JS 基础知识点及常考面试题（一）

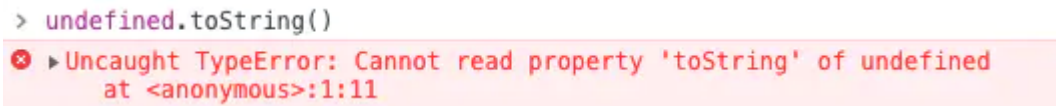
原始（Primitive）类型

> 涉及面试题：原始类型有哪几种？null 是对象嘛？

在 JS 中，存在着 6 种原始值，分别是：

- `boolean`
- `null`
- `undefined`
- `number`
- `string`
- `symbol`

首先原始类型存储的都是值，是没有函数可以调用的，比如 `undefined.toString()`



此时你肯定会有疑问，这不对呀，明明 `'1'.toString()` 是可以使用的。其实在这种情况下，`'1'` 已经不是原始类型了，而是被强制转换成了 `String` 类型也就是对象类型，所以可以调用 `toString` 函数。

除了会在必要的情况下强转类型以外，原始类型还有一些坑。

其中 JS 的 `number` 类型是浮点类型的，在使用中会遇到某些 Bug，比如 `0.1 + 0.2 !== 0.3`，但是这一块的内容会在进阶部分讲到。`string` 类型是不可变的，无论你在 `string` 类型上调用何种方法，都不会对值有改变。

另外对于 `null` 来说，很多人会认为他是个对象类型，其实这是错误的。虽然 `typeof null` 会输出 `object`，但是这只是 JS 存在的一个悠久 Bug。在 JS 的最初版本中使用的是 32 位系统，为了性能考虑使用低位存储变量的类型信息，`000` 开头代表是对象，然而 `null` 表示为全零，所以将它错误的判断为 `object` 。虽然现在的内部类型判断代码已经改变了，但是对于这个 Bug 却是一直流传下来。

对象（Object）类型

> 涉及面试题：对象类型和原始类型的不同之处？函数参数是对象会发生什么问题？

在 JS 中，除了原始类型那么其他的都是对象类型了。对象类型和原始类型不同的是，原始类型存储的是值，对象类型存储的是地址（指针）。当你创建了一个对象类型的时候，计算机会在内存中帮我们开辟一个空间来存放值，但是我们需要找到这个空间，这个空间会拥有一个地址（指针）。

```
const a = []
```

对于常量 `a` 来说，假设内存地址（指针）为 `#001`，那么在地址 `#001` 的位置存放了值 `[]`，常量 `a` 存放了地址（指针） `#001`，再看以下代码

```
const a = []
const b = a
b.push(1)
```

当我们将变量赋值给另外一个变量时，复制的是原本变量的地址（指针），也就是说当前变量 `b` 存放的地址（指针）也是 `#001`，当我们进行数据修改的时候，就会修改存放在地址（指针） `#001` 上的值，也就导致了两个变量的值都发生了改变。

接下来我们来看函数参数是对象的情况

```
function test(person) {
  person.age = 26
  person = {
    name: 'yyy',
    age: 30
  }

  return person
}
const p1 = {
  name: 'yck',
  age: 25
}
const p2 = test(p1)
console.log(p1) // -> ?
console.log(p2) // -> ?
```

对于以上代码，你是否能正确的写出结果呢？接下来让我为你解析一番：

- 首先，函数传参是传递对象指针的副本
- 到函数内部修改参数的属性这步，我相信大家都知道，当前 `p1` 的值也被修改了

- 但是当我们重新为 `person` 分配了一个对象时就出现了分歧，请看下图

所以最后 `person` 拥有了一个新的地址（指针），也就和 `p1` 没有任何关系了，导致了最终两个变量的值是不相同的。

typeof vs instanceof

> 涉及面试题：typeof 是否能正确判断类型？instanceof 能正确判断对象的原理是什么？

`typeof` 对于原始类型来说，除了 `null` 都可以显示正确的类型

```
typeof 1 // 'number'
typeof '1' // 'string'
typeof undefined // 'undefined'
typeof true // 'boolean'
typeof Symbol() // 'symbol'
```

