CHAPTER 2

JavaScript概览

介绍

JavaScript是基于原型、面向对象、弱类型的动态脚本语言。它从函数式语言中借鉴了一些强大的特性，如闭包和高阶函数，这也是 JavaScript语言本身有意思的地方。

从技术层面上来说，JavaScript是根据ECMAScript语言标准来实现的。这里有一点非常重要：由于Node使用了V8的原因，其实现很接近标准，另外，它还提供了一些标准之外实用的附加功能。换句话说，在Node中书写的JavaScript和浏览器上口碑不是很好的JavaScript有着重要的不同。

另外，Node中你书写的绝大多数JavaScript代码都符合Douglas Crockford在他那本著名的书《JavaScript语言精粹》中提到的 JavaScript语言的“精粹”。

本章分为以下两个部分：

- JavaScript基础。语言基础。适用于：Node、浏览器以及语言标准。
- V8中的JavaScript。V8提供的一些特性是浏览器不支持的，IE就更不用说了，因为这些特性是最近才纳入标准的。除此之外，V8还提供一些非标准的特性，它们能辅助解决一些常见的基本需求。

除此之外，下一章还会介绍Node中对语言的扩展和特性。

16 JavaScript基础

本章默认你对JavaScript及其语法有一定的了解。本章会介绍学习Node.js必须要掌握的JavaScript基础知识。

类型

JavaScript类型可以简单地分为两组：基本类型和复杂类型。访问基本类型，访问的是值，而访问复杂类型，访问的是对值的引用。

- 基本类型包括number、boolean、string、null以及undefined。
- 复杂类型包括array、function以及object。

如下述例子所示：

```
// 基本类型
var a = 5;
var b = a;
b = 6;
a; // 结果为5
b; // 结果为6

// 复杂类型
var a = ['hello', 'world'];
var b = a;
b[0] = 'bye';
a[0]; // 结果为“bye”
b[0]; // 结果为“bye”
```

上述例子中的第二部分，b和a包含了对值的相同引用。因此，当通过b修改数组的第一个元素时，a相应的值也更改了，也就说a[0] === b[0]。

类型的困惑

要在JavaScript中准确无误地判断变量值的类型并非易事。

因为对于绝大部分基本类型来说，JavaScript与其他面向对象语言一样有相应的构造器，比方说，你可以通过如下两种方式来创建一个字符串：

```
var a = 'woot';
var b = new String('woot');
a + b; // 'woot woot'
```

17 然而，要是对这两个变量使用typeof和instanceof操作符，事情就变得有意思了：

```
typeof a; // 'string'
typeof b; // 'object'
a instanceof String; // false
b instanceof String; // true
```

而事实上，这两个变量值绝对都是货真价实的字符串：

```
a.substr == b.substr; // true
```

并且使用==操作符判定时两者相等，而使用===操作符判定时并不相同：

```
a == b; // true
a === b; // false
```

考虑到有此差异，我建议你始终通过直观的方式进行定义，避免使用new。

有一点很重要，在条件表达式中，一些特定的值会被判定为false：null、undefined、' '，还有0：

```
var a = 0;
if (a) {
  // 这里始终不会被执行到
}
a == false; // true
a === false; // false
```

另外值得注意的是，typeof不会把null识别为类型为null：

```
typeof null == 'object'; // 很不幸，结果为true
```

数组也不例外，就算是通过[]这种方式定义数组也是如此：

```
typeof [] == 'object'; // true
```

这里要感谢V8给我们提供了判定是否为数组类型的方式，能够让我们免于使用hack的方式。

在浏览器环境中，我们通常要查看对象内部的[[Class]]值：Object.prototype.toString.call([]) == '[object Array]'。该值是不可变的，有利于我们在不同的上下文中（如浏览器窗口）对数组类型进行判定，而instanceof Array这种方式只适用于与数组初始化在相同上下文中才有效。

