Files

haitaoxin / jsoo

No description, website, or topics provided.

59 commits 1 branch 0 releases

Branch: master ▾ New pull request Create new file Upload files

haitaoxin Merge pull request #1 from Gcaufy/patch-1 ···

README.md fixed serval indents

README.md

JavaScript Object-Oriented Programming: Princip Practices

面向对象的JavaScript编程：原理与实践

1. 简介

初衷

笔者早年编写代码使用过 Motorola 68000 和 x86汇编、C/C++、Pascal、Fortran、PowerBuilder、短暂接触过Java、Python和Swift。最近这些年来使用最多的是JavaScript（正式名称是ECMAScript JS）。众所周知，JS问世之初是“难登大雅之堂”的小玩意儿，还要靠冠名Java以壮声势。但是随着浏行的平台成熟起来，再加上NodeJS项目的兴旺，JS本身的进步和普及程度已经使它成为每一个前台软必不可少的工具。

虽然这种语言已经强大到适用于大多数类型的后台服务器程序（更不用说它是前台开发的唯一正式语言其不够严谨，争论它是不是完全符合面向对象（Object-Oriented）语言的特征。更多的初学者和程序机制和使用多少有些含糊不清。这是因为JS的对象跟以前常见的C++或者Java的对象的确有所不同，最初就不像C++或者Java那样经过深思熟虑、严格定义的。但是幸运的是，发展到今天，JS对OO的支编程需要。

本书并不想参与任何无谓的或者纯理论的讨论，只想理清JS的对象到底是怎么回事，以及怎么使用JS和它的局限性在哪儿。

感谢

非常感谢 Jiang Hao 提供的宝贵意见。

适合的读者

本书不是写给初学者看的。读者需要有一定的JS基础以及“**知其然且知其所以然**”的态度。如果你学习过JavaScript，但是对有些概念还是理解得似是而非，这本书就是试图讲解那些概念背后的来龙去脉（当Specification 里都有，但是我相信大多数读者不会喜欢去读那些枯燥的文档）。如果你以前用 C++ 和对象、函数、继承、类等等总感觉有些别扭，那这本书可以帮你澄清这些容易混淆的概念。如果你没有基础，可以说是喜忧参半：喜的是你不会把 JavaScript 和 C++/Java 里同名不同义的概念混淆，没有忧的是有些面向对象的概念你可能需要补一补课。

我没有凑字数给出版社的压力，所以本书的内容基本全是干货——我故意避免有任何废话、显而易见的我会重复好几次，这或者是因为此概念非常重要，或者是因为它不那么容易理解。我相信大多数人和我并不是看一遍就过目不忘、运用自如。

所有代码都已使用 Node 6.x LTS 环境或者在最新版本的Chrome浏览器里验证过。这些代码绝大多数并且可在 console 里看到输出结果的。我非常鼓励你把这些代码运行通过后，再按自己的想法改一改，的。

Files

另外需要抱歉的是有些英文专用单词也许翻译得不是最常见的用法。任何技术上或者翻译上的错误，或
细致的地方，还望读者不吝赐教。

引用资料

本书的动力之一就是我读了 Nicolas Zakas 的《The Principles of Object-Oriented JavaScript》所
的组织结构和代码举例可能会引用其书。本书永久免费，但是我也鼓励有英文阅读能力和财力的读者购

其它好的参考书籍和网站我会逐步列出。

2. 基础数据类型（Primitive）和引用数据类型（Ref

和任何其它现代的编程语言一样，JS对普通的算数运算和循环都有很容易理解的使用方法。JS语言本
度上在于它对数据的表达和处理。根据其在内存里的存储和管理方式，JS支持的数据类型可以分为两大
（Primitive）和引用数据类型（Reference）。二者的区别对于我们理解对象有很关键的作用。

基础数据类型

基础数据类型用于存储比较简单这几类数据：

- **boolean**：布尔值，取值范围只有 `true` 和 `false`
- **number**：任何整数和小数（浮点数）。⚠️ JS里另有 `Number` 这种对象，与此处number数据类型
- **string**：字符串，包括一个字符（JS没有'char'类型），支持Unicode。⚠️ 跟 `Number` 类似，Java
  对象 `String` 对应 `string`
- **null**：“空”。此类型只有一个值就是 `null`
- **undefined**：“未定义”。此类型只有一个值就是 `undefined` ，最常见的场景是你的代码定义了一个
  时，其值为 `undefined`
- **symbol**：在ES6里引入的一种新数据类型，用于定义对象成员的键值，跟面向对象编程关系也不大
  希望有机会再写一本ES6的书吧。

所有的基础数据类型都是一个具体的值存储于内存某处，而你定义的变量就直接被赋予这个值（而不是
是和引用类型最大的区别）。而每一个变量所存贮的值并不跟其它变量共享，即便其二者的内容是一样

```
// strings
var name = "Jack";
var answer = "a";

// numbers
var count = 25;
var result = -0;
var totalCost = 34.56;

// boolean
let default = true;

// null
var obj = null;

// undefined
let k = undefined;
let q;  // 自动被赋值为 undefined

// 每个变量都有自己的存贮地址
let val1 = 50;
let val2 = val1;  // val2 现在也将‘50’这个值存于自己的存储空间内
val1 = 25;
console.log(val2);  // 50；val2的值并不随着val1而改变；注意跟后边引用数据类型的例子对比
```

识别基础数据类型

由于JS的语法规定，在定义一个变量的时候，并不需要声明变量类型，其后赋值时也不进行类型检查，
要动态地检查其所持有的值的类型。为此JS提供了 `typeof` 操作符:

```
// 继续上面的代码
console.log(typeof name);       // 'string'
console.log(typeof count);      // 'number'
console.log(typeof totalCost);  // 'number'
console.log(typeof default);    // 'boolean'
console.log(typeof k);          // 'undefined'
```

Files

```
console.log(typeof q);          // 'undefined'
console.log(typeof obj);        // 'object' 这个结果有问题
```

以上这段代码通俗易懂，但是有两点需要说明：

1. 严格来说， `typeof` 是作用于其后变量当时所被赋予的值，而不是变量本身。因为JS对变量本身并
2. 最后一句里，变量 `obj` 的值是 `null` ，但是 `typeof` 返回的值却是 `object` 。这已经被确认为一个J
   为这个bug已经存在很久了，如果改正反倒可能让一些以前运行正常的程序出错。所以这个bug就

在我们的代码里正确判断 `null` 也很简单：

```
// 如何判断一个值为 null
console.log(obj === null);    // true
console.log(k === null);      // false; k is undefined
```

带强制类型转换的比较

当你用双等号 == 比较两个不同基础数据类型的值时，JavaScript引擎会试图进行强制数据转换之后再
=== 不进行强制类型转换，JavaScript引擎会既比较数据类型也比较数据值。比如

```
console.log("5" == 5);      // true
console.log("5" === 5);     // false

console.log(undefined == null);   // true
console.log(undefined === null);  // false
```

如果双等号的效果恰好是你需要的，当然这样的代码会比你自己转换数据类型后再比较更简洁。反之这
bug。最好的办法是养成习惯：每次写比较语句都先问自己一下：该使用双等号还是三等号？除非你确
的，都应该使用三等号。

基础数据类型的变量也可以调用一些方法（methods）

在传统的面向对象语言里，方法是只有对象才有的特性，是让对象比数据更强大的主要因素。但是在J
number、boolean这三种基础数据类型的变量也有一些自带的方法可以调用，比如

```
var street = "East First Street";
var lowercaseStreet = street.toLowerCase();   // 'east first street'
var firstChar = street.charAt(0);    // 'E'
var middleStreet = street.substring(5, 10);   // 'First'

var price = 10;
var fixedPrice = price.toFixed(2);    // 10.00
var hexCount = price.toString(16);    // 'a'；十六进制表达的10

var isDone = true;
var isDoneFlag = isDone.toString();   // 'true'

var p = "East First Street".toLowerCase();    // type error
var q = 10.toFixed(2);    // syntax error
```

由以上代码的最后两行可以看出来，基础数据类型的值是没有方法可以调用的；但是代表这些值的变量
JS对应这三种基础数据类型有三种已经内建好的标准对象类型：String，Number，和Boolean，并且
了一些最常用的方法以简化程序员的工作。为了把这样的好处也带给相应的基础数据类型的变量，JS引
`price.toFixed(2)` 这样的调用方法时，其实执行了类似下面的代码：

```
let __price__ = new Number(price);      //  构建一个 Number 对象
return __price__.toFixed(2);
```

所以一个基础数据类型变量可以使用的方法，其实就是它对应的内建对象的方法，这些方法都可以在J
些标准内建对象（standard built-in objects，也有一个更容易混淆的名字叫全局对象 global objects

引用数据类型：初步认识对象

我们先看一个典型的例子，帮助理解基础数据类型和引用数据类型的不同

```
let x = new Boolean(false);
if (x) {
```

Files

```
        console.log("x is a Boolean");
    }

    x = false;      // 现在 x 是个 boolean
    if (x) {
        console.log("x is a boolean");  // 不会被执行到
    }
```

如果你对这段代码的输出有点儿疑惑，正好可以带着这个问题继续学习，完成这章之后再返回来理解。

引用数据类型可以描述为用一个地址索引指向的对象。注意这里我用了“对象”（object）这个词，是因
数据类型和对象其实就是一回事。换一种说法，所有不是基础类型的数据结构都是对象。像函数、数组
然我们很少叫它们“函数对象”、“数组对象”，但是作为一个对象的本质特征它们都有。在本书里我们“
型”这两个名词是通用的。

引用数据类型的变量很像 C 语言的指针，你可以将其理解为里面存储的是指向对象的地址。但是不同于
取这个地址的绝对值的（实际上你也不需要）。每次你读取一个变量，如果变量里存的是基础数据类型
返回给你；如果变量里存的是对象的地址，JS引擎就自动按地址取得对象送给你。以下的代码有助于你

```
let a = { 'count': 5 };    // a 里存了指向一个对象的地址
let b = a;                  // b 被赋值为 a，现在 b 里存储的是和 a 一样的地址，所以 b 也指
a.count = 9;                // 通过 a 改变对象里的一个值
console.log(b.count);       // 9；b 的值也跟着改变了，因为它和 a 指向同一个地址
```

对比于

```
let a = 5;
let b = a;
a = 9;
console.log(b);             // 5
```

就可以看出来JS对于不同数据类型存取的不同方法。

对象内部对数值的存储可以粗略地看作是一张哈希表，这张表里每个成员的键值（ key ）必须是字符串
`symbol` 数据类型也可以当作 key，希望我们有机会在ES6的书里专门讲解），而对应的值（ value ）
者另外一个对象（包括函数、数组等等）。

在JavaScript里，“对象（object）”一词其实有两层含义。狭义的内涵是指我们用花括弧标识的若干对
定的数据类型，在英文里第一个字母大写（ `Object` ）。而当我们说“所有的引用数据类型都是对象”的
取其广义，是指狭义的对象数据类型加上其它各种内建的对象类型，即便这些内建类型的表现形式不一
对，比如函数、数组、Map、Set等等。需要特别说明的是函数，它们都是 `Function` 类型的对象，但
没有的特征，所以也就有了其它对象没有的行为，比如可以被调用。我们会在下一章专门讲解函数。

构建对象（Object）

构建一个狭义的对象，我们通常直接赋值就可以：

```
let laowang = {
    'name': '老王',
    'age': 35,
    'relationship': '邻居'
};
```

或者先定义一个空的对象，再给它的成员赋值：

```
let person = {};                    // 先创建一个空对象
person.name = "老张";               // 以后再填充内容
person.age = 24;
person.relationship = 'undefined';
```

比较少见的另一个形式是使用对象的构建函数：

```
let obj1 = new Object();
obj1['id'] = 10029;
```

这三者的效果完全一样。

常见的标准内建对象

除了 Object 这种对象外，JavaScript还定义了很多标准的对象类型。最常用的有以下几个

- Array： 数组，一组有序、可以索引的值
- Set：集合，无索引、无序的一组值
- Map：无序的 k-v 值
- Date：日期和时间
- Function：函数，将会在下一章详细描述
- RegExp：正则表达式
- Error：通用运行错误。JavaScript还定义了几个专用的错误类型，比如数据类型错误（TypeErr
- WeakMap 和 WeakSet：特殊的Map和Set，在某些情况下对管理内存有重要作用

另外，还有三个我们已经见过的基础数据类型对应的对象，也有人把他们称为基础数据类型的 wrappe
和 Boolean

JavaScript对每个标准对象都提供了相应的自带方法，可以简化很多相关的操作。当然，这些标准对象
方法，比如 toString()，也是支持的。

最后，还有一些标准对象并不是为了让我们用来构建自己的对象，而且用作一组相关方法的容器。我们
些方法。最常见的两个这样的方法容器就是 Math 和 JSON 。⚠️另一个我们常用的对象 console 不是
义的标准对象，而是业界共识，所以如果每个运行环境稍有不同也属正常。

对象成员的使用

不论是读写对象的数值成员，还是调用对象的成员函数，都有两种办法。第一种是在对象和成员的键之

```
let person = {
    name: "Obama",
    job: "retired"
};

let name = person.name;
```

第二种方法是用方括号括住键值，比如

```
let name = person["name"];
var array = [];
array["push"](100);     // now array is [100]
```

这里需要注意的是，如果用第二种方法而且如果键值是具体数值（而不是变量）的话，必须用单引号或
串，而第一种不用。另外，虽然第二种方法看起来有些奇怪（因为很多其它语言不支持这样的用法），
用：方括号里可以是一个变量，按运行时的情况赋值。比如

```
let wantName = true;
let key = wantName ? "name" : "job";
console.log(person[key]);
```

对象成员的增减

通过之前的例子，我们已经看到如何容易地增加一个对象的成员。减少一个成员可以使用 delete 运算

```
let person2 = {
    name: "Josh",
    dept: "Math",
    advisor: "Prof. Obama"
};
delete person2.advisor;
```

如果多余的成员变量并没有占用很大内存空间，通常并没用必要经常专门删除它们。另外，在很多情
成员，并不希望其他使用者去随意增减它们。如何做到这一点，我们会在第四章讲述。

释放对象使用的内存

JavaScript是自带垃圾回收的，所以通常你并不需要考虑内存占用问题。但是如果你使用的对象使用了
程序运行速度降低。JavaScript没有命令去释放一个对象占据的存储空间。但是当一个对象没有变量指
JavaScript的引擎释放。你需要做的仅仅是把原本指向此对象的变量指向 null

Files

```
let obj = new Object{};
obj.data = "something really big data block";
let obj2 = obj;
// 处理 obj、obj2 相关的逻辑

obj2 = null;
obj = null;     // 当没有任何变量指向此对象之后对象占用的空间会被释放
```

识别变量的数据类型

众所周知JavaScript在声明一个变量的时候并不定义其类型，其类型取决于它在运行时所指向数值的类
有时候我们需要识别一个变量的数据类型。一个非常明显的应用场景是实现面向对象编程的多态性：J
同一个函数，根据输入参数的类型和数量，应用不同的逻辑进行处理。但是应该说 JavaScript 识别变
乱的，有以下几种情况：

首先，是我们已经见过的 `typeof` 运算符: 可以正确识别除了 `null` 之外的各种基础数据类型和 `functio`
数据类型全部返回 `object` 。虽然这也不能算错，但是没什么用，毕竟一个 Array 和一个 Error 差很远

```
let num = 99;
console.log(typeof num);        // 'number'
let str = "99";
console.log(typeof str);        // 'string'
let bl = true;
console.log(typeof bl);         // 'boolean'
let foo = function() {return "happy";};
console.log(typeof f);          // 'function'
let obj = {};
console.log(typeof obj);        // 'object'
let arr = [9, 8, 7];
console.log(typeof arr);        // 'object'
let err = new Error("Something Wrong!");
console.log(typeof err);        // 'object'
```

其次，针对数组， `Array` 对象提供了一个静态方法 `.isArray()`

```
// 继续上面的代码
console.log(Array.isArray(arr));    // 'true'
console.log(Array.isArray(obj));    // 'false'
console.log(Array.isArray(err));    // 'false'
```

第三， `instanceof` 运算符可以识别对象类型，但是注意它对一个对象应用于其父类也返回 `true`

```
// 继续上面的代码
console.log(arr instanceof Array);      // 'true'
console.log(err instanceof Error);      // 'true'
console.log(foo instanceof Function);   // 'true'
console.log(arr instanceof Object);     // 'true'
```

最后，可以查询对象的 `constructor` 成员，比如

```
// 继续上面的代码
console.log(arr.constructor === Array);     // 'true'
console.log(arr.constructor === Object);    // 'false'
```

以上最后两种方法不仅适用于标准内建对象，也适用于使用自定义的构建函数生成的对象。我们会在第
数。

3. 函数（Function）

正如我们前面提到的，在 JavaScript 里函数也是一种对象。这跟其它很多语言的语法都不一样，但是
的这种处理方法并不难理解。

比如在 C++ 语言里，一个函数也是放在一块内存里的一段程序，而你的代码里的函数名就是指向这块
上来说，你使用函数也是在读取一个引用数据类型。只不过跟指向一个对象的地址不同的是，使用函数
限，比如你不能动态地改变其内容。

Files

而在 JavaScript 里，除了调用这个函数，你还可以把它当对象来使用（所以函数在 JavaScript 里也被
跟其它数据类型并列）。把它当作一个对象类型的参数输入另一个函数（比如作为回调函数），或者把
果输出，在JavaScript代码里都很常见。在本章里我们会看到这样的例子。在构建函数的章节里，我们
的方法（也就是函数对象的函数成员）。

既然函数不是普通的对象，那必定有它特别之处。表象之下，关键是它比其它对象类型多了一个内部成
）叫做 `[[Call]]` 。**内部成员被 JavaScript 语言定义于很多对象类型上，它们无法被我们的代码读写
会使用它们来正确处理不同种类的对象**。内部成员通常用双方括号 `[[ ]]` 标识，今后我们还会见到很多
成员只有函数这种对象才有，比如 JavaScript 引擎的 `typeof` 运算符就是靠检查这个成员来正确识别
同时，有了个这个成员，JavaScript 引擎也就认为这是个可以调用的函数了。

函数的四种定义方式

使用函数之前当然首先要定义它。定义函数有四种方式：函数声明（ function declaration ）、函数表
expression ）、使用 `Function` 构建函数和箭头函数（ arrow function ）。下面依次说明。

函数声明

函数声明是我们最常见到的函数定义形式。

```
function add(n1, n2) {
  return n1 + n2;
}
```

它的特点是**语句以关键字 `function` 开头**，后面加函数名和参数。函数的代码放在花括弧里。

函数表达式

函数表达式的形式和其它赋值语句一样，都是以等号为关键字，等号左边是变量名（也就是函数名），
数。

```
let sub = function (n1, n2) {
  return n1 - n2;
}
```

显然函数表达式要多输入几个字符，所以比函数声明稍微更少用到一些。

Function 构建函数

在JavaScript里， `Function` 也是一个标准内建对象，也可以使用 `new` （第五章会介绍）来构建一个函
表达式的一个特例。比如

```
var sayHi = new Function("console.log(\"Hi!\");");
sayHi();    // 'Hi!'
```

这种方式是把全部函数内容的语句当作一整个字符串，作为输入参数传给 `Function()` 这个构建函数。
难读，更重要的是难以debug。所以读者知道一下就好，一般不会用到。

只有在一种情况下这个定义方式才显示出来其强大的、很多其它语言没有的功能：因为 `Function()` 的
字符串，你可以在运行时根据当时的条件、按照 JavaScript 语法组合一个你需要的字符串，然后用这
新的函数。

函数表达式和函数声明的异同

以上两种定义方式定义的函数，在使用上基本是一样的。二者最大的区别是 **函数声明会被置顶（hoist
会** 。比如

```
var sum = add(3, 5);
console.log(sum);           // 8
var result = sub(9, 4);     // TypeError: sub is not a function

// 函数的定义在调用之后
function add(n1, n2) {
  return n1 + n2;
}

var sub = function (n1, n2) {
```

