javascript高级部分

Function方法 与 函数式编程

call

语法：call([thisObj[,arg1[, arg2[, [,.argN]]]]])

定义：调用一个对象的一个方法，以另一个对象替换当前对象。

说明：call 方法可以用来代替另一个对象调用一个方法。call 方法可将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。

如果没有提供 thisObj 参数，那么 Global 对象被用作 thisObj。

```
let myName = 'goudan';
let myAge = 13;
function showMsg(msg){
   return (msg + '').toLowerCase();
}

showName(myName); // 'goudan'
```

> 这段代码很容易就能看懂，在实际开发工作中，我们会处理不同的数据集合，这时候声明单一变量已经无法满足胃口，需要通过json的形式来存储数据

```
let person = {
      name:"kyogre",
      age:13,
      hobby:"coding"
      }

let newPerson  ={
      name:'dachui',
      age:99,
      hobby:'eat'
      }
function showMsg(msg){
   return  (msg + '').toLowerCase();
}

showMsg(person.name) // 'kyogre'
showMsg(newPerson.name) // 'dachui'
```

> 存储数据的方式发生了改变，但是我们处理数据的方式还是那么。。。古老，如果业务复杂一点

```
function format(msg){
      return msg+''.toLowerCase();
}
function show(msg){
       return '信息的内容是：'+ format(msg);
}
show(person.name) // '信息内容是：kyogre'
show(newPerson.name) // '信息内容是：dachui'
```

> 显示的传递上下文对象 （穿参）使得业务越来越复杂，这种叠加会让开发变得冗余和难以解读，bug和闭包横飞

那我们看看通过this如何清晰解决问题

> 通常this不会指向函数自身，而是调用函数的对象主体。当然，我们可以强制让function自身成为对象主体，这个以后咱们讨论； json本身就是对象，我们是否可以这样：

```
const person = {
      name:"kyogre",
      age:13,
      hobby:"coding"
      }
const newPerson  ={
   name:'dachui',
   age:99,
   hobby:'eat'
   }

function format(){
      return this.name+''.toLowerCase();

}
```

> 问题来了，不让穿参这个format中的this指向谁呢？ 指向调用format的对象本身,可是调用主体并不明确，可能是person也可能是newPerson，这时回过头看看call的定义吧：调用一个对象的一个方法，以另一个对象替换当前对象。 将函数内部执行上下文对象由原始对象替换为指定对象

```
   const name = '我叫window';   //全局变量 非严格模式下都为 window的属性   window.name;
   function format(){
      return  (this.name + '').toLowerCase();
   }
   format(); //'我叫window';
```

> 不要惊慌，本身他就是这样window会作为调用顶级作用域链函数的对象主体；这里的this默认为 window,用person对象代替window这个默认this主体去执行format会怎么样呢

```
    format.call(person);
    // kyogre
    format.call(newPerson);
    // dachui
    function show(){
        return '信息的内容是：'+ format.call(this);
    }


    show.call(person); // 信息的内容是：kyogre
```

> 感觉自己了解了this和call的小明，已经肆无忌惮的笑了起来，这样他就可以从繁重的回调与参数传递中解脱了，并且能够实现方法的初级模块化。

> 下面可以用call做一些平常的操作

```
 function isArray(object){
    return Object.prototype.toString.call(object) == '[object Array]';
 }// 借用Object原型上的toString方法来验证下对象是否是数组?

 function accumulation(){
    return [].reduce.call(arguments,(a,b)=>{return a+b}
 }//让不能使用数组方法的arguments类数组集合使用借用数组的reduce方法

 return Array.prototype.forEach.call($$('*'),(item)=>{item.style.border = '1px solid red';}
```

apply

语法：func.apply(thisArg, [argsArray])

> call()方法的作用和 apply() 方法类似，区别就是`call()`方法接受的是**参数列表**，而`apply()`方法接受的是**一个参数数组**。

```
const person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," + country;
  }
}
const person1 = {
  firstName:"John",
  lastName: "Doe"
}

person.fullName.apply(person1, ["Oslo", "Norway"]);
```

```
Math.max.apply(null, [1,2,3]); // 3

const arr = [1,2,3];
const otherArr = [3,4,5];
arr.push.apply(arr,otherArr);
console.log(arr); // [1, 2, 3, 3, 4, 5]
```

柯理化函数(currying)

> 在数学和计算机科学中，柯里化是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术。

```
function currying(fn) {
  const args = Array.prototype.slice.call(arguments, 1);

  const inlay = function () {
    if (arguments.length === 0) {
      return fn.apply(this, args);
    }
    if(arguments.length > 0 ){
       Array.prototype.push.apply(args, arguments);
       return inlay;
    }
  }
  return inlay;
}

function add() {
  const vals = Array.prototype.slice.call(arguments);
  return vals.reduce((pre, val) => {
    return pre + val;
  });
}

let newAdd = currying(add, 1, 2, 3);
newAdd(4, 5);
newAdd(6, 7)(6)(2, 3);
console.log(newAdd()); //39
```

bind

> `bind()` 方法创建一个新的函数，在 `bind()` 被调用时，这个新函数的 `this` 被指定为 `bind()` 的第一个参数，而其余参数将作为新函数的参数，供调用时使用。

语法:

function.bind(thisArg[, arg1[, arg2[, ...]]])

参数:

- **thisArg**
  - 调用绑定函数时作为 this 参数传递给目标函数的值。 如果使用[new](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new)运算符构造绑定函数，则忽略该值。当使用 bind 在 setTimeout 中创建一个函数（作为回调提供）时，作为 thisArg 传递的任何原始值都将转换为 object 。如果 bind 函数的参数列表为空，或者 thisArg 是 null 或 undefined ，执行作用域的 this 将被视为新函数的 thisArg 。
- **arg1, arg2, ...**
  - 当目标函数被调用时，被预置入绑定函数的参数列表中的参数。

返回值:

返回一个原函数的拷贝，并拥有指定的 ** this ** 值和初始参数。

```
const OBJ = {
   petName: 'kyogre',
   qq: '2971411050',
   sayHi: function () {
      console.log(`我是${this.petName} 很高兴认识你`)
   }
}

let sayHi = OBJ.sayHi;
sayHi(); //我是undifined 很高兴认识你  ps: this非严格模式指向了window window.petName不存在


let sayHi = OBJ.sayHi.bind(OBJ);
sayHi(); //我是kyogre 很高兴认识你 ps:通过bind强制绑定sayHi函数内部的this指向OBJ
```

偏函数 (partial)

> 在计算机科学中，局部应用(偏函数)是指**固定一个函数的一些参数，然后产生另一个更小元的函数。**（什么是元？元是指函数参数的个数，比如一个带有两个参数的函数被称为二元函数。）
>
> bind() 的另一个最简单的用法是使一个函数拥有预设的初始参数。只要将这些参数（如果有的话）作为 bind() 的参数写在 this 后面。当绑定函数被调用时，这些参数会被插入到目标函数的参数列表的开始位置，传递给绑定函数的参数会跟在它们后面。

```
function list() {
  return Array.prototype.slice.call(arguments);
}

function addArguments(arg1, arg2) {
```

```
  return arg1 + arg2
}

const list1 = list(1, 2, 3); // [1, 2, 3]

const result1 = addArguments(1, 2); // 3

// 创建一个函数，它拥有预设参数列表。
const leadingThirtysevenList = list.bind(null, 37);

// 创建一个函数，它拥有预设的第一个参数
const addThirtySeven = addArguments.bind(null, 37);

const list2 = leadingThirtysevenList();
// [37]

const list3 = leadingThirtysevenList(1, 2, 3);
// [37, 1, 2, 3]

const result2 = addThirtySeven(5);
// 37 + 5 = 42

const result3 = addThirtySeven(5, 10);
// 37 + 5 = 42 ，第二个参数被忽略
```

