

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。

```
> [] + {}
'[] Object'

> {} + []
0

> {} + {}
NaN

> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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:

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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;
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [[].toString(), {}.toString()]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# + [ ] 将等义于
> + Number ( [ ] )
0

# + { } 将等义于
> + Number ( { } )
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。

- 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`：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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采用这种方式“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。

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。

```
> [] + {}
'[object Object]'
```

```
> {} + []
0
```

```
> {} + {}
NaN
```

```
> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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:

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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;
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [[].toString(), {}.toString()]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
```

```
> ???  
NaN  
  
# [] + []  
> '' + ''  
''
```

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。

```
{ } + []  
{ } + { }
```

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：

```
# +[] 将等义于  
> + Number([])  
0  
  
# +{} 将等义于  
> + Number({})  
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`：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
```

```
trueabc
```

```
# 与非字符串做"+"运算时
> 'OK, ' + x
Call toString
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

简单地讲，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。

简单地讲，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。

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，简单地讲，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。

```
> [] + {}
'[object Object]'
```

```
> {} + []
0
```

```
> {} + {}
```

NaN

```
> [] + []  
''
```

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**原本就是原始值，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. 如果x是非对象，则返回x  
> _ToPrimitive(5)  
5
```

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：

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

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**Preferred Type**，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：

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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；
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [ [].toString(), {}.toString() ]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]  
{ } + { }
```

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：

```
# + [ ] 将等义于  
> + Number([ ])  
0
```

```
# + { } 将等义于  
> + Number({ })  
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`：

```
> x = new Object  
> Number(x)  
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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

简单地讲，`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题目，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。

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。

```
> [] + {}  
'[object Object]'
```

```
> {} + []  
0
```

```
> {} + {}  
NaN
```

```
> [] + []  
,,
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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Preferred Type，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：

```
> x = Symbol()
> obj = Object(x)
> obj.valueOf() === x
true
```


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。

```
> Object(5) + Object(5)
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。

```
# 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；
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [[].toString(), {}.toString()]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# +[] 将等义于
> + Number([ ])
0
```

```
# +{ } 将等义于
> + Number({ })
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`：

```
> x = new Object
> Number(x)
NaN
```

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，二者之任一要是字符串的话，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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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。

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。

```
> [] + {}
'[object Object]'
```

```
> {} + []
0
```

```
> {} + {}
NaN
```

```
> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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：

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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；
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [ [].toString(), {}.toString() ]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# +[] 将等义于
> + Number([])
0

# +{} 将等义于
> + Number({})
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`：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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主题，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。

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。

```
> [] + {}
'[] Object'

> {} + []
0

> {} + {}
NaN

> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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：

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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；
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [ [].toString(), {}.toString() ]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# + [ ] 将等义于
> + Number([ ])
0

# + { } 将等义于
> + Number({ })
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：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
```

```
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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。

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。

```
> [] + {}
'[] Object'

> {} + []
0

> {} + {}
NaN

> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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:

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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;
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [[].toString(), {}.toString()]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# + [ ] 将等义于
> + Number ( [ ] )
0

# + { } 将等义于
> + Number ( { } )
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`：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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采用这种方式“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。

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。

```
> [] + {}
'[object Object]'
```

```
> {} + []
0
```

```
> {} + {}
NaN
```

```
> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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:

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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;
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [[].toString(), {}.toString()]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
```

```
> ???  
NaN  
  
# [] + []  
> '' + ''  
''
```

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。

```
{ } + []  
{ } + { }
```

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：

```
# +[] 将等义于  
> + Number([])  
0  
  
# +{} 将等义于  
> + Number({})  
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`：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
```

```
trueabc
```

```
# 与非字符串做"+"运算时
> 'OK, ' + x
Call toString
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

简单地讲，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。

简单地讲，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。

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，简单地讲，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。

```
> [] + {}
'[object Object]'
```

```
> {} + []
0
```

```
> {} + {}
```

NaN

```
> [] + []  
''
```

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**原本就是原始值，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. 如果x是非对象，则返回x  
> _ToPrimitive(5)  
5
```

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：

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

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**Preferred Type**，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：

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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；
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [ [].toString(), {}.toString() ]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]  
{ } + { }
```

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：

