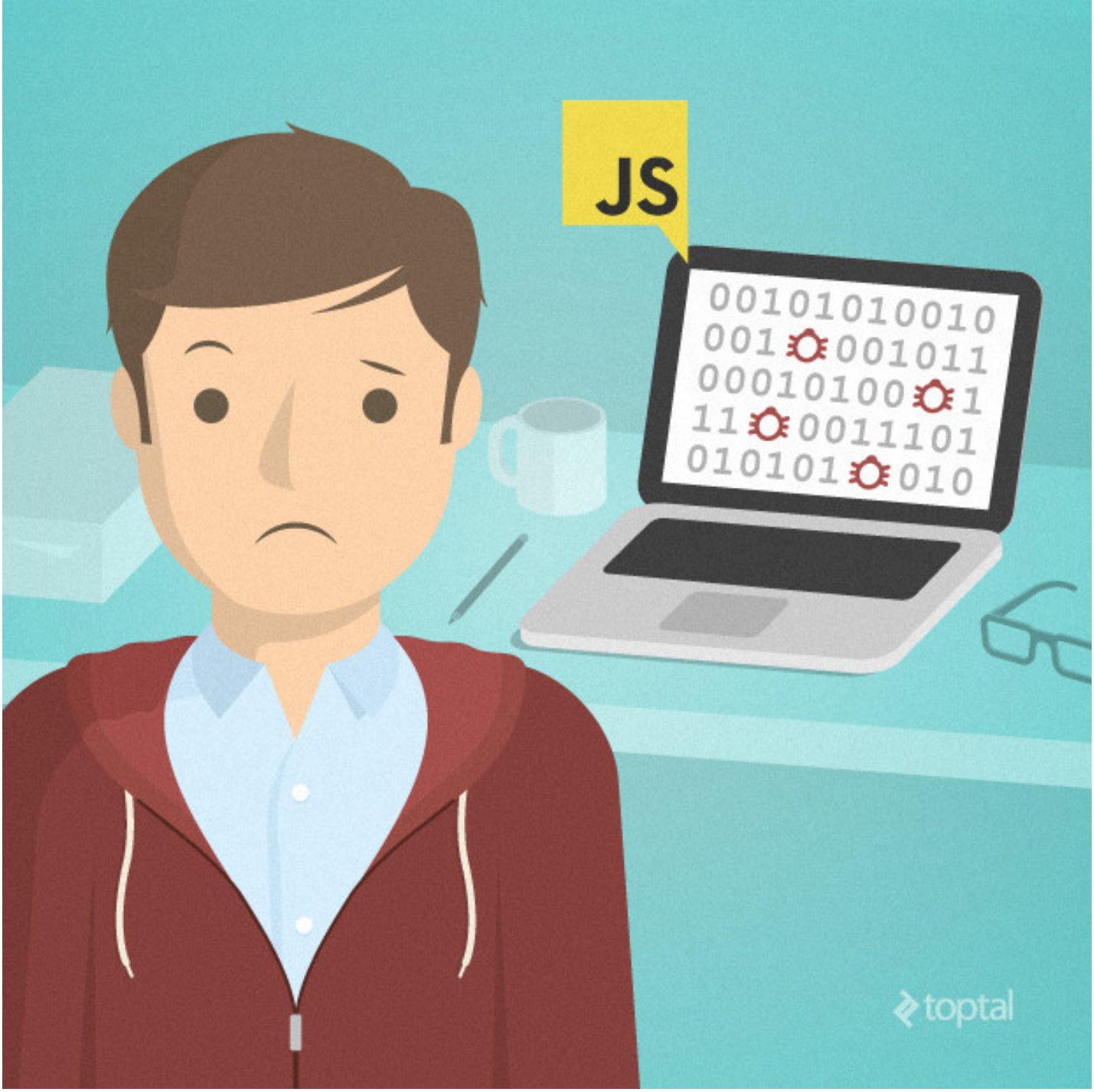
БЛОГ КОМПАНИИ MAIL.RU GROUP, JAVASCRIPT*, РАЗРАБОТКА ВЕБ-САЙТОВ*

10 самых распространённых ошибок при программировании на JavaScript

ПЕРЕВОД

ZaValera 16 августа 2014 в 20:08

Оригинал: Ryan J. Peterson



Сегодня JavaScript лежит в основе большинства современных веб-приложений. При этом за последние годы появилось большое количество JavaScript-библиотек и фреймворков для разработчиков Single Page Application (SPA), графики, анимации и даже серверных платформ. Для веб-разработки JavaScript используется повсеместно, и поэтому качество кода обретает всё большее значение.

