语法专题

数据类型的转换

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。

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

上述代码定义了一个变量`x`，但是`x`的取值不是固定的，取决于变量`y`的值，如果`y`为`true`，则`x`的值为`1`，否则为字符串`a`。

> 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。

运算符的作用是对源数据进行处理，得到一个新的数据。那么针对不同的数据类型，运算符有不同的运算规则。

一般来说，同一个运算符处理的源数据类型是一致的，如果不一致，那么JavaScript会根据一定的规则将源数据进行转换。

强制转换

有时候我们不清楚，JavaScript是如何对数据进行转换的，所以需要手动进行转换。

> 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。

上述三个函数都会分成两种情况：一种是参数是原始类型的值，另一种是参数是对象。

Number()

> `Number`函数可以将任意类型的值转化成数值

原始类型数据

- 数值：转换后还是原来的值
- 字符串：如果可以被解析为数值，则转换为相应的数值，如果不可以被解析为数值，返回 NaN，空字符串转为0
- 布尔值：true 转成 1，false 转成 0
- undefined：转成 NaN
- null：转成0

`Number`与`parseInt`的异同：

1. `Number`比`parseInt`转换方式更加严格
2. `Number`和`parseInt`函数都会自动过滤一个字符串前导和后缀的空格

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。

`Number`函数的转换规则：

1. 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。

2. 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。
3. 第三步，如果`toString`方法返回的是对象，就报错。

案例：

```
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`。

我们可以通过自定义*valueOf*和*toString*方法来使得对象能够进行转换为数值。

```
// 单个对象自定义
var obj = {
    value: 123456,
    valueOf: function(){
        return this.value
    }
}
obj.valueOf(); // => 123456
Number(obj); // => 123456

// 通过原型链对所有对象进行设置
Object.prototype.valueOf = function(){
    return this.value || ''
}
Number({}); // => 0
```

String()

> `String`函数可以将任意类型的值转化成字符串。

原始类型数据

- 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"。

对象类型

> 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。

```
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。

我们可以通过自定义*valueOf*和*toString*方法来使得对象能够进行转换为字符串。

```
// 单个对象自定义
var obj = {
    value: 123456,
    toString: function(){
        return this.value
    }
}
obj.toString(); // => '123456'
String(obj); // => '123456'

// 通过原型链对所有对象进行设置
Object.prototype.toString = function(){
    return this.value || ''
}
String({}); // => ''
```

注意：如果自定义`valueOf`和`toString`返回的都是对象，就会报错

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）

```
Boolean(undefined) // false
Boolean(null) // false
Boolean(0) // false
Boolean(NaN) // false
Boolean('') // 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*。

自动转换

自动转换以强制转换为基础。

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。

1. 第一种情况，不同类型的数据互相运算。
2. 第二种情况，对非布尔值类型的数据求布尔值。
3. 第三种情况，对非数值类型的值使用一元运算符（即+和-）。

> 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。如果该位置既可以是字符串，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。

自动转化为布尔值

> `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）

自动转化为字符串

> `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。

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

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。

```
'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
```

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换成数值。

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

错误处理机制

Error实例对象

> `JavaScript`解析或运行时，一旦发生错误，引擎就会抛出一个错误对象。`JavaScript`原生提供`Error`构造函数，所有抛出的错误都是这个构造函数的实例。

```
var err = new Error('出错了');
err.message // "出错了"
```

> `Error()`构造函数接受一个参数，表示错误提示，可以从实例的`message`属性读到这个参数。抛出`Error`实例对象以后，整个程序就中断在发生错误的地方，不再往下执行。

部分`JavaScript`引擎还提供了`name`和`stack`属性，表示错误的名称和堆栈。

使用`name`和`message`这两个属性，可以对发生什么错误有一个大概的了解。

```
if (error.name) {
  console.log(error.name + ': ' + error.message);
}

function throwit() {
  throw new Error('');
}

function catchit() {
  try {
    throwit();
  } catch(e) {
    console.log(e.stack); // print stack trace
  }
}

catchit()
// Error
```

```
//    at throwit (~/examples/throwcatch.js:9:11)
//    at catchit (~/examples/throwcatch.js:3:9)
//    at repl:1:5
```

原生错误类型

> `Error`实例对象是最一般的错误类型，在它的基础上，`JavaScript`还定义了其他6种错误对象。也就是说，存在Error的6个派生对象。

SyntaxError 对象

语法错误

> `SyntaxError`对象是解析代码时发生的语法错误。

```
// 变量名错误
var 1a;
// Uncaught SyntaxError: Invalid or unexpected token

// 缺少括号
console.log 'hello');
// Uncaught SyntaxError: Unexpected string
```

ReferenceError 对象

引用错误

> `ReferenceError`对象是引用一个不存在的变量时发生的错误。

