你好，我是winter。

在前面的课程中，我已经讲解了JavaScript对象的一些基础知识。但是，我们所讲解的对象，只是特定的一部分，并不能涵盖全部的JavaScript对象。

比如说，我们不论怎样编写代码，都没法绕开Array，实现一个跟原生的数组行为一模一样的对象，这是由于原生数组的底层实现了一个自动随着下标变化的length属性。

并且，在浏览器环境中，我们也无法单纯依靠JavaScript代码实现div对象，只能靠document.createElement来创建。这也说明了JavaScript的对象机制并非简单的属性集合+原型。

我们日常工作中，接触到的主要API，几乎都是由今天所讲解的这些对象提供的。理解这些对象的性质，我们才能真正理解我们使用的API的一些特性。

JavaScript中的对象分类

我们可以把对象分成几类。

- 宿主对象（host Objects）：由JavaScript宿主环境提供的对象，它们的行为完全由宿主环境决定。
- 内置对象（Built-in Objects）：由JavaScript语言提供的对象。
    - 固有对象（Intrinsic Objects ）：由标准规定，随着JavaScript运行时创建而自动创建的对象实例。
    - 原生对象（Native Objects）：可以由用户通过Array、RegExp等内置构造器或者特殊语法创建的对象。
    - 普通对象（Ordinary Objects）：由{}语法、Object构造器或者class关键字定义类创建的对象，它能够被原型继承。

下面我会为你一一讲解普通对象之外的对象类型。

宿主对象

首先我们来看看宿主对象。

JavaScript宿主对象千奇百怪，但是前端最熟悉的无疑是浏览器环境中的宿主了。

在浏览器环境中，我们都知道全局对象是window，window上又有很多属性，如document。

实际上，这个全局对象window上的属性，一部分来自JavaScript语言，一部分来自浏览器环境。

JavaScript标准中规定了全局对象属性，W3C的各种标准中规定了Window对象的其它属性。

宿主对象也分为固有的和用户可创建的两种，比如document.createElement就可以创建一些DOM对象。

宿主也会提供一些构造器，比如我们可以使用new Image来创建img元素，这些我们会在浏览器的API部分详细讲解。

内置对象·固有对象

我们在前面说过，固有对象是由标准规定，随着JavaScript运行时创建而自动创建的对象实例。

固有对象在任何JavaScript代码执行前就已经被创建出来了，它们通常扮演者类似基础库的角色。我们前面提到的“类”其实就是固有对象的一种。

ECMA标准为我们提供了一份固有对象表，里面含有150+个固有对象。你可以通过这个链接查看。

但是遗憾的是，这个表格并不完整。所以在本篇的末尾，我设计了一个小实验（小实验：获取全部JavaScript固有对象），你可以自己尝试一下，数一数一共有多少个固有对象。

内置对象·原生对象

我们把JavaScript中，能够通过语言本身的构造器创建的对象称作原生对象。在JavaScript标准中，提供了30多个构造器。按照我的理解，按照不同应用场景，我把原生对象分成了以下几个种类。

基本类型	基础功能和数据结构	错误类型	二进制操作	带类型的数组
Boolean	Array	Error	ArrayBuffer	Float32Array
String	Date	EvalError	SharedArrayBuffer	Float64Array
Number	RegExp	RangeError	DataView	Int8Array
Symbol	Promise	ReferenceError		Int16Array
Object	Proxy	SyntaxError		Int32Array
	Map	TypeError		UInt8Array
	WeakMap	URIError		UInt16Array
	Set			UInt32Array
	WeakSet			UInt8ClampedArray
	Function			

通过这些构造器，我们可以用new运算创建新的对象，所以我们把这些对象称作原生对象。
几乎所有这些构造器的能力都是无法用纯JavaScript代码实现的，它们也无法用class/extend语法来继承。

这些构造器创建的对象多数使用了私有字段,例如：

- Error: [[ErrorData]]
- Boolean: [[BooleanData]]
- Number: [[NumberData]]
- Date: [[DateValue]]
- RegExp: [[RegExpMatcher]]
- Symbol: [[SymbolData]]
- Map: [[MapData]]

这些字段使得原型继承方法无法正常工作，所以，我们可以认为，所有这些原生对象都是为了特定能力或者性能，而设计出来的“特权对象”。

用对象来模拟函数与构造器：函数对象与构造器对象

我在前面介绍了对象的一般分类，在JavaScript中，还有一个看待对象的不同视角，这就是用对象来模拟函数和构造器。

事实上，JavaScript为这一类对象预留了私有字段机制，并规定了抽象的函数对象与构造器对象的概念。

函数对象的定义是：具有[[call]]私有字段的对象，构造器对象的定义是：具有私有字段[[construct]]的对象。

JavaScript用对象模拟函数的设计代替了一般编程语言中的函数，它们可以像其它语言的函数一样被调用、传参。任何宿主只要提供了“具有[[call]]私有字段的对象”，就可以被 JavaScript 函数调用语法支持。

> [[call]]私有字段必须是一个引擎中定义的函数，需要接受this值和调用参数，并且会产生域的切换，这些内容，我将会在属性访问和执行过程两个章节详细讲述。

我们可以这样说，任何对象只需要实现[[call]]，它就是一个函数对象，可以去作为函数被调用。而如果它能实现[[construct]]，它就是一个构造器对象，可以作为构造器被调用。

对于为JavaScript提供运行环境的程序员来说，只要字段符合，我们在上文中提到的宿主对象和内置对象（如Symbol函数）可以模拟函数和构造器。

当然了，用户用function关键字创建的函数必定同时是函数和构造器。不过，它们表现出来的行为效果却并不相同。

对于宿主和内置对象来说，它们实现[[call]]（作为函数被调用）和[[construct]]（作为构造器被调用）不总是一致的。比如内置对象 Date 在作为构造器调用时产生新的对象，作为函数时，则产生字符串，见以下代码：

```
console.log(new Date); // 1
console.log(Date())
```

而浏览器宿主环境中，提供的Image构造器，则根本不允许被作为函数调用。

```
console.log(new Image);
console.log(Image());//抛出错误
```

再比如基本类型（String、Number、Boolean），它们的构造器被当作函数调用，则产生类型转换的效果。

值得一提的是，在ES6之后 => 语法创建的函数仅仅是函数，它们无法被当作构造器使用，见以下代码：

```
new (a => 0) // error
```

对于用户使用 function 语法或者Function构造器创建的对象来说，[[call]]和[[construct]]行为总是相似的，它们执行同一段代码。

我们看一下示例。

```
function f(){
    return 1;
}
var v = f(); //把f作为函数调用
var o = new f(); //把f作为构造器调用
```

我们大致可以认为，它们[[construct]]的执行过程如下：

- 以 Object.prototype 为原型创建一个新对象；
- 以新对象为 this，执行函数的[[call]]；
- 如果[[call]]的返回值是对象，那么，返回这个对象，否则返回第一步创建的新对象。

这样的规则造成了个有趣的现象，如果我们的构造器返回了一个新的对象，那么new创建的新对象就变成了一个构造函数之外完全无法访问的对象，这一定程度上可以实现“私有”。

```
function cls(){
    this.a = 100;
    return {
        getValue:() => this.a
    }
}
var o = new cls;
o.getValue(); //100
//a在外面永远无法访问到
```

特殊行为的对象

除了上面介绍的对象之外，在固有对象和原生对象中，有一些对象的行为跟正常对象有很大区别。

它们常见的下标运算（就是使用中括号或者点来做属性访问）或者设置原型跟普通对象不同，这里我简单总结一下。

- Array：Array的length属性根据最大的下标自动发生变化。
- Object.prototype：作为所有正常对象的默认原型，不能再给它设置原型了。
- String：为了支持下标运算，String的正整数属性访问会去字符串里查找。
- Arguments：arguments的非负整数型下标属性跟对应的变量联动。
- 模块的namespace对象：特殊的地方非常多，跟一般对象完全不一样，尽量只用于import吧。
- 类型数组和数组缓冲区：跟内存块相关联，下标运算比较特殊。
- bind后的function：跟原来的函数相关联。

结语

在这篇文章中，我们介绍了一些不那么常规的对象，并且我还介绍了JavaScript中用对象来模拟函数和构造器的机制。

这是一些不那么有规律、不那么优雅的知识，而JavaScript正是通过这些对象，提供了很多基础的能力。

我们这次课程留一个挑战任务：不使用new运算符，尽可能找到获得对象的方法。

例子：

```
var o = {}
var o = function(){}
```

请把自己的答案留言给我，我们来比比看谁找到的多。

小实验：获取全部JavaScript固有对象

我们从JavaScript标准中可以找到全部的JavaScript对象定义。JavaScript语言规定了全局对象的属性。

三个值：

Infinity、NaN、undefined。

九个函数：

- eval
- isFinite
- isNaN
- parseFloat
- parseInt
- decodeURI
- decodeURIComponent
- encodeURI
- encodeURIComponent

一些构造器：
Array、Date、RegExp、Promise、Proxy、Map、WeakMap、Set、WeakSet、Function、Boolean、String、Number、Symbol、Object、Error、

EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError、ArrayBuffer、SharedArrayBuffer、DataView、Typed Array、Float32Array、Float64Array、Int8Array、Int16Array、Int32Array、UInt8Array、UInt16Array、UInt32Array、UInt8ClampedArray。

四个用于当作命名空间的对象：

- Atomics
- JSON
- Math
- Reflect

我们使用广度优先搜索，查找这些对象所有的属性和Getter/Setter，就可以获得JavaScript中所有的固有对象。

请你试着先不看我的代码，在自己的浏览器中计算出来JavaScript有多少固有对象。

```
var set = new Set();
var objects = [
    eval,
    isFinite,
    isNaN,
    parseFloat,
    parseInt,
    decodeURI,
    decodeURIComponent,
    encodeURI,
    encodeURIComponent,
    Array,
    Date,
    RegExp,
    Promise,
    Proxy,
    Map,
    WeakMap,
    Set,
    WeakSet,
    Function,
    Boolean,
    String,
    Number,
    Symbol,
    Object,
    Error,
    EvalError,
    RangeError,
    ReferenceError,
    SyntaxError,
    TypeError,
    URIError,
    ArrayBuffer,
    SharedArrayBuffer,
    DataView,
    Float32Array,
    Float64Array,
    Int8Array,
    Int16Array,
    Int32Array,
    Uint8Array,
    Uint16Array,
    Uint32Array,
    Uint8ClampedArray,
    Atomics,
    JSON,
    Math,
    Reflect];
objects.forEach(o => set.add(o));

for(var i = 0; i < objects.length; i++) {
    var o = objects[i]
    for(var p of Object.getOwnPropertyNames(o)) {
        var d = Object.getOwnPropertyDescriptor(o, p)
        if( (d.value !== null && typeof d.value === "object") || (typeof d.value === "function"))
            if(!set.has(d.value))
                set.add(d.value), objects.push(d.value);
        if( d.get )
            if(!set.has(d.get))
                set.add(d.get), objects.push(d.get);
        if( d.set )
            if(!set.has(d.set))
                set.add(d.set), objects.push(d.set);
    }
}
```

