数据类型的转换

概述

JavaScript 是一种动态类型语言，变量没有类型限制，可以随时赋予任意值。

```javascript
var x = y ? 1 : 'a';
```

上面代码中，变量`x`到底是数值还是字符串，取决于另一个变量`y`的值。`y`为`true`时，`x`是一个数值；`y`为`false`时，`x`是一个字符串。这意味着，`x`的类型没法在编译阶段就知道，必须等到运行时才能知道。

虽然变量的数据类型是不确定的，但是各种运算符对数据类型是有要求的。如果运算符发现，运算子的类型与预期不符，就会自动转换类型。比如，减法运算符预期左右两侧的运算子应该是数值，如果不是，就会自动将它们转为数值。

```javascript
'4' - '3' // 1
```

上面代码中，虽然是两个字符串相减，但是依然会得到结果数值`1`，原因就在于 JavaScript 将运算子自动转为了数值。

本章讲解数据类型自动转换的规则。在此之前，先讲解如何手动强制转换数据类型。

强制转换

强制转换主要指使用`Number()`、`String()`和`Boolean()`三个函数，手动将各种类型的值，分别转换成数字、字符串或者布尔值。

Number()

使用`Number`函数，可以将任意类型的值转化成数值。

下面分成两种情况讨论，一种是参数是原始类型的值，另一种是参数是对象。

（1）原始类型值

原始类型值的转换规则如下。

```javascript
// 数值：转换后还是原来的值
Number(324) // 324
```

```
// 字符串：如果可以被解析为数值，则转换为相应的数值
Number('324') // 324

// 字符串：如果不可以被解析为数值，返回 NaN
Number('324abc') // NaN

// 空字符串转为0
Number('') // 0

// 布尔值：true 转成 1，false 转成 0
Number(true) // 1
Number(false) // 0

// undefined：转成 NaN
Number(undefined) // NaN

// null：转成0
Number(null) // 0
```

`Number`函数将字符串转为数值，要比`parseInt`函数严格很多。基本上，只要有一个字符无法转成数值，整个字符串就会被转为`NaN`。

```javascript
parseInt('42 cats') // 42
Number('42 cats') // NaN
```

上面代码中，`parseInt`逐个解析字符，而`Number`函数整体转换字符串的类型。

另外，`parseInt`和`Number`函数都会自动过滤一个字符串前导和后缀的空格。

```javascript
parseInt('\t\v\r12.34\n') // 12
Number('\t\v\r12.34\n') // 12.34
```

（2）对象

简单的规则是，`Number`方法的参数是对象时，将返回`NaN`，除非是包含单个数值的数组。

```javascript
Number({a: 1}) // NaN
Number([1, 2, 3]) // NaN
Number([5]) // 5
```

之所以会这样，是因为`Number`背后的转换规则比较复杂。

第一步，调用对象自身的`valueOf`方法。如果返回原始类型的值，则直接对该值使用`Number`函数，不再进行后续步骤。

第二步，如果`valueOf`方法返回的还是对象，则改为调用对象自身的`toString`方法。如果`toString`方法返回原始类型的值，则对该值使用`Number`函数，不再进行后续步骤。

第三步，如果`toString`方法返回的是对象，就报错。

请看下面的例子。

```javascript
var obj = {x: 1};
Number(obj) // NaN

// 等同于
if (typeof obj.valueOf() === 'object') {
  Number(obj.toString());
} else {
  Number(obj.valueOf());
}
```

上面代码中，`Number`函数将`obj`对象转为数值。背后发生了一连串的操作，首先调用`obj.valueOf`方法，结果返回对象本身；于是，继续调用`obj.toString`方法，这时返回字符串`[object Object]`，对这个字符串使用`Number`函数，得到`NaN`。

默认情况下，对象的`valueOf`方法返回对象本身，所以一般总是会调用`toString`方法，而`toString`方法返回对象的类型字符串（比如`[object Object]`）。所以，会有下面的结果。

```javascript
Number({}) // NaN
```

如果`toString`方法返回的不是原始类型的值，结果就会报错。

```javascript
var obj = {
  valueOf: function () {
    return {};
  },
  toString: function () {
    return {};
  }
};

Number(obj)
// TypeError: Cannot convert object to primitive value
```