通道函数(compose)

> 按照顺序

```
 function toUpperCase(str){
   return str.toUpperCase(); // 将字符串变成大写
}

function add(str){
   return str + '!!!'; // 将字符串拼接
}

function split(str){
   return str.split(''); // 将字符串拆分为数组
}

function reverse(arr){
   return arr.reverse(); // 将数组逆序
}


function join(arr){
```

```
    return arr.join('-'); // 将数组按'-'拼接成字符串
}


function compose(){
    const args = Array.prototype.slice.call(arguments); // 转换为数组使用下面的方法
    return function(x){
        return args.reduceRight(function(result, cb){
            return cb(result);
        }, x);
    }
}

const f = compose(add, join, reverse, split, toUpperCase);
console.log( f('cba') ); // A-B-C!!!
```

柯理化与偏函数区别

- 柯里化是将一个多参数函数转换成多个单参数函数，也就是将一个 n 元函数转换成 n 个一元函数。
- 局部应用则是固定一个函数的一个或者多个参数，也就是将一个 n 元函数转换成一个 n - x 元函数。

面向对象编程

什么是对象

> Everything is object （万物皆对象）

对象到底是什么，我们可以从两次层次来理解。

(1) 对象是单个事物的抽象。

一本书、一辆汽车、一个人都可以是对象，一个数据库、一张网页、一个与远程服务器的连接也可以是对象。当实物被抽象成对象，实物之间的关系就变成了对象之间的关系，从而就可以模拟现实情况，针对对象进行编程。

(2) 对象是一个容器，封装了属性（property）和方法（method）。

属性是对象的状态，方法是对象的行为（完成某种任务）。比如，我们可以把动物抽象为animal对象，使用“属性”记录具体是那一种动物，使用“方法”表示动物的某种行为（奔跑、捕猎、休息等等）。

在实际开发中，对象是一个抽象的概念，可以将其简单理解为：**数据集或功能集。**

ECMAScript-262 把对象定义为：**无序属性的集合，其属性可以包含基本值、对象或者函数。**
严格来讲，这就相当于说对象是一组没有特定顺序的值。对象的每个属性或方法都有一个名字，而每个名字都映射到一个值。

提示：每个对象都是基于一个引用类型创建的，这些类型可以是系统内置的原生类型，也可以是开发人员自定义的类型。

什么是面向对象

> 面向对象不是新的东西，它只是过程式代码的一种高度封装，目的在于提高代码的开发效率和可维护性。

面向对象编程 —— Object Oriented Programming，简称 OOP ，是一种编程开发思想。
它将真实世界各种复杂的关系，抽象为一个个对象，然后由对象之间的分工与合作，完成对真实世界的模拟。

在面向对象程序开发思想中，每一个对象都是功能中心，具有明确分工，可以完成接受信息、处理数据、发出信息等任务。
因此，面向对象编程具有灵活、代码可复用、高度模块化等特点，容易维护和开发，比起由一系列函数或指令组成的传统的过程式编程（procedural programming），更适合多人合作的大型软件项目。

面向对象与面向过程：

- 面向过程就是亲力亲为，事无巨细，面面俱到，步步紧跟，有条不紊
- 面向对象就是找一个对象，指挥得结果
- 面向对象将执行者转变成指挥者
- 面向对象不是面向过程的替代，而是面向过程的封装

面向对象的特性：

- 封装性
  - 封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。
- 继承性
  - 继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。
- 多态性
  - 基于对象所属类的不同，外部对同一个方法的调用，实际执行的逻辑不同。

程序中面向对象的基本体现

在 JavaScript 中，所有数据类型都可以视为对象，当然也可以自定义对象。
自定义的对象数据类型就是面向对象中的类（ Class ）的概念。

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个对象表示：

```
let student1 = { name: 'Michael', score: 98 }
let student2 = { name: 'Bob', score: 81 }
```

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

```
function printScore (student) {
  console.log(`姓名：${student.name}  成绩：${student.score}`);
}
```

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，
而是 `Student` 这种数据类型应该被视为一个对象，这个对象拥有 `name` 和 `score` 这两个属性（Property）。
如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个 `printScore` 消息，让对象自己把自己的数据打印出来。

抽象数据行为模板（Class）：

```
function Student (name, score) {
  this.name = name
  this.score = score
}

Student.prototype.printScore = function () {
  console.log(`姓名：${this.name}  成绩：${this.score}`);
}
```

根据模板创建具体实例对象（Instance）：

```
var std1 = new Student('Michael', 98);
var std2 = new Student('Bob', 81);
```

实例对象具有自己的具体行为（给对象发消息）：

```
std1.printScore() // => 姓名：Michael  成绩：98
std2.printScore() // => 姓名：Bob  成绩 81
```

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。
Class 是一种抽象概念，比如我们定义的 Class——Student ，是指学生这个概念，
而实例（Instance）则是一个个具体的 Student ，比如， Michael 和 Bob 是两个具体的 Student 。

所以，面向对象的设计思想是：

- 抽象出 Class

- 根据 Class 创建 Instance
- 指挥 Instance 得结果

面向对象的抽象程度又比函数要高，因为一个 Class 既包含数据，又包含操作数据的方法。

创建对象

简单方式

我们可以直接通过 `new Object()` 创建：

```
const person = new Object()
person.name = 'Jack'
person.age = 18

person.sayName = function () {
  console.log(this.name)
}
```

每次创建通过 `new Object()` 比较麻烦，所以可以通过它的简写形式对象字面量来创建：

```
const person = {
  name: 'Jack',
  age: 18,
  sayName: function () {
    console.log(this.name)
  }
}
```

对于上面的写法固然没有问题，但是假如我们要生成两个 `person` 实例对象呢？

```
const person1 = {
  name: 'Jack',
  age: 18,
  sayName: function () {
    console.log(this.name)
  }
}

const person2 = {
  name: 'Mike',
  age: 16,
  sayName: function () {
    console.log(this.name)
  }
}
```

通过上面的代码我们不难看出，这样写的代码太过冗余，重复性太高。

简单方式的改进：工厂函数

我们可以写一个函数，解决代码重复问题：

```
function createPerson (name, age) {
  return {
    name: name,
    age: age,
    sayName: function () {
      console.log(this.name)
    }
  }
}
```

然后生成实例对象：

```
let p1 = createPerson('Jack', 18)
let p2 = createPerson('Mike', 18)
```

这样封装确实爽多了，通过工厂模式我们解决了创建多个相似对象代码冗余的问题，
但却没有解决对象识别的问题（即怎样知道一个对象的类型）。

构造函数

内容引导：

- 构造函数语法
- 分析构造函数
- 构造函数和实例对象的关系
  - 实例的 constructor 属性
  - instanceof 操作符
- 普通函数调用和构造函数调用的区别
- 构造函数的返回值
- 构造函数的静态成员和实例成员
  - 函数也是对象
  - 实例成员
  - 静态成员
- 构造函数的问题

更优雅的工厂函数：构造函数

一种更优雅的工厂函数就是下面这样，构造函数：

```
function Person (name, age) {
  this.name = name
  this.age = age
  this.sayName = function () {
    console.log(this.name)
  }
}

let p1 = new Person('Jack', 18)
p1.sayName() // => Jack

let p2 = new Person('Mike', 23)
p2.sayName() // => Mike
```

解析构造函数代码的执行

在上面的示例中，`Person()` 函数取代了 `createPerson()` 函数，但是实现效果是一样的。
这是为什么呢?

