

## 19 | a + b: 动态类型是灾难之源还是最好的特性? (下)

2019-12-27 周爱民

《JavaScript核心原理解析》

[课程介绍 >](#)



讲述：周爱民

时长 20:04 大小 18.38M



你好，我是周爱民。

上一讲，我们说到如何将复杂的类型转换缩减到两条简单的规则，以及两种主要类型。这两条简单规则是：

1. 从值  $x$  到引用：调用 `Object(x)` 函数。
2. 从引用  $x$  到值：调用 `x.valueOf()` 方法；或，调用四种值类型的包装类函数，例如 `Number(x)`，或者 `String(x)` 等等。


两种主要类型则是**字符串**和**数字值**。



当类型转换系统被缩减成这样之后，有些问题就变得好解释了，但也确实有些问题变得更加难解。例如 @graybernhardt 在讲演中提出的灵魂发问，就是：

- 如果将数组跟对象相加，会发生什么？

如果你忘了，那么我们就一起来回顾一下这四个直击你灵魂深处的示例，简单地说，这些示例就是“数组与对象”相加的四种情况，结果都完全不同。

 复制代码

```
1 > [] + {}
2 '[object Object]'
3
4 > {} + []
5 0
6
7 > {} + {}
8 NaN
9
10 > [] + []
11 ''
```

而这个问题，也就是这两讲的标题中“a + b”这个表达式的由来。也就是说，如何准确地解释“两个操作数相加”，与如何全面理解 JavaScript 的类型系统的转换规则，关系匪浅！

## 集中精力办大事

一般来说，运算符很容易知道操作数的类型，例如“a - b”中的减号，我们一看就知道意图，是两个数值求差，所以 a 和 b 都应该是数值；又例如“obj.x”中的点号，我们一看也知道，是取对象 obj 的属性名字字符串 x。

当需要引擎“推断目的”时，JavaScript 设定推断结果必然是三种基础值（boolean、number 和 string）。由于其中的 boolean 是通过查表来进行的，所以就只剩下了 number 和 string 类型需要“自动地、隐式地转换”。

但是在 JavaScript 中，“加号 (+)”是一个非常特别的运算符。像上面那样简单的判断，在加号 (+) 上面就不行，因为它在 JavaScript 中既可能是字符串连结，也可能是数值求和。另外还有一个与此相关的情况，就是 object[x] 中的 x，其实也很难明确地说它是字符串还是数值。因为计算属性（computed property）的名字并不能确定是字符串还是数值；尤其是现在，它还可能是符号类型（symbol）。



NOTE：在讨论计算属性名（computed property name）时，JavaScript 将它作为预期为字符串的一个值来处理，即 `r = ToPrimitive(x, String)`。但是这个转换的结果仍然可能是 5 种值类型之一，因此在得到最终属性名的时候，JavaScript 还会再调用一次 `ToString(r)`。

由于“加号 (+)”不能通过代码字面来判断意图，因此只能在运算过程中实时地检查操作数的类型。并且，这些类型检查都必须是基于“加号 (+) 运算必然操作两个值数据”这个假设来进行。于是，JavaScript 会先调用 `ToPrimitive()` 内部操作来分别得到“a 和 b 两个操作数”可能的原始值类型。

所以，问题就又回到了在上面讲的 `Value vs. Primitive values` 这个东西上面。对象到底会转换成什么？这个转换过程是如何决定的呢？

这个过程包括如下的四个步骤。

## 步骤一

首先，JavaScript 约定：如果 `x` 本身就是原始值，那么 `ToPrimitive(x)` 这个操作直接就返回 `x` 本身。这个很好理解，因为它不需要转换。也就是说（如下代码是不能直接执行的）：


```
1 # 1. 如果x是非对象，则返回x
2 > _ToPrimitive(5)
3 5
```

 复制代码

## 步骤二

接下来的约定是：如果 `x` 是一个对象，且它有对应的五种 `PrimitiveValue` 内部槽之一，那么就直接返回这个内部槽中的原始值。由于这些对象的 `valueOf()` 就可以达成这个目的，因此这种情况下也就是直接调用该方法（步骤三）。相当于如下代码：

```
1 # 2. 如果x是对象，则尝试得到由x.valueOf()返回的原始值
2 > Object(5).valueOf()
3 5
```