Files

```
  return n1 - n2;
}
```

在上面的代码中，变量 add 的声明和内容（也就是它指向的函数）都被JavaScript置顶，所以你在任何
变量 sub 的声明虽然也会被置顶（因为它由 var 定义，如果由 let 定义则不会），但是它的赋值语句
以在第三行调用它的时候，它的值还是 undefined 。回忆我们前面讲过的，JavaScript引擎靠检查对象
内部成员决定其是否是函数。 undefined 显然没有 [[Call]] ，所以会产生 TypeError 。

这两种方式的有个相同点常常被忽视：不论怎样定义，你定义的函数名其实就是个普通的变量。你还可
如果它得到的新值不是函数，你就不可以调用它了。比如以下的语句：

```
// 接上面的代码
add = 100;      // 没问题
sub = 0;        // 没问题
add(1, 2);      // TypeError: add is not a function
```

另外，执行以上语句之后，原先定义的函数就再没有变量指向它们了。别忘了函数也是对象，所以这些
garbage collector 从内存里抹去了。

箭头函数（Arrow Function)

箭头函数是 ECMAScript 2015（ES6）引入的一个新的函数定义方式。它的格式在视觉效果上很直观

- 不需要 function 关键字和函数名
- 用等号加大于号（ => ）分割左边的输入参数和右边的函数内容，看上去很像是从左边的输入得到
- 输入参数用圆括号括住，跟一般的函数定义一样。但是在有且只有一个输入参数的时候，可以直接
  括号。
- 函数内容（等号右边）也是用花括弧括住。但是在有且只有一句代码计算返回值的情况下，可以省
  return 。
- 箭头函数因为被大量用作回调函数（ callback ），所以通常都没有名字，当然你也可以给它用函

下面咱们看几个例子

```
setCallback1( ( err, result ) => {       // NodeJS 的标准回调函数格式
  if (err) {
    console.error(err.message);
  } else {
    console.log(result)
  }
});

setCallback2( result => result * 2);
// 等同于 setCallback2((result) => { return result * 2;});

setCallback3(() => { console.log('done!'); });   // 无输入参数时需要括号!

setCallback4(result => {});   // 函数无内容时也需要花括弧!

let sum = (number1, number 2) => number1 + number2; // 给箭头函数命名并且省略花括弧和'
```

我个人很喜欢在回调函数上使用箭头函数——除了写法更简洁，还有下面会讲到的好处。但是我并不非
花括弧这种简写。仅仅少敲了两个字符是好的，但是会降低代码可读性。尤其对新人来说，看到

```
fs.access('./secret', err => fileReady = err ? false : true);
```

这种语句还要花更多一点儿功夫来断句。

箭头函数的特点

箭头函数除了长得样子不同于一般函数之外，还有几个重要的特点。

首先，最重要的一条就是在**箭头函数范围内的 this 是由此函数定义在语句中的位置决定，而不是被调**
this 的使用不是很清楚的读者，建议你一定要去弄清楚。我们提供一个简单的例子来展现这个差别:

```
var x = 1;

var obj1 = {
        x : 0,
        value : function() {
                console.log(this.x);
```

Files

```
        }
    }

    var obj2 =  {
        x : 0,
        value : () => {
            console.log(this.x);
        }
    }

    obj1.value();           // 0
    obj2.value();           // 在非 strict 模式下: 1 ; strict 模式下: undefined
```

普通的函数是在运行时决定 this 的指向，在这个例子中运行 `obj1.value()` 时，this 就是指向 obj1 这

而箭头函数的 `this.x` 在语句中是被包含在 obj2 的定义语句中, 而 obj2 在定义中父上下文对象就是全局的浏览器里就是 window, 所以在上述例子中箭头函数中定义的 `this.x` 就是 `window.x` , 因此输出是 0。的是，一个对象的成员函数就是一个普通的函数，它的寻址空间并不一定是包含它的对象。

再比如，当箭头函数是一个对象里某句函数调用的回调函数时，箭头函数里的 `this` 总是指向这个对象，就不会因为忘了 `.bind(this)` 而出错。比如

```
// 使用箭头函数
function CheckMyFile(filename) {
  let this.fileReady;
  fs.access(filename, (err) => {
    this.fileReady = err ? false : true;      // 不论何处被调用，这个this永远指向CheckM
  });
}

// 相比较使用传统回调函数
function CheckMyFile(filename) {
  let this.fileReady;
  fs.access(filename, function (err) {
    this.fileReady = err ? false : true;
  }.bind(this));       // 很多人常常忘了 .bind(this) 而在运行时出错——更麻烦的是运行时不报错
};
```

其次，箭头函数里不能使用我们下面马上就要讲到的 `arguments` 这个标准对象。

第三，箭头函数不能做构建函数，也就是不能用关键字 `new` 来调用。

第四，跟以上三条密切相关的，箭头函数不能另外绑定 `this` 、 `arguments` 、 `super` 和 `new.target` 。 `this` ，箭头函数不能（也不需要）使用 `bind()` 方法；使用 `call()` 和 `apply()` 的时候，第一个输入参掉。

第五，箭头函数没有原型（ prototype ）。反正它不能用来做构建函数，所以这条没什么影响。

最后，显而易见地，你不能给两个或更多的输入参数赋予重复的变量名。

如果上面提到的 `new` 、构建函数、原型等概念你还不是很清楚，我们在后面的章节都会讲解。

箭头函数这些特点除了可以减少开发者的错误之外，还可以帮助 JavaScript 引擎进一步优化执行效率，讨论范围之内了。总之，如果你对箭头函数不熟悉，那你应该在回调函数里试着用起来。比如对数组进

```
let arr = [5, 8, 2, 7];
let sorted = arr.sort((a, b) => b - a);
```

generator

ES6 还引入了一种新的特殊函数，generator，用 `function*` 来定义。我认为它更适合放在类似于“ES者“JavaScript Async 编程”这样的书里跟 `iterator` 和 `Promise` 一起介绍比较好。希望我以后有机会再

输入参数

输入参数的随意性

不论是定义还是调用函数，函数名之后立刻就是输入参数了。跟很多语言不同，JavaScript 引擎并不产数，但是参数的次序还是重要的。比如

```
function multiply(first, second) {
  if (typeof second ===  'number') {   // 严格来说，此处还应该检查是正整数
```

Files

```
    switch (typeof first) {
      case 'number':
        return first * second;    //  如果两个输入参数都是数字则做乘法

      case 'string':
        return first.repeat(second);  // 如果第一个参数是字符串则复制

      default:
        return null;
    }
  } else {
    throw TypeError('Second parameter must be positive integer');
  }
}

console.log(multiply(5, 6));         // 30
console.log(multiply('ok', 3));      // 'okokok'
console.log(multiply('ok', 3, 5));   // 'okokok'，最后一个5被无视
console.log(multiply(true, 3));      // 'null'
console.log(multiply(100));          // TypeError
```

从以上的代码可以看出，你调用一个函数的时候，可以给出少于或者多于函数定义里的参数个数，也可
JavaScript 引擎并不会因为这样的调用本身而报错。当然函数里面的代码在使用这些参数的时候，还是
错。比如

```
function doubleAndCallback(value, callback) {
  console.log('Input value is: ', value);
  callback(value * 2);
}

doubleAndCallback(8, 5);      // TypeError; 5 不能作为函数被调用
```

这种输入参数的随意性有好有坏。先看好处：

- 最明显的好处是你**不需要再定义很多个名字相同、参数不同的函数来实现多态性**，一个函数就全部
- 今后再增加参数以扩展函数功能的时候，已经调用这个函数的代码也不需要修改，节省了开发成本
- 因为输入参数可以少于函数定义里的参数，你定义函数的时候可以（也应该）把必须的参数放在前
  后边。这样调用者可以更方便的根据自己的需要来决定。

坏处是：

- 因为没有编译器帮你做数据类型检查，你的函数定义内部往往要检查一下这些参数是否存在及其类
  谁会怎样调用你的函数的情况下。否则很容易出运行错误。
- 在支持多种输入数据类型的情况下，你要更加小心。比如仅仅对输入参数是数字的时候，你希望调
  是因为输入参数是字符串的时候也是调用同一个函数，你要注意避免无关的代码被影响到。在 C+
  overloading 的语言，就没有这样的顾虑。

其实你可能已经注意到了，JavaScript自带的运算符和函数里已经大量使用了这种随意性，比如

```
let a = 8 + 9;      // 17
a = 8 + '9';        // '89'
console.log(10);    // '10'
console.log('10');  // '10'
```

所以不论你喜欢不喜欢这种随意性，它都是 JavaScript 不可避免的一部分，你还是要熟练地掌握它。

缺省的输入参数

arguments 对象

除了箭头函数之外，其它“普通”的函数定义范围内，除了调用时的输入参数之外，还有一个隐含的输入
可以这样理解 `arguments`： JavaScript 引擎把所有调用一个函数时输入的参数（不管函数定义里有没
序排好，放到这个对象里。它对每个输入参数的排序很像是标准数组对象里每个成员的排序，而 Java
数组对象同名的方法让你使用这些参数。但是 `arguments` 不是标准数组对象，当然我们可以轻易地把它

```
function something() {
  console.log(typeof arguments);    // 'object'
  console.log(Array.isArray(arguments););   // 'false'
  let arr = [...arguments];         // arr is an Array
}
```

Files

JavaScript 提供这样一个对象的目的很明确：如果你的函数需要处理输入对象个数不确定的情况，可以
数的个数、遍历每个输入参数；如果你的函数没有这种需要，那你使用函数定义里的输入参数变量名就

`arguments` 对象提供的最重要的两个用法是 `.length` 和方括号索引。比如我们要计算任意个数值的和

```
function sumOfAll() {
  var result = 0,
      i = 0,
      len = arguments.length;

  while(i < len) {
    result += arguments[i];
    i++;
  }

  return result;
}

console.log(sumOfAll(5, 6, 7, 8));  // 26
console.log(sumOfAll());            // 0
```

如果一个函数定义了输入参数名，那这些输入参数其实就是按次序排列的 `arguments` 的元素。比如

```
function twoParam(first, second) {
  console.log((first === arguments[0]) + ' and ' + (second === arguments[1]));
}

twoParam('hello', 9);   // 'true and true'
twoParam(100);   // 'true and true'; second 和 arguments[1] 都是 undefined
```

函数的 length

顺便提一下，函数作为一个对象，它也有 `length` 这个成员，等于函数定义里的参数个数（对比于 `arg`
被调用时实际传入的参数个数）。比如

```
// 接上面两段代码
console.log(sumOfAll.length);   // 0
console.log(twoParam.length);   // 2
```

不过函数的 `length` 好像没什么用，至少我目前还没有机会真的在产品代码里使用。

函数重载（overloading）

前面已经提到过 JavaScript 的函数重载，这里再多啰嗦两句。看看以下的代码

```
function sayMsg(message) {
  console.log(message);
}

function sayMsg() {
  console.log('Have a nice day!')
}

sayMsg();           // Have a nice day!
sayMsg('Hello');    // Have a nice day!
```

这段代码运行不会出错，但是最后一句并没有像其它面向对象语言的函数重载那样去调用第一个函数，
数。这是因为第二个函数定义的时候，因为与第一个重名（即便参数列表不同），已经把第一个函数覆
也无法被调用了；而第二个也不检查是否有输入参数。

所以，在JavaScript里需要判断输入参数的个数、类型等来实现函数重载的效果：

```
    // 上面一段代码可以改做：
function sayMsg(message) {
  if(!message) {                  // 或者 if(arguments.length === 0) {
    message = 'Have a nice day!'
  }
  console.log(message);
}

sayMsg();           // Have a nice day!
sayMsg('Hello');    // Hello
```

Files

另外，如果你定义了一个函数并且不想被别人有意无意地覆盖了（就像以上 `function sayMsg()` 覆盖 `sayMsg(message)` 那样），一个简单的办法就是把你的函数名定义为 const

```
const sayMsg = function (message) {
  console.log(message);
}

function sayMsg() { // SyntaxError: Identifier 'sayMsg' has already been declared
  console.log('Have a nice day!')
}

sayMsg();           // Have a nice day!
sayMsg('Hello');    // Have a nice day!
```

作为对象方法（method）的函数

我们已经知道一个对象可以有任意个成员（ properties ），而每一个成员都可以是基础数据类型或者
对象在 JavaScript 里是“一等公民”，当然也可以做对象的成员。作为另一个对象成员的函数被叫做这
）。这种叫法跟其它面向对象语言一致，很容易理解。但是反之，函数作为对象也有它自己的成员、有
多语言不一样，我们下面也会遇到。

定义方法

定义一个对象的方法跟定义其它类似的对象特征没什么不同，唯一区别就是分号“:”后面跟着的是函数定

```
var person = {
  name: "老王",
  sayName: function() {
    console.log(person.name);
  }
}

person.sayName();   // 老王

person.sayAgain = function() {        // 也可以动态增减对象的方法
  console.log(person.name.repeat(2));
}

person.sayAgain();  // 老王老王
```

以上的代码虽然工作正常，但是有个很大的问题：每当方法需要用到对象的其它成员的时候，必须前缀
如果以后对象的名字改了、被复制了或者用其它方法生成新的对象，这些方法就都不工作了。⚠️ 如果
法就更不能工作了，因为它们找不到一个叫“name”的变量，而JavaScript并不会因为这是某个对象的
的同名成员拿过来用。

这个时候就要用到 `this` 这个关键字了。完全讲透 `this` 的概念大概需要另外一本书，但是在下一节我们
它。

最后需要指出的是，作为方法的函数跟其它函数没有本质区别，唯一的区别就是它被定义在一个对象内
的一个成员被调用而已。

`this` 对象

首先， `this` 也是一个对象（所以你才可以使用 `this.name` ）。在一个函数内部，它就是那个调用此函
的时候动态决定的（箭头函数除外——如果你已经忘了，请翻回去复习）。简略地讲，常见的 `this` 有

- 如果一个非方法的全局函数被调用，它的 `this` 就是“全局对象”。问题在于这个全局对象并不一定
  览器里它通常是 `window` 对象，显然在NodeJS里就不是。所以除非你很清楚知道为什么要在全局
  要用。
- 作为一个对象的方法被此对象调用，是这样的格式： `object.function(parameters)` 。这时 `func`
  面的 `object` ，所以可以用 `this.` 来获取这个对象的任何成员。这是方法里很常见的使用。
- 在回调函数（ callback ）里。既然是回调函数，调用这个函数的对象（甚至不一定是个对象）就
  能知道的。在这种情况下，显然回调函数里的 `this` 基本是不能直接使用的。但是作为一个对象方
  往往要读写此对象的成员（比如更新同一个对象的另一个成员的状态）。这时候你可以使用箭头函
  有例子），也可以使用下面的几个工具。
- 使用 `call()` 、 `apply()` 或者 `bind()` 设定 `this` 。这是我们下节要讲的内容。

Files

设定 this

如前所述，有时候你需要设定一个函数的 this 对象。虽然你不能直接写 function.this = ... ，但是
方便的几个方法（正好温习一下，函数也是对象，也有自己的方法）。⚠️这几个方法都是定义一个 Fu
JavaScript 引擎自动生成的。

call() 方法

函数本来就是被用来调用的，居然它还有一个 call 方法是有点儿奇怪的。我们可以这样理解：如果是
用这个方法；如果你特意用了 call ，那就是要更“高级”地使用这个函数了——这个高级之处，就是设
分一个函数和它所拥有的 call 方法（也是个函数），我们称这个函数为父函数。

call 的使用不复杂：因为它是父函数的方法，它要被用 .call 加在父函数名后面；因为它自己也是函
号和参数。它的第一个参数永远是父函数所需要的 this 所指的对象，其后的所有参数会被完整地按顺

```
function sayName(label) {
  console.log(`In ${label} my name is ${this.name}`);
}

var p1 = { name: 'Jack Ma' };
var p2 = { name: 'YT Jia' };

let name = 'Obama';

sayName.call(p1, 'person1');    // In person1 my name is Jack Ma
sayName.call(p2, 'person2');    // In person2 my name is YT Jia

sayName('global');              // In global my name is result (Chrome v63)
```

以上代码执行到 sayName.call(p1, 'person1'); 这句话时，你可以想象成 JavaScript 引擎先把 sayN
this 用 call 的第一个参数（也就是p1）代替，然后把第二个参数（‘person1’）传给这个替换过 thi
行。

另外，最后一句话是在最新的Chrome浏览器和NodeJS 6.x LTS里执行的结果。你可以看到‘Obama’并
到。这是因为把 this 指向全局太危险----设想你在离这句话很远的地方有个变量叫‘name’，你可能无
改变了，也没有任何报错。所以新的 JavaScript 引擎已经不给你设定全局为 this ，你代码的错误更容
现。

apply() 方法

跟 call() 非常类似的另一个方法是 apply() 。它们的唯一区别是 apply() 只接受两个参数：第一个和
所指的对象，第二个是一个数组，其成员为依次排列的父函数的输入参数。这里要注意两点：

- 即便父函数只有一个输入参数，你也要把它放到一个数组里
- 虽然你在 apply() 用的是数组作为第二个参数，父函数得到的 arguments 还是那个“类似于数组”的
  组。

下面看个简单的例子：

```
let myCalc = {
  base: 0,
  sum: function(first, second) {
    return this.base + first + second;
  }
}

let n1 = { base: 10 },
  n2 = { base: 100 };

let inputs = [2, 3];

console.log(myCalc.sum(2, 3));              // 5
console.log(myCalc.sum.apply(n1, inputs));   // 15
console.log(myCalc.sum.apply(n2, inputs));   // 105;

console.log(myCalc.sum.call(n1, ...inputs));   // 15
```

这段代码演示了你**可以把一个对象的方法作用于另一个对象上**。从这个意义上讲， apply 这个词用得还
的用法不仅说明了 call 和 apply 的相似性，而且在有了 spread operator （ ... ）之后，你可以如此
开”，以至于 apply 显得有点儿多余了。

bind() 方法

Files

在箭头函数出现之前， bind() 方法大概是这三种方法里最重要的了。它的作用跟 call 和 apply 正好
程序运行过程中 this 被动态绑定到其它对象上，而在函数定义的代码里确定一个 this ，这样就不会
对象当作 this 。另外，一个函数被 bind 了之后的结果是生成了一个新的函数，而不是被调用了。这
一样。

下面咱们来看看在对象的方法为什么需要它。

```
// 接上面的代码
myCalc.timeout = function (sec) {
  setTimeout(function () {
    console.log("Timeout!", this.base);
  }, sec * 1000);
}

myCalc.timeout(2);                  // 2秒之后： Timeout! undefined
```

这段代码的第二行里， setTimeout 这个 JavaScript 标准函数的第一个参数必须是个回调函数，这个
JavaScript 引擎调用的。所以这时，回调函数里的 this 是 JavaScript 引擎，它没有 base 这个成员，
undefined 了。其实我们想读取的是 myCalc 对象的 base 值，这时 bind 可以帮我们绑定正确的 this

```
// 上面那段代码修改之后
myCalc.timeout = function (sec) {
  setTimeout(function () {
    console.log("Timeout!", this.base);
  }.bind(this), sec * 1000);      // 加上bind，生成了一个新的回调函数
}

myCalc.timeout(2);                  // 2秒之后： Timeout! 0
```

在 .bind(this) 里， this 是指此语句所在最近范围的对象，也就是 myCalc。当 timer 到时，JavaS
函数的时候， this 就是我们期望的 myCalc 了。如何用箭头函数更简洁地实现同样的效果，就留给读

