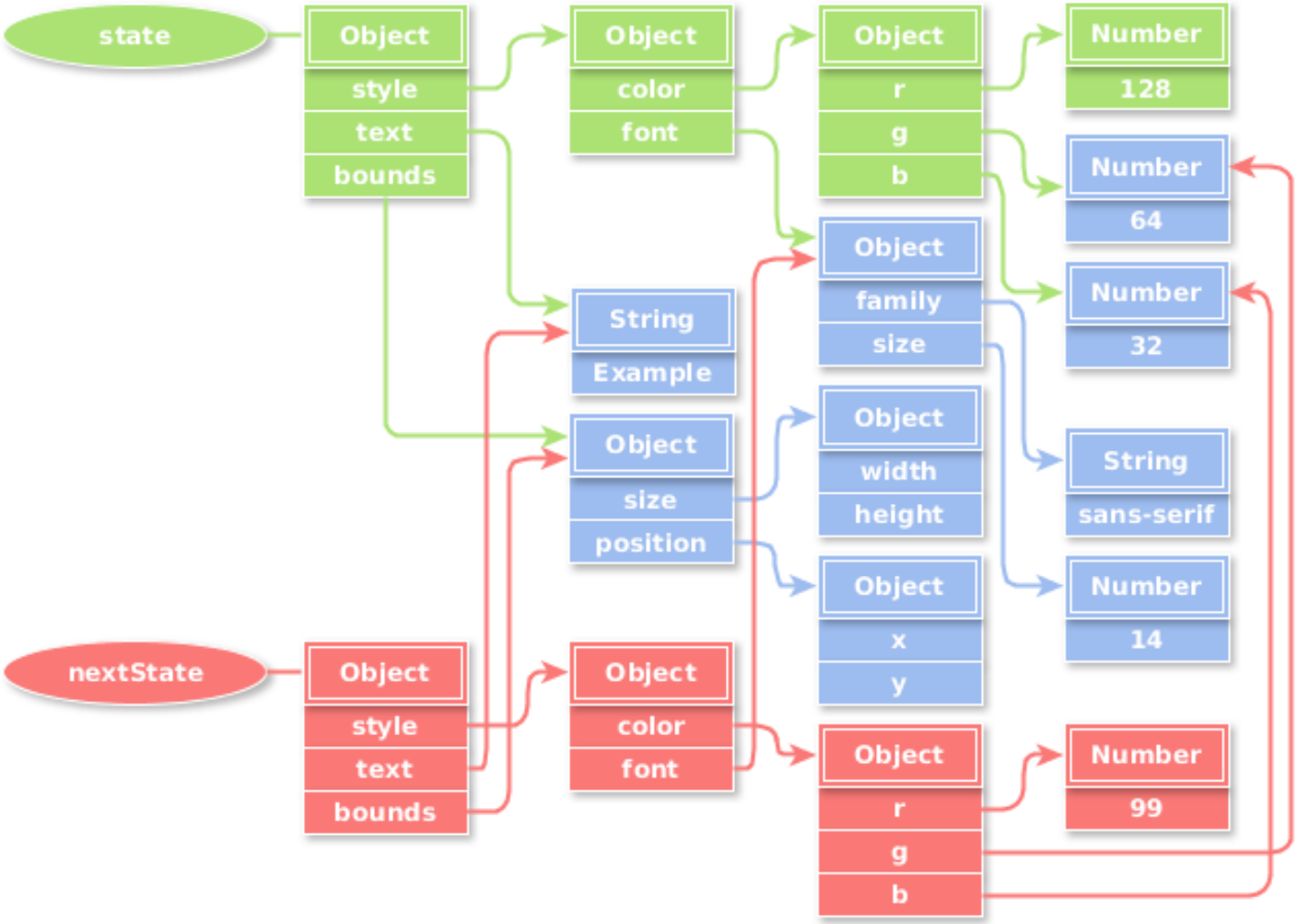
Речь не идет о глубоком копировании: если объект имеет вложенную структуру, то все вложенные объекты, не подвергшиеся модификации, будут переиспользованы.

```
//Use seamless-immutable.js
var state = Immutable({
    style : {
       color : {
          r : 128,
          g : 64,
          b : 32
       },
       font : {
          family : 'sans-serif',
          size : 14
       }
    },
    text : 'Example',
    bounds : {
       size : {
          width : 100,
          height : 200
       },
       position : {
          x : 300,
          y : 400
       }
    }
});

var nextState = state.setIn(['style', 'color', 'r'], 99);

state.bounds === nextState.bounds; //true
state.text === nextState.text; //true
state.style.font === state.style.font; //true
```

В памяти объекты будут представлены следующим образом:



Правда или ложь? Иммутабельные данные в JavaScript

Простое и быстрое отслеживание изменений

Эту возможность активно используют в связке с популярным нынче VirtualDOM (React, Mithril, Riot) для ускорения перерисовки web-страниц.