 复制代码



但是在处理这个约定的时候，JavaScript 有一项特别的设定，就是对“引擎推断目的”这一行为做一个预设。如果某个运算没有预设目的，而 JavaScript 也不能推断目的，那么 JavaScript 就会强制将这个预设为“number”，并进入“传统的”类型转换逻辑（步骤四）。

所以，简单地说（这是一个非常重要的结论）：

如果一个运算无法确定类型，那么在类型转换前，它的运算数将被预设为 number。

NOTE1：预设类型在 ECMAScript 称为 PreferredType，它可以为 undefined 或"default"。但是“default”值是“传统的”类型转换逻辑所不能处理的，这种情况下，JavaScript 会先将它重置为“number”。也就是说，在传统的转换模式中，“number”是优先的。

NOTE2：事实上，只有对象的符号属性 Symbol.toPrimitive 所设置的函数才会被要求处理“default”这个预设。这也是在 Proxy/Reflect 中并没有与类型转换相关的陷阱或方法的原因。

于是，这里会发生两种情况，也就是接下来的步骤三和步骤四。

### 步骤三：作为原始值处理

如果是上述的五种包装类的对象实例（它们有五种PrimitiveValue内部槽之一），那么它们的valueOf()方法总是会忽略掉“number”这样的预设，并返回它们内部确定（即内部槽中所保留的）的原始值。

所以，如果我们为符号创建一个它的包装类对象实例，那么也可以在这种情况下解出它的值。例如：

```
1 > x = Symbol()  
2  
3 > obj = Object(x)  
4  
5 > obj.valueOf() === x  
6 true
```

 复制代码



正是因为对象（如果它是原始值的包装类）中的原始值总是被解出来，所以，你要将数字值5转换成两个对象类型，并且再将这两个对象相加，那么其结果也会是数值 10。

 复制代码

```
1 > Object(5) + Object(5)
2 10
```

这个代码看起来是两个对象“相加”，但是却等效于它们的原始值直接相加。

但是如果考虑“对象属性存取”这样的例子情况就发生了变化，由于“对象属性存取”是一个“有预期”的运算——它的预期是“字符串”，因此会有第二种情况——步骤四。

## 步骤四：进入“传统的类型转换逻辑”

这需要利用到对象的`valueOf()`和`toString()`方法：当预期是“number”时，`valueOf()`方法优先调用；否则就以`toString()`为优先。并且，重要的是，上面的预期只决定了上述的优先级，而当调用优先方法仍然得不到非对象值时，还会顺序调用另一方法。

这带来了一个结果，即：如果用户代码试图得到“number”类型，但`x.valueOf()`返回的是一个对象，那么就还会调用`x.toString()`，并最终得到一个字符串。

到这里，就可以解释前面四种对象与数组相加所带来的特殊效果了。

## 解题 1：从对象到原始值

在`a + b`的表达式中，`a`和`b`是对象类型时，由于“加号 (+)”运算符并不能判别两个操作数的预期类型，因此它们被“优先地”假设为数字值（number）进行类型转换。这样一来，无论是对象，还是数组，它们的`valueOf()`方法调用的结果都将得到它们本身。如果用`typeof()`看一下，结果还仍然是`object`类型。接下来，由于这个调用`valueOf()`方法的结果不是值类型，所以就会再尝试一下调用`toString()`这个方法。

 复制代码




```
1 # 在预期是 'number' 时，先调用 `valueOf()` 方法，但得到的结果仍然是对象类型；
2 > [typeof ([].valueOf()), typeof ({}.valueOf())]
3 [ 'object', 'object' ]
4
5 # 由于上述的结果是对象类型（而非值），于是再尝试 `toString()` 方法来得到字符串
```

```
6 > [].toString(), {}.toString()
7 [ '', '[object Object]' ]
```

在这里，我们就会看到有一点点差异了。空数组转换出来，是一个空字符串，而对象的转换成字符串时是'[object Object]'。

所以接下来的四种运算变成了下面这个样子，它们其实是对字符串相加，也就是字符串连接的结果。

 复制代码

```
1 # [] + {}
2 > '' + '[object Object]'
3 '[object Object]'
4
5 # {} + []
6 > ???
7 0
8
9 # {} + {}
10 > ???
11 NaN
12
13 # [] + []
14 > '' + ''
15 ''
```