需要指出的是，如果你对 timeout 这个成员函数调用其 call 或者 apply 方法，那么它的 this 、包括
this 还是会被改变的：

```
// 接上面的代码
myCalc.timeout.apply(n1, [3]);      // 3秒之后： Timeout! 10
```

除了以上这个 bind() 最常见的用处（尤其是在箭头函数出现之前），它还可以被用来生成“部分输入参
是利用了我们前面说过的 bind() 的特性：它会创建一个新的函数，而不是立刻去调用父函数。看下面

```
// 接上面的代码
let fixedFirst = 1000;
let n1Plus1000 = myCalc.sum.bind(n1, firstFirst);
console.log(n1Plus1000(8));      // 1018 = n1.base(10) + first(1000) + second(8)
```

n1Plus1000 是一个对 myCalc.sum 用 bind 绑定了两个参数而生成的新的函数：第一个参数当然是把
个参数 1000 被绑定在 myCalc.sum 的第一个参数 first 上。所以这条语句可以这样理解：你给我一个
帮你把它的 this 定死了，把它的第一个输入参数也定死了，然后还给你一个新的函数——这个新函数
也就是 myCalc.sum 的第二个参数就够了。

然后在你调用 n1Plus1000(8) 的时候，这个函数其实是调用 myCalc.sum()并且把 this 赋值为 n1、fi
second 赋值为 8，myCalc.sum()里的语句带入变量值计算：

n1.base + first + second = 10 + 1000 + 8 = 1018

这种用法也是其它语言里比较少见的，更常见的是用一个 wrapper 带入固定的参数：

```
function sumOfThree(a, b, c) { return a + b + c; }
function sumOfTwo(a, b) { return sumOfThree(100, a, b); }
console.log(sumOfTwo(1, 2));    // 103
```

二者比较，使用 bind() 可以动态生成需要的新函数、动态绑定 this 、使用动态的固定变量，更灵活

4. 对象：深入了解

我们在第二章讲到引用数据类型的时候，已经介绍了对象。在JavaScript里对象这个概念如此重要，而
向对象编程。所以绝对值得再开辟一章，专门深入讲解它，尤其是我们如何更好地创建自己的对象。

对象的成员（Properties）

我们已经简单了解了如何定义一个对象。跟其它面向对象语言很大的一个不同就是 JavaScript 对象的
下面的例子：

```
var person1 = {
  name: "老张"
};
var person2 = new Object();
person2.name = "老李";

// 其它代码.....
person1.age = 35;
person2.age = 87;

person1.name = "张三";
person2.name = "李四";
```

person1 和 person2 都是对象、都有 name 和 age 的成员（或者叫特征值）。这些成员既可以在对象
person1 的 name），也可以创建对象之后立刻赋值（比如 person2 的 name），还可以在你运行了很
对象上。

特别需要指出的是，`var person2 = new Object();` 这句话运行之后，看似你创建了一个空的对象，
内部（你不可以调用的）和外部（你可以调用的）方法了。其中两个内部方法是 `[[Put]]` 和 `[[Set]]`
没有的成员赋值的时候（比如上面代码里 `person2.age = 87;`），`[[Put]]` 方法就会被调用：回忆我
化地想象成很多 key-value 对组成的哈希表，`[[Put]]` 的效果相当于在这个哈希表里加入一个key（"
（35）。⚠️我特意给 age 加了双引号是因为，虽然代码里 age 看上去好像一个变量名，而我们知道
引号的；但是在对象的 key-value 表里 key 永远是个字符串。

而你给一个已有的对象成员赋值的时候（比如上面的 `person1.name = "张三";`），`[[Set]]` 方法就会
的value。

当然实际上对象对其成员的存储比哈希表要复杂。以上面这样的代码生成的成员叫做"自有成员（own
员只属于这个对象，所以你必须通过这个对象来读写它、改变它。还有一类很重要的成员叫做"原型成
property）"，是我们在下一章要讨论的内容。

另外，有时候你会看到别人代码里对象成员的 key 以下划线开始，比如

```
let person1 = {
        _name: "王宝强",
        getName: function() { return this._name; }
}
```

这只是一种惯例用以标识内部使用的成员，但是并没有任何语法上的作用。也就是说，你照样可以直接
个成员。当然作为一种好的编程习惯，你应该避免这样做。关于如何封装你不想暴露出去的对象成员，
类似于 `private` 这样的关键字，但是还是有方法的，我们后面会讲到。

检查对象的成员

因为JavaScript对象的成员并不是像其它语言那样一经定义就永远存在的，所以有时候你要先检查一下
如你会看到这样的语句：

```
if (person1.name) {                 // 不可靠的检查
    console.log(person1.name);
}
```

这样写起来最简单，但是不可靠。因为我们知道 `if` 是判断后面表达式的真伪（Truthy or Falsy），而
然这个值如果不存在，JavaScript会返回 `undefined`，这的确是个伪值。但是返回伪值的情况还有其
person1.name 的值是空字符串 `""` 、0、 `false` 或者 `null` 的时候。

为此 JavaScript 提供了一个操作符 `in`。这个操作符把左边的字符串在右边的对象的全部键值（keys
找不到为伪，而不检查这个 key 对应的值。使用举例如下

```
var person1 = {
    name: "老张",
    getName: function () {
            return this.name;
    }
};
```

```
person1.age = 35;

console.log("name" in person1);     // true
console.log("age" in person1);      // true
console.log("getName" in person1);  // true
console.log("title" in person1);    // false
```

在以上的代码里，注意两点：

- 需要查询的 key 必须是个字符串或者指向字符串的变量
- 方法也是可以跟其它成员一样查询的

但是这个操作符也有个问题：

```
// 接上面的代码
console.log("toString" in person1); // true
```

显然我们并没有给 person1 定义 “toString” 这个成员。大家知道 “toString” 是 JavaScript 定义在 O
person1 只不过继承过来了它而已，所以它不是 person1 的自有成员。 往往我们只关注对象的自有成
供了一个稍微麻烦一点儿的办法：

```
// 接上面的代码
console.log(person1.hasOwnProperty("name"));        // true
console.log(person1.hasOwnProperty("getName"));     // true
console.log(person1.hasOwnProperty("title"));       // false
console.log(person1.hasOwnProperty("toString"));    // false
console.log(person1.hasOwnProperty("hasOwnProperty"));      // false
```

跟 `toString()` 一样， `hasOwnProperty()` 也是一个所有的对象都有的、从 `Object` 上继承过来的方法。
键值是不是此对象的自有成员。使用这个方法，在下一章里我们就会很容易区分自有成员和原型成员。

删除成员

对象的自有成员可以随时添加，也可以随时删除。比如你临时需要使用一个很大的数组进行一些操作，
对象里给它创建一个成员。一旦操作结束，你就可以删除这个成员以释放内存空间。仅仅把此成员的值
象哈希表里的这一项。你需要用 `delete` 操作彻底删除它：

```
// 接以上的代码
console.log(person1.age);   // 35
delete person1.age;
console.log(person1.age);   // undefined
console.log("age" in person1);      // false
```

在这个操作的背后，是对象内部的另一个标准方法 `[[Delete]]` 。它会被 `delete` 操作符调用来去除相
而如何防止有用的对象成员被删除，我们本章也会讲到。

枚举成员

你给一个对象定义的成员缺省是可以枚举（一一列举出来）的，这是因为每个成员内部都有一个 `[[Enu`
值缺省为 `true` 。（⚠️既然是内部属性，我们的代码就不能直接修改。不过下面我们会看到如何改变它

枚举使用的运算符是 `for...in` ，比如，

```
let car = {
        make: "Toyota",
        brand: "Camry",
        drive: function() { console.log("let's go!");}
}

for (let property in car) {
        console.log(`Name: ${property} -- Value: ${car[property]}`);
}

// 输出结果如下
// Name: make -- Value: Toyota
// Name: brand -- Value: Camry
// Name: drive -- Value: function () { console.log("let's go!");}
```

Files

我们看到 `for...in` 循环帮我们把 car 这个对象里我们定义的成员 key-value 表的每一个 key 放入变量
过，这个 key 是字符串，所以必须用方括号的方法读取它的值。如果试图读取 `car.property` 则会得至

`Object` 对象还提供了一个方法 `keys()`，用来把一个对象的所有成员的 key 放到一个数组里：

```
// 接上面的程序
let carKeys = Object.keys(car), len = carKeys.length;

for (var i = 0; i < len; i++) {
    console.log(`Name: ${carKeys[i]} -- Value: ${car[carKeys[i]]}`);
}
```

输出跟上一段代码是一样的。⚠️但是 `Object.keys()` 跟 `for...in` 循环其实有一个不太引人注意的区别
遍历对象的自有成员，而 `for...in` 循环把对象继承过来的成员也获取了，只要此成员的 `[[Enumerabl`

有个办法知道一个成员的 `[[Enumerable]]` 是否为 `true`： 每个对象都从 `Object` 继承过来了一个 `prop`
用以查询：

```
// 接上面的程序
if ("make" in car) {
        console.log(car.propertyIsEnumerable("make"));  // true
}
console.log("toString" in car);         // true
console.log(car.propertyIsEnumerable("toString")); // false
console.log(car.propertyIsEnumerable("propertyIsEnumerable")); // false
console.log(carKeys.propertyIsEnumerable("length")); // false
```

在以上的代码中，我们知道 car 这个对象是一定有 `toString` 这个成员的，而 `propertyIsEnumerable`
显然这个成员也是存在的。 `carKeys` 是个数组，它当然有 `length` 成员。但是所有这几个成员的 `[[Enu`
`false` 。

关于对象成员的枚举、或者叫遍历，我个人的使用经验是：

- 如果对象的创建者已经把一个成员设为不可枚举，自然有他的道理，尤其是 JavaScript 标准的内
  一个一个成员检查它的 `[[Enumerable]]`
- 普通对象的成员其实用到枚举或者叫遍历的机会不多，但是一些特殊的对象，比如 Array、Map、
- 从上面的代码大家可以看到，JavaScript提供的方法有的你必须从 `Object` 上调用（比如 `Object.`
  每一个对象上（比如 `car.propertyIsEnumerable()`）。这方面的确是有点儿混乱，给程序员添加

成员的类型

我们知道其它面向对象的语言经常把类的成员分为数据（ data member ）和函数（也就是方法，met
的（ public ）和私有的（ private ）。而 JavaScript 的方法其实就是一个对象，除了可以调用之外跟
质区别。JavaScript 对象也没有私有成员，所有成员都是公有的。

但是 JavaScript 的成员可以按另外一种方法分成两类：*data property* 和 *accessor property*。我们在
象成员都是 data property，也是增加对象成员时缺省的方法。accessor property 的使用跟 data pro
是创建它需要定义 getter 或者 setter 或者二者皆有。定义一个 accessor property 的 getter 和 sette
`get` 和 `set`，使用方法如下

```
let myNum = {
        _int: 0,

        get int() {
                console.log("myNum -> get int");
                return this._int;
        },

        set int(val) {
                // 可以在这加代码检查输入值是否为 number
                console.log(`myNum -> set int: ${val}`);
                this._int = Math.ceil(val);
        },

        get sq() {
                console.log("myNum -> get sq");
                return Math.pow(this._int, 2);
        }
}

myNum.int = 8.75;                       // myNum -> set int: 8.75
console.log(myNum["int"]);              // myNum -> get int; 9
```

Files

```
console.log(myNum.sq);          // myNum -> get sq; 81
```

以上这段代码，我们先快进到最后三行，可以看到对 *int* 和 *sq* 这两个变量的读写完全和普通的对象数
在 JavaScript 里不是关键字）。这也是 accessor property 的一个重要好处：对象的使用者不需要什
用方括号也可以用 `.` 读写成员。如果你检查它们的类型，结果也没什么特殊：

```
// 接上面的程序
console.log("int" in myNum);                // true
console.log("sq" in myNum);         // true
console.log(typeof myNum.int);              // myNum -> get int; number
console.log(typeof myNum.sq);               // myNum -> get sq; number
```

我们再来看对象定义的内部。`get` 和 `set` 语句看上去都很像是个函数定义，但是没有 `function` 关键
int 和 *sq* 虽然看上去很像是函数名，其实在使用的时候就是普通的变量名。再后面的花括弧里的语句，
或者写的时候被调用的。在定义对象的 accessor property 的时候需要注意以下几点：

- 因为 `get` 是读取数值的，它不需要输入参数，但是必须返回一个值
- 而 `set` 是赋值语句，所以必须有且只有一个输入参数，也就是给这个对象成员赋值时，等号右边
  个！）
- `get` 和 `set` 不需要都有，但是没有 `get` 当然你就不能读取（write-only），没有 `set` 你就不能赋
  你是在创建有 accessor property 的对象给其他人使用，遇到这种情况你应该提供清晰的文档，
  了。⚠️尤其是读取没有 getter 的值和在非 strict 模式下写入没有 setter 的值，程序并不会报错！
- 至于对象内部是否定义一个成员变量对应这个 accessor property 要看你的需要和设计。在我们
  映射到 myNum._int 上，而 myNum.sq 是动态计算出来的，没有内部对应值。**后一种做法更好地**
  我们已经知道，myNum._int 并没有被封装，其实也是可以从外部读写的。

成员的特性

我们在前面已经看到，对象的每个成员，即便只是个基础类型数据，也不是真的简单到只包含一个数值
JavaScript 不同于其它语言（比如 C++）的地方之一。具体来说，每个成员还有四个已经被 JavaScr
这些特性以前只是内部使用的，但是在 ES5 里它们变得可以被我们的程序使用了。它们是：

- `value`： 此成员的值，不论是基础数据类型还是对象、函数；缺省为 `undefined`
- `enumerable`： 布尔值，缺省为 `true`， 标识此成员是否可以被枚举
- `configurable`： 布尔值，缺省为 `true`， 标识此成员的特性是否可以被修改
- `writable`： 布尔值，缺省为 `true`， 标识此成员是否可以被赋以新的值（也就是它的 `value` 是否
- `value` 和 `writable` 两个特性只有 data property 才有，accessor property 没有。这是因为 acc
  是否可写都是由它的 getter 和 setter 决定的，没必要再用额外的特性标识了。

这四个特性里，第一个和第四个很容易理解。第二个 `enumerable` 我们在前面已经见过了，它如果为 `f`
在 `for...in` 循环里不会出现（这是很多标准对象自带方法的设定）。⚠️如果一个成员的 `writable` 为
模式下对其赋值，JavaScript引擎只会默默地把你赋的新值扔掉，你完全不会察觉。所以除非很特殊的
码的兼容性），一定要 `"use strict"`

比较有意思的是第三个。在英文里"configurable"是个可大可小的概念，在这里它包括这两件事：

1. 此成员是否可以用操作符 `delete` 删除
2. 此成员的除了 `value` 之外的三个特性是否可以被从 `false` 改成 `true`。这句话内容很丰富，我们把
   - `value` 是否可以赋值永远由 `writable` 的真伪决定
   - `configurable` 一旦设为 `false` 就再也没法改为 `true` 了
   - `enumerable` 和 `writable`， 在 `configurable` 为 `false` 的情况下可以由 `true` 改为 `false`，
     `true`

如果这样的解释太烧脑，你可以这样理解： `configurable` 设为 `false` 是件"开弓没有回头箭"的事，而
`{configurable: false}` 把 `enumerable` 和 `writable` 这两个本来"开弓还有回头箭"的特性也变成了"
你还是晕，我实在想不出来更通俗易懂的解释了，但是下面的代码应该会有帮助。

设定 data property 的特性

JavaScript 在 `Object` 上提供了一个方法 `defineProperty()` 来设定对象成员的特性。这个方法不是被
所以调用它的时候要提供对象的名字，再加上成员的名字和你要设定的特性。最后这个输入参数常被叫
descriptor。它就是一个简单的对象，成员是以上四个特性中的一个或者多个。下面咱们看看具体的代

Files

```
// 'use strict'
        let myNum = {
                        int: 0,
        };

        myNum.int = 10;
        console.log(myNum.int);                 // 10
        console.log(myNum.propertyIsEnumerable("int"));         // true

        Object.defineProperty(myNum, "int", {   // 这个 property descriptor 包括两个特性
                enumerable: false,
                writable: false
        });
        console.log(myNum.propertyIsEnumerable("int"));         // false

        myNum.int = 20;         // TypeError 在 strict 模式下
        console.log(myNum.int);         // 10，在非 strict 模式下赋值失败，但是没有出错!

        Object.defineProperty(myNum, "int", {
                configurable: false
        });                                             // “开弓没有回头箭”

        Object.defineProperty(myNum, "int", {
                writable: true
        });                                             // TypeError: Cannot redefine
```

以上代码的逻辑应该是比较容易看懂的：myNum.int 的 emunerable 和 writable 特性被设为 false 之举、也不可以被赋值了。而它的 configurable 特性被设为 false 之后，它的 writable 也没法被改回为

另外请注意我们使用的方法的名字是 “defineProperty” 而不是 “changeProperty”，说明这个方法是i新的成员。尤其是这个成员的后面三个属性不是缺省值的时候，这个方法还是挺好用的。比如

```
        'use strict'
        let circle = {
                r: 0
        }

        Object.defineProperty(circle, "Pi", {
                value: 3.1415926,
                enumerable: false,
                writable: false,
                configurable: false
        });

        circle.size = function() {
                return this.Pi * (Math.pow(this.r, 2));
        }

        circle.r = 15;
        console.log(circle.size());     // 706.858335

        delete circle.Pi;       // TypeError: Cannot delete property 'Pi' of #<Object
```

这里我们给对象 circle 定义了一个不可更改、不可删除、不可枚举的常量 Pi，这些特点显然是我们需要 circle.r 的 configurable 也应该设为 false 而另外两个特性为 true，因为我们不希望使用者可以删除这圆显然是没意义的。

设定 accessor property 的特性

我们前面已经说过，accessor property 没有 value 和 writable 特性。但是除了 configurable 和 enu外，它还有另外两个特性：get 和 set——它们分别指向前面见过的 `get something()` 和 `set somethi` 比前面章节用过的例子，我们来看一下如何使用它们：

```
        let myNum = {
                _int: 0,
        }

        // 下面语句中"int"就是成员的名字，也就是上面代码中 "get int() {...}"里的那个 int
        Object.defineProperty(myNum, "int", {

                // 这个 property descriptor 有四项：get, set, enumerable, configurable
                get: function() {
                        console.log("myNum -> get int");
                        return this._int;
                },
```

Files

```
        set: function(val) {
                console.log(`myNum -> set int: ${val}`);
                this._int = Math.ceil(val);
        },

        enumerable: true,
        configurable: true
});

Object.defineProperty(myNum, "sq", {

// 这个 property descriptor 只有一项：get
        get: function() {
                console.log("myNum -> get sq");
                return Math.pow(this._int, 2);
        }
});

myNum.int = 8.75;                       // myNum -> set int: 8.75
console.log(myNum["int"]);              // myNum -> get int; 9
console.log(myNum.sq);          // myNum -> get sq; 81
```

这段代码和之前讲解 accessory property 的章节的代码举例的输出是一样的。⚠️ accessor property enumerable 这两个特性缺省值是 false。比如在上一段代码里的 sq 这个成员我们没有设定 configura 果我们继续运行下面的代码就可以验证：

```
console.log("sq" in myNum);           // true
console.log(myNum.propertyIsEnumerable("sq"));        // false
delete myNum.sq;      // silently failed
console.log("sq" in myNum);   // true；sq 没有被删除
```

这种跟 data property 的不同性偶尔会造成困扰，大家使用的时候要留心。

读取特性

既然对象成员的这些特性可以设定，当然也应该可以读取。JavaScript 为此在 `Object` 上提供了一个方 `Object.getOwnPropertyDescriptor()` 。从名字就可以看出来，这个方法只能读取对象自有成员的特 要查询的对象和成员的 key，输出是一个对象，其内容跟我们之前使用的 property descriptor 一样。

```
let myNum = {
        int: 100,
}

let descriptor = Object.getOwnPropertyDescriptor(myNum, "int");

console.log(JSON.stringify(descriptor));
// {"value":100,"writable":true,"enumerable":true,"configurable":true}
```