我们注意到，`Person()` 中的代码与 `createPerson()` 有以下几点不同之处：

- 没有显示的创建对象
- 直接将属性和方法赋给了 `this` 对象
- 没有 `return` 语句
- 函数名使用的是大写的 `Person`

而要创建 `Person` 实例，则必须使用 `new` 操作符。
以这种方式调用构造函数会经历以下 4 个步骤：

1. 创建一个新对象
2. 将构造函数的作用域赋给新对象（因此 this 就指向了这个新对象）
3. 执行构造函数中的代码
4. 返回新对象

下面是具体的伪代码：

```
function Person (name, age) {
  // 当使用 new 操作符调用 Person() 的时候，实际上这里会先创建一个对象
  // var instance = {}
  // 然后让内部的 this 指向 instance 对象
  // this = instance
  // 接下来所有针对 this 的操作实际上操作的就是 instance

  this.name = name
  this.age = age
  this.sayName = function () {
    console.log(this.name)
  }

  // 在函数的结尾处会将 this 返回，也就是 instance
  // return this

}
```

构造函数和实例对象的关系

使用构造函数的好处不仅仅在于代码的简洁性，更重要的是我们可以识别对象的具体类型了。
在每一个实例对象中的*_proto_*中同时有一个 `constructor` 属性，该属性指向创建该实例的构造函数：

```
console.log(p1.constructor === Person) // => true
console.log(p2.constructor === Person) // => true
console.log(p1.constructor === p2.constructor) // => true
```

对象的 `constructor` 属性最初是用来标识对象类型的，
但是，如果要检测对象的类型，还是使用 `instanceof` 操作符更可靠一些：

```
console.log(p1 instanceof Person) // => true
console.log(p2 instanceof Person) // => true
```

总结：

- 构造函数是根据具体的事物抽象出来的抽象模板
- 实例对象是根据抽象的构造函数模板得到的具体实例对象
- 每一个实例对象都具有一个 `constructor` 属性，指向创建该实例的构造函数
    - 注意： `constructor` 是实例的属性的说法不严谨，具体后面的原型会讲到
- 可以通过实例的 `constructor` 属性判断实例和构造函数之间的关系
    - 注意：这种方式不严谨，推荐使用 `instanceof` 操作符，后面学原型会解释为什么

构造函数的问题

使用构造函数带来的最大的好处就是创建对象更方便了，但是其本身也存在一个浪费内存的问题：

```
function Person (name, age) {
  this.name = name
  this.age = age
  this.type = 'human'
  this.sayHello = function () {
    console.log('hello ' + this.name)
  }
}

let p1 = new Person('lpz', 18)
let p2 = new Person('Jack', 16)
```

在该示例中，从表面上好像没什么问题，但是实际上这样做，有一个很大的弊端。
那就是对于每一个实例对象， `type` 和 `sayHello` 都是一模一样的内容，
每一次生成一个实例，都必须为重复的内容，多占用一些内存，如果实例对象很多，会造成极大的内存浪费。

```
console.log(p1.sayHello === p2.sayHello) // => false
```

对于这种问题我们可以把需要共享的函数定义到构造函数外部：

```
function sayHello = function () {
  console.log('hello ' + this.name)
}

function Person (name, age) {
  this.name = name
  this.age = age
  this.type = 'human'
  this.sayHello = sayHello
}

let p1 = new Person('lpz', 18)
let p2 = new Person('Jack', 16)

console.log(p1.sayHello === p2.sayHello) // => true
```

这样确实可以了，但是如果有多个需要共享的函数的话就会造成全局命名空间冲突的问题。

你肯定想到了可以把多个函数放到一个对象中用来避免全局命名空间冲突的问题：

```
const fns = {
  sayHello: function () {
    console.log('hello ' + this.name)
  },
  sayAge: function () {
    console.log(this.age)
  }
}

function Person (name, age) {
  this.name = name
  this.age = age
  this.type = 'human'
  this.sayHello = fns.sayHello
  this.sayAge = fns.sayAge
}

let p1 = new Person('lpz', 18)
let p2 = new Person('Jack', 16)

console.log(p1.sayHello === p2.sayHello) // => true
console.log(p1.sayAge === p2.sayAge) // => true
```

至此，我们利用自己的方式基本上解决了构造函数的内存浪费问题。
但是代码看起来还是那么的格格不入，那有没有更好的方式呢？

小结

- 构造函数语法

- 分析构造函数
- 构造函数和实例对象的关系
  - 实例的 constructor 属性
  - instanceof 操作符
- 构造函数的问题

原型

内容引导：

- 使用 prototype 原型对象解决构造函数的问题
- 分析 构造函数、prototype 原型对象、实例对象 三者之间的关系
- 属性成员搜索原则：原型链
- 实例对象读写原型对象中的成员
- 原型对象的简写形式
- 原生对象的原型
  - Object
  - Array
  - String
  - ...
- 原型对象的问题
- 构造的函数和原型对象使用建议

更好的解决方案：`prototype`

Javascript 规定，每一个构造函数都有一个 `prototype` 属性，指向另一个对象。
这个对象的所有属性和方法，都会被构造函数的实例继承。

这也就意味着，我们可以把所有对象实例需要共享的属性和方法直接定义在 `prototype` 对象上。

```
function Person (name, age) {
  this.name = name
  this.age = age
}

console.log(Person.prototype)

Person.prototype.type = 'human'

Person.prototype.sayName = function () {
  console.log(this.name)
}

let p1 = new Person(...)
let p2 = new Person(...)

console.log(p1.sayName === p2.sayName) // => true
```

这时所有实例的 `type` 属性和 `sayName()` 方法，
其实都是同一个内存地址，指向 `prototype` 对象，因此就提高了运行效率。

构造函数、实例、原型三者之间的关系

任何函数都具有一个 `prototype` 属性，该属性是一个对象。

```
function F () {}
console.log(F.prototype) // => object

F.prototype.sayHi = function () {
  console.log('hi!')
}
```

构造函数的 `prototype` 对象默认都有一个 `constructor` 属性，指向 `prototype` 对象所在函数。

```
console.log(F.constructor === F) // => true
```

通过构造函数得到的实例对象内部会包含一个指向构造函数的 `prototype` 对象的指针 `__proto__`。

```
var instance = new F()
console.log(instance.__proto__ === F.prototype) // => true
```

`__proto__` 是非标准属性。

实例对象可以直接访问原型对象成员。

```
instance.sayHi() // => hi!
```

总结：

- 任何函数都具有一个 `prototype` 属性，该属性是一个对象
- 构造函数的 `prototype` 对象默认都有一个 `constructor` 属性，指向 `prototype` 对象所在函数
- 通过构造函数得到的实例对象内部会包含一个指向构造函数的 `prototype` 对象的指针 `__proto__`
- 所有实例都直接或间接继承了原型对象的成员

属性成员的搜索原则：原型链

了解了 **构造函数-实例-原型对象** 三者之间的关系后，接下来我们来解释一下为什么实例对象可以访问原型对象中的成员。

每当代码读取某个对象的某个属性时，都会执行一次搜索，目标是具有给定名字的属性

- 搜索首先从对象实例本身开始
- 如果在实例中找到了具有给定名字的属性，则返回该属性的值
- 如果没有找到，则继续搜索指针指向的原型对象，在原型对象中查找具有给定名字的属性
- 如果在原型对象中找到了这个属性，则返回该属性的值