```
# + [ ] 将等义于  
> + Number([ ])  
0
```

```
# + { } 将等义于  
> + Number({ })  
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`：

```
> x = new Object  
> Number(x)  
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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

简单地讲，`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采用这种方式“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。

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。

```
> [] + {}
'[] Object'

> {} + []
0

> {} + {}
NaN

> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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Preferred Type，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：

```
> x = Symbol()
> obj = Object(x)
> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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；
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [[].toString(), {}.toString()]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# +[] 将等效于
> + Number([ ])
0
```

```
# +{ } 将等效于
> + Number({ })
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`：

```
> x = new Object
> Number(x)
NaN
```

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，二者之任一要是字符串的话，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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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。

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。

```
> [] + {}
'[object Object]'
```

```
> {} + []
0
```

```
> {} + {}
NaN
```

```
> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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：

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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；
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [ [].toString(), {}.toString() ]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# +[] 将等义于
> + Number([])
0

# +{} 将等义于
> + Number({})
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`：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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主题，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。

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。

```
> [] + {}
'[] Object'

> {} + []
0

> {} + {}
NaN

> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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：

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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；
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [ [].toString(), {}.toString() ]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# + [ ] 将等义于
> + Number([ ])
0

# + { } 将等义于
> + Number({ })
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：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
```

```
Call valueOf
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，上述两个操作中都并没有调用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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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。

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。

```
> [] + {}
'[] Object'

> {} + []
0

> {} + {}
NaN

> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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:

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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;
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [[].toString(), {}.toString()]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# + [ ] 将等义于
> + Number ( [ ] )
0

# + { } 将等义于
> + Number ( { } )
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。

- 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`：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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采用这种方式“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。

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。

```
> [] + {}
'[object Object]'
```

```
> {} + []
0
```

```
> {} + {}
NaN
```

```
> [] + []
''
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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:

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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;
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [[].toString(), {}.toString()]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
```

```
> ???  
NaN  
  
# [] + []  
> '' + ''  
''
```

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。

```
{ } + [ ]  
{ } + { }
```

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：

```
# + [ ] 将等义于  
> + Number([ ])  
0  
  
# + { } 将等义于  
> + Number({ })  
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`：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
```

```
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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

简单地讲，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。

简单地讲，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。

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，简单地讲，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。

```
> [] + {}
'[object Object]'
```

```
> {} + []
0
```

```
> {} + {}
```

NaN

```
> [] + []  
''
```

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**原本就是原始值，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. 如果x是非对象，则返回x  
> _ToPrimitive(5)  
5
```

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：

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

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**Preferred Type**，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：

```
> x = Symbol()

> obj = Object(x)

> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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；
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [ [].toString(), {}.toString() ]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# + [ ] 将等义于
> + Number([ ])
0
```

```
# + { } 将等义于
> + Number({ })
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`：

```
> x = new Object
> Number(x)
NaN
```

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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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

简单地讲，`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采用这种方式“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。

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。

```
> [] + {}  
'[object Object]'
```

```
> {} + []  
0
```

```
> {} + {}  
NaN
```

```
> [] + []  
,,
```

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. 如果x是非对象，则返回x
> _ToPrimitive(5)
5
```

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：

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

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Preferred Type，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：

```
> x = Symbol()
> obj = Object(x)
> obj.valueOf() === x
true
```

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。

```
> Object(5) + Object(5)
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。

```
# 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；
> [typeof ([].valueOf()), typeof ({}.valueOf())]
[ 'object', 'object' ]

# 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
> [[].toString(), {}.toString()]
[ '', '[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。

```
# [] + {}
> '' + '[object Object]'
'[object Object]'

# {} + []
> ???
0

# {} + {}
> ???
NaN

# [] + []
> '' + ''
''
```

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。

```
{ } + [ ]
{ } + { }
```

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：

```
# +[] 将等义于
> + Number([ ])
0

# +{} 将等义于
> + Number({ })
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`：

```
> x = new Object
> Number(x)
NaN
```

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，二者之任一要是字符串的话，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：

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

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：

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

```
# 例2：与字符串做“+”运算时
> 'OK, ' + x
Call valueOf
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：

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

测试如下：

```
# 示例
> x = new MyDate;

# 与非字符串做“+”运算时
> true + x
Call toString
trueabc

# 与非字符串做“+”运算时
> 'OK, ' + x
Call toString
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。