你好，我是winter。

在前面的课程中，我已经讲解了JavaScript对象的一些基础知识。但是，我们所讲解的对象，只是特定的一部分，并不能涵盖全部的JavaScript对象。

比如说，我们不论怎样编写代码，都没法绕开Array，实现一个跟原生的数组行为一模一样的对象，这是由于原生数组的底层实现了一个自动

随着下标变化的length属性。

并且，在浏览器环境中，我们也无法单纯依靠JavaScript代码实现div对象，只能靠document.createElement来创建。这也说明了JavaScript的对象机制并非简单的属性集合+原型。

我们日常工作中，接触到的主要API，几乎都是由今天所讲解的这些对象提供的。理解这些对象的性质，我们才能真正理解我们使用的API的一些特性。

JavaScript中的对象分类

我们可以把对象分成几类。

- 宿主对象（host Objects）：由JavaScript宿主环境提供的对象，它们的行为完全由宿主环境决定。
- 内置对象（Built-in Objects）：由JavaScript语言提供的对象。
    - 固有对象（Intrinsic Objects ）：由标准规定，随着JavaScript运行时创建而自动创建的对象实例。
    - 原生对象（Native Objects）：可以由用户通过Array、RegExp等内置构造器或者特殊语法创建的对象。
    - 普通对象（Ordinary Objects）：由{}语法、Object构造器或者class关键字定义类创建的对象，它能够被原型继承。

下面我会为你一一讲解普通对象之外的对象类型。

宿主对象

首先我们来看看宿主对象。

JavaScript宿主对象千奇百怪，但是前端最熟悉的无疑是浏览器环境中的宿主了。

在浏览器环境中，我们都知道全局对象是window，window上又有很多属性，如document。

实际上，这个全局对象window上的属性，一部分来自JavaScript语言，一部分来自浏览器环境。

JavaScript标准中规定了全局对象属性，W3C的各种标准中规定了Window对象的其它属性。

宿主对象也分为固有的和用户可创建的两种，比如document.createElement就可以创建一些DOM对象。

宿主也会提供一些构造器，比如我们可以使用new Image来创建img元素，这些我们会在浏览器的API部分详细讲解。

内置对象·固有对象

我们在前面说过，固有对象是由标准规定，随着JavaScript运行时创建而自动创建的对象实例。

固有对象在任何JavaScript代码执行前就已经被创建出来了，它们通常扮演者类似基础库的角色。我们前面提到的“类”其实就是固有对象的一种。

ECMA标准为我们提供了一份固有对象表，里面含有150+个固有对象。你可以通过这个链接查看。

但是遗憾的是，这个表格并不完整。所以在本篇的末尾，我设计了一个小实验（小实验：获取全部JavaScript固有对象），你可以自己尝试一下，数一数一共有多少个固有对象。

内置对象·原生对象

我们把JavaScript中，能够通过语言本身的构造器创建的对象称作原生对象。在JavaScript标准中，提供了30多个构造器。按照我的理解，按照不同应用场景，我把原生对象分成了以下几个种类。

基本类型	基础功能和数据结构	错误类型	二进制操作	带类型的数组
Boolean	Array	Error	ArrayBuffer	Float32Array
String	Date	EvalError	SharedArrayBuffer	Float64Array
Number	RegExp	RangeError	DataView	Int8Array
Symbol	Promise	ReferenceError		Int16Array
Object	Proxy	SyntaxError		Int32Array
	Map	TypeError		Uint8Array
	WeakMap	URIError		Uint16Array
	Set			Uint32Array
	WeakSet			Uint8ClampedArray
	Function			

通过这些构造器，我们可以用new运算创建新的对象，所以我们把这些对象称作原生对象。
几乎所有这些构造器的能力都是无法用纯JavaScript代码实现的，它们也无法用class/extend语法来继承。

这些构造器创建的对象多数使用了私有字段,例如：

- Error: [[ErrorData]]
- Boolean: [[BooleanData]]
- Number: [[NumberData]]
- Date: [[DateValue]]
- RegExp: [[RegExpMatcher]]
- Symbol: [[SymbolData]]
- Map: [[MapData]]

这些字段使得原型继承方法无法正常工作，所以，我们可以认为，所有这些原生对象都是为了特定能力或者性能，而设计出来的“特权对象”。

用对象来模拟函数与构造器：函数对象与构造器对象

我在前面介绍了对象的一般分类，在JavaScript中，还有一个看待对象的不同视角，这就是用对象来模拟函数和构造器。

事实上，JavaScript为这一类对象预留了私有字段机制，并规定了抽象的函数对象与构造器对象的概念。

函数对象的定义是：具有[[call]]私有字段的对象，构造器对象的定义是：具有私有字段[[construct]]的对象。

JavaScript用对象模拟函数的设计代替了一般编程语言中的函数，它们可以像其它语言的函数一样被调用、传参。任何宿主只要提供了“具有[[call]]私有字段的对象”，就可以被 JavaScript 函数调用语法支持。

> [[call]]私有字段必须是一个引擎中定义的函数，需要接受this值和调用参数，并且会产生域的切换，这些内容，我将会在属性访问和执行过程两个章节详细讲述。

我们可以这样说，任何对象只需要实现[[call]]，它就是一个函数对象，可以去作为函数被调用。而如果它能实现[[construct]]，它就是一个构造器对象，可以作为构造器被调用。

对于为JavaScript提供运行环境的程序员来说，只要字段符合，我们在上文中提到的宿主对象和内置对象（如Symbol函数）可以模拟函数和构造器。

当然了，用户用function关键字创建的函数必定同时是函数和构造器。不过，它们表现出来的行为效果却并不相同。

对于宿主和内置对象来说，它们实现[[call]]（作为函数被调用）和[[construct]]（作为构造器被调用）不总是一致的。比如内置对象 Date 在作为构造器调用时产生新的对象，作为函数时，则产生字符串，见以下代码：

```
    console.log(new Date); // 1
    console.log(Date())
```

而浏览器宿主环境中，提供的Image构造器，则根本不允许被作为函数调用。

```
console.log(new Image);
console.log(Image());//抛出错误
```

再比如基本类型（String、Number、Boolean），它们的构造器被当作函数调用，则产生类型转换的效果。

值得一提的是，在ES6之后 => 语法创建的函数仅仅是函数，它们无法被当作构造器使用，见以下代码：

```
    new (a => 0) // error
```

对于用户使用 function 语法或者Function构造器创建的对象来说，[[call]]和[[construct]]行为总是相似的，它们执行同一段代码。

我们看一下示例。

```
function f(){
    return 1;
}
var v = f(); //把f作为函数调用
var o = new f(); //把f作为构造器调用
```

我们大致可以认为，它们[[construct]]的执行过程如下：

- 以 Object.prototype 为原型创建一个新对象；
- 以新对象为 this，执行函数的[[call]]；
- 如果[[call]]的返回值是对象，那么，返回这个对象，否则返回第一步创建的新对象。

这样的规则造成了个有趣的现象，如果我们的构造器返回了一个新的对象，那么new创建的新对象就变成了一个构造函数之外完全无法访问的对象，这一定程度上可以实现“私有”。

```
function cls(){
    this.a = 100;
    return {
        getValue:() => this.a
    }
```

```
}
var o = new cls;
o.getValue(); //100
//a在外面永远无法访问到
```

特殊行为的对象

除了上面介绍的对象之外，在固有对象和原生对象中，有一些对象的行为跟正常对象有很大区别。

它们常见的下标运算（就是使用中括号或者点来做属性访问）或者设置原型跟普通对象不同，这里我简单总结一下。

- Array：Array的length属性根据最大的下标自动发生变化。
- Object.prototype：作为所有正常对象的默认原型，不能再给它设置原型了。
- String：为了支持下标运算，String的正整数属性访问会去字符串里查找。
- Arguments：arguments的非负整数型下标属性跟对应的变量联动。
- 模块的namespace对象：特殊的地方非常多，跟一般对象完全不一样，尽量只用于import吧。
- 类型数组和数组缓冲区：跟内存块相关联，下标运算比较特殊。
- bind后的function：跟原来的函数相关联。

结语

在这篇文章中，我们介绍了一些不那么常规的对象，并且我还介绍了JavaScript中用对象来模拟函数和构造器的机制。

这是一些不那么有规律、不那么优雅的知识，而JavaScript正是通过这些对象，提供了很多基础的能力。

我们这次课程留一个挑战任务：不使用new运算符，尽可能找到获得对象的方法。

例子：

```
var o = {}
var o = function(){}
```

请把自己的答案留言给我，我们来比比看谁找到的多。

小实验：获取全部JavaScript固有对象

我们从JavaScript标准中可以找到全部的JavaScript对象定义。JavaScript语言规定了全局对象的属性。

三个值：

Infinity、NaN、undefined。

九个函数：

- eval
- isFinite
- isNaN
- parseFloat
- parseInt
- decodeURI
- decodeURIComponent
- encodeURI
- encodeURIComponent

一些构造器：
Array、Date、RegExp、Promise、Proxy、Map、WeakMap、Set、WeakSet、Function、Boolean、String、Number、Symbol、Object、Error、EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError、ArrayBuffer、SharedArrayBuffer、DataView、Typed Array、Float32Array、Float64Array、Int8Array、Int16Array、Int32Array、UInt8Array、UInt16Array、UInt32Array、UInt8ClampedArray。

四个用于当作命名空间的对象：

- Atomics
- JSON
- Math
- Reflect

我们使用广度优先搜索，查找这些对象所有的属性和Getter/Setter，就可以获得JavaScript中所有的固有对象。

请你试着先不看我的代码，在自己的浏览器中计算出来JavaScript有多少固有对象。

```
var set = new Set();
var objects = [
    eval,
    isFinite,
    isNaN,
    parseFloat,
    parseInt,
    decodeURI,
```

```
    decodeURIComponent,
    encodeURI,
    encodeURIComponent,
    Array,
    Date,
    RegExp,
    Promise,
    Proxy,
    Map,
    WeakMap,
    Set,
    WeakSet,
    Function,
    Boolean,
    String,
    Number,
    Symbol,
    Object,
    Error,
    EvalError,
    RangeError,
    ReferenceError,
    SyntaxError,
    TypeError,
    URIError,
    ArrayBuffer,
    SharedArrayBuffer,
    DataView,
    Float32Array,
    Float64Array,
    Int8Array,
    Int16Array,
    Int32Array,
    Uint8Array,
    Uint16Array,
    Uint32Array,
    Uint8ClampedArray,
    Atomics,
    JSON,
    Math,
    Reflect];
objects.forEach(o => set.add(o));

for(var i = 0; i < objects.length; i++) {
    var o = objects[i]
    for(var p of Object.getOwnPropertyNames(o)) {
        var d = Object.getOwnPropertyDescriptor(o, p)
        if( (d.value !== null && typeof d.value === "object") || (typeof d.value === "function"))
            if(!set.has(d.value))
                set.add(d.value), objects.push(d.value);
        if( d.get )
            if(!set.has(d.get))
                set.add(d.get), objects.push(d.get);
        if( d.set )
            if(!set.has(d.set))
                set.add(d.set), objects.push(d.set);
    }
}
```