也就是说，在我们调用 `person1.sayName()` 的时候，会先后执行两次搜索：

- 首先，解析器会问：“实例 person1 有 sayName 属性吗？”答：“没有。
- ”然后，它继续搜索，再问：“ person1 的原型有 sayName 属性吗？”答：“有。
- ”于是，它就读取那个保存在原型对象中的函数。
- 当我们调用 person2.sayName() 时，将会重现相同的搜索过程，得到相同的结果。

而这正是多个对象实例共享原型所保存的属性和方法的基本原理。

总结：

- 先在自己身上找，找到即返回
- 自己身上找不到，则沿着原型链向上查找，找到即返回
- 如果一直到原型链的末端还没有找到，则返回 `undefined`

实例对象读写原型对象成员

读取：

- 先在自己身上找，找到即返回
- 自己身上找不到，则沿着原型链向上查找，找到即返回
- 如果一直到原型链的末端还没有找到，则返回 `undefined`

值类型成员写入（`实例对象.值类型成员 = xx`）：

- 当实例期望重写原型对象中的某个普通数据成员时实际上会把该成员添加到自己身上
- 也就是说该行为实际上会屏蔽掉对原型对象成员的访问

引用类型成员写入（`实例对象.引用类型成员 = xx`）：

- 同上

复杂类型修改（`实例对象.成员.xx = xx`）：

- 同样会先在自己身上找该成员，如果自己身上找到则直接修改
- 如果自己身上找不到，则沿着原型链继续查找，如果找到则修改
- 如果一直到原型链的末端还没有找到该成员，则报错（`实例对象.undefined.xx = xx`）

更简单的原型语法

我们注意到，前面例子中每添加一个属性和方法就要敲一遍 `Person.prototype` 。
为减少不必要的输入，更常见的做法是用一个包含所有属性和方法的对象字面量来重写整个原型对象：

```
function Person (name, age) {
  this.name = name
  this.age = age
}

Person.prototype = {
  type: 'human',
  sayHello: function () {
    console.log('我叫' + this.name + '，我今年' + this.age + '岁了')
  }
}
```

在该示例中，我们将 `Person.prototype` 重置到了一个新的对象。
这样做的好处就是为 `Person.prototype` 添加成员简单了，但是也会带来一个问题，那就是原型对象丢失了 `constructor` 成员。

所以，我们为了保持 `constructor` 的指向正确，建议的写法是：

```
function Person (name, age) {
  this.name = name
  this.age = age
}

Person.prototype = {
  constructor: Person, // => 手动将 constructor 指向正确的构造函数
  type: 'human',
  sayHello: function () {
    console.log('我叫' + this.name + '，我今年' + this.age + '岁了')
  }
}
```

原生对象的原型

所有函数都有 prototype 属性对象。

- Object.prototype
- Function.prototype
- Array.prototype
- String.prototype
- Number.prototype
- Date.prototype
- ...

原型对象的问题

- 共享数组
- 共享对象

如果真的希望可以被实例对象之间共享和修改这些共享数据那就不是问题。但是如果不希望实例之间共享和修改这些共享数据则就是问题。

一个更好的建议是，最好不要让实例之间互相共享这些数组或者对象成员，一旦修改的话会导致数据的走向很不明确而且难以维护。

原型对象使用建议

- 私有成员（一般就是非函数成员）放到构造函数中
- 共享成员（一般就是函数）放到原型对象中
- 如果重置了 `prototype` 记得修正 `constructor` 的指向

prototype 与 __proto__

prototype

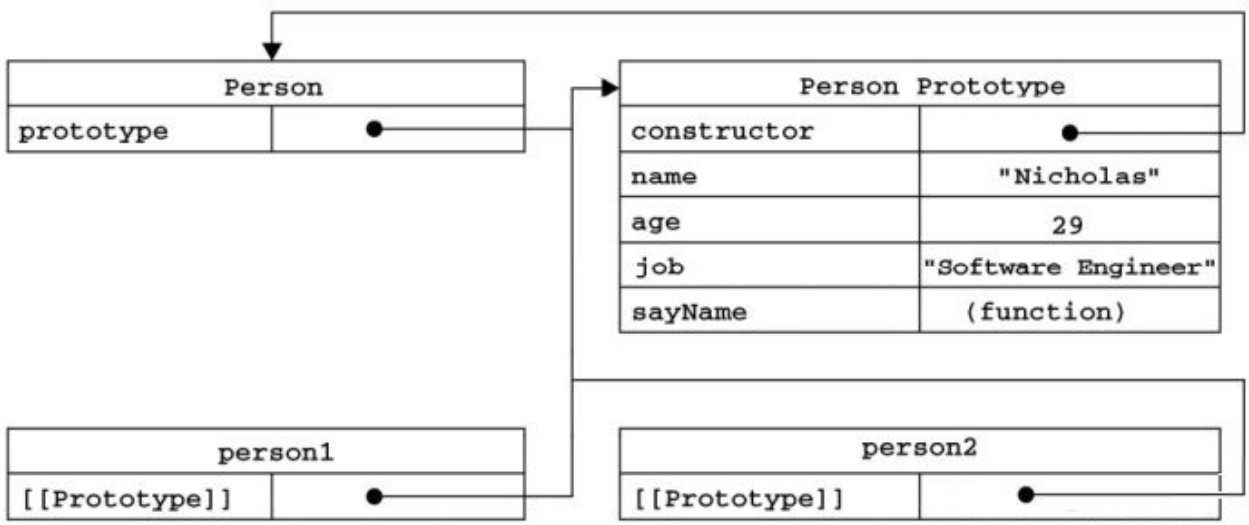
每个函数都有一个prototype属性，该属性是一个指针，指向一个对象（构造函数的原型对象），这个对象包含所有实例共享的属性和方法。原型对象都有一个`constructor`属性，这个属性指向所关联的构造函数。使用这个对象的好处就是可以让所有实例对象共享它所拥有的属性和方法。**这个属性只用js中的类(或者说能够作为构造函数的对象)才会有。**

__proto__

每个实例对象都有一个**proto**属性，用于指向构造函数的原型对象（`protitype`）。__proto__属性是在调用构造函数创建实例对象时产生的。**该属性存在于实例和构造函数的原型对象之间，而不是存在于实例与构造函数之间。**

```
function Person(name, age, job){
   this.name = name;
   this.age = age;
   this.job = job;
   this.sayName = function(){
      console.log(this.name);
   }; // 与声明函数在逻辑上是等价的
}
let person1 = new Person("Nicholas", 29, "Software Engineer");
console.log(person1);
console.log(Person);
console.log(person1.prototype);//undefined
console.log(person1.__proto__);
console.log(Person.prototype);
console.log(person1.__proto__ === Person.prototype);//true
```

1、调用构造函数创建的实例对象的prototype属性为"undefined",构造函数的prototype是一个对象。
2、proto属性是在调用构造函数创建实例对象时产生的。
3、调用构造函数创建的实例对象的proto属性指向构造函数的prototype，本质上就是继承构造函数的原型属性。
4、在默认情况下，所有原型对象都会自动获得一个constructor(构造函数)属性，这个属性包含一个指向prototype属性所在函数的指针。



