# 一 对象的创建

## 1 方式一：字面量

|  |
| --- |
| var obj = {  name : "张学友",  sayHello : function () {  console.log("Hello World");  }  } |

缺点：不能把JSON对象当做模板来进行new操作，只能使用点语法一个一个赋值。

## 2 方式二：Object创建

var p = new Object();

p.age = 13;

缺点同字面量创建。

## 3 方式三：自定义构造函数

|  |
| --- |
| **function** Person() { *//首字母大写用来强调自己是构造函数* **this**.name = **"尼古拉斯凯奇"**; //不添加this类似于window.name 会污染全局   **this**.age = 50;  **this**.sayHello = **function** () {  console.log(**"Hello World"**);  } } **var** p = **new** Person(); |

注意：

1-构造函数是用来初始化对象的，new才是用来创建对象的；

2-构造函数如果没有参数，new调用构造函数时，可以省略括号；

3-构造函数如果没有return，会把构造函数内的this对象返回给对象。

4-new的执行过程是：

粗略过程：申请存储空间，this指向当前对象，设置属性方法，返回this；

详细过程：

a创建一个空对象

b把this指向这个空对象

c把空对象的内部原型 指向 构造函数的原型

d构造函数如果没有return，把空对象返回

如果return了基本数据类型，则仍然返回这个空对象

如果return了复杂数据类型，则替换掉空对象

代码演示---非可运行实际代码：

|  |
| --- |
| function Cat(){  this = {}; //1  this.\_\_proto\_\_ = Cat.prototype; //2  this.age = 18; //3  this.show = function(){console.log(123)};  return this; //4  } |

## 4演进：原型创建

如果使用多个构造函数创建对象，多个构造函数的内部的方法一样，这时候会造成资源浪费，我们可以在外部将这个通用函数书写出来，但是又在成了全局污染：

|  |
| --- |
| **function** studyMethod() {  console.log(**this**.name + **"刻苦学习"**); } **function** Student(stuName) {  **this**.name = stuName;  **this**.study = studyMethod; } **var** stu = **new** Student(**"lis"**); stu.study(); studyMethod(); *//没有调用者，会输出 undefined刻苦学习* |

解决办法：使用原型创建。

在构造函数被创建出来的时候，系统会默认帮构造函数创建并关联一个空对象，即——原型对象，获取原型的方式：

构造函数.prototype

原型中的成员都可以被相应构造函数创建的对象所使用。利用这个特点，我们可以将构造函数中的的函数，放到原型中存储，就解决了全局变量污染的问题。

|  |
| --- |
| *//构造函数：构造函数中的方法将覆盖原型中的同名方法* **function** Cat(name){  **this**.name = name;  **this**.eat = **function** () {  console.log(**"构造函数eat..."**);  } }  *//原型添加成员的两种方式* Cat.prototype.eat = **function** () {  console.log(**"原型eat...."**); } Cat.prototype[**"dance"**] = **function** () {  console.log(**"原型dance..."**); }  **var** c = **new** Cat(**"Green"**); c.eat(); c.dance(); |

注意：

1-点语法赋值时，会直接给对象设置属性，对象没有该属性，将直接添加。不会去原型中查找、修改。

2-原型中的属性是引用类型时，所有对象共有该属性，且一个对象对其进行修改，其他对象都会受到影响。

3-一般情况下，不会将属性放在原型中，原型中只存放共享方法。

## 5 工厂模式创建对象

# 二 对象的使用

## 1 对象添加成员

JS中的键值对组合其实就是对象。对象中值类型存储的是对应类型的值，引用类型存储的是引用地址，其数据类型为object，会在内存中单独存储。

我们可以直接通过 点语法 或者 [] 来给对象添加成员。

|  |
| --- |
| **var** obj = {  name : **"lisi"**,  age : 30 }; obj[0] = **function** () {  console.log(**"添加成功"**); } obj[{}] = **function** () { *//类似添加了 obj["object Object"]* console.log(**"添加成功"**); } console.log(obj);  *//{ '0': [Function],name: 'lisi',age: 30,'[object Object]': [Function] }* |

## 2方法调用顺序

调用方法、属性的顺序：

1-在对象自己内部查找

2-在原型中查找

3-以上属性没找到即undefined，方法没找到就会报错 not a function

## 3 如何访问原型

构造函数访问原型： 构造函数.prototype

实例对象访问原型： 实例对象.\_\_proto\_\_

由于部分浏览器不支持实例对象访问原型的方法，所以不推荐这样使用，仅在调试时使用。

## 4 对象属性

对象在创建出来的时候，这个空对象包含2个属性：

1-默认有一个 constructor 属性，指向构造函数；

2-部分浏览器有 \_\_proto\_\_ 属性，指向自己；

3-原型中还有实例对象需要的方法；

4-原型上如果有属性，只有原型自己能修改这个属性；

如果实例对象也创建了一个和原型一样的属性（属性名一样），那么不会去修改原型的值，而是给实例对象自己添加了属性。读取属性先找实例对象自己的，如果没有再去找原型。

注意：如果使用替换的方式替换原型，如果新原型没有constructor属性，会影响三角关系，为保证 构造函数--原型---对象 三者的合理，应在替换时候手动加入constructor属性。

Person.prototype = {

constructor : Person

};