函数

18

在JavaScript中，函数最为重要。

它们都属于一等函数：可以作为引用存储在变量中，随后可以像其他对象一样，进行传递：

```
var a = function () {}
console.log(a); // 将函数作为参数传递
```

JavaScript中所有的函数都可以进行命名。有一点很重要，就是要能区分出函数名和变量名。

```
var a = function a () {
  'function' == typeof a; // true
};
```

THIS、FUNCTION#CALL以及FUNCTION#APPLY

下述代码中函数被调用时，this的值是全局对象。在浏览器中，就是window对象：

```
function a () {
  window == this; // true;
};

a();
```

调用以下函数时，使用.call和.apply方法可以改变this的值：

```
function a () {
  this.a == 'b'; // true
}

a.call({ a: 'b' });
```

call和apply的区别在于，call接受参数列表，而apply接受一个参数数组：

```
function a (b, c) {
  b == 'first'; // true
  c == 'second'; // true
}

a.call({ a: 'b' }, 'first', 'second')
a.apply({ a: 'b' }, ['first', 'second']);
```

19 函数的参数数量

函数有一个很有意思的属性——参数数量，该属性指明函数声明时可接收的参数数量。在JavaScript中，该属性名为length：

```
var a = function (a, b, c);
a.length == 3; // true
```

尽管这在浏览器端很少使用，但是，它对我们非常重要，因为一些流行的Node.js框架就是通过此属性来根据不同参数个数提供不同的功能的。

闭包

在JavaScript中，每次函数调用时，新的作用域就会产生。

在某个作用域中定义的变量只能在该作用域或其内部作用域（该作用域中定义的作用域）中才能访问到：

```
var a = 5;

function woot () {
  a == 5; // true

  var a = 6;

  function test () {
    a == 6; // true
  }

  test();
};

woot();
```

自执行函数是一种机制，通过这种机制声明和调用一个匿名函数，能够达到仅定义一个新作用域的作用。

```
var a = 3;

(function () {
  var a = 5;
})();

a == 3 // true;
```

自执行函数对声明私有变量是很有用的，这样可以让私有变量不被其他代码访问。

类

20

JavaScript中没有class关键词。类只能通过函数来定义：

```
function Animal () { }
```

要给所有Animal的实例定义函数，可以通过prototype属性来完成：

```
Animal.prototype.eat = function (food) {
  // eat method
}
```

这里值得一提的是，在prototype的函数内部，this并非像普通函数那样指向global对象，而是指向通过该类创建的实例对象：

```
function Animal (name) {
  this.name = name;
}

Animal.prototype.getName () {
```

```
  return this.name;
};

var animal = new Animal('tobi');
a.getName() == 'tobi'; // true
```

继承

JavaScript有基于原型的继承的特点。通常，你可以通过以下方式来模拟类继承。

定义一个要继承自Animal的构造器：

```
function Ferret () { };
```

要定义继承链，首先创建一个Animal对象，然后将其赋值给Ferret.prototype。

```
// 实现继承
Ferret.prototype = new Animal();
```

随后，可以为子类定义属性和方法：

```
// 为所有ferrets实例定义type属性
Ferret.prototype.type = 'domestic';
```

21 还可以通过prototype来重写和调用父类函数：

```
Ferret.prototype.eat = function (food) {
  Animal.prototype.eat.call(this, food);
  // ferret特有的逻辑写在这里
}
```

这项技术很赞。它是同类方案中最好的（相比其他函数式技巧），而且它不会破坏instanceof操作符的结果：

```
var animal = new Animal();
animal instanceof Animal // true
animal instanceof Ferret // false

var ferret = new Ferret();
ferret instanceof Animal // true
ferret instanceof Ferret // true
```

它最大的不足就是声明继承的时候创建的对象总要进行初始化（Ferret.prototype = new Animal），这种方式不好。一种解决该问题的方法就是在构造器中添加判断条件：

```
function Animal (a) {
  if (false !== a) return;
  // 初始化
}

Ferret.prototype = new Animal(false)
```