- `__proto__` ： 是`对象`就会有这个属性(强调是对象);`函数`也是对象,那么函数也有这个属性咯,它指向`构造函数`的原型对象;
- `prototype` ： 是`函数`都会有这个属性(强调是函数)，`普通对象`是没有这个属性的（JS 里面，一切皆为对象，所以这里的`普通对象`不包括`函数对象`）.它是构造函数的原型对象;
- `constructor` ： 这是`原型对象`上的一个指向`构造函数`的属性。

总结:

- 每一个对象都有`__proto__`属性,`__proto__` ==> `Object.prototype`(Object 构造函数的原型对象);
- 每个函数都`__proto__`和`prototype`属性;
- 每个`原型对象`都有`constructor` 和 `__proto__`属性,其中`constructor`指回'构造函数', 而`__proto__`指向`Object.prototype`;
- `object`是有对象的祖先,所有对象都可以通过`__proto__`属性找到它;
- `Function`是所有函数的祖先,所有函数都可以通过`__proto__`属性找到它;
- 每个函数都有一个`prototype`,由于`prototype`是一个对象,指向了构造函数的原型对象
- 对象的`__proto__`属性指向`原型`,`__proto__`将对象和原型链接起来组成了原型链

isPrototypeOf()

> 虽然在所有实现中都无法访问到**proto**，但可以通过 isPrototypeOf()方法来确定对象之间是否存在这种关系。

```
alert(Person.prototype.isPrototypeOf(person1)); //true
alert(Person.prototype.isPrototypeOf(person2)); //true
```

Object.getPrototypeOf()

> 在所有支持的实现中，这个方法返回**proto**的值。例如：

```
//这里的person1是下面实例
alert(Object.getPrototypeOf(person1) == Person.prototype); //true
alert(Object.getPrototypeOf(person1).name); //"Nicholas"  person1
```

注意： 虽然可以通过对象实例访问保存在原型中的值，但却不能通过对象实例重写原型中的值。如果我们在实例中添加了一个属性，而该属性与实例原型中的一个属性同名，那我们就在实例中创建该属性，该属性将会**屏蔽**原型中的那个属性。

hasOwnProperty()

可以检测一个属性是存在于实例中，还是存在于原型中。返回值为true表示该属性存在实例对象中，其他情况都为false。

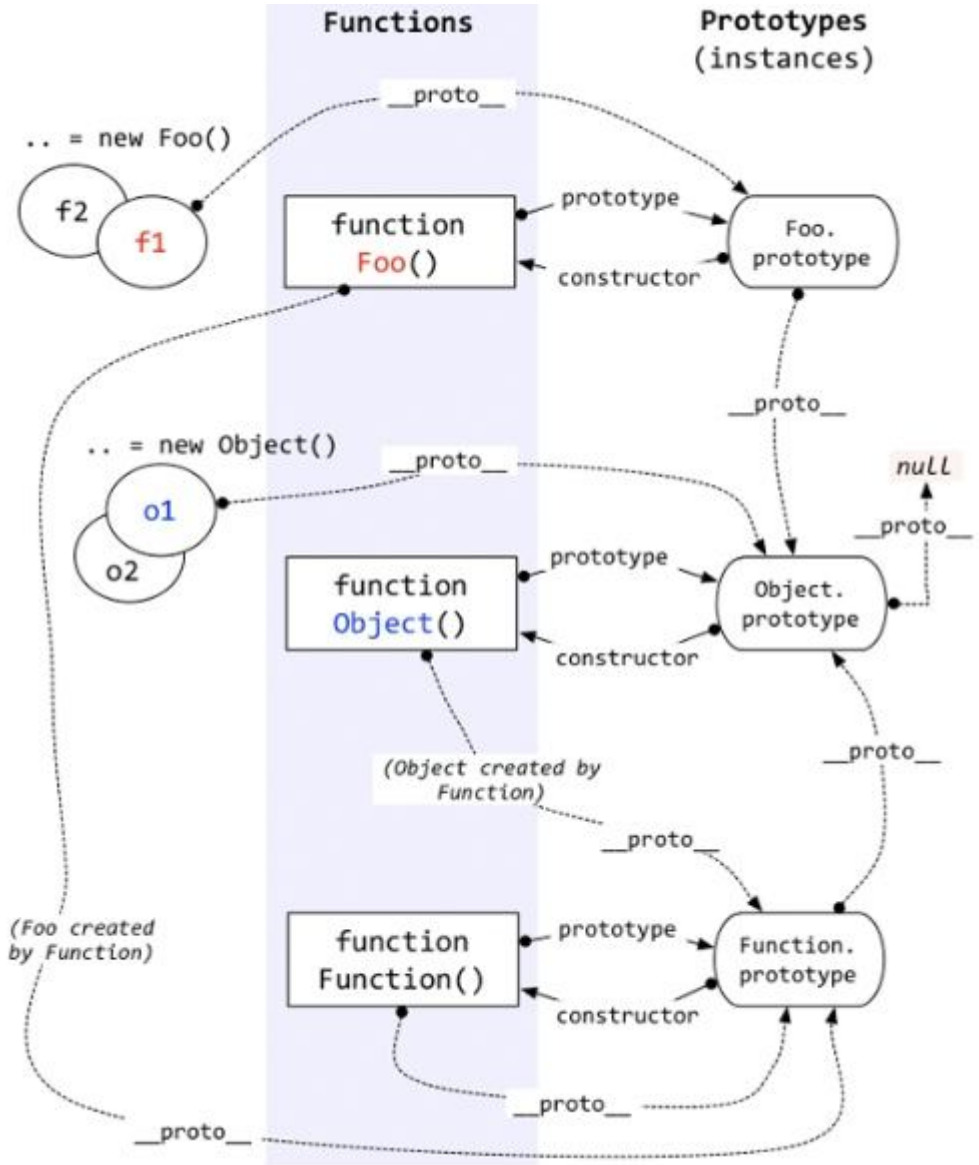
in 操作符

无论该属性存在于实例中还是原型中。只要存在对象中，都会返回true。但是可以同时使用 hasOwnProperty()方法和 in 操作符，就可以确定该属性到底是存在于对象中，还是存在于原型中。

```
var person1 = new Person();
var person2 = new Person();
alert(person1.hasOwnProperty("name")); //false
alert("name" in person1); //true
person1.name = "Greg";
alert(person1.name); //"Greg" —— 来自实例
alert(person1.hasOwnProperty("name")); //true
alert("name" in person1); //true
alert(person2.name); //"Nicholas" —— 来自原型
alert(person2.hasOwnProperty("name")); //false
alert("name" in person2); //true
delete person1.name;
alert(person1.name); //"Nicholas" —— 来自原型
alert(person1.hasOwnProperty("name")); //false
alert("name" in person1); //true
```

原型链与继承

> 对于使用过基于类的语言 (如 Java 或 C++) 的开发人员来说，JavaScript 有点令人困惑，因为它是动态的，并且本身不提供一个 `class` 实现。（在 ES2015/ES6 中引入了 `class` 关键字，但那只是语法糖，JavaScript 仍然是基于原型的）。
>
> 当谈到继承时，JavaScript 只有一种结构：对象。每个实例对象（ object ）都有一个私有属性（称之为 **proto** ）指向它的构造函数的原型对象（**prototype** ）。该原型对象也有一个自己的原型对象(**proto**)，层层向上直到一个对象的原型对象为 `null`。根据定义，`null` 没有原型，并作为这个**原型链**中的最后一个环节。
>
> 几乎所有 JavaScript 中的对象都是位于原型链顶端的 `Object` 的实例。
>
> 尽管这种原型继承通常被认为是 JavaScript 的弱点之一，但是原型继承模型本身实际上比经典模型更强大。例如，在原型模型的基础上构建经典模型相当简单。