```
// 使用一个不存在的变量
unknownVariable
// Uncaught ReferenceError: unknownVariable is not defined

// 将一个值分配给无法分配的对象，比如对函数的运行结果赋值。
// 等号左侧不是变量
console.log() = 1
// Uncaught ReferenceError: Invalid left-hand side in assignment
```

RangeError 对象

范围错误

> `RangeError`对象是一个值超出有效范围时发生的错误。主要有几种情况，一是数组长度为负数，二是`Number`对象的方法参数超出范围，以及函数堆栈超过最大值。

```
// 数组长度不得为负数
new Array(-1)
// Uncaught RangeError: Invalid array length
```

TypeError 对象

类型错误

> `TypeError`对象是变量或参数不是预期类型时发生的错误。比如，对字符串、布尔值、数值等原始类型的值使用`new`命令，就会抛出这种错误，因为`new`命令的参数应该是一个构造函数。

```
new 123
// Uncaught TypeError: 123 is not a constructor

var obj = {};
obj.unknownMethod()
// Uncaught TypeError: obj.unknownMethod is not a function
```

URIError 对象

URI错误

> `URIError`对象是`URI`相关函数的参数不正确时抛出的错误，主要涉及`encodeURI()`、`decodeURI()`、`encodeURIComponent()`、`decodeURIComponent()`、`escape()`和`unescape()`这六个函数。

EvalError 对象

> `eval`函数没有被正确执行时，会抛出`EvalError`错误。该错误类型已经不再使用了，只是为了保证与以前代码兼容，才继续保留。

总结

> 以上这6种派生错误，连同原始的Error对象，都是构造函数。开发者可以使用它们，手动生成错误对象的实例。这些构造函数都接受一个参数，代表错误提示信息（message）。

在开发过程中如需要向控制台输出如http请求500/404等错误，可以用这些错误提示抛出错误显示在控制台。

在一些项目中也可以用于某些特殊情况可能产生错误地方的警告或者提示。

```
var err1 = new Error('出错了！');
var err2 = new RangeError('出错了，变量超出有效范围！');
var err3 = new TypeError('出错了，变量类型无效！');

err1.message // "出错了！"
err2.message // "出错了，变量超出有效范围！"
err3.message // "出错了，变量类型无效！"
```

自定义错误

用户可以通过自定义错误的构造函数，然后修改原型链，使其继承`Error`对象就可以生成自定义对象。

```
function UserError(message) {
  this.message = message || '默认信息';
  this.name = 'UserError';
}

UserError.prototype = new Error();
UserError.prototype.constructor = UserError;

new UserError('这是自定义的错误！');
```

throw语句

> throw语句的作用是手动中断程序执行，抛出一个错误。

throw可以抛出任何类型的值。

```
// 抛出一个错误
throw new Error('出错了')

// 抛出一个字符串
throw 'Error! ';
// Uncaught Error!

// 抛出一个数值
throw 42;
// Uncaught 42

// 抛出一个布尔值
throw true;
// Uncaught true
```

try...catch

在`JavaScript`中，如果发生错误，程序会立即停止执行，这和`JavaScript`的特性有关。

我们可以使用`try...catch`结构对错误进行进一步处理，从而不影响后续程序执行。

```
try {
  throw new Error('出错了!');
} catch (e) {
  console.log(e.name + ": " + e.message);
  console.log(e.stack);
}
// Error: 出错了!
//    at <anonymous>:3:9
//    ...
console.log('我会继续执行')
```

> 如果你不确定某些代码是否会报错，就可以把它们放在`try...catch`代码块之中，便于进一步对错误进行处理。

如果使用`Nodejs`开发服务端的话，调用数据库会经常使用到`try...catch`语句，因为无法预知读取数据过程中会出现哪些问题。

finally代码块

> `try...catch`结构允许在最后添加一个`finally`代码块，表示不管是否出现错误，都必需在最后运行的语句。

```
function cleansUp() {
  try {
    throw new Error('出错了……');
    console.log('此行不会执行');
  } finally {
    console.log('完成清理工作');
  }
```

```
}

cleansUp()
// 完成清理工作
// Uncaught Error: 出错了……
//    at cleansUp (<anonymous>:3:11)
//    at <anonymous>:10:1
```

由于没有`catch`语句块，一旦发生错误，代码就会中断执行。中断执行之前，会先执行`finally`代码块，然后再向用户提示报错信息。

如果`try`或者`catch`语句存在`return`语句，`finally`仍然会执行，并且`return`语句在`finally`之前执行，但是会等待`finally`执行完毕后返回。