这里读取出来的值当然都是缺省值了。

固化对象

JavaScript 对象的使用是非常灵活的。在没有设定以上特性的情况下，你不仅可以随时改变其成员的 类型、增减成员（包括方法）等等。这种灵活性有时候可以让你的代码无比强大，有时候却会给你带来 如果你做好了一个对象给别人使用，使用者拿过来却任意涂改，结果就完全不可控了。比如他把你的对 向另一个函数，或者干脆删除了这个方法，那其他用到这个对象、这个方法的人就完蛋了。在 C++ 和 的，你只能继承父类并扩展成你自己定义的子类，而不能修改父类。JavaScript 显然也需要这样的“固 JavaScript 里我们可以在三个层级上固化一个对象。从宽到严它们依次是：防止扩展、密封、冻结。

防止扩展对象

每个对象内部都有一个 `[[Extensible]]` 成员。如果它的值为 false，这个对象就再也不能增加新的成员 直接读写，但是 JavaScript 提供了这样的方法： `Object.isExtensible()` 读取这个特征， `Object.pr` 其设为 false（缺省为 true ）。⚠️ 并没有方法把它从 false 设回为 true，所以这也是一个“开弓没有回 样做是相当合理的，读者可以自行思考为什么。

下面我们看看怎么使用这两个方法

```
let myNum = {
        int: 0
```

Files

```
}

console.log(Object.isExtensible(myNum));        // true

Object.preventExtensions(myNum);
console.log(Object.isExtensible(myNum));        // false

myNum.getInt = function() { return this.int; }  // silently failed
console.log(myNum.getInt());    // TypeError: myNum.getInt is not a function
```

以上代码可以看出来 `Object.preventExtensions(myNum);` 这句话之后，我们再也不能给 myNum 添加成
`= function()...` 这句在 strict 模式下会出错。

密封对象

密封一个对象比防止扩展对象更进一步：除了把对象设为不可扩展外，它还把所有对象成员的 configu
`false` 。这个动作也是不可逆的，所以密封的对象也是不可以"解封"或者退化到仅仅是不可扩展的程度
`configurable` 为 `false` 的成员是不可删除的，也就是说密封了的对象是不能增减成员的（但是还可以
样就不会出现你辛辛苦苦做好的一个对象，某些方法被使用者不小心删除了的情况。

JavaScript 提供的密封对象和读取其密封状态的方法名字直截了当，分别是 `Object.seal()` 和 Object
只有一个输入参数，就是你关注的那个对象。下面是使用举例

```
let myNum = {
        int: 0
}

console.log(Object.isExtensible(myNum));        // true
console.log(Object.isSealed(myNum));    // false

Object.seal(myNum);
console.log(Object.isExtensible(myNum));        // false
console.log(Object.isSealed(myNum));            // true

myNum.getInt = function() { return this.int; }  // nothing happened
console.log("getInt" in myNum); // false

delete myNum.int;
console.log("int" in myNum);    // true

myNum.int = 10;
console.log(myNum.int);                 // 10
```

检查以上代码的输出，可以看出来在密封 myNum 对象之后，我们既不能增加也不是减少它的成员了。
还是可以更改的。这个状态的对象跟 C++ 和 Java 的对象是最类似的。所以如果你希望自己构建的对象
象的行为，你应该把它密封好。另外，再提醒读者一次，请使用 `use strict` ——这样别人如果试图增
会得到报错而不是悄悄地调用失败。

冻结对象

冻结对象是比密封对象更进一步：这个动作不仅密封了对象，而且连对象成员的赋值也不能改变了。这
能会怀疑这样的对象还有用吗？但是如果你需要提供一个库，这个库里有一组固定的方法封装在一个对
可以包括一些常量）是不可以被使用者改变的，那你就应该冻结它。

跟密封对象类似，对象的冻结是不可逆的，并且被冻结的对象一定都是被密封的，所以也都是不可扩展
冻结对象和读取对象冻结状态的方法也跟密封对象类似： `Object.freeze()` 和 `Object.isFrozen()` ，
是你关注的对象。下面看看如何使用它们：

```
let myCar = {
        model: 'Honda Fit',
        year: 2005,
    status: {
        mileage: 92111,
                changeOil: false
    }
}

console.log(Object.isExtensible(myCar));        // true
console.log(Object.isSealed(myCar));    // false
console.log(Object.isFrozen(myCar));    // false

Object.freeze(myCar);
console.log(Object.isExtensible(myCar));        // false
console.log(Object.isSealed(myCar));            // true
```

Files

```
console.log(Object.isFrozen(myCar));    // true

myCar.getYear = function() { return this.year; }        // nothing happened
console.log("getYear" in myCar);        // false

delete myCar.model;
console.log("model" in myCar);  // true

myCar.year = 2007;
console.log(myCar.year);                // 2005

myCar.status.mileage = 92555;
console.log(myCar.status.mileage);      // 92555
```

在这段代码里我们首先定义了 myCar 对象，它的最后一个成员也是个对象 status。在被冻结之前 my
封、未冻结。一旦被冻结，它也是密封的和不可扩展的了。我们既不能删除myCar.model 成员也不能
了。

最后两条两句显示，我们还是可以改变它成员对象所包含的成员。所以这种冻结不是一种 deep froze
是 myCar.status 的值，也就是说 myCar.status 不能再指向任何其它对象或者持有基础数据类型的值
还是个普通的对象，还可以读写、增减，除非我们也进行这样的操作： `Object.freeze(myCar.status`

5. 构建函数（Constructor）和原型（Prototype）

我刚开始接触 JavaScript 的时候，有个问题困扰了我一段时间：JavaScript 对象的概念不难理解，但
class ）呢？因为以前有 C++ 和 Java 的经验，我们已经很熟悉面向对象编程的套路：定义 interface
类 -> 实现此类的一个或者多个对象。如果没有类，难道是从一个对象复制另一个对象？（⚠️ ES6 终
是彼类，跟 C++ 或者 Java 的类不完全是一回事。待我们到第七章再分解。）

我们知道如果有了一个对象，可以把它赋值给另一个变量，也可以用 `Object.create()` 创建一个新的对
不一定是你想要的。比如看个简单的例子

```
let myCar = {
        year: 2005
}

let myCar1 = myCar;
let myCar2 = Object.create(myCar);


console.log(myCar1.year);               // 2005
console.log(myCar2.year);               // 2005
myCar.year = 2009;
console.log(myCar1.year);               // 2009
console.log(myCar2.year);               // 2009
```

在上面的例子里，不论是变量赋值的 myCar1，还是新创建的对象 myCar2，它们的 `.year` 成员都跟
变。这显然和 C++ 里由类生成的对象不一样，也让新生成的对象不好用了。myCar1 的原因容易理解，
向同一个对象的另一个变量而已；myCar2 的行为是由 `Object.create()` 这个方法决定的，本章后面我

说完不能用的，咱们来讲能用的并且应该用的方法。本章先讲使用构建函数来创建类似于 C++ 和 Java
其它方法的基础。

构建函数（constructor）

有 C++ 基础的读者都知道，在 C++ 里构建函数是一个类里跟类同名的那个成员函数，每个对象被创建
JavaScript 的构建函数虽然名字一样，但完全是另外一个概念。你要先丢掉 C++ 构建函数的概念在你
果你没学过 C++ 也许更好。

从字面上来看，JavaScript 的构建函数其实名字更贴切：**它就是那种专门用来构建对象的函数**。所以，
重用的对象，是用构建函数定义和实现的——它的作用跟 C++ 或者 Java 的类是基本相同的。

其实我们已经见过几个标准内建的构建函数，比如 `Object()` ， `Array()` 和 `Function()` 。你回忆一下
你要的对象。

⚠️ 构建函数传统上都是用大写字母开头（而其它函数、方法用小写）。虽然这不是语法限制的，但是
意义的——别人一目了然就知道你的函数是个构建函数，也就知道不应该像其它函数那样调用它了。所
你玩儿，就要遵守这个惯例。

构建函数的样子（除了名字第一个字母大写）跟其它函数没什么区别，对其内容也没什么特殊要求。比

Files

```
function Person() {
        // 暂时内容为空
}
```

我们的第一个构建函数就做好了！当然这个函数如果你就像以前一样调用： `Person();` ，什么效果也没 `undefined` 。把这个函数当作构建函数调用是有特殊语法的，必须使用关键字 `new` 。比如

```
// 接上面的代码
var person1 = new Person();
var person2 = new Person();
console.log(typeof person1);            // object; 说明构建函数返回了一个对象
console.log(person1 instanceof Person); // true; 证明 person1 是属于 Person 这一
console.log(person1 === person2);       // false; 说明两个变量不是同一个对象
```

`new` 这个关键字的意思也很容易理解，就是要"新建"。新建什么呢？当然是后面跟着的那个构建函数返 对象除了可以用 `instanceof` 来确认它的归属外，还可以检查它的 constructor。⚠️ 这个 constructo 个成员，不是构建函数。此成员就指向此对象的构建函数，所以它们叫同一个名字也有道理。比如我们

```
// 继续上面的代码
console.log(person1 instanceof Object); // true; 因为 person1 也是一个 Object
console.log(person1.constructor === Person);    // true; 精确定位对象的构建函数
console.log(person1.constructor === Object);    // false; person1并不是由 Objec

var fakePerson = Person();              // 没有 new 也可以运行...
console.log(fakePerson instanceof Object);      // false; 但是结果并不是我们想要的
```

最后两句显示如果你忘了使用 new 关键字，程序不会报错，这样的 bug 最难发现。但是如果构建函数 这个错误就更容易被肉眼或者静态代码分析的程序发现了。

构建函数的输入参数和返回值

构建函数跟其它函数一样，也可以有一个或者多个输入参数。当你用它构建对象时，通常就会输入这个 如我们下面要看到的 "name"。构建函数内部当然也可以像其它函数一样使用这些输入参数。

构建函数的输出需要注意（这里都是指被当作构建函数调用、也就是使用 `new` 关键字的情况下）有两种

- 如果此函数最后执行了返回语句 `return` 并且返回值是个对象，那这个对象就是返回值
- 如果此函数没有执行 `return` 或者 `return` 的是个基础数据类型，那就返回此函数创建的对象

第一种情况容易理解（但是比较少见）；第二种情况所谓"此函数创建的对象"到底是哪个对象？或者更 个什么样子、有什么成员呢？这是我们下一节要回答的问题。

构建对象的成员

我们知道函数也是对象，所以它也有自己的成员。在其内部，你可以用 `this.key = value;` 来定义一个 里这样被定义的成员，不论是基础数据类型还是函数、或者其它对象，就都是其新创建对象的成员了； 函数，那它就是新对象的方法。下面我们看个例子

```
function Person(name) {
        this.name = name;
        let food = "meat and vegetable";
        this.sayName = function() {
                console.log(`${this.name} eats ${food}`);
        }
}

let jack = new Person('Jack');
let jenny = new Person('Jenny');

jack.sayName();         // Jack eats meat and vegetable
jenny.sayName();                // Jenny eats meat and vegetable

console.log(p1.name);   // Jack
console.log(p1.food);   // undefined
```

跟上一版比，这一版的 `Person` 构建函数更"高级"了：它现在接受一个输入参数，"name"，并且把它赋 "name" 成员上。新对象还有一个方法叫 "sayName"，你可以在新对象建好后调用。而且我们构建的两 显然不是同一个。现在这样的对象就很像是我们在 C++ 里用类实现的对象了。

Files

⚠️ 从代码的最后一句我们可以看到，如果你在构建函数内部定义了一个变量（ food ）而没有前缀 th
函数范围内的变量而已；它可以被同命名空间内的代码使用：

```
console.log(`${this.name} eats ${food}`);
```

但是它不是新构建对象的一个成员，你也不能在构建函数外部读写它。从这个角度来说，它有些类似私

除了 data property，我们还可以在构建函数里定义 accessor property，也可以设定对象成员的特性

```
function Person(name) {
        Object.defineProperty(this, "name", { // 这个 “name" 是对象成员的 key
                get: function() {
                        return name;            // 这个 name 就是输入参数的那个 n
                },
                set: function(newName) {
                        name = newName; // 这个 name 还是输入参数的那个 name
                },
                enumerable: true,
                configurable: true
        });

        this.sayName = function() {
                console.log(this.name);         // 这个 name 是对象成员的 key
        };
}
```

这段代码里“name”比较多（JavaScript 代码里输入参数跟函数内部变量名相同的情况很多），大家不
的“name”是我们给构建的对象添加的成员的key（所以它必须是个字符串）；第四行的 name 就是输入
时候它在函数内部是个变量了，跟上一段代码里的变量 food 没什么不同，所以它也可以被当作一个“私
部使用了；在第七行里的 name 也是那个输入参数，我们在这里把它当一个内部变量使用。最后在 say
取 this.name，那当然是对象的成员才可以这么使用——这句话会调用成员 name 的 getter，返回的恰
值。如果读完我的解释你还晕，那请再读一遍，直到读懂。

判读构建函数的调用方法

前面已经说过，因为构建函数就是个普通函数，所以代码也可以不带关键字 new 来调用它。这样得到
函数创建的对象。而我们作为构建函数的定义者，往往也不希望别人这样使用它。

我们来看几个使用标准内建构建函数的例子

```
let a = new String("hello");
console.log(a instanceof String);       // true; 是一个 String 对象
a = String("hello");
console.log(typeof a);  // string; 是一个基础数据类型的 string

a = new Date();
console.log(a instanceof Date); // true; 是一个 Date 对象
a = Date();
console.log(typeof a);  // string; 又是一个 string

a = new Map();
console.log(a instanceof Map);  // true; 是一个 Map 对象
a = Map();      // TypeError: Constructor Map requires 'new'
```

我们看到对 String() 和 Date() 调用没有使用 new 都不会报错。如果说 String() 得到的结果还在
Date() 的结果就有点儿意外了。而如果调用比较新的构建函数 Map() 你忘了 new ， JavaScript 引擎
为， Map() 的做法是对的。因为绝大多数情况下，构建函数调用前没有 new ，或者是因为程序员马虎
手，对构建函数还不熟悉。如果你疏忽而忘记了 new ，然后程序像 a = Date(); 那样给你返回一个值
bug并不容易发现——你得到的结果可能并不立即使用，甚至根本不使用而传给其它模块了；你的单元
每个变量赋值都检查它的对象类型。

我们不能百分百避免马虎的错误，但是可以在自己的构建函数里实现类似于 Map() 那样的检查和报错，
发现问题。我们需要的工具是 new.target 这个成员（严格来讲 new 并不是一个对象，不过这是 JavaS
在我们讨论之列）。如果构建函数被调用的时候使用了关键字 new ，那么在构建函数内部，这个成员就
值为 undefined 。在构建函数里，通常一开始就判断 new.target 的值而决定继续执行还是报错：

```
function Person(name) {
        if (!new.target) {      // 判断 new.target 的真伪
                throw new TypeError("Constructor Person requires \'new\'");
        }
```

Files

```
        // 正常执行语句
        this.name = name;
        // ...
}

let jack = Person("Jack");      // TypeError: Constructor Person requires 'new
```

在本书后面章节，我们还会介绍另外一个方法实现同样的效果。

原型（Prototypes）

但是以上的构建函数创建的对象里藏了一个大问题。我们来看代码：

```
function Person(name) {
        this.name = name;
        let food = "meat and vegetable";
        this.sayName = function() {
                console.log(`${this.name} eats ${food}`);
        }
}

let jack = new Person('Jack');
let jenny = new Person('Jenny');

jack.sayName();         // Jack eats meat and vegetable
jenny.sayName();                // Jenny eats meat and vegetable

console.log(jack.sayName === jenny.sayName);    // false
```

以上最后一句代码的运行结果为 false，说明这两个变量不是指向同一个引用数据类型，也就是说同样
了两份。而每个函数的存储除了我们写的语句，还有它自带的各种成员，占用的内存空间并不是小到可
们在代码里定义一个“Student”构建函数，它返回的对象有十个方法。那我们创建1000个“Student”类
方法就被在内存里重复存储了1000次！在显然是不能接受的。我们需要的是 C++ 那种“数据独立、方
就是让我们定义共享的对象成员的途径。对 C++ 或者 Java 不熟悉的读者可以这样理解：原型就好比
红柿炒鸡蛋，怎么做这个菜的方法就是原型，它可以是写在菜谱上、人人都读的同一篇文章，但不是
红柿和鸡蛋做你的菜，别人做别人的。你们共享同一个菜谱，但是各有各的鸡蛋和西红柿、做出自己的

其实我们前面已经很多次使用作为原型的方法了。比如 `defineOwnProperty()` 这个方法就是定义在 `O`
的，并且可以被任何从 `Object` 继承而来的对象共享和使用。我们用代码来看一下更清楚

```
let book1 = {
        title: "JavaScript Basic"
}

console.log("title" in book1);  // true
console.log(book1.hasOwnProperty("title"));     // true; book1 可以使用 hasOwnP

console.log("hasOwnProperty" in book1); // true; hasOwnProperty 是 book1 的成员

console.log(book1.hasOwnProperty("hasOwnProperty"));    // false; 但 hasOwnPro
有的成员

console.log(Object.hasOwnProperty("hasOwnProperty"));   // false; hasOwnPrope
自有的成员

console.log(Object.prototype.hasOwnProperty("hasOwnProperty")); // true; 这下
到底是定义在哪里的

let book2 = new Object({"title": "JavaScript Advanced"});
console.log(book1.hasOwnProperty === book2.hasOwnProperty);     // true; 两个不
函数
```

仔细阅读以上的代码，你会发现：

- hasOwnProperty 是 book1 的一个方法，book1 可以调用它
- 但是它不是 book1 的自有方法；它甚至也不是 book1 的构建函数、也就是 `Object` 的一个自有成
- `Object` 有一个 key 为 "prototype" 的成员，它指向一个对象； `hasOwnProperty` 就是这个对象的
  追根溯源， `hasOwnProperty` 最初是定义在 `Object.prototype` 这个对象上的
- book2 的定义更清楚地让我们看到一个普通对象的构建函数就是 `Object()` ，而 book1 的定义方
  `Object(...)` 构建函数更常见的简化写法而已。

- 因为 book1 和 book2 都是从 `Object()` 构建出来的，所以它们共享 `Object.prototype` 提供的方
（hasOwnProperty），不需要每个对象自己存储一遍。

上面的例子和解释已经说明，像 `hasOwnProperty` 这种原型成员（ prototype property）就是我们需要
法，它的行为跟 C++ 类的方法基本是一样的。 **而原型（prototype）也是一个对象，它的成员就是所**
说，原型这个对象就是为了容纳原型成员而存在的。既然原型是个对象，它里面当然既可以有函数成员
员。但是既然面向对象的原理就是要求“数据独立、方法共享”，显然原型里主要应该是方法了。如果你
面，你要非常小心：任何一个使用它的对象都可能把共享的数值改变了（除非你把它的 writable 设为
象。

因为这是 JavaScript 面向对象编程里非常重要的概念，我希望大家一定要理解清楚。所以我们回忆一
个角度再理一遍。

我们已经知道，每个对象的成员都分为两类：自有成员（ own property ）和原型成员（ prototype p
否有某个成员可以用操作符 `in` 来检查；此成员是否为自有成员可以用方法 `hasOwnProperty` 来检查；
一个方法来检查一个成员是不是一个对象的原型成员。因为对象的任何一个成员如果不是自有成员就一
们可以容易地自己写这样一个函数：

```
// 接上面的代码
function isPrototypeProperty(object, key) {
        return key in object && !object.hasOwnProperty(key);
}

console.log(isPrototypeProperty(book1, "title"));       // false; title 是自有
console.log(isPrototypeProperty(book1, "hasOwnProperty"));      // true; hasO
```