1. 原型链继承

> 基本思想: 利用原型让一个引用类型继承另一个引用类型的属性和方法

核心：原型链对象 变成 父类实例，子类就可以调用父类方法和属性。

```
function Parent() {
}
Parent.prototype.age = 18
Parent.prototype.getName = function () {
   return this.name

}
```

```
function Child(name) {
  this.name = name
}
Child.prototype = new Parent()

var child = new Child('leo')
// 这样子类就可以调用父类的属性和方法
console.log(child.getName())  // leo
console.log(child.age)        // 18
```

优点：** 实现简单。

缺点：

1. 引用类型值的原型属性会被所有实例共享。
2. 不能向父类传递参数。

```
function Parent() {
  this.likeFood = ['水果', '鸡', '烤肉']
}
Parent.prototype.age = 18
Parent.prototype.getName = function () {
  return this.name
}

function Child(name) {
  this.name = name
}
Child.prototype = new Parent()

var chongqiChild = new Child('重庆孩子')
var guangdongChild = new Child('广东孩子')

// 重庆孩子还喜欢吃花椒。。。
chongqiChild.likeFood.push('花椒')
console.log(chongqiChild.likeFood)    // ["水果", "鸡", "烤肉", "花椒"]
console.log(guangdongChild.likeFood)  // ["水果", "鸡", "烤肉", "花椒"]
```

这时，会发现明明只是 **重庆孩子** 爱吃花椒，**广东孩子** 莫名奇妙得也变得爱吃了？？？？这个共享是存在问题的，不科学的。（可能重庆孩子和广东孩子一起黑脸问号。。。）

至于第二个问题，其实也显而易见了，没有传递参数的途径。因此，第二种继承方式出来啦。

2. 借用构造函数继承

遗留问题:

1. 父类引用属性共享。
2. 不能传参数到父类。

核心：子类构造函数内部调用父类构造函数，并传入 this指针。

```
// 2. 借用构造函数
function Parent(name) {
    this.name = name
    this.likeFood = ["水果", "鸡", "烤肉"]
}
function Child(name) {
    Parent.call(this, name)
}
Parent.prototype.getName = function() {
    return this.name
}
var chongqingChild = new Child('重庆孩子')
var guangdongChild = new Child('广东孩子')
chongqingChild.likeFood.push('花椒')

console.log(chongqingChild.likeFood)    // ["水果", "鸡", "烤肉", "花椒"]
console.log(guangdongChild.likeFood)    // ["水果", "鸡", "烤肉"]
console.log(chongqingChild.name)        // "重庆孩子"
console.log(chongqingChild.getName())   // Uncaught TypeError: chongqingChild.getName is not a function
```

值得庆幸的是，这次只有我们 **重庆孩子** 喜欢吃花椒，**广东孩子** 没被标记爱吃花椒啦。并且，我们通过 call 方法将我们的参数也传入到了父类，解决了之前的遗留问题啦。

但是，**原型链继承** 是可以调用父类方法的，但是**借用构造函数**却不可以了，这是因为 当前子类的原型链并不指向父类了。因此，结合 第一，第二种继承方式，第三种继承方式应运而生啦。

3. 组合继承

核心： 前两者结合，进化更高级。

```
function Parent(name) {
    this.name = name
    this.likeFood = ["水果", "鸡", "烤肉"]
}
function Child(name, age) {
    Parent.call(this, name)
    this.age = age
}
Parent.prototype.getName = function() {
    return this.name
}
Child.prototype = new Parent()
Child.prototype.constructor = Child
```

```
Child.prototype.getAge = function() {
    return this.age
}

var chongqingChild = new Child('重庆孩子', 18)
var guangdongChild = new Child('广东孩子', 19)
chongqingChild.likeFood.push('花椒')

console.log(chongqingChild.likeFood)    // ["水果", "鸡", "烤肉", "花椒"]
console.log(guangdongChild.likeFood)    // ["水果", "鸡", "烤肉"]
console.log(chongqingChild.name)       // "重庆孩子"
console.log(chongqingChild.getName())  // "重庆孩子"
console.log(chongqingChild.getAge())    // 18
```

这样：

1. 原型引用类型传参共享问题
2. 传参问题
3. 调用父类问题都解决啦。

- Javascript 的经典继承。
- 但是有一个小缺点：在给 Child 原型赋值会执行一次Parent构造函数。所以，无论什么情况下都会调用两次父类构造函数

---

4. 原型式继承

这是在2006年一个叫 道格拉斯·克罗克福德 的人，介绍的一种方法，**这种方法并没有使用严格意义上的构造函数。**

他的想法是 **借助原型可以基于已有的对象创建新对象，同时还不必因此创建自定义类型。**

这之前的三种继承方式，我们都需要自己写自定义函数(例如，Parent和Child)。假如，现在已经有一个对象了，并且，我也只是想用你的属性，不想搞得那么麻烦的自定义很多函数。那怎么办呢?

核心： 我们需要创建一个临时的构造函数，并将作为父类的对象作为构造函数的原型，并返回一个新对象。

```
/*
    @function 实现继承 函数
    @param parent 充当父类的对象
*/
function realizeInheritance(parent) {
    // 临时函数
    function tempFunc() {}
    tempFunc.prototype = parent
    return new tempFunc()
}
```

核心点说了，我们来尝试一下。

```
// 这个就是已有的对象
var baba = {
    name: "爸爸",
    likeFoods: ["水果", "鸡", "烤肉"]
}
/*
    var newChild = {} <==> baba  这两个对象建立关系就是这种继承的核心了。
*/
var child1 = realizeInheritance(baba)
var child2 = realizeInheritance(baba)
child1.likeFoods.push('花椒')
console.log(child1.likeFoods) //    ["水果", "鸡", "烤肉", "花椒"]
console.log(child2.likeFoods) //    ["水果", "鸡", "烤肉", "花椒"]
```

我们可以发现，父类的属性对于子类来说都是共享的。所以，如果我们只是想一个对象和另一个对象保持一致，这将是不二之选。

ES5 新增了个 Object.create(parentObject) 函数来更加便捷的实现上述继承

```
var baba = {
    name: "爸爸",
    likeFoods: ["水果", "鸡", "烤肉"]
}
var child1 = Object.create(baba)
var child2 = Object.create(baba)
child1.likeFoods.push('花椒')
console.log(child1.likeFoods) //    ["水果", "鸡", "烤肉", "花椒"]
console.log(child2.likeFoods) //    ["水果", "鸡", "烤肉", "花椒"]
```

效果和上面相同~

5. 寄生式继承

这种继承是基于**原型式继承**，是同一个人想出来的，作者觉得，这样不能有子类的特有方法，似乎不妥。就用来一个种工厂模式的方式来给予子类一些独特的属性。

```
function realizeInheritance(parent) {
    // 临时函数
    function tempFunc() {}
    tempFunc.prototype = parent
    return new tempFunc()
}
// Parasitic: 寄生的    inheritance:  继承    一个最简单的工厂函数。
function parasiticInheritance(object) {
```

```
    var clone = realizeInheritance(object)  // 这是用了原型式继承，但是只要是任何可以返回对象的方法都可以。
    clone.sayName = function() {
        console.log('我是'+this.name)
    }
    return clone
}
var baba = {
    name: "爸爸",
    likeFoods: ["水果", "鸡", "烤肉"]
}
var child = parasiticInheritance(baba)
child.name = '儿子'
child.sayName() // 我是儿子
```