```
var count = 0;
function countUp() {
  try {
    return count;
  } finally {
    count++;
  }
}

countUp()
// 0
count
// 1
```

编程风格

> “编程风格”（programming style）指的是编写代码的样式规则。不同的程序员，往往有不同的编程风格。

虽然一个人的编程风格可以是自由的、任意的，但是为了能够让代码更加清晰易读，减少错误的发生，我们应当选择一种能够清晰表达代码意图的风格。

> 如果你选定了一种“编程风格”，就应该坚持遵守，切忌多种风格混用。如果你加入他人的项目，就应该遵守现有的风格。

缩进

行首的空格和`Tab`键，都可以产生代码缩进效果（indent）。

不同编辑器的`Tab`键产生的缩进不一定相同，通常会有四个空格和两个空格。

不要混合使用空格和`Tab`键。

区块

当循环和判断语句的代码块只有一行代码时，我们可以省略大括号。但是这会产生很多问题，由于编辑器的空格和换行会自动被忽略，所以浏览器解析代码的时候可能会出现问题。

```
// 代码
if (a)
  b();
  c();
```

```
// 本意
if (a) {
  b();
  c();
}

// 实际
if (a) {
  b();
}
  c();
```

因此，建议总是使用大括号表示区块。

当我们使用一些语句，如循环、判断等语句时，大括号的位置既可以写在关键字后面，也可以另起一行。

```
if(true){

}
if(true)
{

}
```

但是我们知道，语句是以分号结尾，如果没有浏览器会自动加上分号，所以会出现一些特殊情况。

```
return
{
  key: value
};
// 本意
return {
  key: value
};
// 实际
return;
{
  key: value
};
```

这就会产生较为严重的语法问题。因此，建议大括号始终跟在关键字后面。

圆括号

> 圆括号（parentheses）在 JavaScript 中有两种作用，一种表示函数的调用，另一种表示表达式的组合（grouping）。

我们可以使用空格区分这两种不同的括号。

1. 表示函数调用时，函数名与左括号之间没有空格。
2. 表示函数定义时，函数名与左括号之间没有空格。
3. 其他情况时，前面位置的语法元素与左括号之间，都有一个空格。

行尾的分号

分号表示一条语句的结束。JavaScript 允许省略行尾的分号。但是在一些特殊情况下，不加分号会出现意料之外的错误。

不需要使用分号的情况

- `for`和`while`循环体（注意：do...while有分号）
- 分支语句：`if`，`switch`，`try`
- 函数的声明语句（注意：函数表达式需要添加分号）

分号的自动添加

除了上述三种情况，所有语句都应该使用分号。但是，如果没有使用分号，大多数情况下，`JavaScript`会自动添加。

> 这种语法特性被称为“分号的自动添加”（Automatic Semicolon Insertion，简称 ASI）。

强烈要求始终在语句的结尾添加分号！！！

全局变量

`JavaScript`最大的语法缺点，可能就是全局变量对于任何一个代码块，都是可读可写。这对代码的模块化和重复使用，非常不利。

因此，建议避免使用全局变量。如果不得不使用，可以考虑用大写字母表示变量名，这样更容易看出这是全局变量，比如`UPPER_CASE`。

另外注意任何时候赋值的时候都要是一个已存在的变量，否则会产生全局变量。

变量声明

由于变量提升和函数提升的问题，强烈要求所有变量在代码块头部声明，所有的函数在调用前声明，函数内部的变量在函数头部声明。

with语句

with语句可能会和全局变量产生相关冲突，尽量减少使用，建议不用。

相等和全等

相等运算符会自动转换变量类型，这可能会造成不可预知的问题。

建议不要使用相等运算符（`==`），只使用严格相等运算符（`===`）。

自增和自减

自增和自减的运算符放在前面和后面的效果不一样，即使是多年经验的开发者也会经常搞混淆。

我们可以使用`+=1`来代替自增，使用`-=1`来代替自减。

switch...case 结构

> `switch...case`结构要求，在每一个`case`的最后一行必须是`break`语句，否则会接着运行下一个`case`。这样不仅容易忘记，还会造成代码的冗长。
> 而且，switch...case不使用大括号，不利于代码形式的统一。

建议使用对象结构代替：