好的，你应该已经注意到了，在第二和第三种转换的时候我打了三个问号“???”。因为如果按照上面的转换过程，它们无非是字符串拼接，但结果它们却是两个数字值，分别是 0，还有 NaN。

怎么会这样？！！

## 解题 2：“加号 (+)”运算的戏分很多

现在看看这两个表达式。

 复制代码

```
1 {} + []
2 {} + {}
```



你有没有一点熟悉感？嗯，很不幸，它们的左侧是一对大括号，而当它们作为语句执行的时候，会被优先解析成——块语句！并且大括号作为结尾的时候，是可以省略掉语句结束符“分号 (;)”的。


所以，你碰到了 JavaScript 语言设计历史中最大的一块铁板！就是所谓“自动分号插入 (ASI)”。这个东西的细节我这里就不讲了，但它的结果是什么呢？上面的代码变成下面这个样子：

- `{}; +[]`
- `{}; +{}`

实在是不幸啊！这样的代码仍然是可以通过语法解析，并且仍然是可以进行表达式计算求值的！

于是后续的结论就比较显而易见了。

由于“+”号同时也是“正值运算符”，并且它很明显可以准确地预期后续操作数是一个数值，所以它并不需要调用 `ToPrimitive()` 内部操作来得到原始值，而是直接使用“`ToNumber(x)`”来尝试将 `x` 转换为数字值。而上面也讲到，“将对象转换为数字值，等效于使用它的包装类来转换，也就是 `Number(x)`”。所以，上述两种运算的结果就变成了下面的样子：

 复制代码

```
1 # +[] 将等义于
2 > + Number([])
3 0
4
5 # +{} 将等义于
6 > + Number({})
7 NaN
```

### 解题 3：预期 vs. 非预期

但是你可能会注意到：当使用“`... + {}`”时，`ToPrimitive()` 转换出来的，是字符串“`[object Object]`”；而在使用“`+ {}`”时，`ToNumber(x)` 转换出来的却是值 `NaN`。所以，在不同的预期下面，“对象 -> 值”转换的结果却并不相同。



这之间有什么规律吗？

我们得先理解哪些情况下，JavaScript 是不能确定用户代码的预期的。总结起来，这其实很有限，包括：

1. “加号 (+)”运算中，不能确定左、右操作数的类型；
2. “等值 (==)”运算中，不能确定左、右操作数的类型；（JavaScript 认为，如果左、右操作数之一为 string、number、bigint 和 symbol 四种基础类型之一，而另一个操作数是对象类型 (x)，那么就需要将对象类型“转换成基础类型 (ToPrimitive(x))”来进行比较。操作数将尽量转换为数字来进行比较，即最终结果将等效于：Number(x) == Number(y)。)
3. “new Date(x)”中，如果 x 是一个非 Date() 实例的对象，那么将尝试把 x 转换为基础类型 x1；如果 x1 是字符串，尝试从字符串中 parser 出日期值；否则尝试 x2 = Number(x1)，如果能得到有效的数字值，则用 x2 来创建日期对象。
4. 同样是在 Date() 的处理中，（相对于缺省时优先 number 类型来说，）JavaScript 内部调整了 Date 在转换为值类型时的预期。一个 Date 类型的对象 (x) 转换为值时，将优先将它视为字符串，也就是先调用 x.toString()，之后再调用 x.valueOf()。

其他情况下，JavaScript 不会为用户代码调整或假设预期值。这也就是说，按照 ECMAScript 内部的逻辑与处理过程，其他的运算（运算符或其他内置操作）对于“对象 x”，都是有目标类型明确的、流程确定的方法来转换为“（值类型的）值”的。

## 其他

### 显式的 vs. 隐式的转换

很大程度上来说，显式的转换其实只决定了“转换的预期”，而它内部的转换过程，仍然是需要“隐式转换过程”来参与的。例如，你调用 Number() 函数来转换对象x：

```
1 > x = new Object
2 > Number(x)
3 NaN
```

 复制代码



对于这样的一个显式转换，Number() 只决定它预期的目标是‘number’类型，并最终将调用 ToPrimitive(x, 'Number')来得到结果。然而，一如之前所说的，ToPrimitive() 会接受



