JavaScript 高级

基本概念复习

> 由于 JavaScript 高级还是针对 JavaScript 语言本身的一个进阶学习，我们对所学的 JavaScript 相关知识点做一个复习总结。

JavaScript 是什么

- 解析执行：轻量级解释型的，或是 JIT 编译型的程序设计语言
- 语言特点：动态
- 执行环境：在宿主环境（host environment）下运行，浏览器是最常见的 JavaScript 宿主环境
  - 但是在很多非浏览器环境中也使用 JavaScript ，例如 node.js
- 编程范式：基于原型、多范式的动态脚本语言，并且支持面向对象、命令式和声明式（如：函数式编程）编程风格

JavaScript 的组成

组成部分	说明
Ecmascript	描述了该语言的语法和基本对象
DOM	描述了处理网页内容的方法和接口
BOM	描述了与浏览器进行交互的方法和接口

基本概念

> 本小节快速过即可，主要是对学过的内容做知识点梳理。

- 语法
  - 区分大小写
  - 标识符
  - 注释
  - 严格模式
  - 语句
- 关键字和保留字
- 变量
- 数据类型
  - typeof 操作符
  - Undefined
  - Null

    - Boolean
    - Number
    - String
    - Object
- 操作符
- 流程控制语句
- 函数

JavaScript 中的数据类型

JavaScript 有 5 种简单数据类型：`Undefined、Null、Boolean、Number、String` 和 1 种复杂数据类型 `Object` 。

基本类型（值类型）

- Undefined
- Null
- Boolean
- Number
- String

复杂类型（引用类型）

- Object
- Array
- Date
- RegExp
- Function
- 基本包装类型
    - Boolean
    - Number
    - String
- 单体内置对象
    - Global
    - Math

类型检测

- `typeof`
- `instanceof`
- `Object.prototype.toString.call()`

值类型和引用类型在内存中的存储方式

- 值类型按值存储
- 引用类型按引用存储

值类型复制和引用类型复制

- 值类型按值复制
- 引用类型按引用复制

值类型和引用类型参数传递

- 值类型按值传递
- 引用类型按引用传递

值类型与引用类型的差别

- 基本类型在内存中占据固定大小的空间，因此被保存在栈内存中
- 从一个变量向另一个变量复制基本类型的值，复制的是值的副本
- 引用类型的值是对象，保存在堆内存
- 包含引用类型值的变量实际上包含的并不是对象本身，而是一个指向该对象的指针
- 从一个变量向另一个变量复制引用类型的值的时候，复制是引用指针，因此两个变量最终都指向同一个对象

小结

- 类型检测方式
- 值类型和引用类型的存储方式
- 值类型复制和引用类型复制
- 方法参数中 值类型数据传递 和 引用类型数据传递

JavaScript 执行过程

JavaScript 运行分为两个阶段：

- 预解析
    - 全局预解析（所有变量和函数声明都会提前；同名的函数和变量函数的优先级高）
    - 函数内部预解析（所有的变量、函数和形参都会参与预解析）
        - 函数
        - 形参
        - 普通变量
- 执行

先预解析全局作用域，然后执行全局作用域中的代码，在执行全局代码的过程中遇到函数调用就会先进行函数预解析，然后再执行函数内代码。

JavaScript 面向对象编程

面向对象介绍

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。

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
- 继承性
- [多态性]

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：

```
var std1 = { name: '小张', score: 98 }
var std2 = { name: '小李', 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）：

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('小张', 98)
var std2 = new Student('小李', 81)
```

实例对象具有自己的具体行为（给对象发消息）：

```
std1.printScore() // => 姓名：小张  成绩：98
std2.printScore() // => 姓名：小李  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

对象字面量来创建：

```
var person = {
  name: '小张',
  age: 18,
  sayHi: 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？

```
var person1 = {
  name: '小张',
  age: 18,
  sayHi: function () {
    console.log(this.name)
  }
}
var person2 = {
  name: '小李',
  age: 16,
  sayHi: 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。

`new Object()` 创建：