上面代码的`valueOf`和`toString`方法，返回的都是对象，所以转成数值时会报错。

从上例还可以看到，`valueOf`和`toString`方法，都是可以自定义的。

```javascript
Number({
  valueOf: function () {
    return 2;
  }
})
// 2

Number({
  toString: function () {
    return 3;
  }
})
// 3

Number({
  valueOf: function () {
    return 2;
  },
  toString: function () {
    return 3;
  }
})
// 2
```

上面代码对三个对象使用`Number`函数。第一个对象返回`valueOf`方法的值，第二个对象返回`toString`方法的值，第三个对象表示`valueOf`方法先于`toString`方法执行。

String()

`String`函数可以将任意类型的值转化成字符串，转换规则如下。

（1）原始类型值

- **数值**：转为相应的字符串。
- **字符串**：转换后还是原来的值。
- **布尔值**：`true`转为字符串`"true"`，`false`转为字符串`"false"`。
- **undefined**：转为字符串`"undefined"`。
- **null**：转为字符串`"null"`。

```javascript
String(123) // "123"
String('abc') // "abc"
String(true) // "true"
String(undefined) // "undefined"
```

```
String(null) // "null"
```

（2）对象

`String`方法的参数如果是对象，返回一个类型字符串；如果是数组，返回该数组的字符串形式。

```javascript
String({a: 1}) // "[object Object]"
String([1, 2, 3]) // "1,2,3"
```

`String`方法背后的转换规则，与`Number`方法基本相同，只是互换了`valueOf`方法和`toString`方法的执行顺序。

1. 先调用对象自身的`toString`方法。如果返回原始类型的值，则对该值使用`String`函数，不再进行以下步骤。

2. 如果`toString`方法返回的是对象，再调用原对象的`valueOf`方法。如果`valueOf`方法返回原始类型的值，则对该值使用`String`函数，不再进行以下步骤。

3. 如果`valueOf`方法返回的是对象，就报错。

下面是一个例子。

```javascript
String({a: 1})
// "[object Object]"

// 等同于
String({a: 1}.toString())
// "[object Object]"
```

上面代码先调用对象的`toString`方法，发现返回的是字符串`[object Object]`，就不再调用`valueOf`方法了。

如果`toString`法和`valueOf`方法，返回的都是对象，就会报错。

```javascript
var obj = {
  valueOf: function () {
    return {};
  },
  toString: function () {
    return {};
  }
};
```

```
String(obj)
// TypeError: Cannot convert object to primitive value
```

下面是通过自定义`toString`方法，改变返回值的例子。

```javascript
String({
  toString: function () {
    return 3;
  }
})
// "3"

String({
  valueOf: function () {
    return 2;
  }
})
// "[object Object]"

String({
  valueOf: function () {
    return 2;
  },
  toString: function () {
    return 3;
  }
})
// "3"
```

上面代码对三个对象使用`String`函数。第一个对象返回`toString`方法的值（数值3），第二个对象返回的还是`toString`方法的值（`[object Object]`），第三个对象表示`toString`方法先于`valueOf`方法执行。

Boolean()

`Boolean()`函数可以将任意类型的值转为布尔值。

它的转换规则相对简单：除了以下五个值的转换结果为`false`，其他的值全部为`true`。

- `undefined`
- `null`
- `0`（包含`-0`和`+0`）
- `NaN`
- `''`（空字符串）

```javascript
Boolean(undefined) // false
Boolean(null) // false
```

```
Boolean(0) // false
Boolean(NaN) // false
Boolean('') // false
```

当然，`true`和`false`这两个布尔值不会发生变化。

```javascript
Boolean(true) // true
Boolean(false) // false
```

注意，所有对象（包括空对象）的转换结果都是`true`，甚至连`false`对应的布尔对象`new Boolean(false)`也是`true`（详见《原始类型值的包装对象》一章）。

```javascript
Boolean({}) // true
Boolean([]) // true
Boolean(new Boolean(false)) // true
```