任何一个“原始值”作为结果x1返回（并且要留意的是，在这里 null 值也是原始值），因此它并不保证结果符合预期 'number'。

所以，最终 Number() 还会再调用一次转换过程，尝试将x1转换为数字。

## 字符串在“+”号中的优先权

另一方面，在“+”号运算中，由于可能的运算包括数据和字符串，所以按照隐式转换规则，在不确定的情况下，优先将运算数作为数字处理。那么就是默认“+”号是做求和运算的。

但是，在实际使用中，结果往往会是字符串值。

这是因为字符串在“+”号运算中还有另一层面的优先级，这是由“+”号运算符自己决定的，因而并不是类型转换中的普遍规则。

“+”号运算符约定，对于它的两个操作数，在通过ToPrimitive()得到两个相应的原始值之后，二者之任一为字符串的话，就优先进行字符串连接操作。也就是说，这种情况下另一个操作数会发生一次“值 -> 值”的转换，并最终连接两个字符串以作为结果返回。

那么，我们怎么理解这个行为呢？比如说，如果对象 x 转换成数字和字符串的效果如下：

 复制代码

```
1 x = {  
2   valueOf() { console.log('Call valueOf'); return Symbol() },  
3   toString() { console.log('Call toString'); return 'abc' }  
4 }
```

我声明了一个对象 x，它带有两个定制的 toString() 和 valueOf() 方法，用来观察类型转换的过程；并且，其中 valueOf() 会返回一个 symbol 符号，也就是说，它是“值类型”，但既不是字符串，也不是数字值。

接下来我们尝试用它跟一个任意值做“+”号运算，例如：



 复制代码

```
1 # 例1：与非字符串做“+”运算时  
2 > true + x  
3 Call valueOf
```

```
4 TypeError: Cannot convert a Symbol value to a number
```

“+”号运算在处理这种情况（用对象与非字符串值做加号运算）时，会先调用x的 valueOf() 方法，然后由于“+”号的两个操作数都不是字符串，所以将再次尝试将它们转换成数字并求和。又例如：

 复制代码

```
1 # 例2：与字符串做“+”运算时
2 > 'OK, ' + x
3 Call valueOf
4 TypeError: Cannot convert a Symbol value to a string
```

这种情况下，由于存在一个字符串操作数，因此“字符串连接”运算被优先，于是会尝试将x转换为字符串。

然而需要注意的是，上述两个操作中都没有调用 x.toString()，而“都仅仅是”在 ToPrimitive() 内部操作中调用了 x.valueOf()。也就是说，在检测操作数的值类型“是否是字符串”之后，再次进行的“值 -> 值”的转换操作是基于 ToPrimitive() 的结果，而非原对象x的。

这也是之前在“解题 3”中特别讲述 Date() 对象这一特例的原因。因为 Date() 在“调用 ToPrimitive()”这个阶段的处理顺序是反的，所以它会先调用 x.toString，从而产生不一样的效果。例如：

 复制代码

```
1 // 创建MyDate类，覆盖valueOf()和toString()方法
2 class MyDate extends Date {
3   valueOf() { console.log('Call valueOf'); return Symbol() }
4   toString() { console.log('Call toString'); return 'abc' }
5 }
```

测试如下：

 复制代码

```
1 # 示例
2 > x = new MyDate;
3
4 # 与非字符串做“+”运算时
```



```
5 > true + x
6 Call toString
7 trueabc
8
9 # 与非字符串做“+”运算时
10 > 'OK, ' + x
11 Call toString
12 OK, abc
```

那么对于 `Date()` 这个类来说，这又是如何做到的呢？

## Symbol.toPrimitive 的处理

简单地说，`Date` 类重写了原型对象 `Date.prototype` 上的符号属性 `Symbol.toPrimitive`。任何情况下，如果用户代码重写了对象的 `Symbol.toPrimitive` 符号属性，那么 `ToPrimitive()` 这个转换过程就将由用户代码负责，而原有的顺序与规则就失效了。

我们知道，由于调用 `ToPrimitive(hint)` 时的入口参数 `hint` 可能为 `default/string/number` 这三种值之一，而它要求返回的只是“值类型”结果，也就是说，结果可以是所有 5 种值类型之任一。因此，用户代码对 `ToPrimitive(hint)` 的重写可以“参考”这个 `hint` 值，也可以无视之，也可以在许可范围内返回任何一种值。