```
function customSwitch(action) {
  var actions = {
    'hack': function () {
      return 'hack';
    },
    'slash': function () {
      return 'slash';
    },
    'run': function () {
      return 'run';
    }
  };

  if (typeof actions[action] !== 'function') {
    throw new Error('Invalid action.');
  }

  return actions[action]();
}
```

console对象与控制台

`console`对象是`JavaScript`的原生对象，可以输出各种信息到控制台，并且还提供了很多有用的辅助方法。

我们可以通过`console`来调试网页代码，也可以通过它接收网页报出的错误和警告信息。

浏览器自带的开发者工具还提供了命令行接口用来与网页互动以及运行简易的代码。

console 对象的静态方法

`console`对象提供的各种静态方法，用来与控制台窗口互动。

`console.log`方法用于在控制台输出信息。它可以接受一个或多个参数，将它们连接起来输出。

```
console.log('Hello World')
// Hello World
console.log('a', 'b', 'c')
// a b c
```

`console.log`方法支持以下占位符，不同类型的数据必须使用对应的占位符。

- %s 字符串
- %d 整数
- %i 整数
- %f 浮点数
- %o 对象的链接
- %c CSS 格式字符串

```
var number = 11 * 9;
var color = 'red';

console.log('%d %s balloons', number, color);
// 99 red balloons
```

`console.info`是`console.log`方法的别名，用法完全一样。只不过console.info方法会在输出信息的前面，加上一个蓝色图标。

> `console`对象的所有方法，都可以被覆盖。因此，可以按照自己的需要，定义`console.log`方法。

console.warn()，console.error()

> warn方法和error方法也是在控制台输出信息，它们与log方法的不同之处在于，warn方法输出信息时，在最前面加一个黄色三角，表示警告；error方法输出信息时，在最前面加一个红色的叉，表示出错。

console.table()

对于某些复合类型的数据，console.table方法可以将其转为表格显示。

```
var languages = [
  { name: "JavaScript", fileExtension: ".js" },
  { name: "TypeScript", fileExtension: ".ts" },
  { name: "CoffeeScript", fileExtension: ".coffee" }
];

console.table(languages);
```

-	name	fileExtension
0	JavaScript	.js
1	TypeScript	.ts
2	CoffeeScript	.coffee

如果是对象， 则键名会变成对应的索引。

console.count()

count方法用于计数，输出它被调用了多少次。

该方法可以接受一个字符串作为参数，作为标签，对执行次数进行分类。

```
function greet(user) {
  console.count(user);
  return "hi " + user;
}

greet('bob')
// bob: 1
// "hi bob"

greet('alice')
// alice: 1
// "hi alice"

greet('bob')
// bob: 2
// "hi bob"
```

console.dir()，console.dirxml()

`dir`方法用来对一个对象进行检查（inspect），并以易于阅读和打印的格式显示。

`dir`方法的输出结果，比`log`方法更易读，信息也更丰富。

```
console.log({f1: 'foo', f2: 'bar'})
// Object {f1: "foo", f2: "bar"}

console.dir({f1: 'foo', f2: 'bar'})
// Object
//   f1: "foo"
//   f2: "bar"
//   __proto__: Object
```

如果我们使用`document.body`获取`body`元素的话，使用`log`只能显示`body`的`HTML`代码，但是使用`dir`可以显示所有属性。

```
console.dir(document.body)
```

在node环境中，可以配置高亮。

```
console.dir(obj, {colors: true})
```

`dirxml`方法主要用于以目录树的形式，显示`DOM`节点。

`dirxml`方法的使用仅限于参数是`DOM`节点的情况下，如果是普通的`JavaScript`对象，`console.dirxml`等同于`console.dir`。

console.assert()

`console.assert`方法主要用于程序运行过程中，进行条件判断，如果不满足条件，就显示一个错误，但不会中断程序执行。这样就相当于提示用户，内部状态不正确。

第一个参数是表达式，第二个参数是字符串。只有当第一个参数为false，才会提示有错误。

```
console.assert(false, '判断条件不成立')
// Assertion failed: 判断条件不成立

console.assert(true, '判断条件成立')
// 不显示任何内容
```

console.time()，console.timeEnd()

这两个方法用于计时，可以算出一个操作所花费的准确时间。需要配合使用。

```
console.time('Array initialize');

var array= new Array(1000000);
for (var i = array.length - 1; i >= 0; i--) {
  array[i] = new Object();
};

console.timeEnd('Array initialize');
// Array initialize: 1914.481ms
```

time方法表示计时开始，timeEnd方法表示计时结束。它们的参数是计时器的名称。
调用timeEnd方法之后，控制台会显示“计时器名称: 所耗费的时间”。

console.trace()，console.clear()

`console.trace`方法显示当前执行的代码在堆栈中的调用路径。

`console.clear`方法用于清除当前控制台的所有输出，将光标回置到第一行。如果用户选中了控制台的“Preserve log”选项，console.clear方法将不起作用。