На первый взгляд, этот язык может показаться довольно простым. Встраивание в веб-страницу базового функционала JavaScript — это не проблема для любого опытного разработчика, даже если он ранее не сталкивался с этим языком. Однако это обманчивое впечатление, поскольку JavaScript гораздо сложнее, мощнее и чувствительнее к нюансам, чем кажется поначалу. Немало тонкостей в этом языке приводит к большому количеству распространённых ошибок. Сегодня мы рассмотрим некоторые из них. На эти ошибки нужно обратить особое внимание, если вы хотите отлично программировать на JavaScript.

1. Неправильные ссылки на this

В связи с тем, что за последние годы программирование на JavaScript сильно усложнились, соответственно возросло количество случаев появления функций обратного вызова и замыканий, которые часто являются причиной путаницы с ключевым словом `this`.

Например, выполнение этого кода:

```
Game.prototype.restart = function () {
    this.clearLocalStorage();
    this.timer = setTimeout(function() {
        this.clearBoard();    // что здесь "this"?
    }, 0);
};
```

Приводит к ошибке:

```
Uncaught TypeError: undefined is not a function
```

Почему это происходит? Всё дело в контексте. Когда вы вызываете

`setTimeout()`, то на самом деле вызываете `window.setTimeout()`. В результате, анонимная функция, передаваемая в `setTimeout()`, определяется в контексте объекта `window`, который не имеет метода `clearBoard()`.

Традиционное решение, совместимое со старыми браузерами, предполагает простое сохранение ссылки на `this` в переменной, которая может быть сохранена в замыкании:

```
Game.prototype.restart = function () {
    this.clearLocalStorage();
    var self = this;   // сохраним ссылку на 'this', пока это все
еще 'this'!
    this.timer = setTimeout(function(){
        self.clearBoard();    // все в порядке
    }, 0);
};
```

Для новых браузеров можно использовать метод `bind()`, позволяющий связать функцию с контекстом исполнения:

```
Game.prototype.restart = function () {
    this.clearLocalStorage();
    this.timer = setTimeout(this.reset.bind(this), 0);  // связываем
'this'
};

Game.prototype.reset = function(){
    this.clearBoard();    // возвращаемся в контекст правильного
'this'!
};
```

2. Область видимости на уровне блоков

Разработчики часто считают, что JavaScript создаёт новую область видимости для каждого блока кода. Хоть это и справедливо для

многих других языков, но в JavaScript этого не происходит. Посмотрим на этот код:

```
for (var i = 0; i < 10; i++) {
    /* ... */
}
console.log(i);  // что здесь выведется?
```

Если вы думаете, что вызов `console.log()` повлечёт за собой вывод `undefined` или ошибку, то вы ошибаетесь: будет выведено «10». Почему? В большинстве других языков этот код привёл бы к появлению ошибки, потому что область видимости переменной `i` была бы ограничена блоком `for`. Однако в JavaScript эта переменная остаётся в области видимости даже после завершения цикла `for`, сохраняя своё последнее значение (такое поведение известно как «var hoisting»). Надо заметить, что поддержка области видимости на уровне блоков введена в JavaScript начиная с версии 1.7 с помощью дескриптора `let`.

3. Утечки памяти

Утечки памяти практически неизбежны, если во время работы вы не будете их сознательно избегать. Существует немало причин для появления утечек, но мы остановимся лишь на самых частых.