```
var person = new Object()
person.name = '小何'
person.age = 18
person.sayHi = function () {
  console.log(this.name)
}
```

工厂函数创建对象

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,
    sayHi: function () {
      console.log(this.name)
    }
  }
}
var p1 = createPerson('小张', 18)
var p2 = createPerson('小李', 18)
//创建对象---->实例化一个对象,的同时对属性进行初始化
    /*
    * 共同点:都是函数,都可以创建对象,都可以传入参数
    * 工厂模式:
    * 函数名是小写
    * 有new,
    * 有返回值
    * new之后的对象是当前的对象
    * 直接调用函数就可以创建对象
    * 自定义构造函数:

    * 函数名是大写(首字母)
```

```
    * 没有new
    * 没有返回值
    * this是当前的对象
    * 通过new的方式来创建对象
    * */
```

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

构造函数

> 自定义构造函数创建对象做的事情

```
function Person (name, age) {
  this.name = name
  this.age = age
  this.sayHi = function () {
    console.log(this.name)
  }
}
var p1 = new Person('小张', 18)
p1.sayHi() // => 小张
var p2 = new Person('小李', 23)
p2.sayHi() // => 小李
 /*
    * 1.开辟空间存储对象
    * 2.把this设置为当前的对象
    * 3.设置属性和方法的值
    * 4.把this对象返回
    * */
```

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.sayHi = 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

```
//面向对象的思想是----抽象的过程---->实例化的过程
    //小李这个人,姓名,年龄,性别, 吃饭,打招呼,睡觉
    //自定义构造函数----->实例化对象
    function Person(name,age,sex) {
      this.name=name;
      this.age=age;
      this.sex=sex;
      this.eat=function () {
        console.log("我想吃肉");
      };
    }
    //构造函数---->创建对象
    var p1=new Person("小何",38,"女");
    //per.eat();//吃
    //实例对象是通过构造函数来创建
    //实例对象会指向自己的构造函数(暂时理解,是错误的)
    //把这个对象的结构显示出来

    console.dir(p1);
```

```
    console.dir(Person);
    //实例对象的构造器(构造函数)
    //实例对象的构造器是指向Person的,结果是true,所以,这个实例对象p1就是通过Person来创建的
    console.log(p1.constructor==Person);//
    console.log(p1.__proto__.constructor==Person);
    console.log(p1.__proto__.constructor==Person.prototype.constructor);
    //构造函数
    function Animal(name) {
      this.name=name;
    }
    //实例对象
    var dog=new Animal("小白狗");
    console.dir(dog);//实例对象
    console.dir(Animal);//构造函数的名字
    console.log(dog.__proto__.constructor==Person);
    console.log(dog.__proto__.constructor==Animal);
    //判断这个对象是不是这种数据类型
    console.log(dog.constructor==Animal);
    console.log(dog instanceof Person);
    //总结:
    /*
    * 实例对象和构造函数之间的关系:
    * 1. 实例对象是通过构造函数来创建的---创建的过程叫实例化
    * 2.如何判断对象是不是这个数据类型?
    *  1) 通过构造器的方式 实例对象.构造器==构造函数名字
    *  2) 对象 instanceof 构造函数名字
    *  尽可能的使用第二种方式来识别,为什么?原型讲完再说
    * */
```

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：

> 构造函数创建对象带来的问题

```
//    function Person(name,age) {
//      this.name=name;
//      this.age=age;
//      this.eat=function () {
//        console.log("我想吃蔬菜");
//      }
//    }
//    var p1=new Person("小张",20);
//    var p2=new Person("小李",30);
//    console.dir(p1);
//    console.dir(p2);
function eat() {
  console.log("我想吃蔬菜");
}
var eat=10;
function Person(name,age) {
  this.name=name;
  this.age=age;
  this.eat=eat;
}
var p1=new Person("小张",20);
var p2=new Person("小李",30);
console.dir(p1);
console.dir(p2);
console.log(p1.eat==p2.eat);
//通过原型来解决---------数据共享,节省内存空间,作用之一
//    per1.eat();
//    per2.eat();
//    //不是同一个方法
//    console.log(p1.eat==p2.eat);
//    for(var i=0;i<100;i++){
//      var p=new Person("嘎嘎",20);
//      p.eat();
//    }
```