缺点：使用寄生式继承来为对象添加函数，会由于不能做到函数复用而降低效率（每一个函数都是新的）；这一点与构造函数继承类似。

6.寄生组合式继承

我们先回顾之前的 **组合继承**

```
function Parent(name) {
    this.name = name
    this.likeFood = ["水果", "鸡", "烤肉"]
}
function Child(name, age) {
    Parent.call(this, name)        // 第二次调用
    this.age = age
}
Parent.prototype.getName = function() {
    return this.name
}
Child.prototype = new Parent()     // 第一次调用
Child.prototype.constructor = Child
Child.prototype.getAge = function() {
    return this.age
}
```

这个两次调用的问题之前有提及过。过程大致：

- 第一次调用，Child 的原型被赋值了 name 和 likeFood 属性
- 第二次调用，注入this，会在Child 的实例对象上注入 name 和 likeFood 属性，这样就屏蔽了原型上的属性。

只要了问题，我们就来解决这个问题~

```
function Parent(name) {
    this.name = name

    this.likeFood = ["水果", "鸡", "烤肉"]
```

```
}
function Child(name, age) {
    Parent.call(this, name)
    this.age = age
}
Parent.prototype.getName = function() {
    return this.name
}

// Child.prototype = new Parent()  使用新方法解决
// Child.prototype.constructor = Child
inheritPrototype(Child, Parent)
function inheritPrototype(childFunc, parentFunc) {
    var prototype = realizeInheritance(parentFunc.prototype)   //创建对象,我们继续是用原型式继承的创建
    prototype.constructor = childFunc            //增强对象
    childFunc.prototype = prototype              //指定对象
}
function realizeInheritance(parent) {
    // 临时函数
    function tempFunc() {}
    tempFunc.prototype = parent
    return new tempFunc()
}

Child.prototype.getAge = function() {
    return this.age
}

var chongqingChild = new Child('重庆孩子', 18)
var guangdongChild = new Child('广东孩子', 19)
chongqingChild.likeFood.push('花椒')

console.log(chongqingChild.likeFood)    // ["水果", "鸡", "烤肉", "花椒"]
console.log(guangdongChild.likeFood)    // ["水果", "鸡", "烤肉"]
console.log(chongqingChild.name)       // "重庆孩子"
console.log(chongqingChild.getName())   // "重庆孩子"
console.log(chongqingChild.getAge())    // 18
```

这种方法的核心思想：

- 首先，用一个空对象建立和父类关系。
- 然后，再用这个空对象作为子类的原型对象。

这样，中间的对象就不存在new 构造函数的情况(这个对象本来就没有自定义的函数)，这样就避免了执行构造函数，这就是高效率的体现。并且，在中间对象继承过程中，父类构造器也没有执行。所以，没有在子类原型上绑定属性。

这种继承方式也被开发人员普遍认为是引用类型最理想的继承范式。

总结

- 模式(简述)：

    - 工厂模式：创建中间对象，给中间对象赋添加属性和方法，再返回出去。
    - 构造函数模式：就是自定义函数，并用过 new 关键子创建实例对象。缺点也就是无法复用。
    - 原型模式： 使用 prototype 来规定哪一些属性和方法能被共享。
- 继承
    - 原型链继承:
        - 优点：只调用一次父类构造函数，能复用原型链属性
        - 缺点：部分不想共享属性也被共享，无法传参。
    - 构造函数继承：
        - 优点：可以传参，同属性可以不被共享。
        - 缺点：无法使用原型链上的属性
    - 组合继承
        - 优点：可以传参，同属性可以不被共享，能使用原型链上的属性。
        - 缺点：父类构造函数被调用2次，子类原型有冗余属性。
    - 原型式继承：(用于对象与对象之间)
        - 优点：在对象与对象之间无需给每个对象单独创建自定义函数即可实现对象与对象的继承，无需调用构造函数。
        - 缺点：父类属性被完全共享。
    - 寄生式继承：
        - 优点：基于原型式继承仅仅可以为子类单独提供一些功能(属性)，无需调用构造函数。
        - 缺点：父类属性被完全共享。
    - 寄生组合继承：
        - 优点：组合继承+寄生式继承，组合继承缺点在于调用两次父类构造函数，子类原型有冗余属性，寄生式继承的特性规避了这类情况，集寄生式继承和组合继承的优点与一身，是实现基于类型继承的最有效方式。

Object.create()

> `Object.create()`方法创建一个新对象，使用现有的对象来提供新创建的对象的**proto**。

```
const person = {
  isHuman: false,
  printIntroduction: function() {
    console.log(`我的名字是 ${this.name}. 我是人吗? ${this.isHuman}`);
  }
};

const me = Object.create(person);

me.name = 'Matthew';
me.isHuman = true;

me.printIntroduction();
// "我的名字是 Matthew. 我是人吗? true"
```

用于原型继承

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

  Animal.prototype.showName = function () {
    console.log(this.name, `我是${this.constructor.name}类`);
  }
  Animal.prototype.showAge = function () {
    console.log(this.age, `我是${this.constructor.name}类`);
  }

  function Pig(name, age, sex = "公") {
    Animal.call(this, name, age);
    this.sex = sex;
  }
  Pig.prototype = Object.create(Animal.prototype);
  Pig.prototype.constructor = Pig;
  Pig.prototype.showSex = function () {
    console.log(this.sex, `我是${this.constructor.name}类`);
  }

  let pig = new Pig('佩奇', 1, '母');
  console.log(pig);//Pig {name: "佩奇", age: 1, sex: "母"}
  pig.showName(); //佩奇
  pig.showAge(); //1
  pig.showSex(); //母
```

多重继承

```
function Parent1(name) {
    this.name = name;
  }
  Parent1.prototype.showName = function () {
    console.log(this.name)
  }
  Parent1.prototype.showAge = function () {
    console.log(this.age)
  }

  function Parent2(age) {
    this.age = age;
  }
  Parent2.prototype.showSomething = function () {
    console.log('something')
  }
```

```
function Child(name, age, address) {
  Parent1.call(this, name);
  Parent2.call(this, age);
  this.address = address;
}

function mixProto(targetClass, parentClass, otherParent) {
  targetClass.prototype = Object.create(parentClass.prototype);
  Object.assign(targetClass.prototype, otherParent.prototype);
}

mixProto(Child, Parent1, Parent2)
var child = new Child('佩奇', 3, '火星');
console.log(child); //Child {name: "佩奇", age: 3, address: "火星"}
child.showName();//佩奇
child.showAge();//3
child.showSomething(); //something
```

Object 深入

Object方法

```
Object.getOwnPropertyNames()
//方法返回一个由指定对象的所有自身属性的属性名（包括不可枚举属性但不包括Symbol值作为名称的属性）组成的数组。

Object.getPrototypeOf()
//方法返回指定对象的原型（内部[[Prototype]]属性的值）。

Object.getOwnPropertyDescriptors()
//方法用来获取一个对象的所有自身属性的描述符。

Object.getOwnPropertyDescriptor()
//方法返回指定对象上一个自有属性对应的属性描述符。（自有属性指的是直接赋予该对象的属性，不需要从原型链上进行查找的属性）