你好，我是winter。

在前面的课程中，我已经讲解了JavaScript对象的一些基础知识。但是，我们所讲解的对象，只是特定的一部分，并不能涵盖全部的JavaScript对象。

比如说，我们不论怎样编写代码，都没法绕开Array，实现一个跟原生的数组行为一模一样的对象，这是由于原生数组的底层实现了一个自动随着下标变化的length属性。

并且，在浏览器环境中，我们也无法单纯依靠JavaScript代码实现div对象，只能靠document.createElement来创建。这也说明了JavaScript的对象机制并非简单的属性集合+原型。

我们日常工作中，接触到的主要API，几乎都是由今天所讲解的这些对象提供的。理解这些对象的性质，我们才能真正理解我们使用的API的一些特性。

JavaScript中的对象分类

我们可以把对象分成几类。

- 宿主对象（host Objects）：由JavaScript宿主环境提供的对象，它们的行为完全由宿主环境决定。
- 内置对象（Built-in Objects）：由JavaScript语言提供的对象。
    - 固有对象（Intrinsic Objects ）：由标准规定，随着JavaScript运行时创建而自动创建的对象实例。
    - 原生对象（Native Objects）：可以由用户通过Array、RegExp等内置构造器或者特殊语法创建的对象。
    - 普通对象（Ordinary Objects）：由{}语法、Object构造器或者class关键字定义类创建的对象，它能够被原型继承。

下面我会为你一一讲解普通对象之外的对象类型。

宿主对象

首先我们来看看宿主对象。

JavaScript宿主对象千奇百怪，但是前端最熟悉的无疑是浏览器环境中的宿主了。

在浏览器环境中，我们都知道全局对象是window，window上又有很多属性，如document。

实际上，这个全局对象window上的属性，一部分来自JavaScript语言，一部分来自浏览器环境。

JavaScript标准中规定了全局对象属性，W3C的各种标准中规定了Window对象的其它属性。

宿主对象也分为固有的和用户可创建的两种，比如document.createElement就可以创建一些DOM对象。

宿主也会提供一些构造器，比如我们可以使用new Image来创建img元素，这些我们会在浏览器的API部分详细讲解。

内置对象·固有对象

我们在前面说过，固有对象是由标准规定，随着JavaScript运行时创建而自动创建的对象实例。

固有对象在任何JavaScript代码执行前就已经被创建出来了，它们通常扮演者类似基础库的角色。我们前面提到的“类”其实就是固有对象的一种。

ECMA标准为我们提供了一份固有对象表，里面含有150+个固有对象。你可以通过这个链接查看。

但是遗憾的是，这个表格并不完整。所以在本篇的末尾，我设计了一个小实验（小实验：获取全部JavaScript固有对象），你可以自己尝试一下，数一数一共有多少个固有对象。

内置对象·原生对象

我们把JavaScript中，能够通过语言本身的构造器创建的对象称作原生对象。在JavaScript标准中，提供了30多个构造器。按照我的理解，按照不同应用场景，我把原生对象分成了以下几个种类。

基本类型	基础功能和数据结构	错误类型	二进制操作	带类型的数组
Boolean	Array	Error	ArrayBuffer	Float32Array
String	Date	EvalError	SharedArrayBuffer	Float64Array
Number	RegExp	RangeError	DataView	Int8Array
Symbol	Promise	ReferenceError		Int16Array
Object	Proxy	SyntaxError		Int32Array
	Map	TypeError		UInt8Array
	WeakMap	URIError		UInt16Array
	Set			UInt32Array
	WeakSet			UInt8ClampedArray
	Function			

通过这些构造器，我们可以用new运算创建新的对象，所以我们把这些对象称作原生对象。
几乎所有这些构造器的能力都是无法用纯JavaScript代码实现的，它们也无法用class/extend语法来继承。

这些构造器创建的对象多数使用了私有字段,例如：

- Error: [[ErrorData]]
- Boolean: [[BooleanData]]
- Number: [[NumberData]]
- Date: [[DateValue]]
- RegExp: [[RegExpMatcher]]
- Symbol: [[SymbolData]]
- Map: [[MapData]]

这些字段使得原型继承方法无法正常工作，所以，我们可以认为，所有这些原生对象都是为了特定能力或者性能，而设计出来的“特权对象”。

用对象来模拟函数与构造器：函数对象与构造器对象

我在前面介绍了对象的一般分类，在JavaScript中，还有一个看待对象的不同视角，这就是用对象来模拟函数和构造器。

事实上，JavaScript为这一类对象预留了私有字段机制，并规定了抽象的函数对象与构造器对象的概念。

函数对象的定义是：具有[[call]]私有字段的对象，构造器对象的定义是：具有私有字段[[construct]]的对象。

JavaScript用对象模拟函数的设计代替了一般编程语言中的函数，它们可以像其它语言的函数一样被调用、传参。任何宿主只要提供了“具有[[call]]私有字段的对象”，就可以被 JavaScript 函数调用语法支持。

> [[call]]私有字段必须是一个引擎中定义的函数，需要接受this值和调用参数，并且会产生域的切换，这些内容，我将会在属性访问和执行过程两个章节详细讲述。

我们可以这样说，任何对象只需要实现[[call]]，它就是一个函数对象，可以去作为函数被调用。而如果它能实现[[construct]]，它就是一个构造器对象，可以作为构造器被调用。

对于为JavaScript提供运行环境的程序员来说，只要字段符合，我们在上文中提到的宿主对象和内置对象（如Symbol函数）可以模拟函数和构造器。

当然了，用户用function关键字创建的函数必定同时是函数和构造器。不过，它们表现出来的行为效果却并不相同。

对于宿主和内置对象来说，它们实现[[call]]（作为函数被调用）和[[construct]]（作为构造器被调用）不总是一致的。比如内置对象 Date 在作为构造器调用时产生新的对象，作为函数时，则产生字符串，见以下代码：

```
    console.log(new Date); // 1
    console.log(Date())
```

而浏览器宿主环境中，提供的Image构造器，则根本不允许被作为函数调用。

```
console.log(new Image);
console.log(Image());//抛出错误
```

再比如基本类型（String、Number、Boolean），它们的构造器被当作函数调用，则产生类型转换的效果。

值得一提的是，在ES6之后 => 语法创建的函数仅仅是函数，它们无法被当作构造器使用，见以下代码：

```
    new (a => 0) // error
```

对于用户使用 function 语法或者Function构造器创建的对象来说，[[call]]和[[construct]]行为总是相似的，它们执行同一段代码。

我们看一下示例。

```
function f(){
    return 1;
}
var v = f(); //把f作为函数调用
var o = new f(); //把f作为构造器调用
```

我们大致可以认为，它们[[construct]]的执行过程如下：

- 以 Object.prototype 为原型创建一个新对象；
- 以新对象为 this，执行函数的[[call]]；
- 如果[[call]]的返回值是对象，那么，返回这个对象，否则返回第一步创建的新对象。

这样的规则造成了个有趣的现象，如果我们的构造器返回了一个新的对象，那么new创建的新对象就变成了一个构造函数之外完全无法访问的对象，这一定程度上可以实现“私有”。

```
function cls(){
    this.a = 100;
    return {
        getValue:() => this.a
    }
}
var o = new cls;
o.getValue(); //100
//a在外面永远无法访问到
```

特殊行为的对象

除了上面介绍的对象之外，在固有对象和原生对象中，有一些对象的行为跟正常对象有很大区别。

它们常见的下标运算（就是使用中括号或者点来做属性访问）或者设置原型跟普通对象不同，这里我简单总结一下。

- Array：Array的length属性根据最大的下标自动发生变化。
- Object.prototype：作为所有正常对象的默认原型，不能再给它设置原型了。
- String：为了支持下标运算，String的正整数属性访问会去字符串里查找。
- Arguments：arguments的非负整数型下标属性跟对应的变量联动。
- 模块的namespace对象：特殊的地方非常多，跟一般对象完全不一样，尽量只用于import吧。
- 类型数组和数组缓冲区：跟内存块相关联，下标运算比较特殊。
- bind后的function：跟原来的函数相关联。

结语

在这篇文章中，我们介绍了一些不那么常规的对象，并且我还介绍了JavaScript中用对象来模拟函数和构造器的机制。

这是一些不那么有规律、不那么优雅的知识，而JavaScript正是通过这些对象，提供了很多基础的能力。

我们这次课程留一个挑战任务：不使用new运算符，尽可能找到获得对象的方法。

例子：

```
var o = {}
var o = function(){}
```

请把自己的答案留言给我，我们来比比看谁找到的多。

小实验：获取全部JavaScript固有对象

我们从JavaScript标准中可以找到全部的JavaScript对象定义。JavaScript语言规定了全局对象的属性。

三个值：

Infinity、NaN、undefined。

九个函数：

- eval
- isFinite
- isNaN
- parseFloat
- parseInt
- decodeURI
- decodeURIComponent
- encodeURI
- encodeURIComponent

一些构造器：
Array、Date、RegExp、Promise、Proxy、Map、WeakMap、Set、WeakSet、Function、Boolean、String、Number、Symbol、Object、Error、EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError、ArrayBuffer、SharedArrayBuffer、DataView、Typed Array、Float32Array、Float64Array、Int8Array、Int16Array、Int32Array、UInt8Array、UInt16Array、UInt32Array、UInt8ClampedArray。

四个用于当作命名空间的对象：

- Atomics
- JSON
- Math
- Reflect

我们使用广度优先搜索，查找这些对象所有的属性和Getter/Setter，就可以获得JavaScript中所有的固有对象。

请你试着先不看我的代码，在自己的浏览器中计算出来JavaScript有多少固有对象。

```
var set = new Set();
var objects = [
    eval,
    isFinite,
    isNaN,
    parseFloat,
    parseInt,
    decodeURI,
    decodeURIComponent,
    encodeURI,
    encodeURIComponent,
    Array,
    Date,
    RegExp,
    Promise,
    Proxy,
    Map,
    WeakMap,
    Set,
    WeakSet,
    Function,
    Boolean,
    String,
    Number,
    Symbol,
    Object,
    Error,
    EvalError,
    RangeError,
    ReferenceError,
    SyntaxError,
    TypeError,
    URIError,
```

```
    ArrayBuffer,
    SharedArrayBuffer,
    DataView,
    Float32Array,
    Float64Array,
    Int8Array,
    Int16Array,
    Int32Array,
    Uint8Array,
    Uint16Array,
    Uint32Array,
    Uint8ClampedArray,
    Atomics,
    JSON,
    Math,
    Reflect];
objects.forEach(o => set.add(o));

for(var i = 0; i < objects.length; i++) {
    var o = objects[i]
    for(var p of Object.getOwnPropertyNames(o)) {
        var d = Object.getOwnPropertyDescriptor(o, p)
        if( (d.value !== null && typeof d.value === "object") || (typeof d.value === "function"))
            if(!set.has(d.value))
                set.add(d.value), objects.push(d.value);
        if( d.get )
            if(!set.has(d.get))
                set.add(d.get), objects.push(d.get);
        if( d.set )
            if(!set.has(d.set))
                set.add(d.set), objects.push(d.set);
    }
}
```