Возьмем пример с `state`, приведенный чуть выше. После модификации объекта `state` нужно сравнить его с объектом `nextState` и узнать, что конкретно в нем изменилось. Иммутабельность сильно упрощает нам задачу: вместо того,

чтобы сравнивать значение каждого поля каждого вложенного в `state` объекта с соответствующим значением из `nextState`, можно просто сравнивать ссылки на соответствующие объекты и отсеивать таким образом целые вложенные ветки сравнений.

```
state === nextState //false
state.text === nextState.text //true
state.style === nextState.style //false
state.style.color === nextState.style.color //false
state.style.color.r === nextState.style.color.r //false
state.style.color.g === nextState.style.color.g //true
state.style.color.b === nextState.style.color.b //true
state.style.font === nextState.style.font; //true
//state.style.font.family === nextState.style.font.family; //true
//state.style.font.size === nextState.style.font.size; //true
state.bounds === nextState.bounds //true
//state.bounds.size === nextState.bounds.size //true
//state.bounds.size.width === nextState.bounds.size.width //true
//state.bounds.size.height === nextState.bounds.size.height //true
//state.bounds.position === nextState.bounds.position //true
//state.bounds.position.x === nextState.bounds.position.x //true
//state.bounds.position.y === nextState.bounds.position.y //true
```

Внутри объектов `bounds` и `style.font` операции сравнения производить не нужно, так как они иммутабельны, и ссылки на них не изменились.

Безопаснее использовать и легче тестировать

Нередки случаи, когда переданные в функцию данные могут быть случайно испорчены, и отследить такие ситуации очень сложно.

```
var arr = [2, 1, 3, 5, 4, 0];

function render(items) {
   return arr
```

```
      .sort(function(a, b) {return a < b ? -1 : a > b ? 1 : 0})
      .map(function(item){
         return '<div>' + item + '</div>';
      });
}

render(arr);
console.log(arr); // [0, 1, 2, 3, 4, 5]
```

Здесь иммутабельные данные спасли бы ситуацию. Функция `sort` была бы запрещена.

```
//Use seamless-immutable.js
var arr = [2, 1, 3, 5, 4, 0];

function render(items) {
   return items
      .sort(function(a, b) {return a < b ? -1 : a > b ? 1 : 0})
      .map(function(item){
         return '<div>' + item + '</div>';
      });
}

render(arr); //Uncaught Error: The sort method cannot be invoked on
an Immutable data structure.
console.log(arr);
```

Или вернула бы новый отсортированный массив, не меняя старый:

```
//Use immutable.js
var arr = Immutable.fromJS([2, 1, 3, 5, 4, 0]);

function render(items) {
   return arr
      .sort(function(a, b) {return a < b ? -1 : a > b ? 1 : 0})
      .map(function(item){
         return '<div>' + item + '</div>';
      });
}
```

```
render(arr);
console.log(arr.toJS()); // [2, 1, 3, 5, 4, 0]
```

Больший расход памяти

Каждый раз при модификации иммутабельного объекта создается его копия с необходимыми изменениями. Это приводит к большему расходу памяти, чем при работе с обычным объектом. Но поскольку иммутабельные объекты никогда не меняются, они могут быть реализованы с помощью стратегии, называемой «общие структуры» (structural sharing), которая порождает гораздо меньшую издержку в затратах на память, чем можно было бы ожидать. В сравнении со встроенными массивами и объектами издержка все еще будет существовать, но она будет иметь фиксированную величину и обычно может компенсироваться другим преимуществами, доступными благодаря неизменяемости.

~~Легче кешировать (мемоизировать)~~

В большинстве случаев кешировать легче не станет. Этот пример прояснит ситуацию:

```
var step_1 = Immutable({
   data : {
      value : 0
   }
});
var step_2 = step_1.setIn(['data', 'value'], 1);
var step_3 = step_2.setIn(['data', 'value'], 0);
step_1.data === step_3.data; //false
```

Несмотря на то, что `data.value` из первого шага не отличается от `data.value` из последнего шага, сам объект `data` уже другой, и ссылка на него тоже изменилась.

~~Отсутствие побочных эффектов~~

Это тоже неправда:

```
function test(immutableData) {
   var value = immutableData.get('value');
   window.title = value;
   return immutableData.set('value', 42);
}
```

Гарантий того, что функция станет чистой, или что у нее будут отсутствовать побочные эффекты — нет.

~~Ускорение кода. Больше простора для оптимизаций~~

Здесь не все так очевидно, и производительность зависит от конкретной реализации иммутабельных структур данных, с которой приходится работать. Но если взять и просто заморозить объект при помощи `Object.freeze`, то обращение к нему и его свойствам быстрее не станет, а в некоторых браузерах станет даже медленее.

~~Thread safety~~

JavaScript — однопоточный, и говорить тут особо не о чем. Многие путают асинхронность и многопоточность — это не одно и тоже.

