JavaScript的设计缺陷

在本文中，笔者将会为大家介绍一些JavaScript在发展过程中的一些**重要的**设计缺陷，这些缺陷往往使得JS新手在开发过程中**容易犯错**。

过时的功能

有状态的 RegExp 函数

stop talking show me the code

```
const re = /example/g;
console.log(re.test('example test'));
console.log(re.test('example test'));
```

按照大家对于正则的理解，上面的返回结果应该是`true true`

先来看看在控制台上运行以上代码的结果：

```
> const re = /example/g;
  console.log(re.test('example test'));
  console.log(re.test('example test'));
  true
  false
```

最后代码运行的结果和我们的预期结果不一致，导致这样的结果是因为在JS的RegExp中存在一个lastIndex属性。

只有正则表达式使用了表示全局检索的 "g" 标志时，该属性才会起作用。此时应用下面的规则：
* 如果 lastIndex 大于字符串的长度，则 regexp.test 和 regexp.exec 将会匹配失败，然后 lastIndex 被设置为 0。
* 如果 lastIndex 等于字符串的长度，且该正则表达式匹配空字符串，则该正则表达式匹配从 lastIndex 开始的字符串。（then the regular expression matches input starting at lastIndex.）
* 如果 lastIndex 等于字符串的长度，且该正则表达式不匹配空字符串，则该正则表达式不匹配字符串，lastIndex 被设置为 0.。

* 否则，lastIndex 被设置为紧随最近一次成功匹配的下一个位置。

复杂的类型系统

隐士转换之运算

不管是JS新手还是很有经验的老司机，JS的类型转换可以说是这门语言的一个让人“又爱又恨”的地方了，让我们先来几个例子看看

```
('foo' + + 'bar') === 'fooNaN'
'3' + 1
'3' - 1
'222' - - '111'
```

在下面公布代码运行结果之前，大家可以先自己给出一份结果，然后验证一下

```
('foo' + + 'bar') === 'fooNaN'
true
'3' + 1
"31"
'3' - 1
2
'222' - - '111'
333
```

隐士转换之等于

先看代码：

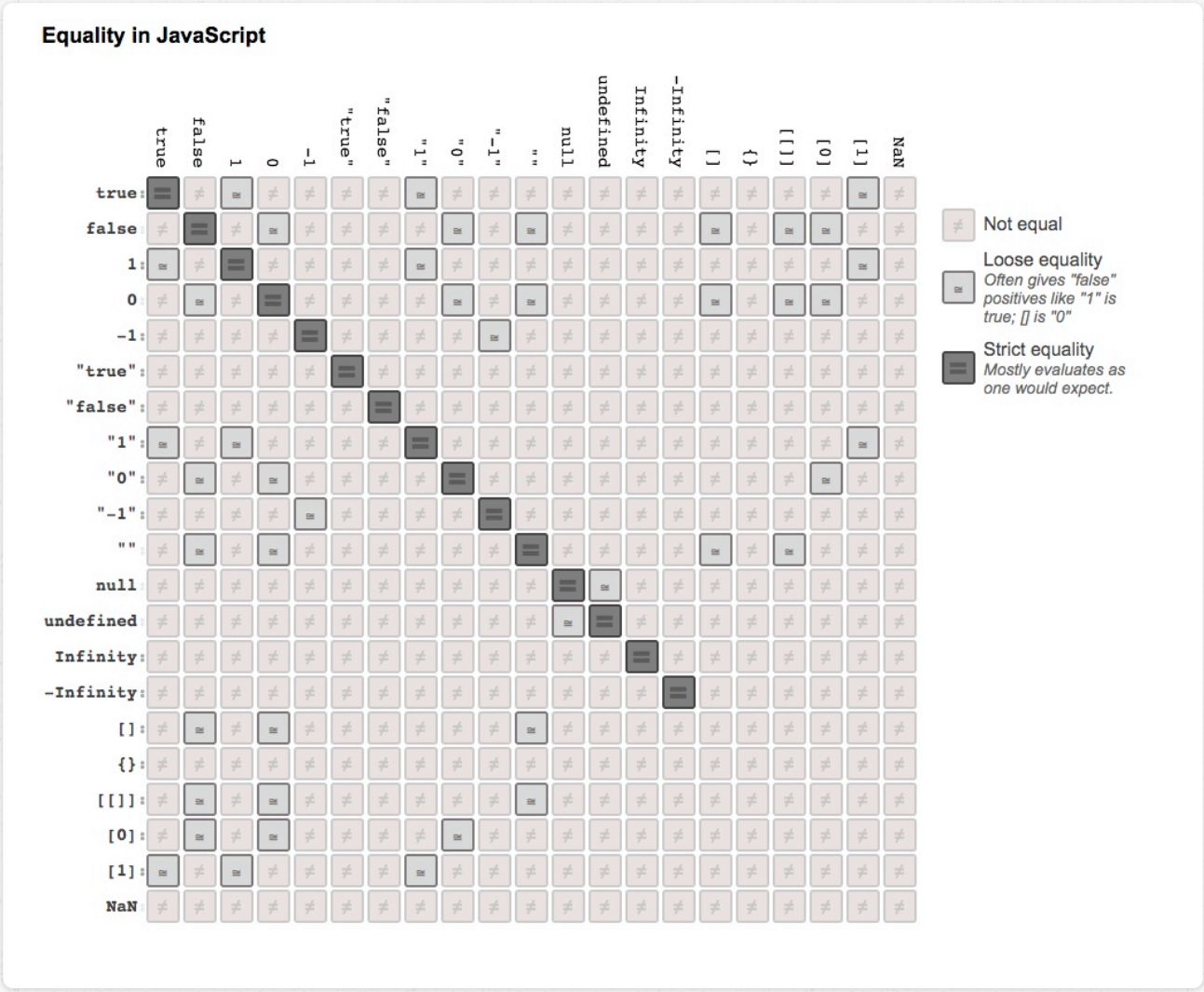
```
[] == ![]
3 == '3'

+0 === -0
1 / +0 === 1 / -0
```