简单地说，它就是一个超强版的 `valueOf()`。

事实上，一旦用户代码声明了符号属性 `Symbol.toPrimitive`，那么 `valueOf()` 就失效了，ECMAScript 采用这个方式“一举”摧毁了原有的隐式转换的全部逻辑。这样一来，包括预期的顺序与重置，以及 `toString` 和 `valueOf` 调用等等都“不复存焉”。

一切重归于零：定制 `Symbol.toPrimitive`，返回值类型；否则抛出异常。

NOTE： `Date()` 类中仍然是会调用 `toString` 或 `valueOf` 的，这是因为在它的 `Symbol.toPrimitive` 实现中仅是调整了两个方法的调用顺序，而之后仍然是调用原始的、内置的 `ToPrimitive()` 方法的。对于用户代码来说，可以自行决定该符号属性（方法）的调用结果，无需依赖 `ToPrimitive()` 方法。



## 结语与思考

今天我们更深入地讲述了类型转换的诸多细节，除了这一讲的简单题解之外，对于“+”号运算也做了一些补充。

总地来讲，我们是在讨论 JavaScript 语言所谓“动态类型”的部分，但是动态类型并不仅限于此。也就是说 JavaScript 中并不仅仅是“类型转换”表现出来动态类型的特性。例如一个更简单的问题：

“`x === x`”在哪些情况下不为 `true`？


这原本是这两讲的另一个备选的标题，它也是讨论动态类型问题的。只不过这个问题所涉及的范围太窄，并不适合展开到这两讲所涵盖的内容，因此被弃用了。这里把它作为一个小小的思考题留给你，你可以试着找找答案。

NOTE1：我可以告诉你答案不只一个，例如“`x` 是 NaN”。^^.

NOTE2：“`x` 是 NaN”这样的答案与动态类型或动态语言这个体系没什么关系，所以它不是我在这里想与你讨论的主要话题。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

分享给需要的人，Ta购买本课程，你将得 20 元

 生成海报并分享

 赞 4  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | `a + b`：动态类型是灾难之源还是最好的特性？（上）

下一篇 20 | `(0, eval)("x = 100")`：一行让严格模式形同虚设的破坏性设计（上）



# JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费 



## 精选留言 (15)

 写留言



weineel 

2019-12-27

强行找到一种方法, 但和本节所讲没啥关系:

```
Object.defineProperty(global, 'x', {  
  get: function() {  
    return Math.random();  
  }  
})
```

```
x === x // false
```

作者回复: 赞!

的确, 这是除NaN之外我认为最可行的一个答案。事实上, 这也是我在课程中提升“动态语言特性”这个方向的原因: 一部分动态特性是基于OOP来实现的, 这正是JavaScript的混合语言特性的应用。

不过这个例子其实可以变成更简单。例如:

...

```
Object.defineProperty(global, 'x', { get: Symbol })
```



```
// 或
Object.defineProperty(global, 'x', { get: Math.random })
...
```

AND, @晓小东 给出的Symbol()方案对这个getter方法是一个很好的补充, 很好地利用了“symbol总是唯一”的特性。

共 2 条评论 >

👍 10



陈强

2020-04-08

这里是一个拓展阅读: <https://2ality.com/2012/01/object-plus-object.html>

为什么{} + {} 在浏览器中打印的是 "[object Object][object Object]"

因为“the Node.js REPL parses its input differently from either Firefox or Chrome (which even uses the same V8 JavaScript engine as Node.js). The following input is parsed as expressions and the results are less surprising:”

实际上chrome自己加了console.log(), 相当于({} + {}), firefox的控制台会输出NaN。

// test in chrome

```
{ } + { } // "[object Object][object Object]"
```

```
{ } + { } === "[object Object][object Object]" // false
```

```
({ } + { }) === "[object Object][object Object]" // true
```

作者回复: 谢谢 🍊+3



👍 7



Geek\_185c7d

2020-01-19

请问老师, 为什么{} + {} 在浏览器中打印的是 "[object Object][object Object]"

作者回复: 浏览器把引擎包了一层, 你执行的不是真正的引擎环境。

在node中用如下命令行试试:

...

```
> node -p -e '{} + {}'
```

NaN

```
> node -p -e '{} + []'
```



0  
...

