

Полезные приёмы работы с массивами в JavaScript

В большинстве приложений, которые разрабатываются в наши дни, требуется взаимодействовать с некими наборами данных. Обработка элементов в коллекциях — это часто встречающаяся операция, с которой вы, наверняка, сталкивались. При работе, например, с массивами, можно, не задумываясь, пользоваться обычным циклом `for`, который выглядит примерно так: `for (var i=0; i < value.length; i++) {}`. Однако, лучше, всё-таки, смотреть на вещи шире.

A large, bold, dark grey 'JS' logo is centered on a solid yellow rectangular background.

Предположим, нам надо вывести список товаров, и, при необходимости, разбивать его на категории, фильтровать, выполнять по нему поиск, модифицировать этот список или его элементы. Возможно, требуется быстро выполнить некие вычисления, в которые будут вовлечены элементы списка. Скажем, надо что-то с чем-то сложить, что-то на что-то умножить. Можно ли найти в JavaScript такие средства, которые позволяют решать подобные задачи быстрее и удобнее, чем с использованием обычного цикла `for`?

На самом деле, такие средства в JavaScript имеются. Некоторые из них рассмотрены в материале, перевод которого мы представляем сегодня вашему вниманию. В частности, речь идёт об операторе расширения, о цикле `for...of`, и о методах `includes()`, `some()`, `every()`, `filter()`, `map()` и `reduce()`. Здесь мы, в основном, будем говорить о массивах, но рассматриваемые здесь методики обычно подходят и для работы с объектами других типов.

Надо отметить, что обзоры современных подходов к разработке на JS обычно включают в себя примеры, подготовленные с использованием стрелочных функций. Возможно, вы не

особенно часто пользуетесь ими — может быть из-за того, что вам они не нравятся, может быть потому, что не хотите тратить слишком много времени на изучение чего-то нового, а, возможно, они просто вам не подходят. Поэтому здесь, в большинстве ситуаций, будут показаны два варианта выполнения одних и тех же действий: с использованием обычных функций (ES5) и с применением стрелочных функций (ES6). Для тех, у кого нет опыта работа со стрелочными функциями, отметим, что стрелочные функции не являются эквивалентами объявлений функций и функциональных выражений. Не стоит механически заменять одно на другое. В частности, это связано с тем, что в обычных и стрелочных функциях ключевое слово `this` ведёт себя по-разному.

1. Оператор расширения

Оператор расширения (spread operator) позволяет «раскрывать» массивы, подставляя в то место, где использован этот оператор, вместо массивов, их элементы. Похожий подход предложен и для литералов объектов.

■ Сильные стороны оператора расширения

- Это — простой и быстрый способ «вытащить» из массива его отдельные элементы.
- Этот оператор подходит для работы с литералами массивов и объектов.
- Это — быстрый и интуитивно понятный метод работы с аргументами функций.
- Оператор расширения не занимает много места в коде — он выглядит как три точки (...).

■ Пример

Предположим, перед вами стоит задача вывести список ваших любимых угощений, не используя при этом цикл. С помощью оператора расширения это делается так:



```
const favoriteFood = ['Pizza', 'Fries', 'Swedish-meatballs'];  
  
console.log(...favoriteFood);  
// Pizza Fries Swedish-meatballs
```

2. Цикл for...of

Оператор `for...of` предназначен для обхода итерируемых объектов. Он даёт доступ к отдельным элементам таких объектов (в частности — к элементам массивов), что, например, позволяет их модифицировать. Его можно считать заменой обычному циклу `for`.

Сильные стороны цикла for...of

- Это — простой способ для добавления или обновления элементов коллекций.
- Цикл `for...of` позволяет выполнять различные вычисления с использованием элементов (суммирование, умножение, и так далее).
- Им удобно пользоваться при необходимости выполнения проверки каких-либо условий.
- Его использование ведёт к написанию более чистого и читабельного кода.

Пример

Предположим, у вас имеется структура данных, описывающая содержимое ящика с инструментами и вам надо показать эти инструменты. Вот как это сделать с помощью цикла `for...of`:

```
const toolbox = ['Hammer', 'Screwdriver', 'Ruler'];

for(item of toolbox){

    console.log(item);

}
// Hammer
// Screwdriver
// Ruler
```

3. Метод includes()

Метод `includes()` используется для проверки наличия в коллекции некоего элемента, в частности, например, определённой строки в массиве, содержащем строки. Этот метод возвращает `true` или `false` в зависимости от результатов проверки. Пользуясь им, стоит учитывать, что он чувствителен к регистру символов. Если, например, в коллекции есть строковый элемент `SCHOOL`, а проверка на его наличие с помощью `includes()` выполняется по строке `school`, метод вернёт `false`.

Сильные стороны метода includes()

- Метод `includes()` полезен в деле создания простых механизмов поиска данных.
- Он даёт разработчику интуитивно понятный способ определения наличия неких данных в массиве.
- Его удобно использовать в условных выражениях для модификации, фильтрации элементов, и для выполнения других операций.
- Его применение ведёт к улучшению читабельности кода.

Пример

Предположим, у вас имеется гараж, представленный массивом со списком автомобилей, и вы не знаете, есть в этом гараже некий автомобиль, или нет. Для того чтобы решить эту проблему, надо написать код, который позволяет проверять наличие автомобиля в гараже. Воспользуемся методом `includes()`:

```
const garage = ['BMW', 'AUDI', 'VOLVO'];  
const findCar = garage.includes('BMW');  
console.log(findCar);  
// true
```

4. Метод `some()`

Метод `some()` позволяет проверить, существуют ли некоторые из искомых элементов в массиве. Он, по результатам проверки, возвращает `true` или `false`. Он похож на вышерассмотренный метод `includes()`, за исключением того, что его аргументом является функция, а не, например, обычная строка.