По умолчанию есть только один поток, который асинхронно обслуживает очередь сообщений.
В браузере для многопоточности есть WebWorkers, но единственное возможное общение между потоками осуществляется через отправку строк или сериализованного JSON; к одним и тем же переменным из разных воркеров обратиться нельзя.

Возможности языка

Ключевое слово const

Использование `const` вместо `var` или `let` не говорит от том, что значение является константой или что оно иммутабельно (неизменяемо). Ключевое слово `const` просто указывает компилятору следить за тем, что переменной больше не будет присвоено никаких других значений.

В случае использования `const` современные JavaScript-движки могут выполнить ряд дополнительных оптимизаций.

Пример:

```
const obj = { text : 'test'};
obj.text = 'abc';
obj.color = 'red';
console.log(obj); //Object {text: "abc", color: "red"}
obj = {}; //Uncaught TypeError: Assignment to constant variable.(…)
```

Object.freeze

Метод `Object.freeze` замораживает объект. Это значит, что он предотвращает добавление новых свойств к объекту, удаление старых свойств из объекта и изменение существующих свойств или значения их атрибутов перечисляемости, настраиваемости и записываемости. В сущности, объект становится эффективно неизменным. Метод возвращает замороженный объект.

Сторонние библиотеки

Seamless-Immutable

Библиотека предлагает иммутабельные структуры данных, обратно совместимые с обычными массивами и объектами. То есть доступ к значениям по ключу или по индексу не будет отличаться от привычного, будут работать стандартные циклы, а также все это можно использовать в связке со специализированными высокопроизводительными библиотеками для манипуляций с данными, вроде Lodash или Underscore.

```
var array = Immutable(["totally", "immutable", {hammer: "Can’t Touch
This"}]);

array[1] = "I'm going to mutate you!"
array[1] // "immutable"

array[2].hammer = "hm, surely I can mutate this nested object..."
array[2].hammer // "Can’t Touch This"

for (var index in array) {
   console.log(array[index]);
}
// "totally"
// "immutable"
// { hammer: 'Can’t Touch This' }
```

```
JSON.stringify(array) // '["totally","immutable",{"hammer":"Can't
Touch This"}]'
```

Для работы эта библиотека использует `Object.freeze`, а также запрещает использование методов, которые могут изменить данные.

```
 Immutable([3, 1, 4]).sort()
// This will throw an ImmutableError, because sort() is a mutating
method.
```

Некоторые браузеры, например Safari, имеют проблемы с производительностью при работе с замороженными при помощи `Object.freeze` объектами, так что в `production` сборке это отключено для увеличения производительности.

Immutable.js

Благодаря продвижению со стороны Facebook эта библиотека для работы с иммутабельными данными стала самой распространенной и популярной среди web-разработчиков. Она предоставляет следующие неизменяемые структуры данных:

- List — иммутабельный аналог JavaScript Array

```
var list = Immutable.List([1, 3, 2, 4, 5]);
console.log(list.size); //5
list = list.pop().pop(); //[1, 3, 2]
list = list.push(6); //[1, 3, 2, 6]
list = list.shift(); //[3, 2, 6]
list = list.concat(9, 0, 1, 4); //[3, 2, 6, 9, 0, 1, 4]
list = list.sort(); //[0, 1, 2, 3, 4, 6, 9]
```

- Stack — иммутабельный список элементов, организованных по принципу LIFO (last in — first out, «последним пришел — первым вышел»)

```
var stack = new Immutable.Stack();
stack = stack.push( 2, 1, 0 );
stack.size;
stack.get(); //2
stack.get(1); //1
stack.get(2); //0
stack = stack.pop(); // [1, 0]
```

- Map — иммутабельный аналог JavaScript Object

```
var map = new Immutable.Map();
map = map.set('value', 5); //{value : 5}
map = map.set('text', 'Test'); //{value : 5, text : "Test"}
map = map.delete('text'); // {value : 5}
```

- OrderedMap — иммутабельный аналог JavaScript Object, гарантирующий такой же порядок обхода элементов, какой он был при записи

```
var map = new Immutable.OrderedMap();
map = map.set('m', 5); //{m : 5}
map = map.set('a', 1); //{m : 5, a : 1}
map = map.set('p', 8); //{m : 5, a : 1, p : 8}
for(var elem of map) {
console.log(elem);
}
```

- Set — иммутабельное множество для хранения уникальных значений

```
var s1 = Immutable.Set( [2, 1] );
var s2 = Immutable.Set( [2, 3, 3] );
var s3 = Immutable.Set( [1, 1, 1] );
console.log( s1.count(), s2.size, s3.count() ); // 2 2 1
console.log( s1.toJS(), s2.toArray(), s3.toJSON() ); // [2, 1]
[2, 3] [1]
var s1S2IntersectArray = s1.intersect( s2 ).toJSON(); // [2]
```