Ссылки на несуществующие объекты. Проанализируем этот код:

```
var theThing = null;
var replaceThing = function () {
    var priorThing = theThing;
    var unused = function () {
        // 'unused' - единственное место, где используется
'priorThing',
        // но 'unused' никогда не вызывается
        if (priorThing) {
```

```
            console.log("hi");
        }
    };
    theThing = {
        longStr: new Array(1000000).join('*'),  // создаем 1Мб
объект
        someMethod: function () {
            console.log(someMessage);
        }
    };
};
setInterval(replaceThing, 1000);    // вызываем 'replaceThing'
каждую секунду
```

Если запустить выполнение этого кода, то можно обнаружить массивную утечку памяти со скоростью около мегабайта в секунду. Создается впечатление, что мы теряем память выделенную под `longStr` при каждом вызове `replaceThing`. В чём причина?

Каждый объект `theThing` содержит свой собственный объект `longStr` размером 1Мб. Каждую секунду при вызове `replaceThing`, функция сохраняет ссылку на предыдущий объект `theThing` в переменной `priorThing`. Это не проблема, ведь каждый раз предыдущая ссылка `priorThing` будет перетерта (`priorThing = theThing;`). Так в чём же причина утечки?

Типичный способ реализации замыкания — это создание связи между каждым объектом-функцией и объектом-словарем, представляющим собой лексическую область видимости для этой функции. Если обе функции (`unused` и `someMethod`), определенные внутри `replaceThing`, реально используют `priorThing`, то важно понимать, что они получают один и тот же объект, даже если `priorThing` переписывается снова и снова, так

как обе функции используют одну и ту же лексическую область видимости. И как только переменная используется в любом из замыканий, то она попадает в лексическую область видимости, используемую всеми замыканиями в этой области видимости. И этот маленький нюанс приводит к мощной утечке памяти.

Циклические ссылки. Рассмотрим пример кода:

```
function addClickHandler(element) {
    element.click = function onClick(e) {
        alert("Clicked the " + element.nodeName)
    }
}
```

Здесь `onClick` имеет замыкание, в котором сохраняется ссылка на `element`. Назначив `onClick` в качестве обработчика события `click` для `element`, мы создали циклическую ссылку: `element` -> `onClick` -> `element` -> `onClick` -> `element`...

Даже если удалить `element` из DOM, то циклическая ссылка скроет `element` и `onClick` от сборщика мусора и произойдет утечка памяти. Как лучше всего избегать возникновения утечек? Управление памятью в JavaScript (и в частности сборка мусора) в значительной степени основано на понятии достижимости объекта. Следующие объекты считаются достижимыми и известны как корневые:

- ссылки на которые содержатся в стеке вызова (все локальные переменные и параметры функций, которые в настоящий момент вызываются, а также все переменные в области видимости замыкания);
- все глобальные переменные.

Объекты сохраняется в памяти лишь до тех пор, пока доступны из корневых по ссылке или цепочке ссылок.

В браузерах встроен сборщик мусора, который очищает память от недостижимых объектов. То есть объект будет удалён из памяти **только если** сборщик мусора решит, что он недостижим. К сожалению, довольно легко могут накопиться неиспользуемые большие объекты, которые считаются «достижимыми».

4. Непонимание равенства

Одним из преимуществ JavaScript является то, что он автоматически преобразует любое значение к булевому значению, если оно используется в булевом контексте. Однако бывают случаи, когда это удобство может ввести в заблуждение:

```
// Все эти сравнения выдадут 'true'!
console.log(false == '0');
console.log(null == undefined);
console.log(" \t\r\n" == 0);
console.log('' == 0);

// И эти тоже!
if ({}) // ...
if ([]) // ...
```

С учётом последних двух строк, даже будучи пустыми, `{}` и `[]` фактически являются объектами. А любой объект в JavaScript соответствует булевому значению `true`. Однако многие разработчики считают, что значение будет `false`.

Как показывают два приведённых примера, автоматическое преобразование типа иногда может мешать. Как правило лучше

использовать `===` и `!==` вместо `==` и `!=`, чтобы избежать побочных эффектов преобразования типов.

Кстати, сравнение `NaN` с чем-либо (даже с `NaN`!) всегда даст результат `false`. Таким образом, нельзя использовать операторы равенства (`==`, `===`, `!=`, `!==`) для определения соответствия значения `NaN`. Вместо этого нужно использовать встроенную глобальную функцию `isNaN()`:

```
console.log(NaN == NaN);    // false
console.log(NaN === NaN);   // false
console.log(isNaN(NaN));    // true
```

5. Нерациональное использование DOM

В JavaScript можно легко работать с DOM (в том числе добавлять, изменять и удалять элементы), но часто разработчики делают это неэффективно. Например, добавляют серии элементов по одному за раз. Однако операция добавления элементов весьма затратна, и последовательного её выполнения нужно избегать.