仔细读懂上面这段小程序，你就会对成员、自有成员、原型成员三者的关系很清楚了。

[[Prototype]] 成员

我们已经看了很多使用原型成员方法的例子（所有标准内建对象的方法都是原型方法，我还没见过例外
是怎么来的、怎么使用呢？我们自己的构建函数里怎么定义它呢？我们本小节先回答第一个问题，后面
个。

内部成员我们已经见过几个。JavaScript 还给每个对象定义了一个内部成员 `[[Prototype]]` 。它是个
对象使用的原型（ prototype ）。当你使用构建函数创建一个新的对象的时候（比如上面的 book1 和
象的 `[[Prototype]]` 就自动地被指向了构建函数的 `prototype` 对象——这个步骤是 JavaScript 引擎
也不需要干预 。但是 JavaScript 提供了一个 `Object.getPrototypeOf()` 方法让你得到一个对象的原型
`[[Prototype]]` 指向的对象）：

```
// 接上面的代码
var prototype1 = Object.getPrototypeOf(book1);
var prototype2 = Object.getPrototypeOf(book2);

console.log(prototype1 === Object.prototype);   // true; book1 的原型指向构建函数
console.log(prototype1 === prototype2); // true; 两个对象的原型指向同一个对象
```

还有一个更简单的办法得到对象的原型。除了 Internet Explorer 之外的三大主流浏览器 Chrome, Fire
JavaScript 引擎（当然也包括 Node.js）都给对象添加了一个 `__proto__` 成员（"proto"前后都是双下
象的原型；换句话说，它的值跟 `Object.getPrototypeOf()` 的结果是一样的。本来这只是浏览器厂商的
（负责ECMAScript标准化的技术委员会）认为不如把它标准化，这样会避免很多兼容性问题，所以就
准。它的使用很简单：

```
// 接上面的代码
console.log(book2.__proto__ === Object.prototype);      // true; book2.__proto
```
函数的 “prototype” 成员对象

原型方法的重载

我们已经说过，一个构建函数创建的所有对象共享此构建函数的原型方法。但是如果其中一个对象需要
己的与众不同的行为，也是允许的。比如

```
// 接上面的代码
console.log(book2.toString());  // [object Object]
console.log(isPrototypeProperty(book2, "toString")); // true; toString 显然也是
```
里继承过来的原型方法

```
book2.toString = function() {   // 给 book2 定义新的 toString() 方法
        return "Book: " + this.title;
```

Files

```
        }

        console.log(book2.toString());  // Book: JavaScript Advanced
        console.log(isPrototypeProperty(book2, "toString"));    // false; 现在的 toStr
了!
```

给 book2 重载 toString() 方法很简单，跟定义其它方法没任何不同；JavaScript 引擎也不会因为你使
有的名字而出错。然后你就可以愉快地使用自己定义的方法了！

这段代码还揭示了一个重要的事实： **JavaScript 寻找对象成员的次序是先自有再原型。** `book2.toSt`
时候，JavaScript 引擎在 book2 的自有成员里找不到这个成员，就去它的 `[[Prototype]]` 成员指向
建函数 `Object()` 的 `prototype` 成员 ）里找，结果找到了就执行（再找不到还会逐级上溯——这是我
容）；如果一直都没找到，就是报错 "TypeError: object.method is not a function"。相比之下， `boo`
被调用的时候，JavaScript 引擎在 book2 的自有成员里找到了这个方法并且执行了，当然也就没原型

这时候如果你又想恢复使用原型方法（希望你不需要这么折腾），你可以删除自己定义的自有方法：

```
        // 接上面的代码
        delete book2.toString();
        console.log(book2.toString());  // [object Object]
        console.log(isPrototypeProperty(book2, "toString")); // true; toString 恢复为原
```

给构建函数添加原型成员

我们讲了半天原型是什么和怎么使用它，现在终于要开始定义原型了。如果你已经真正理解了什么是原
儿都不复杂。我们直接看代码：

```
        // 先定义一个最简单的构建函数
        function Person(name) {
                this.name = name;       // 唯一的自有成员被赋予输入参数的值
        }

        // 接着我们来检查一下这个构建函数里已经有什么了
        console.log("name" in Person);  // true; "name"显然是 Person 的成员
        console.log("prototype" in Person);     // true; JavaScript 引擎已经帮我们在 Per
"prototype" 这个成员
        console.log(typeof Person.prototype);   // object; "prototype" 就是个普通的对象

        // 现在我们往这个 Person.prototype 对象里添加一个成员，也就是一个原型方法
        Person.prototype.sayName = function() {
                console.log(`My name is ${this.name}`);
        }

        // 创建两个对象试试看
        let p1 = new Person("Jack");
        let p2 = new Person("Jenny");

        p1.sayName();           // My name is Jack
        p2.sayName();           // My name is Jenny

        // 确认一下 sayName() 真的是原型成员
        console.log(p1.hasOwnProperty("sayName"));      // false; 不是自有成员，必定是原型
```

简单来说，在你自己的构建函数里添加原型成员分三步：

1. 声明构建函数，并且把其创建对象需要的自有成员定义在函数体内部，比如上面的 `this.name` 。
   个构建函数添加一个叫做 "prototype" 的对象
2. 给上面那个 "prototype" 对象（而不是构建函数本身）添加你需要的方法，也就是 `Person.proto`
   `function() {...}` 这一步。这条赋值语句跟其它对象添加成员没任何区别。
3. 用 "new" 关键字创建新的对象，这些新的对象自然就可以使用构建函数的所有自有成员和原型成员

以上的第二步虽然并不一定非要在声明构建函数之后立刻执行，我还是建议你尽量这样做。否则不仅你
得很差，而且构建出来的对象在什么时候可以使用哪些原型方法也很头疼。偶尔你会见到有人喜欢把另
添加方法，比如

```
        console.log("double" in String.prototype);      // false; String 没有自带叫做 do

        // 我们给 String 添加一个原型方法 double，它就是把自己重复一遍
        String.prototype.double = function{ return this.repeat(2); };

        let a = "test";
        console.log(a.double());        // testtest; double 已经可以使用啦
```

Files

JavaScript 的新手请非常谨慎地使用这招，尤其是要先检查构建函数是否已经有了同名的方法、不要覆
构建函数的这种改动不仅影响到你创建的对象，也影响到别人创建的对象！老手有时候用这个办法制作
的。

另外，我们前面已经提过，原型成员不仅仅可以是方法，也可以是数据。但是这样的数据如果是引用数
那它是被所有此构建函数创建的对象共享的，所以要谨慎使用。比如对于上面 Person 那个构建函数我
员：

```
Person.prototype.city = "";
Person.prototype.schools = [];

p1.city = "铁岭";
p2.city = "沈阳";

console.log(p1.city);   // 铁岭
console.log(p2.city);   // 沈阳

p1.schools.push("铁岭一小");
p2.schools.push("沈阳二校");

console.log(p1.schools);        // ["铁岭一小", "沈阳二校"]
console.log(p2.schools);        // ["铁岭一小", "沈阳二校"]
```

`city` 因为是个基础数据类型，所以每个对象的数据结构里存的就是自己得到的赋值。而 `schools` 是引
的数据结构里存的是指向同一个数组的指针。所以你增减 p1 的 schools，p2 的也跟着变了；反之亦然
要共享的成员都挪到 Person 的函数体内部：

```
function Person(name) {
        this.name = name;
        this.city = "";
        this.schools = [];
}

// 接原型方法的定义...
```

定义构建函数的原型

我们在定义自己的构建函数的时候，它需要的原型方法往往有很多个。我们当然可以像之前的例子里
`Person.prototype.sayName = function() {...}` 那样一个一个添加。但是我们也可以把所有要定义
象里一下子赋值给构建函数的原型。比如

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

Person.prototype = {
        sayName: function() {
                console.log(`My name is ${this.name}`);
        },

        toString: function() {
                return `[Person ${this.name}]`;
        }
}
```

这样的写法要求你把所有的原型成员都放在一起，这往往是个好习惯。但是上面的代码有一个隐藏的问
时候，JavaScript 都悄悄地给它内建了一个 "constructor" 成员。构建函数的原型对象的 constructo
数的。在上面对 Person.prototype 赋值语句里，我们不是给已有的 Person.prototype 添加方法，而
对象，也就是等号右边这个对象——因为这个新的对象就是个普通的对象，它的 constructor 被指向了
一些混乱，比如

```
// 接上面的代码
let p1 = new Person("Jack");

p1.sayName();           // My name is Jack; 没什么问题
console.log(p1.toString());     // [Person Jack]; 也OK
console.log(p1.constructor === Person); // false; 这就不对了

console.log(Person.prototype.constructor === Object);   // true
console.log(p1.constructor === Object); // true
```

解决的方法也不难，直接在给 Person.prototype 赋值的对象里把 constructor 设定好（通常是放在第
了）：

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

Person.prototype = {
        constructor: Person,    // 首先设定 constructor

        sayName: function() {
                console.log(`My name is ${this.name}`);
        },

        toString: function() {
                return `[Person ${this.name}]`;
        }
};

let p1 = new Person("Jack");

p1.sayName();           // My name is Jack
console.log(p1.toString());     // [Person Jack]
console.log(p1.constructor === Person); // true；一切正常
```

对上面代码这样对构建函数的原型赋值之后，如果需要，还是可以随时加减原型成员的。比如

```
// 接上面的代码
Person.prototype.sayHi = function(yourName) {
        console.log(`Hi ${yourName}, my name is ${this.name}`);
}

p1.sayHi("Jenny");      // Hi Jenny, my name is Jack
```

构建函数、原型和对象的联系

借上面的例子，我们理一理构建函数、其原型和其构建的对象这三者是如何联系在一起的。这三个对象
别是 Person, Person.prototype, 和 p1。它们之间是靠每个对象的 [[Prototype]]、constructor、或 p
联起来的：

- Person 的 prototype 成员指向 Person.prototype 对象
- Person.prototype 的 constructor 成员指向 Person
- p1 的 [[Prototype]] 内部成员指向 Person.prototype；p1 的 constructor 成员指向 Person

弄清楚了这些关系，我们在设计构建函数及其原型，以及使用构建函数的时候就要时时提醒自己，这些
则出了 bug 很难查。

标准内建对象的原型

所有 JavaScript 标准的内建对象都自带很多方法以方便我们使用（没有这些方法，那标准对象也没存
面已经提到，这些方法基本都是原型方法。而且我们还给 String 添加了一个 double 原型方法。这样的
都是不允许的，所以对很多读者来说还比较陌生。下面咱们再看一个例子。

数组 Array 也是我们常用的标准对象。当然它也提供了很多现成的方法。但是对数组的操作可以说是无
方法不可能全部支持。比如你的程序在处理数组类型的数据时（假设都是数值），需要经常计算所有成
以定义这样一个方法，然后很方便的调用

```
if (!("average" in Array.prototype)) {  // 先确认一下没有这个原型方法
        Array.prototype.average = function() {
        if (this.length === 0) {
        return NaN;     // 如果数组为空则返回 NaN；严格来讲还应该检查每项是否为有效数值
    } else {
        return this.reduce((first, second) => first + second) / this.length;
    }
  }
}

let arr = [6, 8, 19];
console.log(arr.average());     // 11

let arr0 = [];
console.log(arr0.average());    // NaN
```

Files

⚠️再强调一遍：给标准对象添加原型方法可以谨慎地使用，但是覆盖一个已有的原型方法是非常危险的

6. 继承（Inheritance）

弄懂了原型，我们才能做好准备学习面向对象编程的另一个重要概念：继承。传统面向对象语言的继承现的。JavaScript 的类是个比较新的概念，是我们下一章的内容。在没有类的情况下 JavaScript 是如就是我们上一章讲的原型。

原型链（Prototype Chaining）

首先我们回顾一下上一章用过的简单例子和概念

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

Person.prototype = {
        constructor: Person,

        sayName: function() {
                console.log(`My name is ${this.name}`);
        },

        toString: function() {
                return `[Person ${this.name}]`;
        }
};

let p1 = new Person("Jack");

p1.sayName();           // My name is Jack
```

注意看最后一句：p1 并没有自有的方法叫做 "sayName"。当我们调用这个方法的时候，JavaScript 引没有找到，它就自动去 p1 的内部成员 `[[Prototype]]` 所指的那个对象去找；而这个对象就是 Person. 方法叫做 "sayName"。JavaScript 引擎就把这个方法拿过来用了。其实这个过程不就是对象方法的继子类里找不到一个方法，它父类里的方法就会被找到和调用，它们的这种继承关系靠的是父类-子类的 里，靠的是原型的链接。这种链接机制被叫做“原型链”（Prototype Chaining），也被叫做“原型继承 Inheritance）。

Object.prototype

既然讲原型链，那就让我们从这条链的起头开始。我们知道绝大多数对象都是从标准对象 `Object` 继承可以使用 `Object` 的原型方法。比如

```
let book = {
        title: "Good Part"
};

console.log(book.toString()); // [object Object]
console.log(book.toString === Object.prototype.toString);       // true; book
就是 Object.prototype 提供的

let bookProp = Object.getPrototypeOf(book);
console.log(bookProp === Object.prototype);     // true; bookProp 的内部 [[Pro
Object.prototype
```

Object.prototype 提供的方法

`Object` 作为 JavaScript 最基础的对象，也提供了几个最基础的原型方法。其中有的我们已经见过了：

- **hasOwnProperty()**：已经见过的，检查一个成员是不是对象的自有成员
- **propertyIsEnumerable()**：检查一个自有成员是不是可以枚举的（它的 `[[Enumerable]]` 内部特
- **isPrototypeOf()**：判断此对象是否为另一个对象的原型
- **valueOf()**：返回此对象的基础数据类型的值。这个方法听上去很多余，你也不需要使用；它主要用的
- **toString()**：见过很多次了，返回代表此对象的一个字符串

这些方法既然是原型方法，当然就是可以被其它对象继承和使用的，类似于这样：`myObj.toString()`

Files

⚠️ Object 还有很多方法不是原型方法，比如我们之前使用过的 defineProperty() 、 getPrototype 等。这些方法直接定义在 Object 上，而不是 Object.prototype 上，所以它们不会被其它对象继承。 Object. 前缀，比如 Object.freeze(myObj) 。

对象继承

我们在本章第一节里已经讲了，继承的关键是建起来原型链，让"子对象"的 [[Prototype]] 指向被继 方法建立这种原型链；本节介绍第一个，对象继承（ Object Inheritance ）；下一节介绍另一种，构 Constructor Inheritance ）。

对象继承的概念很简单：首先你已经有了一个对象（我们姑且称之为父对象），它有一些有用的成员 个新的对象（叫它子对象），并且希望子对象可以继承过来父对象的方法重用。换句话说，这是 **从对**

如果你以为以前从来没见过这样做的例子，其实是因为这件事是悄悄地发生的——你每次定义一个普通 JavaScript 都帮你做了这件事。JavaScript 做这件事使用的方法也是我们以前见过但是还不熟悉的： 先你应该注意到它不是个原型方法，所以你必须带着 Object. 来调用它。从名字就可以看出来它是用 是子对象）。它只需要两个输入参数：第一个是新对象的原型，也就是父对象；第二个可选，是子对 descriptors (我们在讲解成员特性的章节用到过)。

我们通过代码来看看

```
// 我们一般是这样定义对象的
let book = {
        title: "Good Parts"
}

// JavaScipt 引擎悄悄地把它翻译成等同于以下的代码
let book = Object.create(Object.prototype, {
        title: {
                configurable: true,
        enumerable: true,
        value: "Good Parts",
        writable: true
    }
});
```

从第二种表达形式我们可以清楚地看到，对象 book 的原型（也就是它的 [[Prototype]] 内部成员） Object.prototype （ Object.create() 的第一个输入参数)。这样 book 对象就可以使用上一小节列 原型方法了。换句话说，在本节的语义范围内，book 这个子对象的父对象是 Object.prototype （ 而

如果是我们自建的父对象，这个方法也一样好用。比如

```
let objParent = {       // 先建一个父对象
        name: "Dad",

        getName: function() {           // 它有一个我们想重用的方法
                return `my name is ${this.name}`;
        }
}

// 下面建子对象；注意Object.create()调用的第一个参数就是父对象
let objChild = Object.create(objParent, {
        name: {         // 子对象的第一个成员
                configurable: true,
                enumerable: true,
                value: "Kid",
                writable: true
        },

        age: {          // 子对象的第二个成员
                configurable: true,
                enumerable: true,
        value: 11,
        writable: true
        }
});

console.log(objChild.getName());        // my name is Kid; 子对象可以使用父对象的
console.log(objChild.age);              // 11; 也可以使用自有成员
console.log(objChild.toString());       // [object Object]

console.log(objParent.hasOwnProperty("getName"));       // true
console.log(objChild.hasOwnProperty("getName"));        // false
console.log(objParent.isPrototypeOf(objChild)); // true
```

Files

这段代码最关键的一句显然是 `let objChild = Object.create(objParent, ...)`；我们靠这句把 ob
`[[Prototype]]` 设为 objParent，所以我们才可以通过原型链调用 objParent 的getName() 这个方法

另外一句很有意思的是 `console.log(objChild.toString());`。objChild 自己并没有定义 toString()
objParent 也没有，所以这个方法不是从 objParent 继承来的。对比本小节的第一段代码我们就会发现
Object.prototype，而 toString() 正是在那里定义的。之所以 `objChild.toString()` 可以被执行而不
JavaScript 引擎对成员的搜索是沿原型链逐级上升的，直到找到结果或者到顶。

既然已经讲到 `Object.create()` 这个方法，顺便提一下，如果你想建一个没有任何原型的对象，也是可

```
let nakedObject = Object.create(null);

console.log("toString" in nakedObject); // false; nakedObject 不继承任何 Object
console.log("valueOf" in nakedObject);  // false
```

这是因为我们在 `let nakedObject = Object.create(null);` 这句里输入的参数是 `null`，所以 nakec
然也就没有任何方法被继承过来。

构建函数继承

我们已经学习了如何用构建函数创建对象，包括新创建的对象是如何继承构建函数里的方法的。但这只
形成原型链。这节我们来学习一个构建函数如何继承另一个构建函数。

在 JavaScript 里，每个函数（当然也包括构建函数）在定义的时候，JavaScript 引擎都会给它自动添
"prototype" 的成员，指向一个对象。（⚠️普通对象的定义是没有这个自动添加的成员的。）我们用代

```
let obj = {};
console.log(obj.prototype);     // undefined; 普通对象没有这个成员

function f() {}

/* JavaScript 引擎悄悄地帮你执行了以下语句
f.prototype = Object.create(Object.prototype, {
        constructor: {
                configurable: true,
                enumerable: true,
                value: f,
                writable: true
        }
});
*/

console.log(f.prototype);       // f {};
console.log(typeof f.prototype);        // object
```

这个缺省的 prototype 对象从 Object.prototype 继承而来；它里面只有一个自有成员 "constructor"
在的函数本身。读者应该还记得，在上一章里，我们就是在这个 prototype 对象里添加方法，作为构建
能继承的原型方法。

如果我们想让一个构建函数继承另外一个函数，其实方法很简单：把“子构建函数”的 prototype 指向“
对象，这样就形成了原型链。下面我们来看个具体的例子

```
// Rectangle 是个普通的构建函数
function Rectangle(length, width) {
        this.length = length;                           // 它有两个自有成员：leng
        this.width = width;
}

// 给 Rectangle 定义两个原型方法
Rectangle.prototype.getArea = function() {
        return this.length * this.width;
}

Rectangle.prototype.toString = function() {
        return `Rectangle: ${this.length} x ${this.width}`;
}

// Square 是一种特殊的 Rectangle，它需要继承自 Rectangle
function Square(length) {
        this.length = length;
        this.width = length;    // Rectangle 需要两个自有成员，否则 getArea() 就无
}