Сильные стороны метода `some()`

- Метод `some()` позволяет проверить, имеется ли в массиве хотя бы один из интересующих нас элементов.
- Он выполняет проверку условия с использованием переданной ему функции.
- Он способствует применению декларативного подхода при программировании.
- Этим методом удобно пользоваться.

Пример

Предположим, вы — владелец клуба, и в общем-то, вас не интересует — кто именно в ваш клуб приходит. Однако, некоторым посетителям вход в клуб закрыт, так как они склонны к излишнему потреблению спиртных напитков, по крайней мере, в том случае, если они оказываются в вашем заведении сами, и с ними нет никого, кто может за ними присмотреть. В данном случае группе посетителей можно войти в клуб только при условии, что хотя бы одному из них не меньше 18-ти лет. Для того чтобы автоматизировать проверку подобного рода, воспользуемся методом `some()`. Ниже его применение продемонстрировано в двух вариантах.

ES5

```
const age = [16, 14, 18];  
age.some(function(person){  
    return person >= 18;  
});  
// Output: true
```

ES6



```
const age = [16, 14, 18];  
  
age.some((person) => person >= 18);  
  
// true
```

5. Метод every()

Метод `every()` обходит массив и проверяет каждый его элемент на соответствие некоему условию, возвращая `true` в том случае, если все элементы массива соответствуют условию, и `false` в противном случае. Можно заметить, что он похож на метод `some()`.


Сильные стороны метода every()

- Метод `every()` позволяет проверить соответствие условию всех элементов массива.
- Условия можно задавать с использованием функций.
- Он способствует применению декларативного подхода при программировании.

Пример

Вернёмся к предыдущему примеру. Там вы пропускали в клуб посетителей, не достигших 18 лет, но кто-то написал заявление в полицию, после чего вы попали в неприятную ситуацию. После того, как всё удалось уладить, вы решили, что вам всё это ни к чему и ужесточили правила посещения клуба. Теперь группа посетителей может пройти в клуб только в том случае, если возраст каждого члена группы не меньше 18 лет. Как и в прошлый раз, рассмотрим решение задачи в двух вариантах, но на этот раз будем пользоваться методом `every()`.

ES5




```
const age = [15, 20, 19];

age.every(function(person){
  return person >= 18;
});

// Output: false
```

ES6



```
const age = [15, 20, 19];

age.every((person) => person >= 18);

// false
```


6. Метод filter()

Метод `filter()` позволяет создать, на основе некоего массива, новый массив, содержащий только те элементы исходного массива, которые удовлетворяют заданному условию.

Сильные стороны метода filter()

- Метод `filter()` позволяет избежать модификации исходного массива.
- Он позволяет избавиться от ненужных элементов.
- Он улучшает читабельность кода.

Пример

Предположим, вам надо отобрать из списка цен только те, которые больше или равны 30. Воспользуемся для решения этой задачи методом `filter()`.

ES5

```
// array
const prices = [25, 30, 15, 55, 40, 10];

prices.filter(function(price){
  return price >= 30;
});

// Output: [ 30, 55, 40 ]
```

```
const prices = [25, 30, 15, 55, 40, 10];  
prices.filter((price) => price >= 30);  
// [30, 55, 40]
```

7. Метод `map()`

Метод `map()` похож на метод `filter()` тем, что он тоже возвращает новый массив. Однако он применяется для модификации элементов исходного массива.

Сильные стороны метода `map()`

- Метод `map()` позволяет избежать необходимости изменения элементов исходного массива.
- С его помощью удобно модифицировать элементы массивов.
- Он улучшает читаемость кода.

Пример

Предположим, у вас имеется список товаров с ценами. Вашему менеджеру нужен новый список товаров, цены которых снижены на 25%. Воспользуемся для решения этой задачи

методом `map()`.

ES5

```
const productPriceList = [200, 350, 1500, 5000];  
productPriceList.map(function(item){  
    return item * 0.75;  
});  
// [ 150, 262.5, 1125, 3750 ]
```

ES6

```
const productPriceList = [200, 350, 1500, 5000];  
productPriceList.map((item) => item * 0.75);  
// [ 150, 262.5, 1125, 3750 ]
```

8. Метод `reduce()`

Метод `reduce()`, в его простейшем виде, позволяет суммировать элементы числовых массивов. Другими словами, он сводит массив к единственному значению. Это позволяет использовать его для выполнения различных вычислений.

Сильные стороны метода `reduce()`

- С помощью метода `reduce()` можно посчитать сумму или среднее значение элементов массива.
- Этот метод ускоряет и упрощает проведение вычислений.

Пример

Предположим, вам надо посчитать ваши расходы за неделю, которые хранятся в массиве. Решим эту задачу с помощью метода `reduce()`.

ES5

```
const weeklyExpenses = [200, 350, 1500, 5000, 450, 680, 350];  
weeklyExpenses.reduce(function(first, last){  
    return first + last;  
});  
// 8350
```

ES6



```
const weeklyExpenses = [200, 350, 1500, 5000, 450, 680, 350];  
  
weeklyExpenses.reduce((first, last) => first + last);  
// 8350
```

Итоги

В этом материале мы рассмотрели некоторые полезные приёмы, которые упрощают и ускоряют работу с массивами и улучшают читаемость кода. Если сегодня состоялось ваше первое знакомство с этими приёмами, рекомендуем, пользуясь полученной здесь базой, узнать о них побольше и поэкспериментировать с ними самостоятельно. Уверены, всё это вам пригодится.

Версия #1

Seryak создал Wed, Sep 16, 2020 3:45 AM

Seryak обновил Wed, Sep 16, 2020 3:45 AM