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 sayHi = function () {
  console.log('hello ' + this.name)
}
function Person (name, age) {
  this.name = name
  this.age = age
  this.type = 'human'
  this.sayHi = sayHi
}
var p1 = new Person('李白', 18)
var p2 = new Person('张小凡', 16)
console.log(p1.sayHi === p2.sayHi) // => 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：

```
var p1 = {
  sayHi: 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.sayHi = p1.sayhi
  this.sayAge = p1.sayAge
}

var p1 = new Person('李小凡', 18)
var p2 = new Person('张忠', 16)

console.log(p1.sayHi === p2.sayHi) // => 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。

> 原型添加方法解决数据共享问题

```
function Person(name,age) {
  this.name=name;
  this.age=age;
}
//通过原型来添加方法,解决数据共享,节省内存空间
Person.prototype.eat=function () {
  console.log("吃肉");
};
var p1=new Person("小明",20);
var p2=new Person("小红",30);
console.log(p1.eat==p2.eat);//true
console.dir(p1);
console.dir(p2);
//实例对象中根本没有eat方法,但是能够使用,这是为什么?
```

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。

> 原型

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

```
      this.age=age;
    }
    //通过原型来添加方法,解决数据共享,节省内存空间
    Person.prototype.eat=function () {
      console.log("吃凉菜");
    };
    var p1=new Person("小明",20);
    var p2=new Person("小红",30);
    console.dir(p1);
    console.dir(p2);
    console.dir(Person);
    p1.__proto__.eat();
    console.log(p1.__proto__==Person.prototype);
    /*
    * 原型?
    * 实例对象中有__proto__这个属性,叫原型,也是一个对象,这个属性是给浏览器使用,不是标准的属性----->__proto__----->可以叫原型对象
    * 构造函数中有prototype这个属性,叫原型,也是一个对象,这个属性是给程序员使用,是标准的属性------>prototype--->可以叫原型对象
    * 实例对象的__proto__和构造函数中的prototype相等--->true
    * 又因为实例对象是通过构造函数来创建的,构造函数中有原型对象prototype
    * 实例对象的__proto__指向了构造函数的原型对象prototype
    * */
    //window是对象
    //document是属性,document也是一个对象
    //write()是方法
    //window.document.write("哈哈");
    //对象.style.color=值;
```

> 体会面向过程和面向对象的编程思想

```
 <style>
    div {
      width: 300px;
      height: 200px;
      background-color: red;
    }
  </style>
</head>
<body>
<input type="button" value="显示效果" id="btn"/>
<div id="dv"></div>
<script src="common.js"></script>
<script>
  function ChangeStyle(btnObj, dvObj, json) {
    this.btnObj = btnObj;
    this.dvObj = dvObj;
    this.json = json;
  }
  ChangeStyle.prototype.init = function () {
    //点击按钮,改变div多个样式属性值
    var that = this;
    this.btnObj.onclick = function () {//按钮
```