`typeof` 对于对象来说，除了函数都会显示 `object`，所以说 `typeof` 并不能准确判断变量到底是什么类型

```
typeof [] // 'object'
typeof {} // 'object'
typeof console.log // 'function'
```

如果我们想判断一个对象的正确类型，这时候可以考虑使用 `instanceof`，因为内部机制是通过原型链来判断的，在后面的章节中我们也会自己去实现一个 `instanceof`。

```
const Person = function() {}
const p1 = new Person()
p1 instanceof Person // true

var str = 'hello world'
str instanceof String // false

var str1 = new String('hello world')
str1 instanceof String // true
```

对于原始类型来说，你想直接通过 `instanceof` 来判断类型是不行的，当然我们还是有办法让 `instanceof` 判断原始类型的

```
class PrimitiveString {
  static [Symbol.hasInstance](x) {
    return typeof x === 'string'
  }
}
console.log('hello world' instanceof PrimitiveString) // true
```

你可能不知道 `Symbol.hasInstance` 是什么东西，其实就是一个能让我们自定义 `instanceof` 行为的东西，以上代码等同于 `typeof 'hello world' === 'string'`，所以结果自然是 `true` 了。这其实也侧面反映了一个问题， `instanceof` 也不是百分之百可信的。

类型转换

> 涉及面试题：该知识点常在笔试题中见到，熟悉了转换规则就不惧怕此类题目了。

首先我们要知道，在 JS 中类型转换只有三种情况，分别是：

- 转换为布尔值
- 转换为数字
- 转换为字符串

我们先来看一个类型转换表格，然后再进入正题

> 注意图中有一个错误，Boolean 转字符串这行结果我指的是 true 转字符串的例子，不是说 Boolean、函数、Symblo 转字符串都是 `true`

原始值	转换目标	结果
number	布尔值	除了 0、-0、NaN 都为 true
string	布尔值	除了空串都为 true
undefined、null	布尔值	FALSE
引用类型	布尔值	TRUE
number	字符串	5 => '5'
Boolean、函数、Symbol	字符串	'true'
数组	字符串	[1,2] => '1,2'
对象	字符串	'[object Object]'
string	数字	'1' => 1，'a' => NaN
数组	数字	空数组为0，存在一个元素且为数字转数字，其他情况NaN
null	数字	0
除了数组的引用类型	数字	NaN
Symbol	数字	抛错

转Boolean

在条件判断时，除了 `undefined`， `null`， `false`， `NaN`， `''`， `0`， `-0`，其他所有值都转为 `true`，包括所有对象。

对象转原始类型

对象在转换类型的时候，会调用内置的 `[[ToPrimitive]]` 函数，对于该函数来说，算法逻辑一般来说如下：

- 如果已经是原始类型了，那就不需要转换了
- 如果需要转字符串类型就调用 `x.toString()`，转换为基础类型的话就返回转换的值。不是字符串类型的话就先调用 `valueOf`，结果不是基础类型的话再调用 `toString`
- 调用 `x.valueOf()`，如果转换为基础类型，就返回转换的值
- 如果都没有返回原始类型，就会报错

当然你也可以重写 `Symbol.toPrimitive`，该方法在转原始类型时调用优先级最高。

```
let a = {
  valueOf() {
    return 0
  },
  toString() {
```

```
    return '1'
  },
  [Symbol.toPrimitive]() {
    return 2
  }
}
1 + a // => 3
```

四则运算符

加法运算符不同于其他几个运算符，它有以下几个特点：

- 运算中其中一方为字符串，那么就会把另一方也转换为字符串
- 如果一方不是字符串或者数字，那么会将它转换为数字或者字符串

```
1 + '1' // '11'
true + true // 2
4 + [1,2,3] // "41,2,3"
```

如果你对于答案有疑问的话，请看解析：

- 对于第一行代码来说，触发特点一，所以将数字 `1` 转换为字符串，得到结果 `'11'`
- 对于第二行代码来说，触发特点二，所以将 `true` 转为数字 `1`
- 对于第三行代码来说，触发特点二，所以将数组通过 `toString` 转为字符串 `1,2,3`，得到结果 `41,2,3`