- OrderedSet — иммутабельное множество для хранения уникальных значений, гарантирующее такой же порядок обхода элементов, какой он был при записи.

```
var s1 = Immutable.OrderedSet( [2, 1] );
var s2 = Immutable.OrderedSet( [2, 3, 3] );
var s3 = Immutable.OrderedSet( [1, 1, 1] );
var s1S2S3UnionArray = s1.union( s2, s3 ).toJSON();// [2, 1, 3]
var s3S2S1UnionArray = s3.union( s2, s1 ).toJSON();// [1, 2, 3]
```

- Record — конструктор иммутабельных данных со значениями по умолчанию

```
var Data = Immutable.Record({
value: 5
});
var Test = Immutable.Record({
text: '',
data: new Data()
});
var test = new Test();
console.log( test.get('data').get('value') ); //5 the default
value
```

Mori

Библиотека, которая привносит персистентные структуры данных из ClojureScript (Lists, Vectors, Maps и т.д.) в JavaScript.

Отличия от Immutable.js:

- Функциональное API без публичных методов
- Быстрее
- Больший размер библиотеки

Пример использования:

```
var inc = function(n) {
  return n+1;
};

mori.intoArray(mori.map(inc, mori.vector(1,2,3,4,5)));
// => [2,3,4,5,6]

//Efficient non-destructive updates!

var v1 = mori.vector(1,2,3);
var v2 = mori.conj(v1, 4);
v1.toString(); // => '[1 2 3]'
v2.toString(); // => '[1 2 3 4]'
var sum = function(a, b) {
  return a + b;
};
mori.reduce(sum, mori.vector(1, 2, 3, 4)); // => 10

//Lazy sequences!

var _ = mori;
_.intoArray(_.interpose("foo", _.vector(1, 2, 3, 4)));
// => [1, "foo", 2, "foo", 3, "foo", 4]
```

Проблемы при разработке, с которыми вы столкнетесь

Речь пойдет об использовании Immutable.js (с Mori все примерно также). В случае работы с Seamless-Immutable таких проблем у вас не возникнет из-за обратной совместимости с нативными структурами JavaScript.

Работа с серверным API

Дело в том, что в большинстве случаев серверное API принимает и возвращает данные в формате JSON, который соответствует стандартным объектам и массивам из JavaScript. Это значит, что

нужно будет каким-то образом преобразовывать Immutable-данные в обычные и наоборот.

Immutable.js для конвертации обычных данных в иммутабельные предлагает следующую функцию:

```
Immutable.fromJS(json: any, reviver?: (k: any, v: Iterable<any,
any>) => any): any
```

где с помощью функции `reviver` можно добавлять собственные правила преобразования и управлять существующими.

Предположим, серверное API вернуло нам следующий объект:

```
var response = [
    {_id : '573b44d91fd2f10100d5f436', value : 1},
    {_id : '573dd87b212dc501001950f2', value : 2},
    {_id : '5735f6ae2a380401006af05b', value : 3},
    {_id : '56bdc2e1cee8b801000ff339', value : 4}
]
```

Удобнее всего такой объект будет представить как OrderedMap. Напишем соответствующий `reviver`:

```
var state = Immutable.fromJS(response, function(k, v){
   if(Immutable.Iterable.isIndexed(v)) {
      for(var elem of v) {
         if(!elem.get('_id')) {
            return elem;
         }
      }
      var ordered = [];
      for(var elem of v) {
         ordered.push([elem.get('_id'), elem.get('value')]);
```

```
      }
      return Immutable.OrderedMap(ordered);
   }
   return v;
});

console.log(state.toJS());
//Object {573b44d91fd2f10100d5f436: 1, 573dd87b212dc501001950f2: 2,
5735f6ae2a380401006af05b: 3, 56bdc2e1cee8b801000ff339: 4}
```

Предположим, нам нужно изменить данные и отправить их обратно на сервер:

```
state = state.setIn(['573dd87b212dc501001950f2', 5]);
console.log(state.toJS());
//Object {573b44d91fd2f10100d5f436: 1, 573dd87b212dc501001950f2: 5,
5735f6ae2a380401006af05b: 3, 56bdc2e1cee8b801000ff339: 4}
```

Immutable.js для конвертации иммутабельных данных в обычные предлагает следующую функцию:

```
toJS(): any
```

Как вы видите, `reviver` отсутствует, а это значит, что придется писать собственный внешний `immutableHelper`. И он каким-то образом должен уметь отличать обычный OrderMap от того, который соответствует структуре ваших исходных данных. Унаследоваться от OrderMap вы тоже не можете. В настоящем приложении структуры, скорее всего, окажутся вложенными, что добавит вам дополнительных сложностей.

Можно, конечно, использовать при разработке только List и Map, но тогда зачем же все остальное? И в чем плюсы использования конкретно Immutable.js?

Иммутабельность везде

Если проект раньше работал с нативными структурами данных, то легко перейти на иммутабельные не получится. Придется переписывать весь код, который хоть как-то взаимодействует с данными.