```
      for (var key in that.json) {
        that.dvObj.style[key] = that.json[key];
      }
    };
  };
  //实例化对象
  var json = {"width": "500px", "height": "800px", "backgroundColor": "blue", "opacity": "0.2"};
  var cs = new ChangeStyle(my$("btn"), my$("dv"), json);
  cs.init();//调用方法
  //点击p标签,设置div的样式
</script>
<script>
  //点击按钮,改变div的背景颜色
  //  document.getElementById("btn").onclick=function () {
  //      document.getElementById("dv").style.backgroundColor="yellow";
  //  };
  //面向对象思想----初级的
  //按钮是一个对象,div是一个对象,颜色是一个属性
  //  function ChangeStyle(btnId,dvId,color) {
  //    this.btnObj=document.getElementById(btnId);//按钮对象
  //    this.dvObj=document.getElementById(dvId);//div对象
  //    this.color=color;//颜色
  //  }
  //  //数据共享来改变背景颜色
  //  ChangeStyle.prototype.init=function () {
  //    //console.log(this);//就是实例对象---就是当前对象
  //    var that=this;
  //    //点击按钮,改变div的背景颜色
  //    this.btnObj.onclick=function () {
  //      that.dvObj.style.backgroundColor=that.color;
  //    };
  //  };
  //实例化对象
  //  var cs=new ChangeStyle("btn","dv","yellow");
  //  cs.init();
  //  function Person() {
  //    this.sayHi=function () {
  //      console.log(this);
  //    };
  //  }
  //  var p=new Person();
  //  p.sayHi();
</script>
```

> 复习原型

```
//构造函数
    function Person(sex,age) {
      this.sex=sex;
      this.age=age;
    }
    //通过原型添加方法
    Person.prototype.sayHi=function () {
```

```
    console.log("打招呼,您好");
  };
  var per=new Person("男",20);
  console.log(per.__proto__.constructor==Person.prototype.constructor);//实例对象
  console.dir(Person);//构造函数的名字
  var per2=new Person("女",30);
  console.log(per.sayHi==per2.sayHi);
  //实例对象中有两个属性(这两个属性是通过构造函数来获取的),__proto__这个属性
  //构造函数中并没有sex和age的两个属性
  /*
  * 实例对象中有个属性,__proto__,也是对象,叫原型,不是标准的属性,浏览器使用的
  * 构造函数中有一个属性,prototype,也是对象,叫原型,是标准属性,程序员使用
  * 原型---->__proto__或者是prototype,都是原型对象,
  * 原型的作用:共享数据,节省内存空间
  * */
```

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。

> 构造函数和实例对象和原型对象之间的关系

```
//    //通过构造函数实例对象,并初始化
//    var arr=new Array(10,20,30,40);
//    //join是方法,实例对象调用的方法
//    arr.join("|");
//    console.dir(arr);
//    //join方法在实例对象__proto__原型
//    console.log(arr.__proto__==Array.prototype);
  </script>
  <script>
    //原型的作用之一:共享数据,节省内存空间
    //构造函数
    function Person(age,sex) {
      this.age=age;
      this.sex=sex;
    }
    //通过构造函数的原型添加一个方法
    Person.prototype.eat=function () {
      console.log("明天中午吃完饭就要演讲,好痛苦");
    };
    var per=new Person(20,"男");
   // per.__proto__.eat();
    per.eat();
   // per.eat();
    //构造函数,原型对象,实例对象之间的关系
    console.dir(Person);
   //console.dir(per);
```

> 总结三者之间的关系

```
    /*
    * 构造函数可以实例化对象
    * 构造函数中有一个属性叫prototype,是构造函数的原型对象
    * 构造函数的原型对象(prototype)中有一个constructor构造器,这个构造器指向的就是自己所在的原型对象所在的构造函数
    * 实例对象的原型对象(__proto__)指向的是该构造函数的原型对象
    * 构造函数的原型对象(prototype)中的方法是可以被实例对象直接访问的
    * */
```

> 利用原型共享数据

```
//什么样子的数据是需要写在原型中?
    //需要共享的数据就可以写原型中
    //原型的作用之一:数据共享
    //属性需要共享,方法也需要共享
    //不需要共享的数据写在构造函数中,需要共享的数据写在原型中
    //构造函数
    function Student(name,age,sex) {
      this.name=name;
      this.age=age;
      this.sex=sex;
    }
    //所有学生的身高都是188,所有人的体重都是55
    //所有学生都要每天写500行代码
    //所有学生每天都要吃一个10斤的西瓜
    //原型对象
    Student.prototype.height="188";
    Student.prototype.weight="55kg";
    Student.prototype.study=function () {
      console.log("学习,写500行代码,小菜一碟");
    };
    Student.prototype.eat=function () {
      console.log("吃一个10斤的西瓜");
    };
    //实例化对象,并初始化
    var stu=new Student("晨光",57,"女");
    console.dir(Student);
    console.dir(stu);
//    stu.eat();
//    stu.study();
```

> 原型的简单的语法