另外对于加法还需要注意这个表达式 `'a' + + 'b'`

```
'a' + + 'b' // -> "aNaN"
```

因为 `+ 'b'` 等于 `NaN`，所以结果为 `"aNaN"`，你可能也会在一些代码中看到过 `+ '1'` 的形式来快速获取 `number` 类型。

那么对于除了加法的运算符来说，只要其中一方是数字，那么另一方就会被转为数字

```
4 * '3' // 12
4 * [] // 0
4 * [1, 2] // NaN
```

比较运算符

1. 如果是对象，就通过 `toPrimitive` 转换对象
2. 如果是字符串，就通过 `unicode` 字符索引来比较

```
let a = {
  valueOf() {
    return 0
  },
  toString() {
    return '1'
  }
}
a > -1 // true
```

在以上代码中，因为 `a` 是对象，所以会通过 `valueOf` 转换为原始类型再比较值。

this

> 涉及面试题：如何正确判断 this？箭头函数的 this 是什么？

`this` 是很多人会混淆的概念，但是其实它一点都不难，只是网上很多文章把简单的东西说复杂了。在这一小节中，你一定会彻底明白 `this` 这个概念的。

我们先来看几个函数调用的场景

```
function foo() {
  console.log(this.a)
}
var a = 1
foo()

const obj = {
  a: 2,
  foo: foo
}
obj.foo()

const c = new foo()
```

接下来我们一个个分析上面几个场景

- 对于直接调用 `foo` 来说，不管 `foo` 函数被放在了什么地方，`this` 一定是 `window`
- 对于 `obj.foo()` 来说，我们只需要记住，谁调用了函数，谁就是 `this`，所以在这个场景下 `foo` 函数中的 `this` 就是 `obj` 对象
- 对于 `new` 的方式来说，`this` 被永远绑定在了 `c` 上面，不会被任何方式改变 `this`

说完了以上几种情况，其实很多代码中的 `this` 应该就没什么问题了，下面让我们看看箭头函数中的 `this`

```
function a() {
  return () => {
    return () => {
      console.log(this)
    }
  }
}
console.log(a()()())
```

首先箭头函数其实是没有 `this` 的，箭头函数中的 `this` 只取决包裹箭头函数的第一个普通函数的 `this` 。在这个例子中，因为包裹箭头函数的第一个普通函数是 `a` ，所以此时的 `this` 是 `window` 。另外对箭头函数使用 `bind` 这类函数是无效的。

最后种情况也就是 `bind` 这些改变上下文的 API 了，对于这些函数来说， `this` 取决于第一个参数，如果第一个参数为空，那么就是 `window` 。

那么说到 `bind` ，不知道大家是否考虑过，如果对一个函数进行多次 `bind` ，那么上下文会是什么呢?

```
let a = {}
let fn = function () { console.log(this) }
fn.bind().bind(a)() // => ?
```

如果你认为输出结果是 `a` ，那么你就错了，其实我们可以把上述代码转换成另一种形式

```
// fn.bind().bind(a) 等于
let fn2 = function fn1() {
  return function() {
    return fn.apply()
  }.apply(a)
}
fn2()
```

可以从上述代码中发现，不管我们给函数 `bind` 几次， `fn` 中的 `this` 永远由第一次 `bind` 决定，所以结果永远是 `window` 。

```
let a = { name: 'yck' }
function foo() {
  console.log(this.name)
}
foo.bind(a)() // => 'yck'
```

以上就是 `this` 的规则了，但是可能会发生多个规则同时出现的情况，这时候不同的规则之间会根据优先级最高的来决定 `this` 最终指向哪里。

首先，`new` 的方式优先级最高，接下来是 `bind` 这些函数，然后是 `obj.foo()` 这种调用方式，最后是 `foo` 这种调用方式，同时，箭头函数的 `this` 一旦被绑定，就不会再被任何方式所改变。

如果你还是觉得有点绕，那么就看以下的这张流程图吧，图中的流程只针对于单个规则。