另外一个办法就是再定义一个新的空构造器，并重写它的原型：

```
function Animal () {
  // constructor stuff
}

function f () {};
f.prototype = Animal.prototype;
Ferret.prototype = new f;
```

幸运的是，V8提供了更简洁的解决方案，本章后续部分会做介绍。

TRY {} CATCH {}

try/catch允许进行异常捕获。下述代码会抛出异常：

```
> var a = 5;
> a()
TypeError: Property 'a' of object #<Object> is not a function
```

当函数抛出错误时，代码就停止执行了：

22

```
function () {
  throw new Error('hi');
  console.log('hi'); // 这里永远不会被执行到
}
```

若使用try/catch则可以进行错误处理，并让代码继续执行下去：

```
function () {
  var a = 5;
  try {
    a();
  } catch (e) {
    e instanceof Error; // true
  }

  console.log('you got here!');
}
```

v8中的JavaScript

至此，你已经了解了JavaScript在绝大多数环境下（包括早期浏览器中）的语言特性。

随着Chrome浏览器的发布，它带来了一个全新的JavaScript引擎——V8，它以极速的执行环境，加之时刻保持最新并支持最新ECMAScript特性的优势，快速地在浏览器市场中占据了重要的位置。

其中有些特性弥补了语言本身的不足。另外一部分特性的引入则要归功于像jQuery和PrototypeJS这样的前端类库，因为它们提供了非常实用的扩展和工具，如今，很难想象

JavaScript中要是没有了这些会是什么样子。

本章介绍V8中最有用的特性，并使用这些特性书写出更精准、更高效的代码，与此同时，代码的风格也将借鉴最流行的Node.js框架和库的代码风格。

OBJECT#KEYS

要想获取下述对象的键（a和c）：

```
var a = { a: 'b', c: 'd' };
```

通常会使用如下迭代的方式：

```
for (var i in a) { }
```

23 通过对键进行迭代，可以将它们收集到一个数组中。不过，如果采用如下方式对Object.prototype进行过扩展：

```
Object.prototype.c = 'd';
```

为了避免在迭代过程中把c也获取到，就需要使用hasOwnProperty来进行检查：

```
for (var i in a) {
  if (a.hasOwnProperty(i)) {}
}
```

在V8中，要获取对象上所有的自有键，还有更简单的方法：

```
var a = { a: 'b', c: 'd' };
Object.keys(a); // ['a', 'c']
```

ARRAY#ISARRAY

正如你此前看到的，对数组使用typeof操作符会返回object。然而大部分情况下，我们要检查数组是否真的是数组。

Array.isArray对数组返回true，对其他值则返回false：

```
Array.isArray(new Array) // true
Array.isArray([]) // true
Array.isArray(null) // false
Array.isArray(arguments) // false
```

数组方法

要遍历数组，可以使用forEach（类似jQuery的$.each）：

```
// 会打印出1、2、3
[1, 2, 3].forEach(function (v) {
  console.log(v);
});
```

要过滤数组元素，可以使用filter（类似jQuery的$.grep）：

```
[1, 2, 3].forEach(function (v) {
  return v < 3;
}); // 会返回[1,2]
```

要改变数组中每个元素的值，可以使用map（类似jQuery的$.map）：

```
[5, 10, 15].map(function (v) {
  return v * 2;
}); // 会返回 [10, 20, 30]
```

V8还提供了一些不太常用的方法，如reduce、reduceRight以及lastIndexOf。

24

字符串方法

要移除字符串首末的空格，可以使用：

```
'  hello  '.trim(); // 'hello'
```

JSON

V8提供了JSON.stringify和JSON.parse方法来对JSON数据进行解码和编码。

JSON是一种编码标准，和JavaScript对象字面量很相近，它用于大部分的Web服务和API服务：

```
var obj = JSON.parse('{"a":"b"}')
obj.a == 'b'; // true
```