Object.assign()
//方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。

Object.create()
//方法创建一个新对象，使用现有的对象来提供新创建的对象的__proto__。（请打开浏览器控制台以查看运行结果。）

Object.freeze()
//方法可以冻结一个对象。一个被冻结的对象再也不能被修改；冻结了一个对象则不能向这个对象添加新的属性，不能删除已有属性，不能修改该对象已有属性的可枚举性、可配置性、可写性，以及不能修改已有属性的值。此外，冻结一个对象后该对象的原型也不能被修改。freeze() 返回和传入的参数相同的对象。


Object.isFrozen()
```

```
//方法判断一个对象是否被冻结。

Object.isSealed()
//方法判断一个对象是否被密封。

hasOwnProperty()
//方法会返回一个布尔值，指示对象自身属性中是否具有指定的属性（也就是，是否有指定的键）。

isPrototypeOf()
//方法用于测试一个对象是否存在于另一个对象的原型链上。

Object.is()
//方法判断两个值是否为同一个值。
```

Object.defineProperty()

> 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回此对象。

语法

`Object.defineProperty(obj, prop, descriptor)`

参数

- `obj`

  要定义属性的对象。

- `prop`

  要定义或修改的属性的名称或 `Symbol` 。

- `descriptor`

  要定义或修改的属性描述符。

返回值

被传递给函数的对象。

该方法允许精确地添加或修改对象的属性。通过赋值操作添加的普通属性是可枚举的，在枚举对象属性时会被枚举到（for...in 或 Object.keys 方法），可以改变这些属性的值，也可以删除这些属性。这个方法允许修改默认的额外选项（或配置）。默认情况下，使用 Object.defineProperty() 添加的属性值是不可修改（immutable）的。

对象里目前存在的属性描述符有两种主要形式：数据描述符和存取描述符。数据描述符是一个具有值的属性，该值可以是可写的，也可以是不可写的。存取描述符是由 getter 函数和 setter 函数所描述的属性。一个描述符只能是这两者其中之一；不能同时是两者。

这两种描述符都是对象。它们共享以下可选键值（默认值是指在使用 Object.defineProperty() 定义属性时的默认值）：

- `configurable`

当且仅当该属性的 `configurable` 键值为 `true` 时，该属性的描述符才能够被改变，同时该属性也能从对应的对象上被删除。 **默认为 `false`**。

- `enumerable`

  当且仅当该属性的 `enumerable` 键值为 `true` 时，该属性才会出现在对象的枚举属性中。 **默认为 `false`**。

数据描述符还具有以下可选键值：

- `value`

  该属性对应的值。可以是任何有效的 JavaScript 值（数值，对象，函数等）。 **默认为 `undefined`**。

- `writable`

  当且仅当该属性的 `writable` 键值为 `true` 时，属性的值，也就是上面的 `value`，才能被赋值运算符改变。 **默认为 `false`**。

存取描述符还具有以下可选键值：

- `get`

  属性的 getter 函数，如果没有 getter，则为 `undefined`。当访问该属性时，会调用此函数。执行时不传入任何参数，但是会传入 `this` 对象（由于继承关系，这里的 `this` 并不一定是定义该属性的对象）。该函数的返回值会被用作属性的值。 **默认为 `undefined`**。

- `set`

  属性的 setter 函数，如果没有 setter，则为 `undefined`。当属性值被修改时，会调用此函数。该方法接受一个参数（也就是被赋予的新值），会传入赋值时的 `this` 对象。 **默认为 `undefined`**。

描述符默认值汇总

- 拥有布尔值的键 `configurable`、`enumerable` 和 `writable` 的默认值都是 `false`。
- 属性值和函数的键 `value`、`get` 和 `set` 字段的默认值为 `undefined`。

描述符可拥有的键值

	`configurable`	**`enumerable`**	**`value`**	**`writable`**	**`get`**	**`set`**
数据描述符	可以	可以	可以	可以	不可以	不可以
存取描述符	可以	可以	不可以	不可以	可以	可以

如果一个描述符不具有 `value`、`writable`、`get` 和 `set` 中的任意一个键，那么它将被认为是一个数据描述符。如果一个描述符同时拥有 `value` 或 `writable` 和 `get` 或 `set` 键，则会产生一个异常。

记住，这些选项不一定是自身属性，也要考虑继承来的属性。为了确认保留这些默认值，在设置之前，可能要冻结 `Object.prototype`，明确指定所有的选项，或者通过 `Object.create(null)` 将 `__proto__` 属性指向 `null`。

`Object.defineProperties()` 可以对多条属性进行修改

```
var obj = {};
Object.defineProperties(obj, {
  'property1': {
    value: true,
    writable: true
  },
  'property2': {
    value: 'Hello',
    writable: false
  }
  // etc. etc.
});
```

Object.entries()

> 方法返回一个给定对象自身可枚举属性的键值对数组，其排列与使用 for...in 循环遍历该对象时返回的顺序一致（区别在于 for-in 循环还会枚举原型链中的属性）。

语法

```
Object.entries(obj)
```

参数

- obj

  可以返回其可枚举属性的键值对的对象。

返回值

给定对象自身可枚举属性的键值对数组。

```
const object1 = {
  a: 'somestring',
  b: 42
};

for (const [key, value] of Object.entries(object1)) {
  console.log(`${key}: ${value}`);
}

// expected output:
// "a: somestring"
// "b: 42"
// order is not guaranteed
```

Object.keys()

> 方法会返回一个由一个给定对象的自身可枚举属性组成的数组，数组中属性名的排列顺序和正常循环遍历该对象时返回的顺序一致 。。

语法

```
Object.keys(obj)
```

参数

- `obj`

  要返回其枚举自身属性的对象。

返回值

一个表示给定对象的所有可枚举属性的字符串数组。

```
const arr = ['a', 'b', 'c'];
console.log(Object.keys(arr)); // console: ['0', '1', '2']

// array like object
const obj = { 0: 'a', 1: 'b', 2: 'c' };
console.log(Object.keys(obj)); // console: ['0', '1', '2']
```

Object.fromEntries()

> 方法把键值对列表转换为一个对象。

语法

```
Object.fromEntries(iterable);
```

参数

- **iterable**

  类似 `Array` 、 `Map` 或者其它实现了可迭代协议的可迭代对象。

返回值

一个由该迭代对象条目提供对应属性的新对象

```
const map = new Map([ ['foo', 'bar'], ['baz', 42] ]);
const obj = Object.fromEntries(map);
console.log(obj); // { foo: "bar", baz: 42 }
```

Object.preventExtensions()

> 方法让一个对象变的不可扩展，也就是永远不能再添加新的属性。

语法

```
Object.preventExtensions(obj)
```

参数

- obj

  将要变得不可扩展的对象。

返回值

已经不可扩展的对象。

```
var obj = {};
var obj2 = Object.preventExtensions(obj);
obj === obj2;  // true

// 字面量方式定义的对象默认是可扩展的.
var empty = {};
Object.isExtensible(empty) //=== true

// ...但可以改变.
Object.preventExtensions(empty);
Object.isExtensible(empty) //=== false

// 使用Object.defineProperty方法为一个不可扩展的对象添加新属性会抛出异常.
var nonExtensible = { removable: true };
Object.preventExtensions(nonExtensible);
Object.defineProperty(nonExtensible, "new", { value: 8675309 }); // 抛出TypeError异常

// 在严格模式中,为一个不可扩展对象的新属性赋值会抛出TypeError异常.
function fail()
{
  "use strict";
  nonExtensible.newProperty = "FAIL"; // throws a TypeError
}
fail();const map = new Map([ ['foo', 'bar'], ['baz', 42] ]);
const obj = Object.fromEntries(map);
console.log(obj); // { foo: "bar", baz: 42 }
```