// ⚠️ 继承的关键：把 Square 的原型指向一个由 Rectangle 构建的对象
```

```
Square.prototype = new Rectangle();
Square.prototype.constructor = Square;  // 别忘了把 constructor 改过来

// 我们可以覆盖继承过来的方法
Square.prototype.toString = function() {
        return `Square with border of ${this.length}`;
}

// 两个构建函数都已经做好，可以开始使用了
var rect = new Rectangle(7, 8);
var square = new Square(9);

console.log(rect.getArea());    // 56
console.log(square.getArea());  // 81

console.log(rect.toString());   // Rectangle: 7 x 8
console.log(square.toString()); // Square with border of 9

console.log(rect instanceof Rectangle); // true
console.log(rect instanceof Object);    // true

console.log(square instanceof Square);  // true
console.log(square instanceof Rectangle);       // true
console.log(square instanceof Object);  // true
```

上面的代码很清楚，但我们还是把要点理一下：

- 既然要继承，首先要把被继承的父构建函数定义好（我们前面说过，它的所有原型方法定义最好紧
- 然后定义子构建函数。注意在此函数内部，父构建函数需要的自有成员通常都要定义，否则以后调法的时候，用到这些成员（就像 getArea() 需要 length 和 width）就会出错
- 最关键的一步：把子构建函数（ Square ）的 prototype 指向一个用父构建函数（ Rectangle ）创父构建函数本身）。这是因为这个新创建的对象的内部成员 `[[Prototype]]` 指向 Rectangle.prot成了原型链。
- 经常忽略的一步是把子构建函数的 prototype 对象的 constructor 改回到子构建函数本身（上一句constructor 显然是指向父构建函数的）
- 最后，你可以在子构建函数的 prototype 上定义新的原型方法，也可以定义同名的方法覆盖父构建

以上模式建好之后，从 Square 创建的对象（ square ）在调用一个方法的时候，就会沿着 “自有方法-> Rectangle的原型方法 -> Object的原型方法” 这个次序依次寻找同名的函数使用。驱动这个调用次序的代码也许能帮你更直观地理解这条链：

```
// 接上面的代码；以下所有语句的结果皆为 true
console.log(square.__proto__ === Square.prototype);
console.log(square.__proto__.__proto__ === Rectangle.prototype);
console.log(square.__proto__.__proto__.__proto__ === Object.prototype);
```

不需要创建对象的构建函数继承

理解了以上代码之后，细心的读者可能会问，我们在 `Square.prototype = new Rectangle();` 这句话型的对象哪儿去了？它就放在内存里并没有用、也永远不会被用到。而且由于我们并没有给它输入参数也都是 undefined。但是这些变量的内存还是被分配了，构建函数的语句还是被执行了。其实我们需要 Rectangle.prototype 这个对象。所以如果我们不希望执行父构建函数的语句而达到建立原型链的目的写为

```
Square.prototype = Object.create(Rectangle.prototype, {
        constructor: {  // 顺便把 constructor 也设好了
                configurable: true,
                enumerable: true,
                value: Square,
                writable: true
        }
});
```

这样的代码稍微复杂一些，但是执行的时候更简洁，最重要的是会避免由于没有输入参数而导致的构建用对象的大块内存。

调用父构建函数

到目前为止，我们基本上已经实现了传统面向对象语言所提供的继承机制。但是在 JavaScript 的构建 `super` 这样的对象指向父构建函数。如果在子构建函数里我们希望调用父构建函数或者它的原型方法，

Files

回想我们对函数的了解，其实不论任何函数，除了调用时处理参数不一样，还有一个关键就是当时调用 this 不一样，决定了它不一样的行为。而我们是可以通过 apply() 和 call() 这两个方法改变函数的

根据这个特点，我们从子构建函数调用父构建函数的基本思路就是把子构建函数当作 this 传入父构建构建函数及其方法当作自己的来用了。这样讲还比较抽象，咱们来看代码举例：

```
function Rectangle(length, width) {
        this.length = length;
        this.width = width;
}

Rectangle.prototype.getArea = function() {
        return this.length * this.width;
}

Rectangle.prototype.toString = function() {
        return `Rectangle: ${this.length} x ${this.width}`;
}

function Square(length) {
        Rectangle.call(this, length, length);
        // 以上语句调用 Rectangle 函数，通过 call() 把其 this 赋值为自身（也就是 Squa
        // 并且给 Rectangle 需要的两个输入参数赋值
}

Square.prototype = Object.create(Rectangle.prototype, {
        constructor: {
                configurable: true,
                enumerable: true,
                value: Square,
                writable: true
        }
});

Square.prototype.toString = function() {
        return `Square with border of ${this.length}`;
}

var square = new Square(9);
console.log(square.getArea());  // 81
console.log(square.toString()); // Square with border of 9
```

以上代码关键的一句是 Rectangle.call(this, length, length); 。如果你还记得 call() 的用法，这语句的位置， this 就是 Square 所创建的对象（在使用了关键字 new 的条件下）。我们通过 call 把行时的 this。在下面我们创建对象的时候（ var square = new Square(9); ），会在 Square() 内部调是类似于这个样子

```
Rectangle.call(square, 9, 9);
```

再深一步到 Rectangle() 内部去看，因为这时的 this 是对象 square， this.length = length; 这句就成员赋值为9。 this.width 的赋值同理。

除了赋值之外，很重要并且很常见的是在父构建函数里有一些初始化的工作。通过以上的方法我们就不一遍这些代码了。

调用父构建函数的原型方法

除了父构建函数本身，它定义好的原型方法也经常被子构建函数的原型方法调用。这是因为既然它们是多特性和行为是类似的——有其它面向对象语言基础的读者对此肯定不陌生。在 JavaScript 里实现这非常类似，还是使用 .call() 。比如我们希望 Square.toString() 能重用 Rectangle.toString() 的一些改变如下：

```
function Rectangle(length, width) {
        this.length = length;
        this.width = width;
}

Rectangle.prototype.getArea = function() {
        return this.length * this.width;
}

Rectangle.prototype.toString = function() {
        return `Rectangle: ${this.length} x ${this.width}`;
}
```

```
function Square(length) {
        Rectangle.call(this, length, length);
}

Square.prototype = Object.create(Rectangle.prototype, {
        constructor: {
                configurable: true,
                enumerable: true,
                value: Square,
                writable: true
        }
});

Square.prototype.toString = function() {
        // 调用父构建函数的原型方法，并且设定其 this 为自身
        let text = Rectangle.prototype.toString.call(this);
        return text.replace("Rectangle", "Square");
}

var square = new Square(9);
console.log(square.toString()); // Square: 9 x 9
```

在上面的代码里，我们可以方便地调用父构建函数的原型方法，唯一需要额外做的就是加后缀 `.call(t` 好像是自己的方法。

所以结论是即便没有 `super` ， JavaScript 也允许我们相当容易地得到使用 `super` 的效果。

7. 类（Class）

跟很多程序员一样，我刚开始接触 JavaScript 的时候一直有这样的疑问：居然没有"类"这么核心的概 是面向对象的语言吗？大概 TC39 听到了群众的呼声，终于在 ES6 里增加了类（ Class ）。但是这个 变化，而仅仅是语法上的美化。其实这也不奇怪，因为毕竟在没有类的日子里，我们也已经可以实现组 了。即便如此，我们还是应该熟练掌握这个新的类：不仅仅是因为它会让我们的代码更清晰、更简洁、 来越多的第三方库是用类提供的。

类定义的声明

第一个定义类的方法是声明一个类，它的样子跟声明一个对象非常类似，但是它使用关键字 `class` 开头 其中第一个一般都是 `constructor()` ；每个方法的定义都不需要关键字 `function` ，方法与方法之间也 我们定义一个简单的类

```
class Person {  // 类似于构建函数，类的第一个字母也应该大写

        // 此 constructor 作用就相当于构建函数
        constructor(name) {
                this.name = name;       // 在 constructor 里定义所有的自有成员
        }

        // 此 sayName 相当于构建函数的 Person.prototype.sayName
        sayName() {
                console.log(`My name is ${this.name}`);
        }
}

let jack = new Person("Jack");
jack.sayName(); // My name is Jack.

console.log(jack instanceof Person);    // true
console.log(jack instanceof Object);    // true

console.log(typeof Person);     // function
console.log(typeof Person.prototype.sayName);   // function
```

上面的代码里，我们首先用关键字 `class` 声明我们要定义的类，Person。Person 后面跟着花括弧，花 有方法了。

第一个方法 `constructor` 的名字是关键字；它的作用类似 C++ 的 constructor——你应该在这里完成 始化工作。但是在这之前，**你还应该在 constructor 里声明所有的自有对象**（相当于 C++ 里定义 dat 是因为在 JavaScript 的类里没有单独的地方声明自有成员，而且把这些自有成员的定义集中放在一起 读性。当你在后面的代码里 `new` 一个对象的时候（比如 `let jack = new Person("Jack");` ），这个 会被执行，它的输入参数就是你传给类的输入参数（ `"Jack"` ）。

类里面的其它方法看上去跟函数没什么不一样，并且上面的最后两句代码揭示了很有意思的事实：

1. 类（比如 Person）其实是个 function。什么样的 function 呢？其实就是个构建函数
2. 类里的方法（比如 sayName）并不是定义在类上的，而是定义在类的原型对象（ Person.prototy 个函数，虽然我们没有使用关键字 `function` 定义它

所以我们现在明白了，**类其实就是整容之后的构建函数**。当然这个“整容手术”还做了好几件有用的事：

- 类的定义不会被置顶，JavaScript 内部更像是用 `let` 而不是 `var` 来定义的这个构建函数
- 类内部的代码都是运行于 strict 模式，没有例外
- 类的所有方法都是不可遍历的；换句话说，它们的 `[[Enumerable]]` 内部特性的值都是 `false` 。这 如我们遍历一个数组的时候当然是遍历数组里的每个数值，而不是数组自带的方法
- 类内部的所有方法都不可以被当作构建函数，换句话说不能用 `new` 调用。应该没有人要这样用吧！
- 调用类创建对象的时候必须带 `new` ，否则出错。从“类就是构建函数”的角度说，做这种检查是对的 说，我认为还不如根本不需要 `new` ——既然你调用类，JavaScript 引擎直接就加上 `new` 好了，还 然现在说什么也晚了。
- 在类的方法内部试图改变类名变量的赋值会出错。

前面的五条都容易理解，最后一条有点儿绕口令。虽然我不相信你会用到，但是我们还是看个例子，把

```
class Foo {
        constructor() {
                Foo = "bar";    // Uncaught TypeError: Assignment to constant
        }
}

let foo = new Foo();
```

这里我们看到，“Foo"在类的内部是个常量，你不可以给它赋值。但是在类的外部，它是可以被赋值的 况你应该这么做）：

```
// 接上面的代码
Foo = "bar";    // OK, no error
```

类定义的表达式

跟函数类似地，类也可以用表达式来定义。既然是表达式，就是有等号的：等号左边是变量名，右边是

```
let Person = class {

        constructor(name) {
                this.name = name;       // 在 constructor 里定义所有的自有成员
        }

        sayName() {
                console.log(`My name is ${this.name}`);
        }
}
```

除了第一行，这个定义跟上一节的代码完全一样。我们也可以给等号右边的“无名”类加一个名字，但是 部使用。如果你试图用这个名字创建新的对象，程序会出错

```
let Person = class PersonClass {

        constructor(name) {
                this.name = name;       // 在 constructor 里定义所有的自有成员
        }

        sayName() {
                console.log(`My name is ${this.name}`);
        }
}

let larry = new PersonClass("Larry");   // ReferenceError: PersonClass is not
```

类也是“一等公民”

跟函数一样，在 JavaScript 里类也是“一等公民”，也可拿来像其它对象一样使用，比如作为输入参数

Files

```
function createObj(classDef) {  // 此函数的输入参数是一个类
        return new classDef();  // 返回值是用此类构建的对象
}

let obj = createObj( class {
        sayHi() {
                console.log("吃了没? ");
        }
});

obj.sayHi();    // 吃了没?
```

在 obj 的赋值语句里，我们给函数 createObj 传入一个无名的类，它只有一个方法 sayHi。createObj 这个无名类构建的对象并把它赋值给 obj。然后我们就可以正常使用它了。在实际的产品代码里，情况是道理是一样的。

单例（Singleton）

单例（ Singleton ）是一种比较特殊的面向对象编程方法。它可以理解为“跟我同类的只有我这一个对 建它的目的不是为了使用丰富的面向对象的特性，而往往是为了把相关的一些方法组织起来放在一个特 JavaScript 标准内建的 Math 对象和 JSON 对象。它们并不是为了让你用来构建一个 Math 类型或者 是提供了一组很有用的算数运算方法或者处理 JSON 对象的方法。

生成单例的方法是在定义一个类的同时就调用它创建这个对象、并且不给这个类命名，这样它就不会被 对象了。这个道理跟函数调用的 IIFE（ Immediately-Invoked Function Expression ）类似。咱们来看

```
let person = new class {

        constructor(name) {
                this.name = name;
        }

        sayName() {
                console.log(`My name is ${this.name}`);
        }
}("Obama");

person.sayName(); // My name is Obama
```

注意这段代码跟普通的类定义表达式有两点不同：

1. 在第一行的 `class` 之前用了 `new` 关键字，说明我们要调用这个类创建一个对象
2. 在类定义的最后直接跟着一个括号，括号里面是此类的 constructor 的输入参数。这种写法也明示 成对象，并且给出了参数

因为在表达式 `let person = new class {...}("Obama");` 里，等号的右边定义了一个类并且立刻调用 变量 person 被赋予的是一个此类的对象而不是一个类。并且因为刚被调用的这个类没有名字，所以以 新的对象了。由此我们得到了此类创建的唯一一个对象，也就是个单例。

在实际工作中，这个办法可以被用来封装一组相关的方法，就像 Math 一样。比如你制作了一组处理自 以把它们封装在一个叫做 “NaturalLang” 的对象里给别人使用。

在类里定义 accessor property

我们前面已经说过，对象的自有成员都应该在类的 constructor 里定义好。但是如果你需要使用 acce 以的：

```
class Person {

        constructor(name) {
                this.name = name;
                this.job = {
                        title: "unknown",
                        salary: 0
                };
        }

        get title() {
                return this.job.title;
        }

        set title(val) {
```

Files

```
        this.job.title = val;
    }

    get salary() {
        return this.job.salary;
    }

    set salary(number) {
        this.job.salary = number;
    }
}

var obama = new Person("奥巴毛");

obama.title = "总统";
obama.salary = 198888;

console.log(obama.title);       // 总统
console.log(obama.salary);      // 198888
```

静态方法

传统面向对象语言都有静态方法，也就是那些不需要创建对象就可以使用的方法。JavaScript 的构建函
法。我们先来看构建函数的静态方法

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

// 静态方法：开始计时
Person.timerStart = function() {
    console.time("Person Timer");
}

// 静态方法：停止计时
Person.timerStop = function() {
    console.timeEnd("Person Timer");
}

// 原型方法
Person.prototype.getName = function() {
    console.log(`My name is ${this.name}`);
}

Person.timerStart();    // 直接从构建函数上调用静态方法；开始计时

let p1 = new Person("Jack");
p1.getName();   // My name is Jack

Person.timerStop();     // Person Timer: 2.989013671875ms; 停止计时并输出时长
```

在以上代码中可以看到，静态方法的定义跟原型方法很像，只是在构建函数和方法名之间没有 `prototy`
决定了静态方法和原型方法完全不一样的行为：静态方法直接被构建函数调用，而不能被构建出来的对
上继承了。

在类里定义静态方法更直观，只要在方法名之前使用关键字 `static` 。上面的代码用类定义可以重写为

```
class Person {

    constructor(name) {
        this.name = name;
    }

    getName() {
        console.log(`My name is ${this.name}`);
    }

    // 静态方法
    static timerStart() {
        console.time("Person Timer");
    }

    // 静态方法
    static timerStop() {
        console.timeEnd("Person Timer");
    }
}
```

Files

```
Person.timerStart();    // 直接从构建函数上调用静态方法；开始计时

let p1 = new Person("Jack");
p1.getName();   // My name is Jack

Person.timerStop();     // Person Timer: 2.1669921875ms; 停止计时并输出时长
```

在代码里，我建议你把所有静态函数都写在紧跟 constructor 之后，或者都写在原型方法之后、类定义
两类方法可以更清楚地区分开。

类的继承

作为“美化版的构建函数”，类当然也支持继承，并且继承的语法更简洁易懂。ES6 里跟定义 `class` 关键
关键字用来支持类的继承： `extends` 和 `super` 。它们让 JavaScript 类的继承语法很类似于传统面向对

使用 extends 和 super()

我们还用讲解构建函数继承时使用的例子，把它用类的继承重写一遍

```
class Rectangle {

        constructor(length, width) {
                this.length = length;
                this.width = width;
        }

        getArea() {
                return this.length * this.width;
        }

}

// Square 用 extends 继承 Rectangle
class Square extends Rectangle {

        constructor(length) {
                // 用 super() 调用父类的 constructor
                // 等同于 Rectangle.call(this, length, length)
                super(length, length);
        }
}

var sq8 = new Square(8);

console.log(sq8.getArea());     // 64; 子类的对象调用父类定义的方法
console.log(sq8 instanceof Square);     // true
console.log(sq8 instanceof Rectangle);  // true
```

上述的代码比构建函数继承的代码明显简洁很多。 `class Square extends Rectangle {` 一句话就把我
Square.prototype 、再改变 Square.prototype.constructor 的步骤都搞定，看上去也不那么曲折了。

在子类（ Square ）的 constructor 里，我们可以方便地通过 `super()` 调用父类的 constructor 来运行
然在这里我们还可以加子类特有的逻辑）。需要注意的是，这里 **`super()` 的调用是必须的** （否则就父
了），哪怕父类的 constructor 里是空的。如果子类没有定义 constructor，JavaScript 会帮你调用父
如

```
class Square extends Rectangle {
        // no constructor here
}
```

等同于

```
class Square extends Rectangle {
        constructor(...args) {
                super(...args);
        }
}
```

从第二段代码可以看出来，JavaScript 引擎不仅帮你调用父类的 constructor，还把你传入子类的参数
除了父类需要做的初始化工作，你的子类完全没有其它逻辑需要处理，不提供子类的 constructor 其实

Files

`super()` 的使用还需要注意以下几点：

- `super()` 只能用在子类里。如果你的类不继承任何父类而你使用它，程序会出错。
- 你必须先调用 `super()`，然后才能使用 `this` 指针。这是因为 `this` 的初始化也是 `super()` 工作的
- 唯一不需要在子类的 constructor 里使用 `super()` 的情况就是子类的 constructor 直接返回一个对 见。

覆盖父类的方法

如果子类需要覆盖父类的一个方法，子类只需要定义一个同名的方法就可以了：

```
class Square extends Rectangle {
        constructor(length) {
                super(length, length);
        }

        // 定义一个跟 Rectangle 方法同名的函数
        getArea() {
                return this.length * this.length;
        }
}
```

如果你创建了一个 Square 类型的对象并且调用它的 getArea() 方法，因为 JavaScript 引擎在 Square 方法，这个方法当然就被使用，而不需要继续上溯原型链了。

调用父类的方法

我们也可以在子类的方法里调用父类的方法，这还是通过 `super` 实现的。比如

```
class Square extends Rectangle {
        constructor(length) {
                super(length, length);
        }

        getArea() {
                return super.getArea(); // 调用父类的方法
        }
}
```

继承静态方法

父类的静态方法都会被子类继承过来，而不需要任何代码（再次说明构建函数的这个“美容手术”还是很 用子类继承本章前面的带静态方法的父类，得到

```
class Person {

        constructor(name) {
                this.name = name;
        }

        // 原型方法
        getName() {
                console.log(`My name is ${this.name}`);
        }

        // 静态方法
        static timerStart() {
                console.time("Person Timer");
        }

        // 静态方法
        static timerStop() {
                console.timeEnd("Person Timer");
        }
}

class Student extends Person {

        constructor(name, id) {
                super(name);
                this.id = id;
        }

        getId() {
```