你好，我是winter。

在前面的课程中，我已经讲解了JavaScript对象的一些基础知识。但是，我们所讲解的对象，只是特定的一部分，并不能涵盖全部的JavaScript对象。

比如说，我们不论怎样编写代码，都没法绕开Array，实现一个跟原生的数组行为一模一样的对象，这是由于原生数组的底层实现了一个自动随着下标变化的length属性。

并且，在浏览器环境中，我们也无法单纯依靠JavaScript代码实现div对象，只能靠document.createElement来创建。这也说明了JavaScript的对象机制并非简单的属性集合+原型。

我们日常工作中，接触到的主要API，几乎都是由今天所讲解的这些对象提供的。理解这些对象的性质，我们才能真正理解我们使用的API的一些特性。

JavaScript中的对象分类

我们可以把对象分成几类。

- 宿主对象（host Objects）：由JavaScript宿主环境提供的对象，它们的行为完全由宿主环境决定。
- 内置对象（Built-in Objects）：由JavaScript语言提供的对象。
    - 固有对象（Intrinsic Objects ）：由标准规定，随着JavaScript运行时创建而自动创建的对象实例。
    - 原生对象（Native Objects）：可以由用户通过Array、RegExp等内置构造器或者特殊语法创建的对象。
    - 普通对象（Ordinary Objects）：由{}语法、Object构造器或者class关键字定义类创建的对象，它能够被原型继承。

下面我会为你一一讲解普通对象之外的对象类型。

宿主对象

首先我们来看看宿主对象。

JavaScript宿主对象千奇百怪，但是前端最熟悉的无疑是浏览器环境中的宿主了。

在浏览器环境中，我们都知道全局对象是window，window上又有很多属性，如document。

实际上，这个全局对象window上的属性，一部分来自JavaScript语言，一部分来自浏览器环境。

JavaScript标准中规定了全局对象属性，W3C的各种标准中规定了Window对象的其它属性。

宿主对象也分为固有的和用户可创建的两种，比如document.createElement就可以创建一些DOM对象。

宿主也会提供一些构造器，比如我们可以使用new Image来创建img元素，这些我们会在浏览器的API部分详细讲解。

内置对象·固有对象

我们在前面说过，固有对象是由标准规定，随着JavaScript运行时创建而自动创建的对象实例。

固有对象在任何JavaScript代码执行前就已经被创建出来了，它们通常扮演者类似基础库的角色。我们前面提到的“类”其实就是固有对象的一种。

ECMA标准为我们提供了一份固有对象表，里面含有150+个固有对象。你可以通过这个链接查看。

但是遗憾的是，这个表格并不完整。所以在本篇的末尾，我设计了一个小实验（小实验：获取全部JavaScript固有对象），你可以自己尝试一下，数一数一共有多少个固有对象。

内置对象·原生对象

我们把JavaScript中，能够通过语言本身的构造器创建的对象称作原生对象。在JavaScript标准中，提供了30多个构造器。按照我的理解，按照不同应用场景，我把原生对象分成了以下几个种类。

基本类型	基础功能和数据结构	错误类型	二进制操作	带类型的数组
Boolean	Array	Error	ArrayBuffer	Float32Array
String	Date	EvalError	SharedArrayBuffer	Float64Array
Number	RegExp	RangeError	DataView	Int8Array
Symbol	Promise	ReferenceError		Int16Array
Object	Proxy	SyntaxError		Int32Array
	Map	TypeError		Uint8Array
	WeakMap	URIError		Uint16Array
	Set			Uint32Array
	WeakSet			Uint8ClampedArray
	Function			

通过这些构造器，我们可以用new运算创建新的对象，所以我们把这些对象称作原生对象。
几乎所有这些构造器的能力都是无法用纯JavaScript代码实现的，它们也无法用class/extend语法来继承。

这些构造器创建的对象多数使用了私有字段,例如：

- Error: [[ErrorData]]
- Boolean: [[BooleanData]]
- Number: [[NumberData]]
- Date: [[DateValue]]
- RegExp: [[RegExpMatcher]]
- Symbol: [[SymbolData]]
- Map: [[MapData]]

这些字段使得原型继承方法无法正常工作，所以，我们可以认为，所有这些原生对象都是为了特定能力或者性能，而设计出来的“特权对象”。

用对象来模拟函数与构造器：函数对象与构造器对象

我在前面介绍了对象的一般分类，在JavaScript中，还有一个看待对象的不同视角，这就是用对象来模拟函数和构造器。

事实上，JavaScript为这一类对象预留了私有字段机制，并规定了抽象的函数对象与构造器对象的概念。

函数对象的定义是：具有[[call]]私有字段的对象，构造器对象的定义是：具有私有字段[[construct]]的对象。

JavaScript用对象模拟函数的设计代替了一般编程语言中的函数，它们可以像其它语言的函数一样被调用、传参。任何宿主只要提供了“具有[[call]]私有字段的对象”，就可以被 JavaScript 函数调用语法支持。

> [[call]]私有字段必须是一个引擎中定义的函数，需要接受this值和调用参数，并且会产生域的切换，这些内容，我将会在属性访问和执行过程两个章节详细讲述。

我们可以这样说，任何对象只需要实现[[call]]，它就是一个函数对象，可以去作为函数被调用。而如果它能实现[[construct]]，它就是一个构造器对象，可以作为构造器被调用。

对于为JavaScript提供运行环境的程序员来说，只要字段符合，我们在上文中提到的宿主对象和内置对象（如Symbol函数）可以模拟函数和构造器。

当然了，用户用function关键字创建的函数必定同时是函数和构造器。不过，它们表现出来的行为效果却并不相同。

对于宿主和内置对象来说，它们实现[[call]]（作为函数被调用）和[[construct]]（作为构造器被调用）不总是一致的。比如内置对象 Date 在作为构造器调用时产生新的对象，作为函数时，则产生字符串，见以下代码：

```
console.log(new Date); // 1
console.log(Date())
```

而浏览器宿主环境中，提供的Image构造器，则根本不允许被作为函数调用。

```
console.log(new Image);
console.log(Image());//抛出错误
```

再比如基本类型（String、Number、Boolean），它们的构造器被当作函数调用，则产生类型转换的效果。

值得一提的是，在ES6之后 => 语法创建的函数仅仅是函数，它们无法被当作构造器使用，见以下代码：

```
    new (a => 0) // error
```

对于用户使用 function 语法或者Function构造器创建的对象来说，[[call]]和[[construct]]行为总是相似的，它们执行同一段代码。

我们看一下示例。

```
function f(){
    return 1;
}
var v = f(); //把f作为函数调用
var o = new f(); //把f作为构造器调用
```

我们大致可以认为，它们[[construct]]的执行过程如下：

- 以 Object.prototype 为原型创建一个新对象；
- 以新对象为 this，执行函数的[[call]]；
- 如果[[call]]的返回值是对象，那么，返回这个对象，否则返回第一步创建的新对象。

这样的规则造成了个有趣的现象，如果我们的构造器返回了一个新的对象，那么new创建的新对象就变成了一个构造函数之外完全无法访问的对象，这一定程度上可以实现“私有”。

```
function cls(){
    this.a = 100;
    return {
        getValue:() => this.a
    }
}
var o = new cls;
o.getValue(); //100
//a在外面永远无法访问到
```

特殊行为的对象

除了上面介绍的对象之外，在固有对象和原生对象中，有一些对象的行为跟正常对象有很大区别。

它们常见的下标运算（就是使用中括号或者点来做属性访问）或者设置原型跟普通对象不同，这里我简单总结一下。

- Array：Array的length属性根据最大的下标自动发生变化。
- Object.prototype：作为所有正常对象的默认原型，不能再给它设置原型了。
- String：为了支持下标运算，String的正整数属性访问会去字符串里查找。
- Arguments：arguments的非负整数型下标属性跟对应的变量联动。
- 模块的namespace对象：特殊的地方非常多，跟一般对象完全不一样，尽量只用于import吧。
- 类型数组和数组缓冲区：跟内存块相关联，下标运算比较特殊。
- bind后的function：跟原来的函数相关联。

结语

在这篇文章中，我们介绍了一些不那么常规的对象，并且我还介绍了JavaScript中用对象来模拟函数和构造器的机制。

这是一些不那么有规律、不那么优雅的知识，而JavaScript正是通过这些对象，提供了很多基础的能力。

我们这次课程留一个挑战任务：不使用new运算符，尽可能找到获得对象的方法。

例子：

```
var o = {}
var o = function(){}
```

请把自己的答案留言给我，我们来比比看谁找到的多。

小实验：获取全部JavaScript固有对象

我们从JavaScript标准中可以找到全部的JavaScript对象定义。JavaScript语言规定了全局对象的属性。

三个值：

Infinity、NaN、undefined。

九个函数：

- eval
- isFinite
- isNaN
- parseFloat
- parseInt
- decodeURI
- decodeURIComponent
- encodeURI
- encodeURIComponent

一些构造器：
Array、Date、RegExp、Promise、Proxy、Map、WeakMap、Set、WeakSet、Function、Boolean、String、Number、Symbol、Object、Error、EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError、ArrayBuffer、SharedArrayBuffer、DataView、Typed Array、Float32Array、Float64Array、Int8Array、Int16Array、Int32Array、UInt8Array、UInt16Array、UInt32Array、UInt8ClampedArray。

四个用于当作命名空间的对象：

- Atomics
- JSON
- Math
- Reflect

我们使用广度优先搜索，查找这些对象所有的属性和Getter/Setter，就可以获得JavaScript中所有的固有对象。

请你试着先不看我的代码，在自己的浏览器中计算出来JavaScript有多少固有对象。

```
var set = new Set();
var objects = [
    eval,
    isFinite,
    isNaN,
    parseFloat,
    parseInt,
    decodeURI,
    decodeURIComponent,
    encodeURI,
    encodeURIComponent,
    Array,
    Date,
    RegExp,
    Promise,
    Proxy,
    Map,
    WeakMap,
    Set,
    WeakSet,
    Function,
    Boolean,
    String,
    Number,
    Symbol,
    Object,
    Error,
    EvalError,
    RangeError,
    ReferenceError,
    SyntaxError,
    TypeError,
    URIError,
    ArrayBuffer,
    SharedArrayBuffer,
    DataView,
    Float32Array,
    Float64Array,
    Int8Array,
    Int16Array,
    Int32Array,
    Uint8Array,
    Uint16Array,
    Uint32Array,
    Uint8ClampedArray,
    Atomics,
    JSON,
    Math,
    Reflect];
objects.forEach(o => set.add(o));

for(var i = 0; i < objects.length; i++) {
    var o = objects[i]
    for(var p of Object.getOwnPropertyNames(o)) {
        var d = Object.getOwnPropertyDescriptor(o, p)
        if( (d.value !== null && typeof d.value === "object") || (typeof d.value === "function"))
            if(!set.has(d.value))
                set.add(d.value), objects.push(d.value);
        if( d.get )
```

```
            if(!set.has(d.get))
                set.add(d.get), objects.push(d.get);
        if( d.set )
            if(!set.has(d.set))
                set.add(d.set), objects.push(d.set);
    }
}
```