Если нужно добавить несколько элементов, то, в качестве альтернативы, можно использовать фрагменты документа:

```
var div = document.getElementsByTagName("my_div");

var fragment = document.createDocumentFragment();

for (var e = 0; e < elems.length; e++) {
    fragment.appendChild(elems[e]);
}
div.appendChild(fragment.cloneNode(true));
```

Также рекомендуем сначала создавать и модифицировать

элементы, а потом уже добавлять в DOM, это также существенно повышает производительность.

6. Некорректное использование определений функций внутри циклов for

Рассмотрим пример кода:

```
var elements = document.getElementsByTagName('input');
var n = elements.length;    // предположим, у нас есть 10 элементов
for (var i = 0; i < n; i++) {
    elements[i].onclick = function() {
        console.log("This is element #" + i);
    };
}
```

При клике на любом из 10 элементов появлялось бы сообщение «This is element #10». Причина в том, что к тому времени, когда `onclick` вызывается любым из элементов, вышестоящий цикл `for` будет завершён, а значение `i` будет равно 10.

Пример правильного кода:

```
var elements = document.getElementsByTagName('input');
var n = elements.length;    // предположим, у нас есть 10 элементов
var makeHandler = function(num) {  // внешняя функция
     return function() {   // внутренняя функция
         console.log("This is element #" + num);
     };
};
for (var i = 0; i < n; i++) {
    elements[i].onclick = makeHandler(i+1);
}
```

`makeHandler` немедленно запускается на каждой итерации цикла, получает текущее значение `i+1` и сохраняет его в переменной `num`. Внешняя функция возвращает внутреннюю функцию (которая

также использует переменную `num`) и устанавливает ее в качестве обработчика `onclick`. Это позволяет гарантировать, что каждый `onclick` получает и использует правильное значение `i`.

7. Неправильное наследование через прототипы

Удивительно много разработчиков не имеют ясного понимания механизма наследования через прототипы. Рассмотрим пример кода:

```
BaseObject = function(name) {
    if(typeof name !== "undefined") {
        this.name = name;
    } else {
        this.name = 'default'
    }
};

var firstObj = new BaseObject();
var secondObj = new BaseObject('unique');

console.log(firstObj.name);  // -> в 'default'
console.log(secondObj.name); // -> в 'unique'
```

Но если бы мы написали так:

```
delete secondObj.name;
```

то получили бы:

```
console.log(secondObj.name); // -> в 'undefined'
```

Но не лучше ли вернуть значение к `default`? Это можно легко сделать, если применить наследование через прототипы:

```
BaseObject = function (name) {
    if(typeof name !== "undefined") {
        this.name = name;
```

```
    }
};

BaseObject.prototype.name = 'default';
```

Каждый экземпляр `BaseObject` наследует свойство name своего прототипа, в котором ему присвоено значение `default`. Таким образом, если конструктор вызван без `name`, свойство `name` по умолчанию будет `default`. И точно так же, если свойство `name` будет удалено из экземпляра `BaseObject`, будет произведен поиск по цепочке прототипов и свойство `name` будет получено из объекта `prototype`, в котором оно по-прежнему равно `default`:

```
var thirdObj = new BaseObject('unique');
console.log(thirdObj.name);  // -> в 'unique'

delete thirdObj.name;
console.log(thirdObj.name);  // -> в 'default'
```

8. Создание неправильных ссылок на методы экземпляров

Определим простой конструктор и с помощью него создадим объект:

```
var MyObject = function() {}

MyObject.prototype.whoAmI = function() {
    console.log(this === window ? "window" : "MyObj");
};

var obj = new MyObject();
```

Для удобства, создадим ссылку на метод `whoAmI`:

```
var whoAmI = obj.whoAmI;
```