Сериализация/Десериализация

Immutable.js не предлагает нам ничего кроме функций `fromJS`, `toJS`, которые работают следующим образом:

```
var set = Immutable.Set([1, 2, 3, 2, 1]);
set = Immutable.fromJS(set.toJS());
console.log(Immutable.Set.isSet(set)); //false
console.log(Immutable.List.isList(set)); //true
```

То есть абсолютно бесполезны для сериализации/десериализации.

Существует сторонняя библиотека transit-immutable-js. Пример ее использования:

```
var transit = require('transit-immutable-js');
var Immutable = require('immutable');

var m = Immutable.Map({with: "Some", data: "In"});
```

```
var str = transit.toJSON(m);
console.log(str) // ["~#cmap",["with","Some","data","In"]]

var m2 = transit.fromJSON(str);
console.log(Immutable.is(m, m2));// true
```

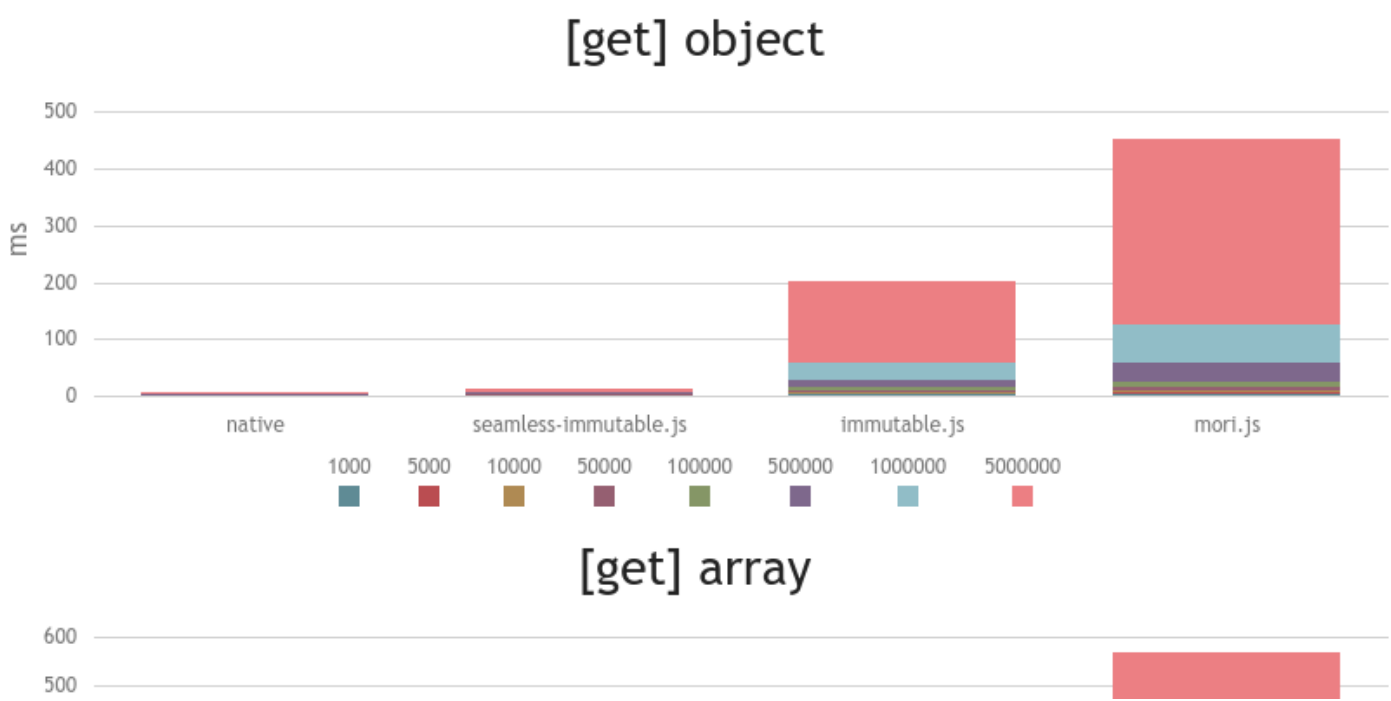
Производительность

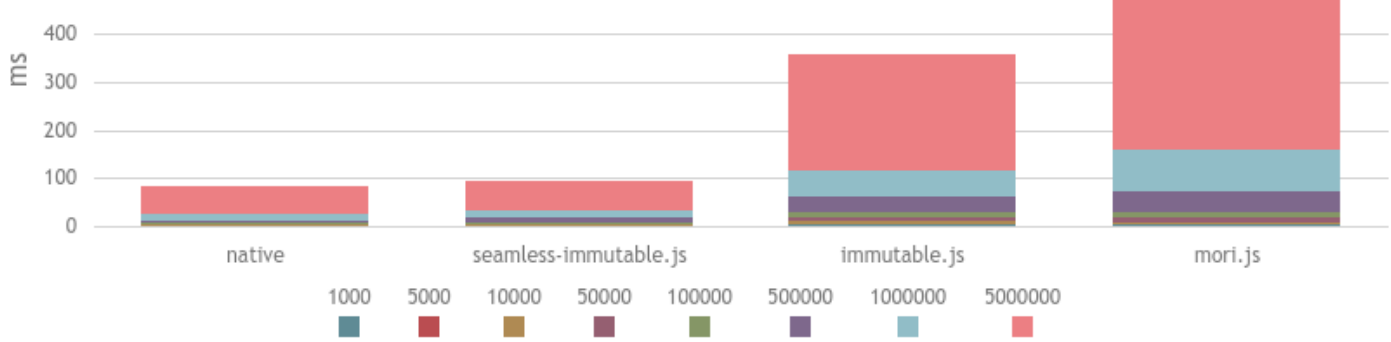
Для тестирования производительности были написаны бенчмарки. Чтобы запустить их у себя, выполните команды:

```
git clone https://github.com/MrCheater/immutable-benchmarks.git
cd ./immutable-benchmarks
npm install
npm start
```

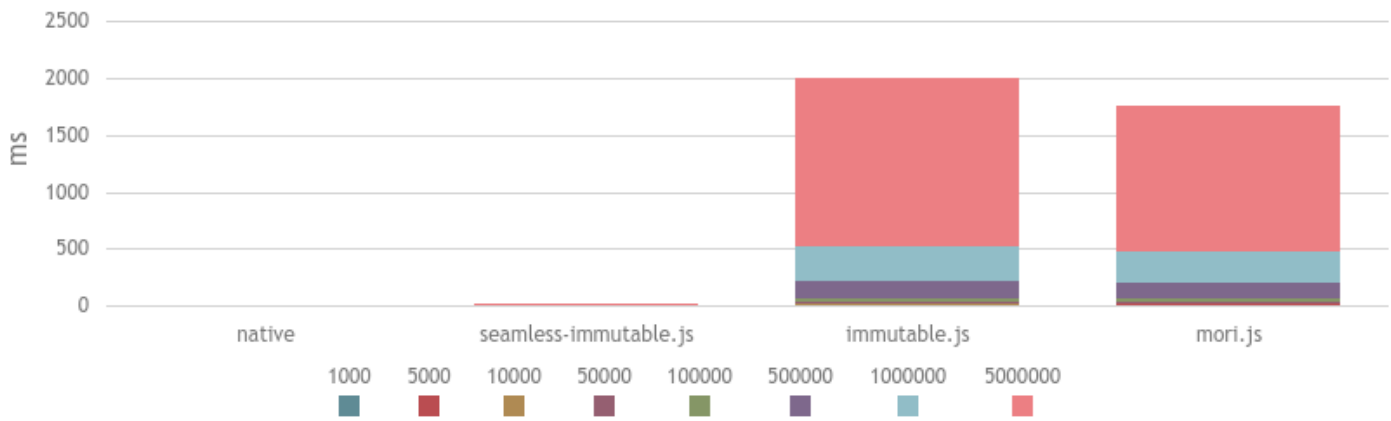
Результаты бенчмарков можно увидеть на графиках (repeats / ms). Чем больше время выполнения, тем хуже результат.

При чтении самыми быстрыми оказались нативные структуры данных и Seamless-immutable.