你好，我是winter。

在前面的课程中，我已经讲解了JavaScript对象的一些基础知识。但是，我们所讲解的对象，只是特定的一部分，并不能涵盖全部的JavaScript对象。

比如说，我们不论怎样编写代码，都没法绕开Array，实现一个跟原生的数组行为一模一样的对象，这是由于原生数组的底层实现了一个自动随着下标变化的length属性。

并且，在浏览器环境中，我们也无法单纯依靠JavaScript代码实现div对象，只能靠document.createElement来创建。这也说明了JavaScript的对象机制并非简单的属性集合+原型。

我们日常工作中，接触到的主要API，几乎都是由今天所讲解的这些对象提供的。理解这些对象的性质，我们才能真正理解我们使用的API的一些特性。

JavaScript中的对象分类

我们可以把对象分成几类。

- 宿主对象（host Objects）：由JavaScript宿主环境提供的对象，它们的行为完全由宿主环境决定。
- 内置对象（Built-in Objects）：由JavaScript语言提供的对象。
    - 固有对象（Intrinsic Objects ）：由标准规定，随着JavaScript运行时创建而自动创建的对象实例。
    - 原生对象（Native Objects）：可以由用户通过Array、RegExp等内置构造器或者特殊语法创建的对象。
    - 普通对象（Ordinary Objects）：由{}语法、Object构造器或者class关键字定义类创建的对象，它能够被原型继承。

下面我会为你一一讲解普通对象之外的对象类型。

宿主对象

首先我们来看看宿主对象。

JavaScript宿主对象千奇百怪，但是前端最熟悉的无疑是浏览器环境中的宿主了。

在浏览器环境中，我们都知道全局对象是window，window上又有很多属性，如document。

实际上，这个全局对象window上的属性，一部分来自JavaScript语言，一部分来自浏览器环境。

JavaScript标准中规定了全局对象属性，W3C的各种标准中规定了Window对象的其它属性。

宿主对象也分为固有的和用户可创建的两种，比如document.createElement就可以创建一些DOM对象。

宿主也会提供一些构造器，比如我们可以使用new Image来创建img元素，这些我们会在浏览器的API部分详细讲解。

内置对象·固有对象

我们在前面说过，固有对象是由标准规定，随着JavaScript运行时创建而自动创建的对象实例。

固有对象在任何JavaScript代码执行前就已经被创建出来了，它们通常扮演者类似基础库的角色。我们前面提到的“类”其实就是固有对象的一种。

ECMA标准为我们提供了一份固有对象表，里面含有150+个固有对象。你可以通过这个链接查看。

但是遗憾的是，这个表格并不完整。所以在本篇的末尾，我设计了一个小实验（小实验：获取全部JavaScript固有对象），你可以自己尝试一下，数一数一共有多少个固有对象。

内置对象·原生对象

我们把JavaScript中，能够通过语言本身的构造器创建的对象称作原生对象。在JavaScript标准中，提供了30多个构造器。按照我的理解，按照不同应用场景，我把原生对象分成了以下几个种类。

基本类型	基础功能和数据结构	错误类型	二进制操作	带类型的数组
Boolean	Array	Error	ArrayBuffer	Float32Array
String	Date	EvalError	SharedArrayBuffer	Float64Array
Number	RegExp	RangeError	DataView	Int8Array
Symbol	Promise	ReferenceError		Int16Array
Object	Proxy	SyntaxError		Int32Array
	Map	TypeError		UInt8Array
	WeakMap	URIError		UInt16Array
	Set			UInt32Array
	WeakSet			UInt8ClampedArray
	Function			

通过这些构造器，我们可以用new运算创建新的对象，所以我们把这些对象称作原生对象。
几乎所有这些构造器的能力都是无法用纯JavaScript代码实现的，它们也无法用class/extend语法来继承。

这些构造器创建的对象多数使用了私有字段,例如：

- Error: [[ErrorData]]
- Boolean: [[BooleanData]]
- Number: [[NumberData]]
- Date: [[DateValue]]
- RegExp: [[RegExpMatcher]]
- Symbol: [[SymbolData]]
- Map: [[MapData]]

这些字段使得原型继承方法无法正常工作，所以，我们可以认为，所有这些原生对象都是为了特定能力或者性能，而设计出来的“特权对象”。

用对象来模拟函数与构造器：函数对象与构造器对象

我在前面介绍了对象的一般分类，在JavaScript中，还有一个看待对象的不同视角，这就是用对象来模拟函数和构造器。

事实上，JavaScript为这一类对象预留了私有字段机制，并规定了抽象的函数对象与构造器对象的概念。

函数对象的定义是：具有[[call]]私有字段的对象，构造器对象的定义是：具有私有字段[[construct]]的对象。

JavaScript用对象模拟函数的设计代替了一般编程语言中的函数，它们可以像其它语言的函数一样被调用、传参。任何宿主只要提供了“具有[[call]]私有字段的对象”，就可以被 JavaScript 函数调用语法支持。

> [[call]]私有字段必须是一个引擎中定义的函数，需要接受this值和调用参数，并且会产生域的切换，这些内容，我将会在属性访问和执行过程两个章节详细讲述。

我们可以这样说，任何对象只需要实现[[call]]，它就是一个函数对象，可以去作为函数被调用。而如果它能实现[[construct]]，它就是一个构造器对象，可以作为构造器被调用。

对于为JavaScript提供运行环境的程序员来说，只要字段符合，我们在上文中提到的宿主对象和内置对象（如Symbol函数）可以模拟函数和构造器。

当然了，用户用function关键字创建的函数必定同时是函数和构造器。不过，它们表现出来的行为效果却并不相同。

对于宿主和内置对象来说，它们实现[[call]]（作为函数被调用）和[[construct]]（作为构造器被调用）不总是一致的。比如内置对象 Date 在作为构造器调用时产生新的对象，作为函数时，则产生字符串，见以下代码：

```
console.log(new Date); // 1
console.log(Date())
```

而浏览器宿主环境中，提供的Image构造器，则根本不允许被作为函数调用。

```
console.log(new Image);
console.log(Image());//抛出错误
```

再比如基本类型（String、Number、Boolean），它们的构造器被当作函数调用，则产生类型转换的效果。

值得一提的是，在ES6之后 => 语法创建的函数仅仅是函数，它们无法被当作构造器使用，见以下代码：

```
new (a => 0) // error
```

对于用户使用 function 语法或者Function构造器创建的对象来说，[[call]]和[[construct]]行为总是相似的，它们执行同一段代码。

我们看一下示例。

```
function f(){
    return 1;
}
var v = f(); //把f作为函数调用
var o = new f(); //把f作为构造器调用
```

我们大致可以认为，它们[[construct]]的执行过程如下：

- 以 Object.prototype 为原型创建一个新对象；
- 以新对象为 this，执行函数的[[call]]；
- 如果[[call]]的返回值是对象，那么，返回这个对象，否则返回第一步创建的新对象。

这样的规则造成了个有趣的现象，如果我们的构造器返回了一个新的对象，那么new创建的新对象就变成了一个构造函数之外完全无法访问的对象，这一定程度上可以实现“私有”。

```
function cls(){
    this.a = 100;
    return {
        getValue:() => this.a
    }
}
var o = new cls;
o.getValue(); //100
//a在外面永远无法访问到
```

特殊行为的对象

除了上面介绍的对象之外，在固有对象和原生对象中，有一些对象的行为跟正常对象有很大区别。

它们常见的下标运算（就是使用中括号或者点来做属性访问）或者设置原型跟普通对象不同，这里我简单总结一下。

- Array：Array的length属性根据最大的下标自动发生变化。
- Object.prototype：作为所有正常对象的默认原型，不能再给它设置原型了。
- String：为了支持下标运算，String的正整数属性访问会去字符串里查找。
- Arguments：arguments的非负整数型下标属性跟对应的变量联动。
- 模块的namespace对象：特殊的地方非常多，跟一般对象完全不一样，尽量只用于import吧。
- 类型数组和数组缓冲区：跟内存块相关联，下标运算比较特殊。
- bind后的function：跟原来的函数相关联。

结语

在这篇文章中，我们介绍了一些不那么常规的对象，并且我还介绍了JavaScript中用对象来模拟函数和构造器的机制。

这是一些不那么有规律、不那么优雅的知识，而JavaScript正是通过这些对象，提供了很多基础的能力。

我们这次课程留一个挑战任务：不使用new运算符，尽可能找到获得对象的方法。

例子：

```
var o = {}
var o = function(){}
```

请把自己的答案留言给我，我们来比比看谁找到的多。

小实验：获取全部JavaScript固有对象

我们从JavaScript标准中可以找到全部的JavaScript对象定义。JavaScript语言规定了全局对象的属性。

三个值：

Infinity、NaN、undefined。

九个函数：

- eval
- isFinite
- isNaN
- parseFloat
- parseInt
- decodeURI
- decodeURIComponent
- encodeURI
- encodeURIComponent

一些构造器：
Array、Date、RegExp、Promise、Proxy、Map、WeakMap、Set、WeakSet、Function、Boolean、String、Number、Symbol、Object、Error、

EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError、ArrayBuffer、SharedArrayBuffer、DataView、Typed Array、Float32Array、Float64Array、Int8Array、Int16Array、Int32Array、UInt8Array、UInt16Array、UInt32Array、UInt8ClampedArray。

四个用于当作命名空间的对象：

- Atomics
- JSON
- Math
- Reflect

我们使用广度优先搜索，查找这些对象所有的属性和Getter/Setter，就可以获得JavaScript中所有的固有对象。

请你试着先不看我的代码，在自己的浏览器中计算出来JavaScript有多少固有对象。

```
var set = new Set();
var objects = [
    eval,
    isFinite,
    isNaN,
    parseFloat,
    parseInt,
    decodeURI,
    decodeURIComponent,
    encodeURI,
    encodeURIComponent,
    Array,
    Date,
    RegExp,
    Promise,
    Proxy,
    Map,
    WeakMap,
    Set,
    WeakSet,
    Function,
    Boolean,
    String,
    Number,
    Symbol,
    Object,
    Error,
    EvalError,
    RangeError,
    ReferenceError,
    SyntaxError,
    TypeError,
    URIError,
    ArrayBuffer,
    SharedArrayBuffer,
    DataView,
    Float32Array,
    Float64Array,
    Int8Array,
    Int16Array,
    Int32Array,
    Uint8Array,
    Uint16Array,
    Uint32Array,
    Uint8ClampedArray,
    Atomics,
    JSON,
    Math,
    Reflect];
objects.forEach(o => set.add(o));

for(var i = 0; i < objects.length; i++) {
    var o = objects[i]
    for(var p of Object.getOwnPropertyNames(o)) {
        var d = Object.getOwnPropertyDescriptor(o, p)
        if( (d.value !== null && typeof d.value === "object") || (typeof d.value === "function"))
            if(!set.has(d.value))
                set.add(d.value), objects.push(d.value);
        if( d.get )
            if(!set.has(d.get))
                set.add(d.get), objects.push(d.get);
        if( d.set )
            if(!set.has(d.set))
                set.add(d.set), objects.push(d.set);
    }
}
```