共 3 条评论 >

👍 4



**westfall**

2021-02-08

关于 chrome 控制台 `{}` + `{}` 打印出 `"[object Object][object Object]"`，找到一个解释，chrome 会将所有已 `'{'` 开头，已 `'}'` 结尾的输入强行用 `()` 包起来，因此输入 `{}` 得到的是一个对象字面量的求值 (`{}`)，而不是一个语句块，同样 `{}` + `{}` 得到的是 `({} + {})`。源码在此：<https://chromium.googlesource.com/chromium/src.git/+/4fd348fdb9c0b3842829acdfb2b82c86dadcd8e0a%5E!/#F2>



👍 3



**晓小东**

2019-12-28

难道是这个吗， 如果作为标识符 `var x` 确实没想出。

```
>> Symbol() === Symbol() // false
```

作者回复: 参见 @sprinty 的答案。呵呵，我自己也不知道有没有更多的可能了。



👍 2



**undefined**

2021-04-30

```
> node -v  
v12.20.1
```

```
> node -p '{} + {}'  
NaN
```

```
> node -p '({} + {})'  
[object Object][object Object]
```

```
> node -p "eval('{} + {}')"  
NaN
```

```
> node -p "eval('({} + {})')"  
[object Object][object Object]
```



作者回复: 这个? 问题是?

1、3是按语句执行，所以是NaN；2、4是按表达式执行，所以是[object Object]\*2。

正文中（解题2）说的都是第一种情况。而第二种情况的分析路径在正文中也是交待了的哦。



1



wheatup

2020-06-10

和sprinty的方法类似，只不过避免了污染全局作用域：

```
with ({ get x() { return Symbol() } }) {  
  console.log(x === x); // false  
}
```

这个能成功的原因是这里的 x 其实是有 [withObj].x 取值的一个操作，然后利用getter返回不同的值造成的。

----

真要说利用本节课学到的内容的话，可以把题目的  $x === x$  换成  $x >= x$ ，单纯的  $x === x$  是不会触发隐式类型转换的，而js在比较大小时一定会将对象转换成数值再进行比较，所以这里就会调用[Symbol.toPrimitive]，这时候就可以重写这个方法做些手脚：

```
let x = {  
  [Symbol.toPrimitive]() {  
    return Math.random();  
  }  
};
```

```
console.log(x >= x); // true和false都有可能
```

作者回复: +1



1



波哥

2022-01-07

老师您好，我在浏览器控制台，node环境控制台或者代码中：`{ } + { }`的执行结果都是"[object Object][object Object]"，但是教程中的结果是NaN，是不是还有其他理解方式？





作者回复: 评论中有解说过这个问题, 是node/控制台包过一次。可以用如下命令行试试:

```

```
> node -p -e '{} + {}'
```

```
NaN
```

```
> node -p -e '{} + []'
```

```
0
```

```



油菜

2020-11-19

老师, `Number([]) == Number([])` 是true, 但 `[] == []` 是false, 动态类型转换如何解释这个案例呢?

-----

```
var x = [];
```

```
x === x // true;
```

```
x === [] // false;
```

```
[] === x // false;
```

```
[] === [] //false;
```

作者回复: 对于`Number([])`来说, 过程如下:

====

```
Number([]) =>
```

```
Number(ToNumber(ToPrimitive([]))) =>
```

```
Number(ToNumber([].valueOf())) =>
```

```
Number(ToNumber("")) =>
```

```
Number(0) =>
```

```
0
```

====

注意这里的结果值是一个值类型, 而不是对象类型, 所以`typeof Number([])`的值将是"number"。所以, 第一个式子等义于`0 == 0`, 当然是true。

对于`[] == []`来说, 两个对象实例的比较总是false。





HoSalt

2020-05-25

「Date() 类中仍然是会调用 toString 或 valueOf 的，这是因为在它的Symbol.toPrimitive实现中仅是调整了两个方法的调用顺序，而之后仍然是调用原始的、内置的ToPrimitive()方法的」

调整顺序是值hint值为default的情况下？

作者回复: 是。如果hint不是default，那么就相当于指定顺序了。

<https://tc39.es/ecma262/#sec-toprimitive>



K4SHIFZ

2020-03-28

自动分号插入并不会将{}+{} 变成{};+{} 吧？因为没有行结束符。

作者回复: 会的。:)