Выведем значение нашей новой переменной `whoAmI`:

```
console.log(whoAmI);
```

В консоли будет выведено:

```
function () {
    console.log(this === window ? "window" : "MyObj");
}
```

А теперь обратите внимание на разницу при вызовах `obj.whoAmI()` и `whoAmI()`:

```
obj.whoAmI();  // выведет "MyObj" (как и ожидалось)
whoAmI();      // выведет "window"
```

Что пошло не так? Когда мы присвоили `var whoAmI = obj.whoAmI;`, новая переменная была определена в глобальном пространстве имён. В результате значение `this` оказалось равным `window`, а не `obj`, экземпляру `MyObject`. Таким образом, если нам действительно нужно создать ссылку на существующий метод объекта, необходимо сделать это в пределах пространства имён этого объекта. Например:

```
var MyObject = function() {}

MyObject.prototype.whoAmI = function() {
    console.log(this === window ? "window" : "MyObj");
};

var obj = new MyObject();
obj.w = obj.whoAmI;   // в пространстве имен объекта

obj.whoAmI();  // выведет "MyObj" (как и ожидалось)
obj.w();       // выведет "MyObj" (как и ожидалось)
```

9. Использование строки в качестве первого аргумента в setTimeout или setInterval

Само по себе это не является ошибкой. И дело тут не только в производительности. Дело в том, что когда вы передаете строковую переменную первым аргументом в `setTimeout` или `setInterval`, она будет передана конструктору `Function` для преобразования в новую функцию. Этот процесс может быть медленным и неэффективным. Альтернативой является использование функции в качестве первого аргумента:

```
setInterval(logTime, 1000);   // передаем функцию logTime в
setInterval

setTimeout(function() {       // передаем анонимную функцию в
setTimeout
    logMessage(msgValue);     // (msgValue здесь всё ещё доступна)
}, 1000);
```

10. Отказ от использования «strict mode»

Это режим, в котором накладывается ряд ограничений на исполняемый код, что повышает безопасность и может предотвратить появление некоторых ошибок. Конечно, отказ от использования «строгого режима» не является ошибкой как таковой. Просто в этом случае вы лишаете себя ряда преимуществ:

- Облегчение процесса отладки. Ошибки в коде, которые были бы проигнорированы или не замечены, приведут к появлению предупреждений и генерации исключений, которые быстрее приведут вас к источнику проблемы.

- Предотвращение случайного появления глобальных переменных. Присвоение значения необъявленной переменной автоматически создаёт глобальную переменную с таким именем. Это одна из наиболее распространённых ошибок в JavaScript. В «строгом режиме» это приведёт к появлению сообщения об ошибке.
- Запрет на дублирование названий свойств или значений параметров. Если при включённом «строгом режиме» у объекта обнаруживается дублирование названий свойств (например, `var object = {foo: "bar", foo: "baz"};`) или названий аргументов у функции, то будет выведено сообщение об ошибке. Это позволяет быстро обнаружить и устранить баг.
- Уменьшение потенциальной опасности `eval()`. В «строгом режиме» переменные и функции, объявленные внутри `eval()`, не создаются в текущей области видимости.
- Получение сообщения об ошибке при ошибочном использовании оператора `delete`. Этот оператор не может быть применён к свойствам объекта, у которых флаг `configurable` равен `false`, и при попытке это сделать будет выведено сообщение об ошибке.

В завершение

Чем лучше понимаешь, как и почему работает JavaScript, тем более надёжным будет код, тем эффективнее можно использовать возможности этого языка. И наоборот, недопонимание заложенных в JavaScript парадигм становится причиной большого количества багов в программных продуктах.

Поэтому изучение нюансов и тонкостей языка является наиболее

эффективной стратегией повышения своего профессионализма и продуктивности, а также поможет избежать многих распространённых ошибок при написании JavaScript-кода.

Похожие публикации

Правда о традиционных JavaScript-бенчмарках

12

ПЕРЕВОД

AloneCoder • 26 декабря 2016 в 15:55

Правильная обработка ошибок в JavaScript

10

ПЕРЕВОД

mitasovr • 21 апреля 2016 в 18:04

Новая ICQ для Windows, открытый код и кое-что еще

107

Dimitryopho • 16 марта 2016 в 17:52

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