你好，我是winter。

在前面的课程中，我已经讲解了JavaScript对象的一些基础知识。但是，我们所讲解的对象，只是特定的一部分，并不能涵盖全部的JavaScript对象。

比如说，我们不论怎样编写代码，都没法绕开Array，实现一个跟原生的数组行为一模一样的对象，这是由于原生数组的底层实现了一个自动

随着下标变化的length属性。

并且，在浏览器环境中，我们也无法单纯依靠JavaScript代码实现div对象，只能靠document.createElement来创建。这也说明了JavaScript的对象机制并非简单的属性集合+原型。

我们日常工作中，接触到的主要API，几乎都是由今天所讲解的这些对象提供的。理解这些对象的性质，我们才能真正理解我们使用的API的一些特性。

JavaScript中的对象分类

我们可以把对象分成几类。

- 宿主对象（host Objects）：由JavaScript宿主环境提供的对象，它们的行为完全由宿主环境决定。
- 内置对象（Built-in Objects）：由JavaScript语言提供的对象。
    - 固有对象（Intrinsic Objects ）：由标准规定，随着JavaScript运行时创建而自动创建的对象实例。
    - 原生对象（Native Objects）：可以由用户通过Array、RegExp等内置构造器或者特殊语法创建的对象。
    - 普通对象（Ordinary Objects）：由{}语法、Object构造器或者class关键字定义类创建的对象，它能够被原型继承。

下面我会为你一一讲解普通对象之外的对象类型。

宿主对象

首先我们来看看宿主对象。

JavaScript宿主对象千奇百怪，但是前端最熟悉的无疑是浏览器环境中的宿主了。

在浏览器环境中，我们都知道全局对象是window，window上又有很多属性，如document。

实际上，这个全局对象window上的属性，一部分来自JavaScript语言，一部分来自浏览器环境。

JavaScript标准中规定了全局对象属性，W3C的各种标准中规定了Window对象的其它属性。

宿主对象也分为固有的和用户可创建的两种，比如document.createElement就可以创建一些DOM对象。

宿主也会提供一些构造器，比如我们可以使用new Image来创建img元素，这些我们会在浏览器的API部分详细讲解。

内置对象·固有对象

我们在前面说过，固有对象是由标准规定，随着JavaScript运行时创建而自动创建的对象实例。

固有对象在任何JavaScript代码执行前就已经被创建出来了，它们通常扮演者类似基础库的角色。我们前面提到的“类”其实就是固有对象的一种。

ECMA标准为我们提供了一份固有对象表，里面含有150+个固有对象。你可以通过这个链接查看。

但是遗憾的是，这个表格并不完整。所以在本篇的末尾，我设计了一个小实验（小实验：获取全部JavaScript固有对象），你可以自己尝试一下，数一数一共有多少个固有对象。

内置对象·原生对象

我们把JavaScript中，能够通过语言本身的构造器创建的对象称作原生对象。在JavaScript标准中，提供了30多个构造器。按照我的理解，按照不同应用场景，我把原生对象分成了以下几个种类。

基本类型	基础功能和数据结构	错误类型	二进制操作	带类型的数组
Boolean	Array	Error	ArrayBuffer	Float32Array
String	Date	EvalError	SharedArrayBuffer	Float64Array
Number	RegExp	RangeError	DataView	Int8Array
Symbol	Promise	ReferenceError		Int16Array
Object	Proxy	SyntaxError		Int32Array
	Map	TypeError		Uint8Array
	WeakMap	URIError		Uint16Array
	Set			Uint32Array
	WeakSet			Uint8ClampedArray
	Function			

通过这些构造器，我们可以用new运算创建新的对象，所以我们把这些对象称作原生对象。
几乎所有这些构造器的能力都是无法用纯JavaScript代码实现的，它们也无法用class/extend语法来继承。

这些构造器创建的对象多数使用了私有字段,例如：

- Error: [[ErrorData]]
- Boolean: [[BooleanData]]
- Number: [[NumberData]]
- Date: [[DateValue]]
- RegExp: [[RegExpMatcher]]
- Symbol: [[SymbolData]]
- Map: [[MapData]]

这些字段使得原型继承方法无法正常工作，所以，我们可以认为，所有这些原生对象都是为了特定能力或者性能，而设计出来的“特权对象”。

用对象来模拟函数与构造器：函数对象与构造器对象

我在前面介绍了对象的一般分类，在JavaScript中，还有一个看待对象的不同视角，这就是用对象来模拟函数和构造器。

事实上，JavaScript为这一类对象预留了私有字段机制，并规定了抽象的函数对象与构造器对象的概念。

函数对象的定义是：具有[[call]]私有字段的对象，构造器对象的定义是：具有私有字段[[construct]]的对象。

JavaScript用对象模拟函数的设计代替了一般编程语言中的函数，它们可以像其它语言的函数一样被调用、传参。任何宿主只要提供了“具有[[call]]私有字段的对象”，就可以被 JavaScript 函数调用语法支持。

> [[call]]私有字段必须是一个引擎中定义的函数，需要接受this值和调用参数，并且会产生域的切换，这些内容，我将会在属性访问和执行过程两个章节详细讲述。

我们可以这样说，任何对象只需要实现[[call]]，它就是一个函数对象，可以去作为函数被调用。而如果它能实现[[construct]]，它就是一个构造器对象，可以作为构造器被调用。

对于为JavaScript提供运行环境的程序员来说，只要字段符合，我们在上文中提到的宿主对象和内置对象（如Symbol函数）可以模拟函数和构造器。

当然了，用户用function关键字创建的函数必定同时是函数和构造器。不过，它们表现出来的行为效果却并不相同。

对于宿主和内置对象来说，它们实现[[call]]（作为函数被调用）和[[construct]]（作为构造器被调用）不总是一致的。比如内置对象 Date 在作为构造器调用时产生新的对象，作为函数时，则产生字符串，见以下代码：

```
    console.log(new Date); // 1
    console.log(Date())
```

而浏览器宿主环境中，提供的Image构造器，则根本不允许被作为函数调用。

```
console.log(new Image);
console.log(Image());//抛出错误
```

再比如基本类型（String、Number、Boolean），它们的构造器被当作函数调用，则产生类型转换的效果。

值得一提的是，在ES6之后 => 语法创建的函数仅仅是函数，它们无法被当作构造器使用，见以下代码：

```
    new (a => 0) // error
```

对于用户使用 function 语法或者Function构造器创建的对象来说，[[call]]和[[construct]]行为总是相似的，它们执行同一段代码。

我们看一下示例。

```
function f(){
    return 1;
}
var v = f(); //把f作为函数调用
var o = new f(); //把f作为构造器调用
```

我们大致可以认为，它们[[construct]]的执行过程如下：

- 以 Object.prototype 为原型创建一个新对象；
- 以新对象为 this，执行函数的[[call]]；
- 如果[[call]]的返回值是对象，那么，返回这个对象，否则返回第一步创建的新对象。

这样的规则造成了个有趣的现象，如果我们的构造器返回了一个新的对象，那么new创建的新对象就变成了一个构造函数之外完全无法访问的对象，这一定程度上可以实现“私有”。

```
function cls(){
    this.a = 100;
    return {
        getValue:() => this.a
    }
```

```
}
var o = new cls;
o.getValue(); //100
//a在外面永远无法访问到
```

特殊行为的对象

除了上面介绍的对象之外，在固有对象和原生对象中，有一些对象的行为跟正常对象有很大区别。

它们常见的下标运算（就是使用中括号或者点来做属性访问）或者设置原型跟普通对象不同，这里我简单总结一下。

- Array：Array的length属性根据最大的下标自动发生变化。
- Object.prototype：作为所有正常对象的默认原型，不能再给它设置原型了。
- String：为了支持下标运算，String的正整数属性访问会去字符串里查找。
- Arguments：arguments的非负整数型下标属性跟对应的变量联动。
- 模块的namespace对象：特殊的地方非常多，跟一般对象完全不一样，尽量只用于import吧。
- 类型数组和数组缓冲区：跟内存块相关联，下标运算比较特殊。
- bind后的function：跟原来的函数相关联。

结语

在这篇文章中，我们介绍了一些不那么常规的对象，并且我还介绍了JavaScript中用对象来模拟函数和构造器的机制。

这是一些不那么有规律、不那么优雅的知识，而JavaScript正是通过这些对象，提供了很多基础的能力。

我们这次课程留一个挑战任务：不使用new运算符，尽可能找到获得对象的方法。

例子：

```
var o = {}
var o = function(){}
```

请把自己的答案留言给我，我们来比比看谁找到的多。

小实验：获取全部JavaScript固有对象

我们从JavaScript标准中可以找到全部的JavaScript对象定义。JavaScript语言规定了全局对象的属性。

三个值：

Infinity、NaN、undefined。

九个函数：

- eval
- isFinite
- isNaN
- parseFloat
- parseInt
- decodeURI
- decodeURIComponent
- encodeURI
- encodeURIComponent

一些构造器：
Array、Date、RegExp、Promise、Proxy、Map、WeakMap、Set、WeakSet、Function、Boolean、String、Number、Symbol、Object、Error、EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError、ArrayBuffer、SharedArrayBuffer、DataView、Typed Array、Float32Array、Float64Array、Int8Array、Int16Array、Int32Array、UInt8Array、UInt16Array、UInt32Array、UInt8ClampedArray。

四个用于当作命名空间的对象：

- Atomics
- JSON
- Math
- Reflect

我们使用广度优先搜索，查找这些对象所有的属性和Getter/Setter，就可以获得JavaScript中所有的固有对象。

请你试着先不看我的代码，在自己的浏览器中计算出来JavaScript有多少固有对象。

```
var set = new Set();
var objects = [
    eval,
    isFinite,
    isNaN,
    parseFloat,
    parseInt,
    decodeURI,
```

```
    decodeURIComponent,
    encodeURI,
    encodeURIComponent,
    Array,
    Date,
    RegExp,
    Promise,
    Proxy,
    Map,
    WeakMap,
    Set,
    WeakSet,
    Function,
    Boolean,
    String,
    Number,
    Symbol,
    Object,
    Error,
    EvalError,
    RangeError,
    ReferenceError,
    SyntaxError,
    TypeError,
    URIError,
    ArrayBuffer,
    SharedArrayBuffer,
    DataView,
    Float32Array,
    Float64Array,
    Int8Array,
    Int16Array,
    Int32Array,
    Uint8Array,
    Uint16Array,
    Uint32Array,
    Uint8ClampedArray,
    Atomics,
    JSON,
    Math,
    Reflect];
objects.forEach(o => set.add(o));

for(var i = 0; i < objects.length; i++) {
    var o = objects[i]
    for(var p of Object.getOwnPropertyNames(o)) {
        var d = Object.getOwnPropertyDescriptor(o, p)
        if( (d.value !== null && typeof d.value === "object") || (typeof d.value === "function"))
            if(!set.has(d.value))
                set.add(d.value), objects.push(d.value);
        if( d.get )
            if(!set.has(d.get))
                set.add(d.get), objects.push(d.get);
        if( d.set )
            if(!set.has(d.set))
                set.add(d.set), objects.push(d.set);
    }
}
```