```
function Student(name, age, sex) {
      this.name = name;
      this.age = age;
      this.sex = sex;
    }
    //简单的原型写法
    Student.prototype = {
      //手动修改构造器的指向
      constructor:Student,
```

```
  height: "188",
  weight: "55kg",
  study: function () {
    console.log("学习好开心啊");
  },
  eat: function () {
    console.log("我要吃好吃的");
  }
};
var stu=new Student("段飞",20,"男");
stu.eat();
stu.study();
console.dir(Student);
console.dir(stu);
//    Student.prototype.height="188";
//    Student.prototype.weight="55kg";
//    Student.prototype.study=function () {
//      console.log("学习,写500行代码,小菜一碟");
//    };
//    Student.prototype.eat=function () {
//      console.log("吃一个10斤的西瓜");
//    };
//    //实例化对象,并初始化
//    var stu=new Student("晨光",57,"女");
//    console.dir(Student);
//    console.dir(stu);
```

> 原型中的方法是可以相互访问的

```
//    function Person(age) {
//      this.age=age;
//      this.sayHi=function () {
//        console.log("考尼奇瓦");
//        //打招呼的同时,直接调用吃的方法
//      };
//      this.eat=function () {
//        console.log("吃东西啦");
//        this.sayHi();
//      };
//    }
//    //实例对象的方法,是可以相互调用的
//
//    //实例化对象，并初始化
//    var per=new Person(20);
//    //调用方法
//    //per.sayHi();
//    per.eat();
//原型中的方法,是可以相互访问的
function Animal(name,age) {
  this.name=name;
  this.age=age;
}
//原型中添加方法
```

```
Animal.prototype.eat=function () {
  console.log("动物吃东西");
  this.play();
};
Animal.prototype.play=function () {
  console.log("玩球");
  this.sleep();
};
Animal.prototype.sleep=function () {
  console.log("睡觉了");
};
var dog=new Animal("小苏",20);
dog.eat();
//原型对象中的方法,可以相互调用
```

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`

> 实例对象使用属性和方法层层的搜索

```
<script>
    function Person(age,sex) {
      this.age=age;//年龄
      this.sex=sex;
      this.eat=function () {
        console.log("构造函数中的吃");
      };
    }
    Person.prototype.sex="女";
    Person.prototype.eat=function () {
      console.log("原型对象中的吃");
    };
    var per=new Person(20,"男");
    console.log(per.sex);//男
    per.eat();
    console.dir(per);
    /*
    * 实例对象使用的属性或者方法,先在实例中查找,找到了则直接使用,找不到则,去实例对象的__proto__指向的原型对象prototype中找,找到了则使用,找不到则报错
    * */
  </script>
```

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

练习：为数组对象和字符串对象扩展原型方法。

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

> 为内置对象添加原型方法

```
<script>
    //为内置对象添加原型方法
//    var arr=new Array(10,20,30,40,50);
    //    arr.join("|");
    //    console.dir(arr);
    //    var str=new String("哦,唛嘎的");
    //    str.indexOf("哦");
    //    console.dir(str);
    //    var dt=new Date();
    //    dt.getFullYear();
    //    console.dir(dt);
    //实例中的方法如果没有,去创建该实例对象的构造函数的原型对象中找
    //我们能否为系统的对象的原型中添加方法,相当于在改变源码
    //我希望字符串中有一个倒序字符串的方法
    String.prototype.myReverse=function () {
      for(var i=this.length-1;i>=0;i--){
        console.log(this[i]);
      }
    };
    var str="abcdefg";
    str.myReverse();
    //为Array内置对象的原型对象中添加方法
    Array.prototype.mySort=function () {
      for(var i=0;i<this.length-1;i++){
          for(var j=0;j<this.length-1-i;j++){
              if(this[j]<this[j+1]){
                  var temp=this[j];
                this[j]=this[j+1];
                this[j+1]=temp;
              }//end if
          }// end for
      }//end for