同上，在公布结果之前，大家可以先给出自己的结果，然后验证一下

```
[] == ![]
true
3 == '3'
true
+0 === -0
true
1 / +0 === 1 / -0
false
```

是不是感受到了，JS的隐式转换的”魅力“所在，而且在社区，大家还总结了一张 **JavaScript-Equality-Table** [高清地址]



在ES6中，可以使用Object.is来解决这个问题

```
Object.is(NaN, NaN); // true
Object.is(+0, -0); // true
```

受Java的历史影响

> *The diktat from upper engineering management was that the language must “look like Java”*

Date的问题

因为JavaScript的Date对象是基于Java1.0而参考实现的，随着Java的版本的迭代，一些“有问题”的方法逐渐被抛弃，但是JS却还保留着...

```
const d = new Date('2016-09-10');
d.getDate();
d.getYear();
d.getMonth();
```

大家粗略一眼看去，可能认为的结果如下:

```
10
2016
8
```

运行代码结果如下:

```
const d = new Date('2016-09-10');
console.log(d.getDate())
console.log(d.getYear());
console.log(d.getMonth());
```

```
10
116
8
```

第二个`getDate()`返回的值不是2016，而是116（通过`2016-1900`）得出，具体为什么要减去1900，大家可以查阅相关资料。

Auto-Semicolon-Insertion (ASI) 自动分号插入

很多的语言都支持不写分号，比如：Swift Python Golang等，但是在JS中你不写分号，有时候会产生一些问题

比如 在`函数return`的时候

```
// return undefined
return
{
    status: true
}

// return {status: true}
return {
    status: true
}
```

比如 在`另起一行写一个IIFE的时候`

```
// Uncaught TypeError: (intermediate value)(...) is not a function
var a = function(x) { console.log(x) }
(function() {
    console.log('hello')
})()

// hello
var a = function(x) { console.log(x) };
(function() {
    console.log('hello')
})()
```

或者是这样的时候:

```
// [4, 9]
var a = [1, [2, 3]]
[3, 2, 1].map(function(num) {
    return num * num;
})

// [9, 4, 1]
var a = [1, [2, 3]];
[3, 2, 1].map(function(num) {
    return num * num;
})
```

有兴趣的同学 点这里了解更多

解决办法

在`([+-/`开始的一行之间加一个`;` 举个例子

```
var a = function(x) { console.log(x) }
;(function() { // do something })()
```

作用域

函数作用域

```
for (var i = 0; i !== 10; ++i) {
    // logs 10 ten times
    setTimeout(function() { console.log(i) }, 0)
}
```

解决方案

使用块作用域(let/const)

```
for (let i = 0; i !== 10; ++i) {
    // logs 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    setTimeout(function() { console.log(i) }, 0)
}
```

变量提升

```
bla = 2;
var bla;
```

```
var bla;
bla = 2;
```

熟悉JS特性的同学肯定知道上面两段代码是等价的

解决方案

Temporay Dear Zone （暂死区）

```
// without TDZ
console.log(a) // undefined
```

```
var a = 1

// with TDZ
console.log(b) // ReferenceError
let b = 2
```

对于TDZ不了解的同学可以参考这篇文章

“关键字”的误导

const

看起来，const关键字是表示来定义一个常量的，实际上不是这样

```
const MY_OBJECT = {"key": "value"};
//  重写会导致抛出异常TypeError
MY_OBJECT = {"OTHER_KEY": "value"};
// 但是，对象的key是可以被修改的
MY_OBJECT.key = "otherValue"; // 可以使用 Object.freeze() 去使得对象不可变
```

Function.prototype

```
[[Prototype]] !== prototype
```

API设计的问题

NaN

```
isNaN(123) // false
isNaN(NaN) // true
isNaN('a string') // true
```

数组初始化

```
Array(1, 2, 3); // [1, 2, 3]
```

```
Array(2, 3); // [2, 3]
Array(3); // [, , ,] ? ? ?
```

Array-like Objects

```
typeof arguments.length // number

Object.prototype.toString.call(arguments) // [object Arguments]

arguments.slice(0, 1) // TypeError: arguments.slice is not a function

var args = Array.prototype.slice.apply(arguments)

Object.prototype.toString.call(arguments) // [object Array]

args.slice(0, 1) // no error
```

ES6解决方案

```
[...arguments]
```

eval

```
function test() {
    var x = 2, y = 4;
    // 直接调用 使用函数内作用域的x, y
    console.log(eval("x + y"))
    var geval = eval;
    // 不是直接调用，使用全局的scope 报错 ReferenceError `x` is undefined
    console.log(geval("x + y"))
}
```

具体的原因 有兴趣的同学可以查看eval规范

大概意思就是如果直接调用就绑定当前上下文执行，否则就是全局上下文执行

结束语

希望大家以后在开发的过程中，能够对上面的一些case能够有一定的印象，能够在开

发过充中避免“踩坑”，如果还有其他你觉得也属于JS的设计缺陷的，欢迎一起讨论，有错误的欢迎指出，谢谢！

> *written by* *wangning.frontend@bytedance.com*