你好，我是winter。

在前面的课程中，我已经讲解了JavaScript对象的一些基础知识。但是，我们所讲解的对象，只是特定的一部分，并不能涵盖全部的JavaScript对象。

比如说，我们不论怎样编写代码，都没法绕开Array，实现一个跟原生的数组行为一模一样的对象，这是由于原生数组的底层实现了一个自动随着下标变化的length属性。

并且，在浏览器环境中，我们也无法单纯依靠JavaScript代码实现div对象，只能靠document.createElement来创建。这也说明了JavaScript的对象机制并非简单的属性集合+原型。

我们日常工作中，接触到的主要API，几乎都是由今天所讲解的这些对象提供的。理解这些对象的性质，我们才能真正理解我们使用的API的一些特性。

JavaScript中的对象分类

我们可以把对象分成几类。

- 宿主对象（host Objects）：由JavaScript宿主环境提供的对象，它们的行为完全由宿主环境决定。
- 内置对象（Built-in Objects）：由JavaScript语言提供的对象。
    - 固有对象（Intrinsic Objects ）：由标准规定，随着JavaScript运行时创建而自动创建的对象实例。
    - 原生对象（Native Objects）：可以由用户通过Array、RegExp等内置构造器或者特殊语法创建的对象。
    - 普通对象（Ordinary Objects）：由{}语法、Object构造器或者class关键字定义类创建的对象，它能够被原型继承。

下面我会为你一一讲解普通对象之外的对象类型。

宿主对象

首先我们来看看宿主对象。

JavaScript宿主对象千奇百怪，但是前端最熟悉的无疑是浏览器环境中的宿主了。

在浏览器环境中，我们都知道全局对象是window，window上又有很多属性，如document。

实际上，这个全局对象window上的属性，一部分来自JavaScript语言，一部分来自浏览器环境。

JavaScript标准中规定了全局对象属性，W3C的各种标准中规定了Window对象的其它属性。

宿主对象也分为固有的和用户可创建的两种，比如document.createElement就可以创建一些DOM对象。

宿主也会提供一些构造器，比如我们可以使用new Image来创建img元素，这些我们会在浏览器的API部分详细讲解。

内置对象·固有对象

我们在前面说过，固有对象是由标准规定，随着JavaScript运行时创建而自动创建的对象实例。

固有对象在任何JavaScript代码执行前就已经被创建出来了，它们通常扮演者类似基础库的角色。我们前面提到的“类”其实就是固有对象的一种。

ECMA标准为我们提供了一份固有对象表，里面含有150+个固有对象。你可以通过这个链接查看。

但是遗憾的是，这个表格并不完整。所以在本篇的末尾，我设计了一个小实验（小实验：获取全部JavaScript固有对象），你可以自己尝试一下，数一数一共有多少个固有对象。

内置对象·原生对象

我们把JavaScript中，能够通过语言本身的构造器创建的对象称作原生对象。在JavaScript标准中，提供了30多个构造器。按照我的理解，按照不同应用场景，我把原生对象分成了以下几个种类。

基本类型	基础功能和数据结构	错误类型	二进制操作	带类型的数组
Boolean	Array	Error	ArrayBuffer	Float32Array
String	Date	EvalError	SharedArrayBuffer	Float64Array
Number	RegExp	RangeError	DataView	Int8Array
Symbol	Promise	ReferenceError		Int16Array
Object	Proxy	SyntaxError		Int32Array
	Map	TypeError		UInt8Array
	WeakMap	URIError		UInt16Array
	Set			UInt32Array
	WeakSet			UInt8ClampedArray
	Function			

通过这些构造器，我们可以用new运算创建新的对象，所以我们把这些对象称作原生对象。
几乎所有这些构造器的能力都是无法用纯JavaScript代码实现的，它们也无法用class/extend语法来继承。

这些构造器创建的对象多数使用了私有字段,例如：

- Error: [[ErrorData]]
- Boolean: [[BooleanData]]
- Number: [[NumberData]]
- Date: [[DateValue]]
- RegExp: [[RegExpMatcher]]
- Symbol: [[SymbolData]]
- Map: [[MapData]]

这些字段使得原型继承方法无法正常工作，所以，我们可以认为，所有这些原生对象都是为了特定能力或者性能，而设计出来的“特权对象”。

用对象来模拟函数与构造器：函数对象与构造器对象

我在前面介绍了对象的一般分类，在JavaScript中，还有一个看待对象的不同视角，这就是用对象来模拟函数和构造器。

事实上，JavaScript为这一类对象预留了私有字段机制，并规定了抽象的函数对象与构造器对象的概念。

函数对象的定义是：具有[[call]]私有字段的对象，构造器对象的定义是：具有私有字段[[construct]]的对象。

JavaScript用对象模拟函数的设计代替了一般编程语言中的函数，它们可以像其它语言的函数一样被调用、传参。任何宿主只要提供了“具有[[call]]私有字段的对象”，就可以被 JavaScript 函数调用语法支持。

> [[call]]私有字段必须是一个引擎中定义的函数，需要接受this值和调用参数，并且会产生域的切换，这些内容，我将会在属性访问和执行过程两个章节详细讲述。

我们可以这样说，任何对象只需要实现[[call]]，它就是一个函数对象，可以去作为函数被调用。而如果它能实现[[construct]]，它就是一个构造器对象，可以作为构造器被调用。

对于为JavaScript提供运行环境的程序员来说，只要字段符合，我们在上文中提到的宿主对象和内置对象（如Symbol函数）可以模拟函数和构造器。

当然了，用户用function关键字创建的函数必定同时是函数和构造器。不过，它们表现出来的行为效果却并不相同。

对于宿主和内置对象来说，它们实现[[call]]（作为函数被调用）和[[construct]]（作为构造器被调用）不总是一致的。比如内置对象 Date 在作为构造器调用时产生新的对象，作为函数时，则产生字符串，见以下代码：

```
    console.log(new Date); // 1
    console.log(Date())
```

而浏览器宿主环境中，提供的Image构造器，则根本不允许被作为函数调用。

```
console.log(new Image);
console.log(Image());//抛出错误
```

再比如基本类型（String、Number、Boolean），它们的构造器被当作函数调用，则产生类型转换的效果。

值得一提的是，在ES6之后 => 语法创建的函数仅仅是函数，它们无法被当作构造器使用，见以下代码：

```
    new (a => 0) // error
```

对于用户使用 function 语法或者Function构造器创建的对象来说，[[call]]和[[construct]]行为总是相似的，它们执行同一段代码。

我们看一下示例。

```
function f(){
    return 1;
}
var v = f(); //把f作为函数调用
var o = new f(); //把f作为构造器调用
```

我们大致可以认为，它们[[construct]]的执行过程如下：

- 以 Object.prototype 为原型创建一个新对象；
- 以新对象为 this，执行函数的[[call]]；
- 如果[[call]]的返回值是对象，那么，返回这个对象，否则返回第一步创建的新对象。

这样的规则造成了个有趣的现象，如果我们的构造器返回了一个新的对象，那么new创建的新对象就变成了一个构造函数之外完全无法访问的对象，这一定程度上可以实现“私有”。

```
function cls(){
    this.a = 100;
    return {
        getValue:() => this.a
    }
}
var o = new cls;
o.getValue(); //100
//a在外面永远无法访问到
```

特殊行为的对象

除了上面介绍的对象之外，在固有对象和原生对象中，有一些对象的行为跟正常对象有很大区别。

它们常见的下标运算（就是使用中括号或者点来做属性访问）或者设置原型跟普通对象不同，这里我简单总结一下。

- Array：Array的length属性根据最大的下标自动发生变化。
- Object.prototype：作为所有正常对象的默认原型，不能再给它设置原型了。
- String：为了支持下标运算，String的正整数属性访问会去字符串里查找。
- Arguments：arguments的非负整数型下标属性跟对应的变量联动。
- 模块的namespace对象：特殊的地方非常多，跟一般对象完全不一样，尽量只用于import吧。
- 类型数组和数组缓冲区：跟内存块相关联，下标运算比较特殊。
- bind后的function：跟原来的函数相关联。

结语

在这篇文章中，我们介绍了一些不那么常规的对象，并且我还介绍了JavaScript中用对象来模拟函数和构造器的机制。

这是一些不那么有规律、不那么优雅的知识，而JavaScript正是通过这些对象，提供了很多基础的能力。

我们这次课程留一个挑战任务：不使用new运算符，尽可能找到获得对象的方法。

例子：

```
var o = {}
var o = function(){}
```

请把自己的答案留言给我，我们来比比看谁找到的多。

小实验：获取全部JavaScript固有对象

我们从JavaScript标准中可以找到全部的JavaScript对象定义。JavaScript语言规定了全局对象的属性。

三个值：

Infinity、NaN、undefined。

九个函数：

- eval
- isFinite
- isNaN
- parseFloat
- parseInt
- decodeURI
- decodeURIComponent
- encodeURI
- encodeURIComponent

一些构造器：
Array、Date、RegExp、Promise、Proxy、Map、WeakMap、Set、WeakSet、Function、Boolean、String、Number、Symbol、Object、Error、EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError、ArrayBuffer、SharedArrayBuffer、DataView、Typed Array、Float32Array、Float64Array、Int8Array、Int16Array、Int32Array、UInt8Array、UInt16Array、UInt32Array、UInt8ClampedArray。

四个用于当作命名空间的对象：

- Atomics
- JSON
- Math
- Reflect

我们使用广度优先搜索，查找这些对象所有的属性和Getter/Setter，就可以获得JavaScript中所有的固有对象。

请你试着先不看我的代码，在自己的浏览器中计算出来JavaScript有多少固有对象。

```
var set = new Set();
var objects = [
    eval,
    isFinite,
    isNaN,
    parseFloat,
    parseInt,
    decodeURI,
    decodeURIComponent,
    encodeURI,
    encodeURIComponent,
    Array,
    Date,
    RegExp,
    Promise,
    Proxy,
    Map,
    WeakMap,
    Set,
    WeakSet,
    Function,
    Boolean,
    String,
    Number,
    Symbol,
    Object,
    Error,
    EvalError,
    RangeError,
    ReferenceError,
    SyntaxError,
    TypeError,
    URIError,
```

```
    ArrayBuffer,
    SharedArrayBuffer,
    DataView,
    Float32Array,
    Float64Array,
    Int8Array,
    Int16Array,
    Int32Array,
    Uint8Array,
    Uint16Array,
    Uint32Array,
    Uint8ClampedArray,
    Atomics,
    JSON,
    Math,
    Reflect];
objects.forEach(o => set.add(o));

for(var i = 0; i < objects.length; i++) {
    var o = objects[i]
    for(var p of Object.getOwnPropertyNames(o)) {
        var d = Object.getOwnPropertyDescriptor(o, p)
        if( (d.value !== null && typeof d.value === "object") || (typeof d.value === "function"))
            if(!set.has(d.value))
                set.add(d.value), objects.push(d.value);
        if( d.get )
            if(!set.has(d.get))
                set.add(d.get), objects.push(d.get);
        if( d.set )
            if(!set.has(d.set))
                set.add(d.set), objects.push(d.set);
    }
}
```