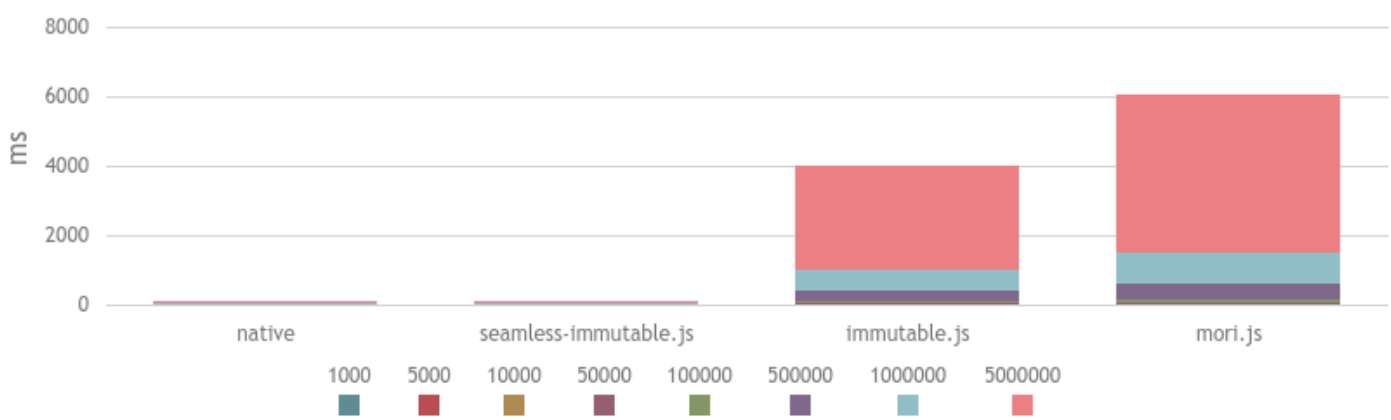




[getIn] object

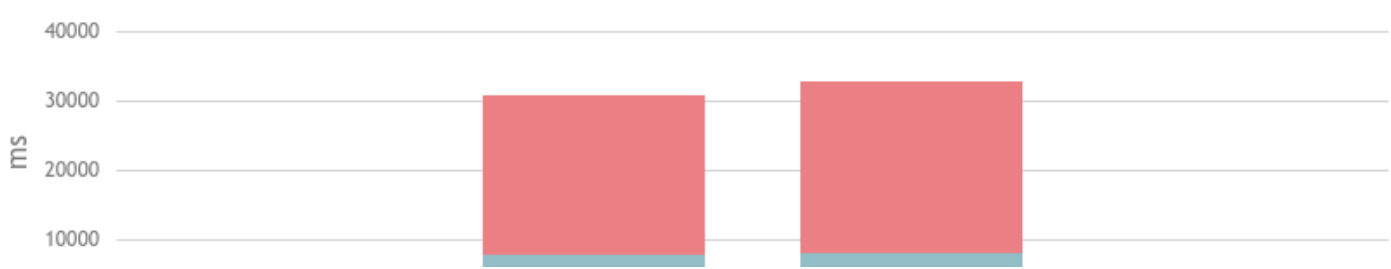


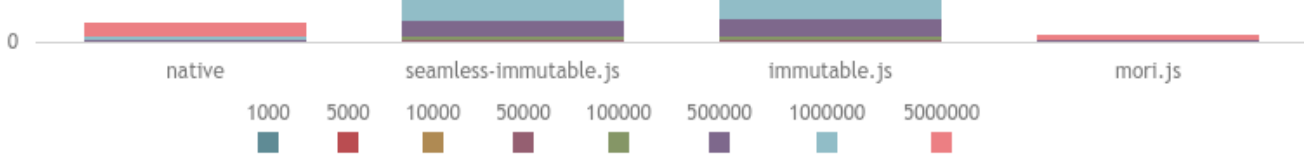
[getIn] array



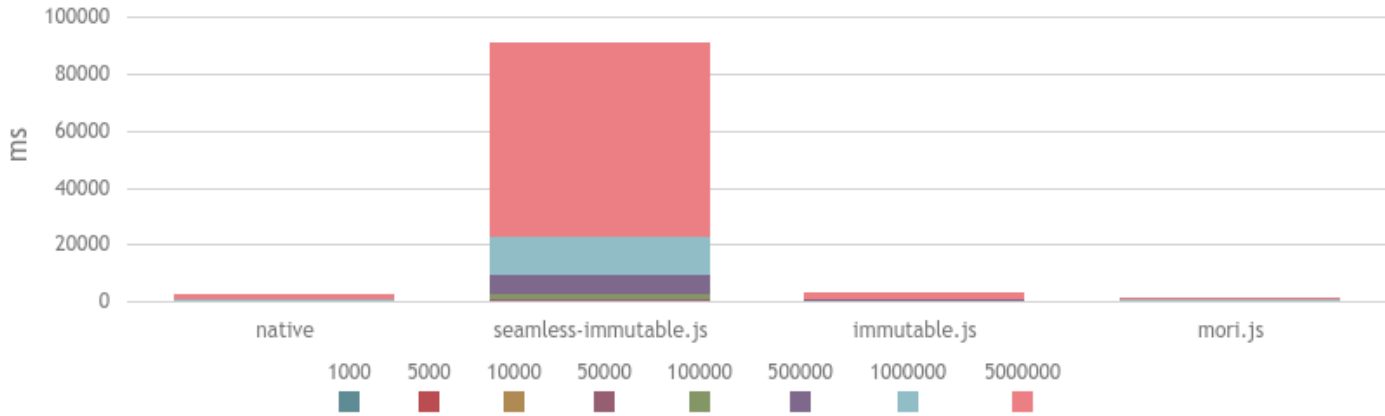
При записи самым быстрым оказался Mori. Seamless-immutable показал наихудший результат.