    };
```

```
    var arr=[100,3,56,78,23,10];
    arr.mySort();
    console.log(arr);
    String.prototype.sayHi=function () {
      console.log(this+"哈哈,我又变帅了");
    };
    //字符串就有了打招呼的方法
    var str2="小杨";
    str2.sayHi();
  </script>
</head>
```

> 把局部变量变成全局变量

```
    //函数的自调用---自调用函数
    //一次性的函数--声明的同时,直接调用了
//    (function () {
//      console.log("函数");
//    })();
    //页面加载后.这个自调用函数的代码就执行完了
//    (function (形参) {
//      var num=10;//局部变量
//    })(实参);
//    console.log(num);
//    (function (win) {
//      var num=10;//局部变量
//      //js是一门动态类型的语言,对象没有属性,点了就有了
//      win.num=num;
//    })(window);
//    console.log(num);
    //如何把局部变量变成全局变量?
    //把局部变量给window就可以了
```

> 把随机数对象暴露给window成为全局对象

```
    //通过自调用函数产生一个随机数对象,在自调用函数外面,调用该随机数对象方法产生随机数
    (function (window) {
      //产生随机数的构造函数
      function Random() {
      }
      //在原型对象中添加方法
      Random.prototype.getRandom = function (min,max) {
        return Math.floor(Math.random()*(max-min)+min);
      };
      //把Random对象暴露给顶级对象window--->外部可以直接使用这个对象
      window.Random=Random;
    })(window);
    //实例化随机数对象
    var rm=new Random();
    //调用方法产生随机数
    console.log(rm.getRandom(0,5));

    //全局变量
```

> 案例随机小方块

```
<style>
    .map{
      width: 800px;
      height: 600px;
      background-color: #CCC;
      position: relative;
    }
  </style>
</head>
<body>
<div class="map"></div>
<script src="common.js"></script>
<script>
  //产生随机数对象的
  (function (window) {
    function Random() {
    }
    Random.prototype.getRandom=function (min,max) {
      return Math.floor(Math.random()*(max-min)+min);
    };
    //把局部对象暴露给window顶级对象,就成了全局的对象
    window.Random=new Random();
  })(window);//自调用构造函数的方式,分号一定要加上

  //产生小方块对象
  (function (window) {
    //console.log(Random.getRandom(0,5));
    //选择器的方式来获取元素对象
    var map=document.querySelector(".map");
    //食物的构造函数
    function Food(width,height,color) {
      this.width=width||20;//默认的小方块的宽
      this.height=height||20;//默认的小方块的高
      //横坐标,纵坐标
      this.x=0;//横坐标随机产生的
      this.y=0;//纵坐标随机产生的
      this.color=color;//小方块的背景颜色
      this.element=document.createElement("div");//小方块的元素
    }
    //初始化小方块的显示的效果及位置---显示地图上
    Food.prototype.init=function (map) {
      //设置小方块的样式
      var div=this.element;
      div.style.position="absolute";//脱离文档流
      div.style.width=this.width+"px";
      div.style.height=this.height+"px";
      div.style.backgroundColor=this.color;
      //把小方块加到map地图中
      map.appendChild(div);
      this.render(map);
```

```
    };
    //产生随机位置
    Food.prototype.render=function (map) {
      //随机产生横纵坐标
      var x=Random.getRandom(0,map.offsetWidth/this.width)*this.width;
      var y=Random.getRandom(0,map.offsetHeight/this.height)*this.height;
      this.x=x;
      this.y=y;
      var div=this.element;
      div.style.left=this.x+"px";
      div.style.top=this.y+"px";
    };
    //实例化对象
    var fd=new Food(20,20,"green");
    fd.init(map);
    console.log(fd.x+"===="+fd.y);
  })(window);
</script>
```