Files

```
                console.log(`My ID is ${this.id}`);
        }
}

Student.timerStart();   // 子类调用父类的静态方法

let jack = new Student("Jack", 128924);
jack.getName(); // My name is Jack
jack.getId();   // My ID is 128924

Student.timerStop();    // Person Timer: 1.991943359375ms
```

从上面的最后一句可以看出，我们运行的还是父类的静态方法。（如果你不喜欢在这里显示父类的定时
习如何带入子类的定时器名字。）

从构建函数继承

到目前为止，我们看到的都是从父类到子类的继承。我们已经知道，所谓类其实就是构建函数。而在 E
前，已经有很多构建函数存在了，当然用户不想把那些代码都用类重写一遍。JavaScript 非常强大的支
的继承！而且语法跟继承父类没什么不同。比如

```
// 假设这是一个已有的构建函数
function Rectangle(length, width) {
        this.length = length;
        this.width = width;
}

Rectangle.prototype.getArea = function() {
        return this.length * this.width;
}

// 新定义一个子类，继承上面的构建函数
class Square extends Rectangle {
        constructor(length) {
                super(length, length);
        }
}

// 使用子类创建对象
let sq7 = new Square(7);
console.log(sq7.getArea());     // 49
console.log(sq7 instanceof Rectangle);  // true
```

`extends` 关键词后面甚至可以跟一个表达式。比如上面的代码可以改为

```
// 假设这是一个已有的构建函数
function Rectangle(length, width) {
        this.length = length;
        this.width = width;
}

Rectangle.prototype.getArea = function() {
        return this.length * this.width;
}

function getParentClass() {
        return Rectangle;
}

// 新定义一个子类，继承的父类是一个函数的返回值
class Square extends getParentClass() {
        constructor(length) {
                super(length, length);
        }
}

// 使用子类创建对象
let sq7 = new Square(7);
console.log(sq7.getArea());     // 49
console.log(sq7 instanceof Rectangle);  // true
```

当然 `extends` 后面也不是什么对象都可以跟的，还有一些限制。目前你只要记住应该放父类或者构造函

混合继承

Files

“混合继承”这个词是我无奈之下发明的，因为它的英文是 "mixin"。在其它面向对象语言里，它通常被一个子类从多于一个父类里继承。这样的做法其实是把好几个父类的方法混合在一起都传承给子类，所理。

JavaScript 并不直接支持混合继承，也就是说你不能在 `extends` 后面放多于一个父类。但是我们可以数”，把几个父类混合成一个，让子类来继承。这样的代码如下

```
// 第一个需要被继承的父构建函数
let SerializableMixin = {
        serialize() {
                return JSON.stringify(this);
        }
};

// 第二个需要被继承的父构建函数
let AreaMixin = {
        getArea() {
                return this.length * this.width;
        }
}

// 关键的 mixin 函数
function mixin(...mixins) {
        let base = function() {};       // 先定义一个空函数当作构建函数的基础
        Object.assign(base.prototype, ...mixins);
        return base;
}

class Square extends mixin(SerializableMixin, AreaMixin) {
        constructor(length) {
                super();
                this.length = length;
                this.width = length;
        }
}

let sq5 = new Square(5);
console.log(sq5.getArea());     // 25
console.log(sq5.serialize());   // {"length":5,"width":5}
```

上面代码关键函数的关键语句是 `Object.assign(base.prototype, ...mixins);` 。这个 Object 的方法作为输入参数；它把第二个及之后每个对象的可以枚举的成员都复制到第一个对象上，这样就达到了混果。⚠️因为我们使用了 `extends` 实现继承，子类的 constructor 里还是要上来就调用 `super()` ，即便的代码要执行。

类里的 new.target

我们记得在构建函数里，可以用成员 `new.target` 来判断当前的调用是否有前缀 `new` 。因为类本质上也 constructor 就是执行构建函数本身的代码，所以 `new.target` 在类的 constructor 里也还是存在的。判断其真伪的逻辑已经被 JavaScript 引擎帮我们完成了。但是在有类的继承的情况下，我们还是可以者到底 new 的是哪个类：

```
class Rectangle {

        constructor(length, width) {
                console.log(`I am ${new.target === Rectangle ? "" : "not "}cr
Rectangle`);
                this.length = length;
                this.width = width;
        }

        getArea() {
                return this.length * this.width;
        }
}

class Square extends Rectangle {

        constructor(length) {
                super(length, length);
        }
}

var sq8 = new Square(8);        // I am not created as Rectangle
console.log(sq8.getArea());     // 64
```

Files

```
var rect = new Rectangle(5, 3); // I am created as Rectangle
console.log(rect.getArea());    // 15
```

在上面代码的父类的 constructor 里，我们用 `new.target === Rectangle` 来判断它是被赋值语句直接
对象, 还是被作为父类调用来生成其它子类的对象。

抽象类

以上这个用法可以让我们实现面向对象语言的另一个功能，就是抽象类。抽象类的作用是仅仅作为其它
用来直接生成对象。在 C++ 里抽象类里必须包括用关键字 `virtual` 标识的纯虚函数成员。在 JavaScr
也没有关键字 `virtual` 。但是我们可以在所谓“抽象类”里检查 `new.target` 来避免它创建对象（⚠️这里
表明在我们的设计中把此类设计成抽象类，但是 JavaScript 语言并没有定义这个概念）。比如

```
// 这里我们用 '_Abs' 前缀来表明“抽象类”，但是并没有语法上的意义
class _AbsShape {

        constructor() {
                if (new.target === _AbsShape) {
                        throw new TypeError("_AbsShape cannot be used to insta
directly.");
                }
        }

        getArea() {}
}

// 类 Circle 继承自 _AbsShape 并且不是一个“抽象类”
class Circle extends _AbsShape {

        constructor(radius) {
                super();
                this.radius = radius;
        }

        get Pi() {
                return 3.1415926;
        }

        getArea() {
                return Math.pow(this.radius, 2) * this.Pi;
        }
}

let circle5 = new Circle(5);
console.log(circle5.getArea()); // 78.539815

let shape1 = new _AbsShape();   // TypeError: _AbsShape cannot be used to inst
```

在上面代码里，虽然我们希望 _AbsShape 是个抽象类，但是因为没有任何语法的限制， `let shape1`
条语句是可以顺利执行的。不过我们在“抽象”的基类 _AbsShape 的 constructor 里判断了它是否被直
错误，由此达到了抽象类的效果。而它的子类的对象初始化（ `let circle5 = new Circle(5);` ）就没

8. Proxy

首先说明一下，本章的内容比较新，也比较抽象，算是本书的“进阶课题”吧。如果你是 JavaScript 初
一遍阅读此书的时候跳过这一章。 Proxy 和 Reflection API 被加入 ES6 的本意并不一定是更好地支持
观上它们达到了这个效果。所以我还是把它们加入此书。如果你从来没有在别人的代码里见到过使用它
成为你们团队里第一个吃螃蟹的人。

Proxy，Trap，和 Reflect 的概念

Proxy

有网络基础知识的读者对 Proxy 这个单词大概不陌生，中文叫做“代理”。在网络上它被架设在客户端和
给服务器的数据包都会被它首先收到。Proxy 可以检查这些数据包、改变其内容、拒绝它、或者把它转

JavaScript 的 `Proxy` 当然不是个网络设备，但是它的作用跟网络代理差不多：它本身是一个对象，架
目标对象，target）和使用目标对象的代码之间。它可以介入目标对象是如何被使用的；换句话说，**你
操作要先从 Proxy 过一道手** 。如果这样说还是抽象，我们来看个具体的例子

Files

```
// 最简单的 Proxy 举例：先定义一个目标对象
let target = {};

// 再用 Proxy() 构建一个 proxy 对象；Proxy() 的第一个输入参数是上面的目标对象，
// 第二个参数是另一个对象，我们称其为 handler；这里先设其成员为空
let proxy = new Proxy(target, {});

// 通过 proxy 给 target 赋值
proxy.name = "proxy";
console.log(proxy.name);        // proxy
console.log(target.name);       // proxy

// 直接给 target 赋值
target.name = "target";
console.log(proxy.name);        // target
console.log(target.name);       // target
```

在这段代码里，我们使用 `let proxy = new Proxy(target, {});` 构建了一个代理 target 对象的 `Pro`
Proxy() 调用的第一个参数就是这个 target；第二个参数也是个对象，我们叫它 handler（顾名思义它
些功能的）。现在 handler 里是空的，可以想象这个 proxy 什么都没做。或者更确切地说，它做的唯-
原原本本地转交给 target。所以接着我们给 proxy 定义新成员并赋值，其实是 target 得到了这个成员
proxy 也把 target 的成员值原封不动地返还给我们。

Trap 和 Reflect

这样的 proxy 当然还没什么用。关键是我们还没给 handler 里加东西。handler 是个对象，它的成员必
JavaScript 语言定义好的方法中的若干个。这样的方法被称作 **Trap** 。每个方法对应一种 JavaScript
操作。这些底层操作原本只在 JavaScript 引擎内部使用，ES6 把它们公开出来是希望让 JavaScript 更

全部 Trap（或者说 handler 支持的全部方法）的列表可以在 MDN 上找到。我们在本章遇到几个常用
Trap 的目的并不是要改变你给对象定义的那些方法的行为——那种改变应该通过类的继承和方法的覆
JavaScript 引擎对对象本身的操作。

`Reflect` 是 JavaScript 语言定义的另一个标准内建对象。它的任务是为上面列表中的每个 Trap 提供
成员是跟 Trap 同名的一组方法，通常参数也是一样的。它的使用不需要新建对象，直接调用 `Reflect`
以，跟我们使用 `Math` 对象是一样的。最常见的使用方法是在一个 Trap 里先完成我们需要完成的任务
后调用同名的 `Reflect` 方法把过滤过的数值传送给目标对象。下面咱们来看几个有实际意义的例子。

Proxy的使用举例

使用 `set` 过滤新的对象成员

我们知道可以随时给对象添加新的成员，而且新成员的命名（只要是字符串或者 symbol ）、赋值都是
性有的时候太宽松了。比如我们有一个对象里放的是一系列食品名和它们对应的价格，这时候我们希望
的赋值（价格）只能是数字。以前我们是没办法在添加对象成员的时候做这种限制的。阅读 MDN 的文
里有一个 `set` 方法，它恰好是对象成员赋值的 trap。（⚠️ 不要把这个 `set` 和我们以前见过的其它 `se`
参数：

- `target`： 目标对象
- `property`： 对象成员的 key
- `value`： 对象成员的赋值
- `receiver`： 收到调用的那个对象，通常就是 `Proxy` 对象本身

基于这个 trap，我们可以构建和使用 proxy 来确保添加成员的值（ value ）只能是数字：

```
// 先定义目标对象
let priceTarget = {
        category: "Food"        // 目标对象自己定义的成员不受 Proxy 的限制
}

// 再定义 proxy
let foodPrice = new Proxy(priceTarget, {

        // 在 handler 里只实现一个 trap : set
        set(target, key, value, receiver) {

                // 如果是目标对象已有的成员，则不做检查
                if (!target.hasOwnProperty(key)) {
                        if (isNaN(value)) {     // 如果输入的 value 不是数字则报错
                                throw new TypeError("Price must be a number")
                        }
```

Files

```
                }

                // 利用 Reflect 添加新的成员或赋值
                return Reflect.set(target, key, value, receiver);
        }
});

// 给 foodPrice 添加一个“合法”的成员
foodPrice.egg = 2.5;
console.log(foodPrice["egg"]);  // 2.5
console.log(priceTarget["egg"]);        // 2.5

// 给 priceTarget 已有的成员赋值
foodPrice.category = "grocery";
console.log(foodPrice.category);        // grocery
console.log(priceTarget.category);      // grocery

// 给 foodPrice(priceTarget) 添加一个“非法”的成员
foodPrice.milk = "expensive";   // TypeError: Price must be a number
```

以上代码中，我们的 proxy 被命名为 foodPrice（我个人不喜欢在此变量名里一定加上"proxy"，我更 proxy 当作一个正常的对象使用，至于它是如何做赋值检查的，那是应该被封装的对象内部逻辑）。在 `Proxy(priceTarget, {` 这行之后的嵌套有四层，大家要看仔细了：

1. 第一层是为了定义 handler 对象，里面只有一个方法 set
2. 第二层是 set 的函数内容
3. 第三层的 `if` 是先检查一下 key：如果是给目标对象已有的成员赋值则跳过下一层
4. 第四层才是我们需要的检查赋值是否为数字 `if (isNaN(value)) {` 。如果不是就报错；如果是就 法，“原来该怎么办还怎么办”。

这个例子很典型地演示了如何不通过继承而改变对象的某些行为。最后，以上这段代码其实有个漏洞， 读者烧脑吧。

使用 `get` 检查对象的 key

JavaScript 有一个常为人诟病的问题：你如果试图读取一个对象并不存在的成员，比如

```
let obj = {
        value: undefined
}

console.log(obj.value); // undefined
console.log(obj.name);  // undefined
```

大多数其它语言在执行第二句 `console.log(obj.name);` 时会出错，因为它试图读取一个不存在的变 里，最后两条语句的返回没有任何区别。这显然不太合理，但是让 JavaScript 在新版本里一下子把这 知道有多少老代码会突然死掉。

如果我们希望自己新定义的对象达到其它语言那样对未定义成员报错的效果，可以使用 Proxy 的 `get` `get` 在每次读取对象成员的值时都会被调用。 `get` 有三个输入参数：target，property，和 receiver。 里是一样的。注意 `get` 不需要输入 value，因为它本身就是要返回 value。

上面的代码可以改写成如下形式，实现对读取不存在的成员报错：

```
let objTarget = {
        value: undefined,
        date: 25
}

let obj = new Proxy(objTarget, {
        get(target, key, receiver) {
                if (key in target) {    // 在 get 里判断 key 是否存在于 target 对
                        return Reflect.get(target, key, receiver);
                } else {
                        throw new ReferenceError(`\"${key}\" not exist in obj
                }
        }
})

console.log(obj.value); // undefined
console.log(obj.date);  // 25
console.log(obj.name);  // ReferenceError: "name" not exist in obj
```

Files

以上的代码通俗易懂，就不多说了。但是要注意在 get trap 里不是可以为所欲为的，比如你不能返回一样的值。在使用之前请先认真学习文档。

防止对象的成员被删除和修改

JavaScript 除了可以随时增添对象的成员，也可以随时使用 delete 运算符删除。但是有的对象里的某没法正常工作了。比如对一个圆来说，它的半径和常量 π 是进行任何计算的基础。如果任何一个消失了会出错。下面我们来构建这样一个 proxy，它可以防止

1. 圆对象的半径或者 π 被删除
2. π 的值被修改

我们可以用已经见过的 set trap 防止对象被修改。删除对象相应的 trap 是 deleteProperty 。 delet入参数：目标对象 target 和要被删除的成员键值 key。我们可以判断 key 是不是不可删除的成员；如

```
let circleTarget = {
        radius: 0,
        Pi: 3.1415926,
        getArea() {
                return Math.pow(this.radius, 2) * this.Pi;
        }
};

let circle = new Proxy(circleTarget, {
        // 使用 set trap 防止常量 Pi 被赋值
        set(target, key, value, receiver) {
                if (key === "Pi") {
                        throw TypeError("Pi cannot be changed");
                } else {
                        Reflect.set(target, key, value, receiver);
                }
        },

// 使用 deleteProperty trap 避免 Pi 或者 radius 被删除
deleteProperty(target, key) {
        let undeletableKeys = new Set(["Pi", "radius"]);
        if (undeletableKeys.has(key)) {
                throw Error("Pi and radius cannot be deleted");
        } else {
                delete target[key];
        }
}
});

circle.radius = 10;     // radius 是可以变化的
console.log(circle.radius);     // 10
console.log(circle.getArea());  // 314.15926
console.log(circle.Pi);         // 3.1415926

circle.color = "red";   // 增加新成员，没有触发任何 trap
delete circle.color;    // 成员 color 可以被删除

circle.Pi = 3.14;       // TypeError: Pi cannot be changed
delete circle.radius;   // Error: Pi and radius cannot be deleted
```

当程序运行到 circle.Pi = 3.14; 这句时， set(target, key, value, receiver) {...} 这个 trap 被更改的成员是 Pi，就会抛出一个 TypeError。

假设没有上一句，或者它抛出的 TypeError 被 catch 住了，下面一句 delete circle.radius; 就会被一个 trap deleteProperty(target, key) {...} 。在这个函数里，我们先把所有不可删除的成员的键样以后可以很容易地扩展需要保护的成员列表。然后我们在 Set 对象里搜索输入的 key，如果找到了也

使用 ownKeys Trap 隐藏对象成员

我们知道 JavaScript 的对象没有所谓私有成员，而且在第四章我们还学习过一个方法 Object.keys() [[Enumerable]] 特性为 true 的对象成员。如果你做好了一个对象提供给别人，但是其中有些成员其实哪怕你没有在文档是提及这些“内部成员”，别人还是可以用 Object.keys() 或者类似的方法发现它们。 Object.defineProperty 一一把这些成员的 [[Enumerable]] 设为 false 之外，我们还有一个叫做 own

ownKeys 会在程序调用以下五个 Object 的方法时被触发：

- Object.keys()
- Object.getOwnPropertyNames() ： 这个功能跟 keys() 非常类似
- Object.getOwnPropertySymbols() ： 这个用于成员的 key 是 symbol 的时候（Symbol 以后放到

Files

- `Object.assign()`：因为 `assign()` 要把一个对象的全部自有成员拷贝到另一个对象上，所以它需有自有成员的键值
- 在 `for (key in object) { ... }` 循环里

`ownKeys()` 的输入只有一个，就是目标对象。`ownKeys` 的返回值必须是一个数组，数组的内容就是你举的那些键值。如果你不想对这些键值做任何过滤，你也可以直接返回 `Reflect.ownKeys(target)`。

下面我们用一个例子看看如何"半隐藏"对象的特定成员。我们以前提及过，对象内部使用的成员的命名但是这只是一种习惯，并没有语法上的意义。用以下的 proxy 可以从逻辑上把以下划线开头的自有成员起来

```
// 先定义一个函数，用来生成会隐藏以"_"开头的成员的 proxy
function hideUnderscore(targetObj) {            // 输入是目标对象
        return new Proxy(targetObj, {   // 输出是 proxy
                ownKeys(target) {
                        // 先调用 Reflect.ownKeys() 得到缺省的键值列表
                        // 再用标准方法 filter() 过滤掉类型是 string 并且以下划线开头
                        return Reflect.ownKeys(target).filter(key => {
                                return typeof key !== "string" || key[0] !== '
                        });
                }
        });
}

// 我们的目标对象
let studentTarget = {
        name: "Jack",
        grade: 7,
        _age: 12,
        _id: 44396
}

let student = hideUnderscore(studentTarget);    // 生成 proxy

let names = Object.getOwnPropertyNames(student);        // 通过 proxy 获取成员列
let keys = Object.keys(student);
// 通过 proxy 获取键值列表

console.log(names.length);      // 2；只返回两个成员
console.log(names);     // [ 'name', 'grade' ]；以下划线开头的成员不见了

console.log(keys.length);       // 2
console.log(keys);      // [ 'name', 'grade' ]
```

跟使用 `Object.defineProperty` 把每一个下划线开头的成员的 `[[Enumerable]]` 特性设为 false 相比，更有扩展性。如果你想在目标对象里新增一个不可以被枚举的对象，只要命名时加上下划线前缀就好了

但是需要指出的是，以上的方法只是"半隐藏"对象的成员。下划线开头的成员虽然不能被枚举，但是还

```
// 接上面的代码
console.log(student._id);       // 44396
```