所有对象的布尔值都是`true`，这是因为 JavaScript 语言设计的时候，出于性能的考虑，如果对象需要计算才能得到布尔值，对于`obj1 && obj2`这样的场景，可能会需要较多的计算。为了保证性能，就统一规定，对象的布尔值为`true`。

自动转换

下面介绍自动转换，它是以强制转换为基础的。

遇到以下三种情况时，JavaScript 会自动转换数据类型，即转换是自动完成的，用户不可见。

第一种情况，不同类型的数据互相运算。

```javascript
123 + 'abc' // "123abc"
```

第二种情况，对非布尔值类型的数据求布尔值。

```javascript
if ('abc') {
  console.log('hello')
}  // "hello"
```

第三种情况，对非数值类型的值使用一元运算符（即`+`和`-`）。

```javascript
+ {foo: 'bar'} // NaN
```

- [1, 2, 3] // NaN
```

自动转换的规则是这样的：预期什么类型的值，就调用该类型的转换函数。比如，某个位置预期为字符串，就调用`String`函数进行转换。如果该位置即可以是字符串，也可能是数值，那么默认转为数值。

由于自动转换具有不确定性，而且不易除错，建议在预期为布尔值、数值、字符串的地方，全部使用`Boolean`、`Number`和`String`函数进行显式转换。

### 自动转换为布尔值

JavaScript 遇到预期为布尔值的地方（比如`if`语句的条件部分），就会将非布尔值的参数自动转换为布尔值。系统内部会自动调用`Boolean`函数。

因此除了以下五个值，其他都是自动转为`true`。

- `undefined`
- `null`
- `+0`或`-0`
- `NaN`
- `''`（空字符串）

下面这个例子中，条件部分的每个值都相当于`false`，使用否定运算符后，就变成了`true`。

```javascript
if ( !undefined
  && !null
  && !0
  && !NaN
  && !''
) {
  console.log('true');
} // true
```

下面两种写法，有时也用于将一个表达式转为布尔值。它们内部调用的也是`Boolean`函数。

```javascript
// 写法一
expression ? true : false

// 写法二
!! expression
```

自动转换为字符串
```

JavaScript 遇到预期为字符串的地方，就会将非字符串的值自动转为字符串。具体规则是，先将复合类型的值转为原始类型的值，再将原始类型的值转为字符串。

字符串的自动转换，主要发生在字符串的加法运算时。当一个值为字符串，另一个值为非字符串，则后者转为字符串。

```javascript
'5' + 1 // '51'
'5' + true // "5true"
'5' + false // "5false"
'5' + {} // "5[object Object]"
'5' + [] // "5"
'5' + function (){} // "5function (){}"
'5' + undefined // "5undefined"
'5' + null // "5null"
```

这种自动转换很容易出错。

```javascript
var obj = {
  width: '100'
};

obj.width + 20 // "10020"
```

上面代码中，开发者可能期望返回`120`，但是由于自动转换，实际上返回了一个字符`10020`。

自动转换为数值

JavaScript 遇到预期为数值的地方，就会将参数值自动转换为数值。系统内部会自动调用`Number`函数。

除了加法运算符（`+`）有可能把运算子转为字符串，其他运算符都会把运算子自动转成数值。

```javascript
'5' - '2' // 3
'5' * '2' // 10
true - 1  // 0
false - 1 // -1
'1' - 1   // 0
'5' * []    // 0
false / '5' // 0
'abc' - 1   // NaN
null + 1 // 1
undefined + 1 // NaN
```

上面代码中，运算符两侧的运算子，都被转成了数值。

> 注意：`null`转为数值时为`0`，而`undefined`转为数值时为`NaN`。

一元运算符也会把运算子转成数值。

```javascript
+'abc' // NaN
-'abc' // NaN
+true // 1
-false // 0
```

参考链接

- Axel Rauschmayer, [What is {} + {} in JavaScript?](http://www.2ality.com/2012/01/object-plus-object.html)
- Axel Rauschmayer, [JavaScript quirk 1: implicit conversion of values](http://www.2ality.com/2013/04/quirk-implicit-conversion.html)
- Benjie Gillam, [Quantum JavaScript?](http://www.benjiegillam.com/2013/06/quantum-javascript/)