共 2 条评论 >



晓小东

2019-12-27

老师我测很多代码得出一个总结：

参与 + 或 - 运算 + - 只认那五种植类型数据，

从包装对象实例（String， Number， Boolean, Symbol），和数组Object 对象调用valueOf 可以看出

只要valueOf 返回五种植类型数据， 就不会toString()方法， 反之如果还是对象类型，即使是包装对象实例，还是会调用toString方法

总结是： 在toPrimitive（）中要获取五种植类型数据包括undefined 和 null， 大部分情况下引擎都是按数据值类型进行预测： 如下：

{}; + []

String(1) + [];

1 - [];

都是valueOf -> toString 过程

最终在toPrimitive() 中 根据 + 或者 - 以及运算符两侧值类型 调用相应String 或者Number 进行转换 求值

所以最终的结果只有三种 字符串、数值、和 NaN



直接toString就是在模板字符串中(目前发现这一种除了Date)  
`\${x}`

测试代码如下

代码链接 <https://repl.it/@XiaoDHuang/MadEuphoricTabs>

```
let valueOf = Array.prototype.valueOf
let toString = Array.prototype.toString;
```

```
Array.prototype.valueOf = function() {
  console.log(this.flag + " valueOf")
  return valueOf.call(this);
}
```

```
Array.prototype.toString = function() {
  console.log(this.flag + ' toString');
  return toString.call(this);
}
```

```
var x = [];
```

```
x.flag = '{}; + []';
{}; + x;
```

```
x.flag = 'String(1) + []';
console.log(1 + x);
```

```
x.flag = '1 - []';
console.log(1 - x);
```

```
x.flag = `${x}`
console.log(`${x}`);
```

作者回复: 是的呀。

> 总结是: 在toPrimitive () 中要获取五种植类型数据包括undefined 和 null, ...

-----

在上一小节里不是讲过了么? 原文是:

> > 一种关于“原始值”的简单解释是: 所有 5 种能放入私有槽 (亦即是说它们有相应的包装类) 的值



(Values)，都是原始值；并且，再加上两个特殊值 undefined 和 null，那么就是所谓原始值 (Primitive values) 的完整集合了。

> 只要valueOf 返回五种植类型数据，就不会toString()方法，反之如果还是对象类型，即使是包装对象实例，还是会调用toString方法...

----

这在这一讲的“步骤4”中也讲到了。原文是：

> > 这需要利用到对象的valueOf()和toString()方法：当预期是“number”时，valueOf()方法优先调用；否则就以toString()为优先。并且，重要的是，上面的预期只决定了上述的优先级，而当调用优先方法仍然得不到非对象值时，还会顺序调用另一方法。

最后，关于Date()类型中顺序相反的问题，本讲里也是解释了的哟哟哟哟~ ^^.



**Astrogladiator-埃蒂...**

2019-12-27

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality\\_comparisons\\_and\\_sameness](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness)

看了下mdn，还真是只有NaN这么一种情况。

If Type(x) is different from Type(y), return false.

If Type(x) is Number or BigInt, then

Return ! Type(x)::equal(x, y).

Return ! SameValueNonNumeric(x, y).

Assert: Type(x) is not Number or BigInt.

Assert: Type(x) is the same as Type(y).

If Type(x) is Undefined, return true.

If Type(x) is Null, return true.

If Type(x) is String, then

If x and y are exactly the same sequence of code units (same length and same code units at corresponding indices), return true; otherwise, return false.

If Type(x) is Boolean, then

If x and y are both true or both false, return true; otherwise, return false.

If Type(x) is Symbol, then

If x and y are both the same Symbol value, return true; otherwise, return false.

If x and y are the same Object value, return true. Otherwise, return false.

按照<https://tc39.es/ecma262/#sec-samevaluenonnumeric>的说明测试了下也没有找到其



他的可能

好奇这个答案是什么

作者回复: 绝对是还有的。至少一个。^^.

共 2 条评论 >



晓小东

2019-12-27

老师这个“其中的 boolean 是通过查表来进行的” 这个查表该如何理解???

作者回复: Here:

<https://tc39.es/ecma262/#sec-toboolean>



潇潇雨歇

2019-12-27

想不出啦.....NaN不是唯一的吗

作者回复: 参见 @sprinty 的答案哟。总算有人给出来这个标准答案了。呵呵~

共 7 条评论 >