使用 `get` 和 `set` trap 阻止这样的读写会有一些意想不到的副作用。在下一章里我们会给出一个完全隐

函数和类的 Proxy

到目前为止我们看到的都是如何使用普通对象的 proxy。函数，包括构建函数和类，作为一类特殊的对的 trap。本节我们就看几个这样的例子。

apply 和 construct

我们知道，函数跟其它对象不同之处就在于它可以被调用。准确来说，它可以被以两种方法调用：不带（也就是当作构建函数）。这两种方法在函数对象里对应两个内部成员，`[[Call]]` 和 `[[Construct]]` 这两个内部成员的）。JavaScript 分别给他们提供了两个 trap：`apply` 和 `construct`。换句话说，一用的时候，此函数的proxy里的 `apply` 这个 trap 会被触发；如果带了 `new`，则是 `construct` 被触发。

`apply()` 及其对应的 `Reflect.apply()` 有三个输入参数：

1. target：目标函数
2. thisArg：当前调用的 `this` 的值
3. argumentsList：输入参数（数组）

Files

`construct()` 及其对应的 `Reflect.construct()` 有三个输入参数：

1. target：目标函数
2. argumentsList：输入参数（数组）
3. newTarget：我们以前讲到构造函数时用过的 `new.target` 变量

最后一个参数对 `Reflect.construct()` 是可选项。

下面我们先看一个不是构建函数的例子

```
// 函数 sum 把它的所有输入参数相加；如果每个参数都是数值则求算术和；
// 如果遇到一个字符串则从那个字符串开始变为字符串相连
function sum (...args) {
        return args.reduce((previous, current) => previous + current, 0);
}

console.log(sum(1, 2, 3));      // 6
console.log(sum(1, 2, '3'));    // '33'

// 构建一个 proxy；它只允许输入任意个数值，并且用上面的函数进行算术求和
// 并且，因为这不是一个构建函数，不允许使用"new"
let sumNumber = new Proxy(sum, {

        // 在没有 “new” 调用的时候
        apply(target, thisArg, argumentList) {
                argumentList.forEach( arg => {  // 检查每个输入参数的类型
                        if (typeof arg !== "number") {
                                throw new TypeError(`\"${arg}\" is not a numb
                        }
                });

                // 如果没有出错则调用目标函数进行计算
                return Reflect.apply(target, thisArg, argumentList);
        },

        // 如果调用带 “new” 则报错
        construct(target, argumentList, newTarget) {
                throw new TypeError("This function can't be called with new."
        }
});

console.log(sumNumber(1, 2, 3));        // 6

console.log(sumNumber(1, 2, "3"));      // TypeError: "3" is not a number

let total = new sumNumber(1, 2, 3);     // TypeError: This function can't be
```

上面例子的效果有点像是个“函数的继承”：我们从一个通用的“sum”生成一个只针对数值的“sum”。这
类的定义上。

类的 Proxy

我们已经讲过，JavaScript 的类本质就是个构建函数，不过被进行了一些特殊处理使其更适合用于创建
最明显的一个就是你必须使用 `new` 来调用它，否则出错。这个行为的实现很类似于我们在上一节的举例
trap，只不过是同样的语句换到了 `apply()` 里。因为 Proxy 提供的是一些 JavaScript 更内部的方法，
`new` 而调用类的情况。跟其它函数一样，没有 `new` 的调用一个类会触发 `apply()` 这个 trap。（可以想
Reflect.apply() 逻辑必定是抛出错误。）下面我们看看怎么在具体代码里使用类的 Proxy。

假设已经有一个类 Person，生成的对象有两个成员：name 和 birthYear。现在我们要构造这个类的 P
人的对象，其出生年份必须在1957年前。另外，如果调用这个 Proxy 的代码忘记了使用 `new` ，我们也
为既然已经是调用类，当然就是要返回对象）而不报错。

```
// 先构造一个普通的类
class Person {
        constructor(name, birthYear) {
                // 检查一下输入参数的类型
                if (typeof name !== "string" || typeof birthYear !== "number"
                        throw TypeError("Name must be string and year must be
                } else  {
                        this.name = name;
                        this.birthYear = birthYear;
                }
        }
}

// 构造 Person 的 proxy
```

Files

```
let Retired = new Proxy(Person, {
        // 如果没有 'new'...
        apply(target, thisArg, argumentList) {
                return new Retired(...argumentList);
                // 注意这里返回的是 new Retired()，因为输入参数还要通过下一个 trap 的
        },

        // 如果有`'new' 则检查输入的出生年份
        construct(target, argumentList, newTarget) {
                if (argumentList[1] < 1957) {
                        return Reflect.construct(target, argumentList);
                } else  {
                        throw TypeError("birthYear must be before 1957");
                }
        }
});

// Retired 可以当作一个新的类来使用
let retired = Retired("老张", 1955);     // 没问题，返回一个对象
let zhang = new Person("小张", 1995);    // 没问题，因为 Person.constructor() 不
let young = Retired("小张", 1995);       // TypeError: birthYear must be befor
```

我们可以看到，除了对忘记 new 的调用更友善之外，上面的代码还从通用的 Person 类“继承”为一个更
当然这不是真正的继承，但是如果你希望对处理的数据有所限制的时候，这种方法还是很好用的。

9. 编程攻略

本书前面所举的很多例子，虽然短小，但是我们力图做到有实用价值。很多对象的构造举例，希望你改
的成员和方法，就可以用到自己的代码里。在这一章里，我们不再讲解新的语言概念，而是重点介绍广
面向对象编程的攻略。

对象成员的封装

前面已经提到过，JavaScript 面向对象编程最为人诟病的大概就是没有 private 关键字，对象的成员
个问题并不是完全无解的。

利用 IIFE 封装对象的私有成员

IIFE（Immediately-Invoked Function Expression）在 JavaScript 里是一个常用的技巧，往往被用
辑。因为 IIFE 里的匿名函数的变量仅存在于它自己的函数范围命名空间内，我们可以把需要隐藏的变量
而把对象的公共成员从这个返回，看上去就是这样：

```
var myObj = (function() {
        // 在这里定义对象的私有

        return {        // 返回给 myObj 的结果对象
                // 这里定义对象的公共成员
        };
}());
```

在上面的伪码中，匿名函数返回的对象会赋值给 myObj，当然里面的对象都是 myObj 可以读写的。而
的函数内部变量，显然是在匿名函数外部无法获取的，当然也不是 myObj 的成员。关键在于，这些**内
语句返回的对象里的方法读写，所以它们的效果等同于对象的私有成员**。

来看个以前用过的例子：架设我们要构建一个 person 对象。显然一个人的名字应该是可以改的，但是
改，而他的年龄是当前年份减去出生年份。假定我们的设计要求当前年份也不可以直接读写，但是可以

```
let person = (function () {
        const birthYear = 1990;
        let currentYear = 2017;

        return {
                name: "jack",

                older: function() {
                        currentYear++;
                },

                get age() {
                return currentYear - birthYear;
                }
        };
```

Files

```
        }());

        console.log(`My name is ${person.name} and age is ${person.age}`);      // My
age is 27
        person.name = "josh";
        person.age = 15;
        console.log(`My name is ${person.name} and age is ${person.age}`);      // My
age is 27; name 可以改而 age 不可以改

        person.birthYear = 1980;
        console.log(`My age is ${person.age}`); // My age is 27; 内部变量 birthYear 不能

        person.currentYear = 2027;
        console.log(`My age is ${person.age}`); // My age is 27; 内部变量 currentYear 也

        person.older(); // older() 方法可以写 currentYear 内部变量
        console.log(`My age is ${person.age}`); // My age is 28
```

在上面的代码里，我们用 IIFE 构造了一个对象并将它返回。IIFE的匿名函数里有两个内部变量：birthY
名函数返回的对象里有一个数据成员 name、一个普通的方法 older() 和一个 accessor property "age
currentYear 赋值，而 age() 需要读取 birthYear 和 currentYear。这两个内部变量都不可以通过最后
而 birthYear 由于不可以被改变，我们直接把它定义为 `const` 。

有的程序员喜欢把尽量多的逻辑写在 IIFE 返回的对象之外，而在返回对象里对每个成员再赋值，相当于

```
        let person = (function () {
                const birthYear = 1990;
                let currentYear = 2017;

                function older() {
                        currentYear++;
                }

                return {
                        name: "jack",

                        older: older,

                        get age() {
                        return currentYear - birthYear;
                        }
                };
        }());
```

这样会让返回对象的成员看得更清楚，更像其它面向对象语言的 interface 定义。

利用构造函数隐藏对象成员

以上的方法虽然不错，但是有个问题。因为 IIFE 只运行一次，所以也只生成一个对象。如果我们要构建
每个对象有不同的参数怎么办?

我们可以把上面例子的 IIFE 打开并且给匿名函数命名，这样其实就是一个构建函数了：

```
        function Person(name, birthYear) {
                let currentYear = 2017;

                function older() {
                        currentYear++;
                }

                return {
                        name: name,
                        older: older,
                        get age() {
                        return currentYear - birthYear;;
                        }
                };
        }

        let p1 = new Person("jack", 1985);
        console.log(`p1\'s name is ${p1.name} and age is ${p1.age}`);   // p1's name
32

        p1.name = "josh";
        p1.age = 40;
        console.log(`p1\'s name is ${p1.name} and age is ${p1.age}`);   //p1's name is
32。可以改 name 但是不能改 age
```

Files

```
        p1.birthYear = 1980;
        console.log(`p1\'s age is ${p1.age}`);  // p1's age is 32

        p1.currentYear = 2019;
        console.log(`p1\'s age is ${p1.age}`);  // p1's age is 32

        p1.older();     // p1 不可以读写 currentYear，但是 p1.older() 方法可以
        console.log(`p1\'s age is ${p1.age}`);  // p1's age is 33

        let p2 = new Person("mike", 1995);      //
        p2.older();
        console.log(`p2\'s age is ${p2.age}`);  // p2's age is 23

        console.log(p1.older === p2.older);     // false
```

以上的构建函数不能返回缺省的对象，而必须在函数内部构建一个对象返回。这样的代码满足了本小节
来了新的问题：最后一句的输出说明此构建函数创建的两个对象并没有共享一个方法的存储。这并不奇
older() 并不是一个原型方法。如果你在 Person() 的定义之后再定义一些原型方法，它们是无法读写 P
量的。

另外，我们看 currentYear 这个成员，它对所有的对象显然应该是同一个数值。换句话说，它应该是个
修改。下面我们看看如何用 IIFE 生成一个构建函数，它既有原型方法也有静态方法。

```
        var Person = (function() {
                let currentYear = 2017;         // 静态“成员”，其实是匿名函数的内部变量

            function _Person(name, birthYear) { // 在匿名函数里定义一个构建函数
                this.name = name;       // 对象的普通成员

                        Object.defineProperty(this, "birthYear", {
                                        get: function() {       // 只读成员
                                                return birthYear;
                                        },
                                        enumerable: true,
                                        configurable: true
                        });
            }

            _Person.older = function() {        // 对象的的静态方法
                                currentYear++;
                        };

            _Person.prototype.getAge = function () {    // 原型方法
                return currentYear - this.birthYear;
            }

                return _Person;
        }());

        let p1 = new Person("jack", 1985);
        console.log(`p1\'s name is ${p1.name} and age is ${p1.getAge()}`);      // p1
age is 32

        p1.birthYear = 1980;
        console.log(`p1\'s age is ${p1.getAge()}`);     // p1's age is 32; 结果没有变化

        p1.currentYear = 2019;
        console.log(`p1\'s age is ${p1.getAge()}`);     // p1's age is 32; 结果没有变化

        let p2 = new Person("mike", 1995);      //
        console.log(`p2\'s age is ${p2.getAge()}`);     // p2's age is 22

        Person.older(); // 调用静态方法
        console.log(`p1\'s age is ${p1.getAge()}`);     // p1's age is 33
        console.log(`p2\'s age is ${p2.getAge()}`);     // p2's age is 23; p1 和 p2 的
```

在这个例子里我们使用了一个 IIFE。它的匿名函数里定义了以下内容：

- 本地变量 currentYear： 因为匿名函数只被调用了一次，所以这个变量在内存里只有一份；换句话
数生成的所有对象共享的，效果相当于“静态成员”
- 构建函数 _Person： 这是我们最后要返回的结果，里面定义了一个普通成员 name 和一个只读成
- 构建函数的静态方法 _Person.older()
- 构建函数的原型方法 _Person.prototype.getAge()： 它的计算需要用到对象的自有成员 birthYea
currentYear

Files

使用这个 IIFE 生成的构建函数，我们可以创建两个对象 p1 和 p2。试图通过 p1 去修改 birthYear 或者
的，只有通过静态方法 `Person.older();` 才可以递增 currentYear，从而递增所有 p1 和 p2 的 getAg

⚠️ 上面代码有个小问题：因为每个对象的 birthYear 是不能共享的，而且我们在原型方法 getAge() 里
以我们必须把它设为只读成员。如果你要求这个成员完全不可以通过对象直接读写，那上面的代码还是
JavaScript 还是没有一种机制让对象的某个成员只允许由其方法读写，而不能由使用对象的代码直接读

Mixin

不同于构建函数或者类的继承， mixin 的目的是把其它对象（我们称之为供应对象 supplier）的成员借
使用的对象（称之为接收对象 receiver ）上，但是不改变接收对象的原型。它不是 JavaScript 正规定
对它有争议，比如 React 的开发者就认为它是有害的，并且要彻底弃用它。但是毕竟还是有很多人在使
量的代码里。不论你是否直接使用它，间接上你很难避免遇到它，比如在第三方的库里。我们在这里不
讲清楚它是怎么工作的，由读者自行判断在你的代码里如何使用。

我们前面已经见过通过调用 `Object.assign()` 方法的 mixin 例子，这里再看另一个途径。先定义一个

```
        function mixin(receiver, supplier) {
                for (var property in supplier) {
                        if (supplier.hasOwnProperty(property) && !(property in receiv
                                receiver[property] = supplier[property];        // 注意
receiver.property = supplier.property!
                        }
                }

                return receiver;
        }
```

这个函数有两个输入参数，分别是 receiver 和 supplier 这两个对象。它的内部逻辑很简单：遍历 sup
如果 receiver 里没有这个成员（如果你希望用 supplier 的同名成员覆盖它，就不做这个判断），就把
⚠️ 这个拷贝是 shallow copy，也就是说如果拷贝的是 supplier 的一个对象成员，那并没有在 receiv
而是在原有对象上新加了一个指针。

咱们把这个函数放到一个具体的例子里来看看。假设你做了一个名为 Circle 的构建函数，它生成的对象
和 color，和一个原型方法 sayName()。使用一段时间后你需要给它添加计算面积和周长的方法。这时
一个有这种运算功能的构建函数 CalcCircleArea。当然我们要尽量重用已有的代码，下面的代码让你可
法：

```
        // 上面代码里的 mixin 函数
        function mixin(receiver, supplier) {
                for (var property in supplier) {
                        if (supplier.hasOwnProperty(property) && !(property in receiv
                                receiver[property] = supplier[property];
                        }
                }
                return receiver;
        }

        // ColorCircle 构建函数
        function ColorCircle(radius, color) {
                this.radius = radius;
                this.color = color;
        }

        ColorCircle.prototype.sayName = function() {
                console.log(`My radius is ${this.radius} and my color is ${this.color
        }

        // CalcCircleArea 构建函数
        function CalcCircleArea(radius) {
                this.radius = radius;
                this.Pi = 3.1415926;

                this.getArea = function() {
                if(typeof this.radius === 'number') {
                        return Math.pow(this.radius, 2) * this.Pi;
                } else {
                        return NaN;
                        }
                };

        this.getCircumference = function() {
                        if(typeof this.radius === 'number') {
                        return  2 * this.radius * this.Pi;
                } else {
```

Files

```
                return NaN;
        }
};
}

// 注意，这里的 receiver 是 ColorCircle.prototype,
// 而 supplier 是一个用 CalcCircleArea() 创建的对象。
mixin(ColorCircle.prototype, new CalcCircleArea());

let colorCircle = new ColorCircle(3, "yellow"); // 用 mixin 之后的构建函数创建对象
colorCircle.sayName();  // My radius is 3 and my color is yellow；原本的方法还可
console.log(colorCircle.getArea());     // 28.2743334；新的方法也可以使用了
console.log(colorCircle.getCircumference());    // 18.849555600000002
```

再来看个稍微变化的情况。架设 ColorCircle 是别人已经定义好、大家共用的构建函数，你不想改变它 ColorCircle 创建的一个对象里，你需要计算面积和周长的方法。上面的代码可以修改如下

```
function mixin(receiver, supplier) {
        for (var property in supplier) {
                if (supplier.hasOwnProperty(property) && !(property in receiv
                        receiver[property] = supplier[property];
                }
        }
        return receiver;
}

function ColorCircle(radius, color) {
        this.radius = radius;
        this.color = color;
}

ColorCircle.prototype.sayName = function() {
        console.log(`My radius is ${this.radius} and my color is ${this.color
}

function CalcCircleArea(radius) {
        this.radius = radius;
        this.Pi = 3.1415926;

        this.getArea = function() {
        if(typeof this.radius === 'number') {
                return Math.pow(this.radius, 2) * this.Pi;
        } else {
                return NaN;
                }
        };

this.getCircumference = function() {
                if(typeof this.radius === 'number') {
                return  2 * this.radius * this.Pi;
        } else {
                return NaN;
        }
};
}

// 使用 mixin 函数直接返回一个“混合”好的对象
let colorCircle = mixin(new ColorCircle(3, "yellow"), new CalcCircleArea());

colorCircle.sayName();  // My radius is 3 and my color is yellow
console.log(colorCircle.getArea());     // 28.2743334

    let cc2 = new ColorCircle(5, "green");
cc2.sayName();      // My radius is 5 and my color is green
console.log(cc2.getArea()); // TypeError: cc2.getArea is not a function
```

这段代码的 mixin() 函数和两个构建函数都跟前一段完全相同。但是我们在 mixin() 的参数上稍加调整 的。因为我们并没有改造 ColorCircle 这个构建函数，它创建出来的其它对象（ cc2 ）当然也不可以使

Scope-Safe 构建函数

我们在前面章节已经讲过构建函数如果被不带 new 前缀调用时候的一些问题，以及如何用 new.target 缀。这一节我们再拓展一下这个话题，并且演示另一个解决问题的方法。先看下面最简单的例子

```
// 运行于非 strict 模式下
var name = "Jack";
```

Files

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

Person.prototype.sayName = function() {
        console.log(`My name is ${this.name}.`);
}

var p1 = Person("Mary");        // 忘了 new
console.log(name);      // Mary
```

这段代码显示，当程序运行在非 strict 模式下的时候， `var p1 = Person("Mary");` 这条语句因为忘记 `this` 指向了全局，结果把全局变量 name 给改变了。这是非常危险的，也是很难发现的 bug 。

这个问题的解决办法就是如果构建函数发现没有 `new` ，我们在代码里帮它加上。而如果正常使用了 `new` 的时候， `this` 指向的就是要创建出来的对象。我们可以利用这个特点把上面的代码改进如下

```
var name = "Jack";

function Person(name) {
  if (this instanceof Person){  // 效果等同于 if (new.target === Person){
        this.name = name;       // 给对象赋值
  } else {
        return new Person(name);        // 否则调用自己创建对象
  }
}

Person.prototype.sayName = function() {
        console.log(`My name is ${this.name}.`);
}

var p1 = Person("Mary");
var p2 = new Person("Mike");

p1.sayName();   // My name is Mary.
p2.sayName();   // My name is Mike.
console.log(name);      // Jack
```

经过这样的处理，忘记了 `new` 的代码不仅可以一样生成对象，而且最重要的是不会无意间改变其它的变