你好，我是winter。

在前面的课程中，我已经讲解了JavaScript对象的一些基础知识。但是，我们所讲解的对象，只是特定的一部分，并不能涵盖全部的JavaScript对象。

比如说，我们不论怎样编写代码，都没法绕开Array，实现一个跟原生的数组行为一模一样的对象，这是由于原生数组的底层实现了一个自动随着下标变化的length属性。

并且，在浏览器环境中，我们也无法单纯依靠JavaScript代码实现div对象，只能靠document.createElement来创建。这也说明了JavaScript的对象机制并非简单的属性集合+原型。

我们日常工作中，接触到的主要API，几乎都是由今天所讲解的这些对象提供的。理解这些对象的性质，我们才能真正理解我们使用的API的一些特性。

JavaScript中的对象分类

我们可以把对象分成几类。

- 宿主对象（host Objects）：由JavaScript宿主环境提供的对象，它们的行为完全由宿主环境决定。
- 内置对象（Built-in Objects）：由JavaScript语言提供的对象。
    - 固有对象（Intrinsic Objects ）：由标准规定，随着JavaScript运行时创建而自动创建的对象实例。
    - 原生对象（Native Objects）：可以由用户通过Array、RegExp等内置构造器或者特殊语法创建的对象。
    - 普通对象（Ordinary Objects）：由{}语法、Object构造器或者class关键字定义类创建的对象，它能够被原型继承。

下面我会为你一一讲解普通对象之外的对象类型。

宿主对象

首先我们来看看宿主对象。

JavaScript宿主对象千奇百怪，但是前端最熟悉的无疑是浏览器环境中的宿主了。

在浏览器环境中，我们都知道全局对象是window，window上又有很多属性，如document。

实际上，这个全局对象window上的属性，一部分来自JavaScript语言，一部分来自浏览器环境。

JavaScript标准中规定了全局对象属性，W3C的各种标准中规定了Window对象的其它属性。

宿主对象也分为固有的和用户可创建的两种，比如document.createElement就可以创建一些DOM对象。

宿主也会提供一些构造器，比如我们可以使用new Image来创建img元素，这些我们会在浏览器的API部分详细讲解。

内置对象·固有对象

我们在前面说过，固有对象是由标准规定，随着JavaScript运行时创建而自动创建的对象实例。

固有对象在任何JavaScript代码执行前就已经被创建出来了，它们通常扮演者类似基础库的角色。我们前面提到的“类”其实就是固有对象的一种。

ECMA标准为我们提供了一份固有对象表，里面含有150+个固有对象。你可以通过这个链接查看。

但是遗憾的是，这个表格并不完整。所以在本篇的末尾，我设计了一个小实验（小实验：获取全部JavaScript固有对象），你可以自己尝试一下，数一数一共有多少个固有对象。

内置对象·原生对象

我们把JavaScript中，能够通过语言本身的构造器创建的对象称作原生对象。在JavaScript标准中，提供了30多个构造器。按照我的理解，按照不同应用场景，我把原生对象分成了以下几个种类。

基本类型	基础功能和数据结构	错误类型	二进制操作	带类型的数组
Boolean	Array	Error	ArrayBuffer	Float32Array
String	Date	EvalError	SharedArrayBuffer	Float64Array
Number	RegExp	RangeError	DataView	Int8Array
Symbol	Promise	ReferenceError		Int16Array
Object	Proxy	SyntaxError		Int32Array
	Map	TypeError		Uint8Array
	WeakMap	URIError		Uint16Array
	Set			Uint32Array
	WeakSet			Uint8ClampedArray
	Function			

通过这些构造器，我们可以用new运算创建新的对象，所以我们把这些对象称作原生对象。
几乎所有这些构造器的能力都是无法用纯JavaScript代码实现的，它们也无法用class/extend语法来继承。

这些构造器创建的对象多数使用了私有字段,例如：

- Error: [[ErrorData]]
- Boolean: [[BooleanData]]
- Number: [[NumberData]]
- Date: [[DateValue]]
- RegExp: [[RegExpMatcher]]
- Symbol: [[SymbolData]]
- Map: [[MapData]]

这些字段使得原型继承方法无法正常工作，所以，我们可以认为，所有这些原生对象都是为了特定能力或者性能，而设计出来的“特权对象”。

用对象来模拟函数与构造器：函数对象与构造器对象

我在前面介绍了对象的一般分类，在JavaScript中，还有一个看待对象的不同视角，这就是用对象来模拟函数和构造器。

事实上，JavaScript为这一类对象预留了私有字段机制，并规定了抽象的函数对象与构造器对象的概念。

函数对象的定义是：具有[[call]]私有字段的对象，构造器对象的定义是：具有私有字段[[construct]]的对象。

JavaScript用对象模拟函数的设计代替了一般编程语言中的函数，它们可以像其它语言的函数一样被调用、传参。任何宿主只要提供了“具有[[call]]私有字段的对象”，就可以被 JavaScript 函数调用语法支持。

> [[call]]私有字段必须是一个引擎中定义的函数，需要接受this值和调用参数，并且会产生域的切换，这些内容，我将会在属性访问和执行过程两个章节详细讲述。

我们可以这样说，任何对象只需要实现[[call]]，它就是一个函数对象，可以去作为函数被调用。而如果它能实现[[construct]]，它就是一个构造器对象，可以作为构造器被调用。

对于为JavaScript提供运行环境的程序员来说，只要字段符合，我们在上文中提到的宿主对象和内置对象（如Symbol函数）可以模拟函数和构造器。

当然了，用户用function关键字创建的函数必定同时是函数和构造器。不过，它们表现出来的行为效果却并不相同。

对于宿主和内置对象来说，它们实现[[call]]（作为函数被调用）和[[construct]]（作为构造器被调用）不总是一致的。比如内置对象 Date 在作为构造器调用时产生新的对象，作为函数时，则产生字符串，见以下代码：

```
console.log(new Date); // 1
console.log(Date())
```

而浏览器宿主环境中，提供的Image构造器，则根本不允许被作为函数调用。

```
console.log(new Image);
console.log(Image());//抛出错误
```

再比如基本类型（String、Number、Boolean），它们的构造器被当作函数调用，则产生类型转换的效果。

值得一提的是，在ES6之后 => 语法创建的函数仅仅是函数，它们无法被当作构造器使用，见以下代码：

```
    new (a => 0) // error
```

对于用户使用 function 语法或者Function构造器创建的对象来说，[[call]]和[[construct]]行为总是相似的，它们执行同一段代码。

我们看一下示例。

```
function f(){
    return 1;
}
var v = f(); //把f作为函数调用
var o = new f(); //把f作为构造器调用
```

我们大致可以认为，它们[[construct]]的执行过程如下：

- 以 Object.prototype 为原型创建一个新对象；
- 以新对象为 this，执行函数的[[call]]；
- 如果[[call]]的返回值是对象，那么，返回这个对象，否则返回第一步创建的新对象。

这样的规则造成了个有趣的现象，如果我们的构造器返回了一个新的对象，那么new创建的新对象就变成了一个构造函数之外完全无法访问的对象，这一定程度上可以实现“私有”。

```
function cls(){
    this.a = 100;
    return {
        getValue:() => this.a
    }
}
var o = new cls;
o.getValue(); //100
//a在外面永远无法访问到
```

特殊行为的对象

除了上面介绍的对象之外，在固有对象和原生对象中，有一些对象的行为跟正常对象有很大区别。

它们常见的下标运算（就是使用中括号或者点来做属性访问）或者设置原型跟普通对象不同，这里我简单总结一下。

- Array：Array的length属性根据最大的下标自动发生变化。
- Object.prototype：作为所有正常对象的默认原型，不能再给它设置原型了。
- String：为了支持下标运算，String的正整数属性访问会去字符串里查找。
- Arguments：arguments的非负整数型下标属性跟对应的变量联动。
- 模块的namespace对象：特殊的地方非常多，跟一般对象完全不一样，尽量只用于import吧。
- 类型数组和数组缓冲区：跟内存块相关联，下标运算比较特殊。
- bind后的function：跟原来的函数相关联。

结语

在这篇文章中，我们介绍了一些不那么常规的对象，并且我还介绍了JavaScript中用对象来模拟函数和构造器的机制。

这是一些不那么有规律、不那么优雅的知识，而JavaScript正是通过这些对象，提供了很多基础的能力。

我们这次课程留一个挑战任务：不使用new运算符，尽可能找到获得对象的方法。

例子：

```
var o = {}
var o = function(){}
```

请把自己的答案留言给我，我们来比比看谁找到的多。

小实验：获取全部JavaScript固有对象

我们从JavaScript标准中可以找到全部的JavaScript对象定义。JavaScript语言规定了全局对象的属性。

三个值：

Infinity、NaN、undefined。

九个函数：

- eval
- isFinite
- isNaN
- parseFloat
- parseInt
- decodeURI
- decodeURIComponent
- encodeURI
- encodeURIComponent

一些构造器：
Array、Date、RegExp、Promise、Proxy、Map、WeakMap、Set、WeakSet、Function、Boolean、String、Number、Symbol、Object、Error、EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError、ArrayBuffer、SharedArrayBuffer、DataView、Typed Array、Float32Array、Float64Array、Int8Array、Int16Array、Int32Array、UInt8Array、UInt16Array、UInt32Array、UInt8ClampedArray。

四个用于当作命名空间的对象：

- Atomics
- JSON
- Math
- Reflect

我们使用广度优先搜索，查找这些对象所有的属性和Getter/Setter，就可以获得JavaScript中所有的固有对象。

请你试着先不看我的代码，在自己的浏览器中计算出来JavaScript有多少固有对象。

```
var set = new Set();
var objects = [
    eval,
    isFinite,
    isNaN,
    parseFloat,
    parseInt,
    decodeURI,
    decodeURIComponent,
    encodeURI,
    encodeURIComponent,
    Array,
    Date,
    RegExp,
    Promise,
    Proxy,
    Map,
    WeakMap,
    Set,
    WeakSet,
    Function,
    Boolean,
    String,
    Number,
    Symbol,
    Object,
    Error,
    EvalError,
    RangeError,
    ReferenceError,
    SyntaxError,
    TypeError,
    URIError,
    ArrayBuffer,
    SharedArrayBuffer,
    DataView,
    Float32Array,
    Float64Array,
    Int8Array,
    Int16Array,
    Int32Array,
    Uint8Array,
    Uint16Array,
    Uint32Array,
    Uint8ClampedArray,
    Atomics,
    JSON,
    Math,
    Reflect];
objects.forEach(o => set.add(o));

for(var i = 0; i < objects.length; i++) {
    var o = objects[i]
    for(var p of Object.getOwnPropertyNames(o)) {
        var d = Object.getOwnPropertyDescriptor(o, p)
        if( (d.value !== null && typeof d.value === "object") || (typeof d.value === "function"))
            if(!set.has(d.value))
                set.add(d.value), objects.push(d.value);
        if( d.get )
```

```
            if(!set.has(d.get))
                set.add(d.get), objects.push(d.get);
        if( d.set )
            if(!set.has(d.set))
                set.add(d.set), objects.push(d.set);
    }
}
```