FUNCTION#BIND

.bind（类似jQuery的$.proxy）允许改变对this的引用：

```
function a () {
  this.hello == 'world'; // true
};

var b = a.bind({ hello: 'world' });
b();
```

FUNCTION#NAME

V8还支持非标准的函数属性名：

```
var a = function woot () {};
a.name == 'woot'; // true
```

该属性用于V8内部的堆栈追踪。当有错误抛出时，V8会显示一个堆栈追踪的信息，会告诉你是哪个函数调用导致了错误的发生：

```
> var woot = function () { throw new Error(); };
```

```
> woot()
Error
    at [object Context]:1:32
```

25 在上述例子中，V8无法为函数引用指派名字。然而，如果对函数进行了命名，v8就能在显示堆栈追踪信息时将名字显示出来：

```
> var woot = function buggy () { throw new Error(); };
> woot()
Error
    at buggy ([object Context]:1:34)
```

为函数命名有助于调试，因此，推荐始终对函数进行命名。

PROTO（继承）

__proto__使得定义继承链变得更加容易：

```
function Animal () { }
function Ferret () { }
Ferret.prototype.__proto__ = Animal.prototype;
```

这是非常有用的特性，可以免去如下的工作：

- 像上一章节介绍的，借助中间构造器。
- 借助OOP的工具类库。无须再引入第三方模块来进行基于原型继承的声明。

存取器

你可以通过调用方法来定义属性，访问属性就使用__defineGetter__、设置属性就使用__defineSetter__。

比如，为Date对象定义一个称为ago的属性，返回以自然语言描述的日期间隔。

很多时候，特别是在软件中，想要用自然语言来描述日期距离某个特定时间点的时间间隔。比如，“某件事情发生在三秒钟前”这种表达，远要比“某件事情发生在×年×月×日”这种表达更容易理解。

下面的例子，为所有的Date实例都添加了ago获取器，它会返回以自然语言描述的日期距离现在的时间间隔。简单地访问该属性就会调用事先定义好的函数，无须显式调用。

```
// 基于John Resig的prettyDate（遵循MIT协议）
Date.prototype.__defineGetter__('ago', function () {
  var diff = (((new Date()).getTime() - this.getTime()) / 1000)
    , day_diff = Math.floor(diff / 86400);
```

26

```
  return day_diff == 0 && (
      diff < 60 && "just now" ||
      diff < 120 && "1 minute ago" ||
```

```
        diff < 3600 && Math.floor( diff / 60 ) + " minutes ago" ||
        diff < 7200 && "1 hour ago" ||
        diff < 86400 && Math.floor( diff / 3600 ) + " hours ago") ||
        day_diff == 1 && "Yesterday" ||
        day_diff < 7 && day_diff + " days ago" ||
        Math.ceil( day_diff / 7 ) + " weeks ago";
});
```

然后，简单地访问ago属性即可。注意，访问属性实际上还会调用定义的函数，只是这个过程透明了而已：

```
var a = new Date('12/12/1990'); // my birth date
a.ago // 1071 weeks ago
```

小结

理解掌握本章的内容对了解语言本身的不足以及大多数糟糕的JavaScript运行环境，如老版本的浏览器，至关重要。

由于JavaScript多年来自身发展缓慢且多少有种被人忽略的感觉，许多开发者投入了大量的时间来开发相应的技术以书写出更高效、可维护的JavaScript代码，同时也总结出了JavaScript一些诡异的工作方式。

V8做了一件很酷的事情，它始终坚定不移地实现最新版本的ECMA标准。Node.js的核心团队也是如此，只要你安装的是最新版本的Node，你总能使用到最新版本的V8。它开启了服务器端开发的新篇章，我们可以使用它提供的更易理解且执行效率更高的API。

希望通过本章的学习，你已掌握了Node开发者常用的一些特性，这些特性是诸多未来JavaScript拥有的特性中的一部分。