[set] object

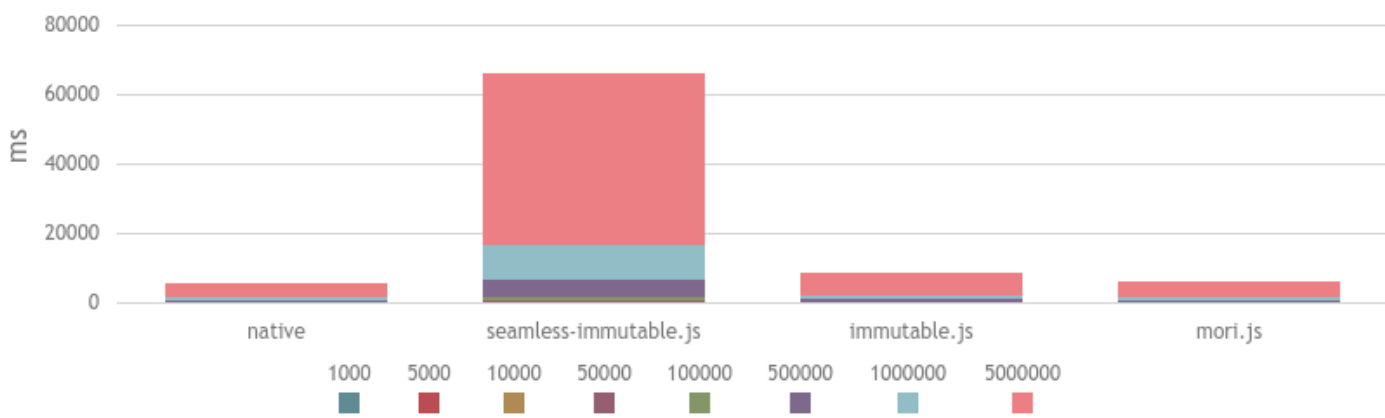




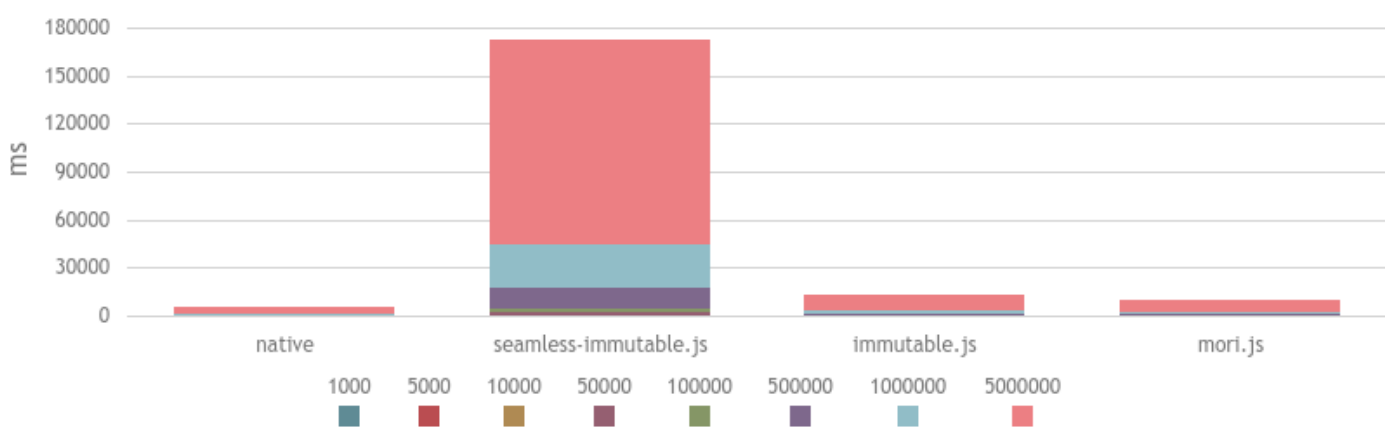
[set] array



[setIn] object



[setIn] array



Заключение

Эта статья будет полезна JavaScript-разработчикам, столкнувшимся с необходимостью использовать иммутабельные данные в своих приложениях для повышения производительности. В частности, это касается frontend-разработчиков, которые работают с фреймворками, использующими VirtualDOM (React, Mithril, Riot), а также Flux/Redux решения.

Подводя итоги, можно сказать, что среди рассмотренных библиотек для иммутабельности в JavaScript самая быстрая, удобная и простая в использовании это Seamless-immutable. Самая стабильная и распространенная — Immutable.js. Самая быстрая на запись и самая необычная — Mori. Надеюсь данное исследование поможет выбрать вам решение для своего проекта. Удачи.

Захвати и визуализируй! Или гистограмма с микрофона средствами Web Audio API 15

tatyana_ryzh • 9 марта 2016 в 12:11

Релиз компонентов DevExpress .NET, HTML5/JS и VCL v2015.2 10

AlexAkimov • 30 декабря 2015 в 15:46

ChartJS — JavaScript-библиотека визуализации данных 42

kaatula • 9 июля 2013 в 11:21

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

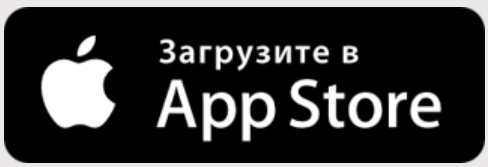
marks • вчера в 14:46

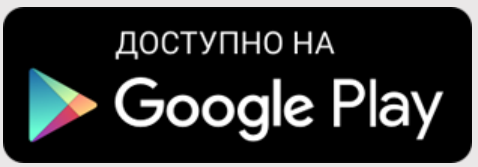
Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32

Мобильное приложение





Полная версия