所以我们创建对象还可以这样创建：

|  |
| --- |
| **function** Person(name) {  **this**.name = name; } **var** p1 = **new** Person(**"zs"**); console.log(Person.prototype.constructor); console.log(p1.constructor); *//直接用对象访问原型中的属性* **var** p2 = **new** p1.constructor(**"lisi"**); *//相当与使用了构造函数创建* console.log(p2.name); |

## 5 标准创建对象

|  |
| --- |
| **function** Person(option){  **this**.\_init(option); }  Person.prototype = {  \_init:**function**(option){ *//（构造函数内部方法需要加\_,不加也可以，但不推荐）* **this**.name = option.name;  **this**.age = option.age;  },  show:**function**(){  console.log(1111);  }  };  **var** p = **new** Person({  name:**'lisi'**,  age:30 }); p.show(); |

这样写的好处：构造函数传参创建对象，以后如果需要新增参数，就会去很多地方进行修改，这种标准书写方式就可以避免。

# 三 面向对象三大特性之继承

面向对象三大特性：封装、继承、多态（大多用于强类型语言）

## 1 混入式继承

|  |
| --- |
| **var** singer1= {  name : **"张学友"**,  sayHello : **function** () {  console.log(**"Hello World"**);  } } **var** singer2 = {  } **for**(**var** k **in** singer1){ *//k只能获得键，singer[k]获得值* singer2[k] = singer1[k]; } console.log(singer2); |

## 2 原型继承

因为原型中的成员可以被相关的不同对象共享。

|  |
| --- |
| **var** father = {  sayHello : **function** () {  console.log(**"实现了继承"**);  } }  **function** Son(name,age) {  **this**.name = name;  **this**.age = age; }  *//我们想让son对象继承 father对象的sayHello函数* Son.prototype = father; **var** son = **new** Son(**"lisi"**,30); son.sayHello(); |

注意：直接替换Son原型会有隐患。原型中的成员可能会丢失，我们也可以对原型进行混入式继承：

for(var k in father){

Person.prototype[k] = father[k];

}

## 3 经典继承

|  |
| --- |
| **function** jicheng(obj) {  **var** o = {};  o.\_\_proto\_\_ = obj;  **return** o; } **var** o = jicheng({name : **"zs"**}); console.log(o.name); |

ECMA根据上述方式在ECMA5中制定了标准继承语法： Object.create(obj) 返回值为一个继承了obj的对象。

|  |
| --- |
| *//制作一个实现继承的工具* **function** create(obj) {  **if**(Object.create){  **return** Object.create(obj);  }**else** { *//解决不兼容问题ECMA5* **function** F() {  }  F.prototype = obj;  **return new** F();  } } |

## 4 扩展内置对象

知识点：我们不推荐扩展内置对象，但是可以使用下列方法更安全的扩展。

|  |
| --- |
| **function** MyArray() {} MyArray.prototype = **new** Array(); *//MyArray即成为了具备和Array一样的功能* |

# 四 原型链

### 4.1 原型链介绍

每个对象都有构造函数；

每个构造函数都有原型对象；

每个构造函数的原型都是一个对象，那么这个原型对象也有构造函数，这个构造函数也有原型对象；

这样形成了一个链式结构，成为原型链；

最终的对象为Object。

### 4.2 属性搜索原则

1、先在对象自身查找

2、再去当前对象的原型中查找

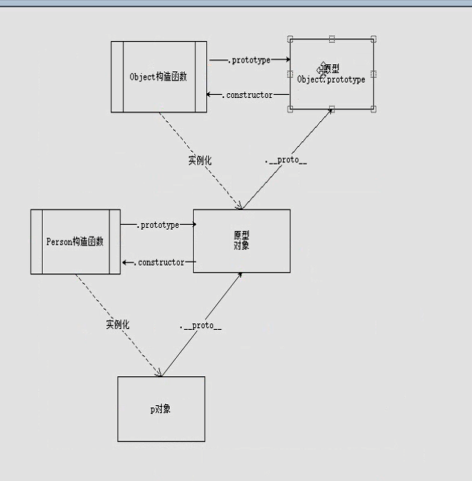
3、再找原型的原型对象...依次向上查找

4、最后查找Objct.prototype

5、找不到报错

|  |
| --- |
| *//原型链展示：动物-人-老师-数学老师* **function** Animal() {  **this**.gender = **"male"**; } Human.prototype = **new** Animal(); Human.prototype.constructor = Human; **function** Human() {  **this**.actionWay = **"run"**; }  Teacher.prototype = **new** Human(); Teacher.prototype.constructor = Teacher; **function** Teacher() {  **this**.skill = **"teach"**; }  MathTeacher.prototype = **new** Teacher(); MathTeacher.prototype.constructor = MathTeacher; **function** MathTeacher() {  **this**.name = **"lisi"**; } **var** t = **new** MathTeacher(); console.log(t); |

### 4.3 原型链关系图



# 5 函数调用模式

|  |
| --- |
| *// 1-函数执行模式* **function** fn1(a,b){  *// console.log(this); //this 即是window* **return** a + b; } console.log(fn1(2,3));  *//2-对象方法调用模式 所有的事件响应方法都是对象方法调用模式 //3-构造器调用模式 this指向构造出来的对象。 //4-call和applye调用模式* **function** fn2(a,b) {  **this**.result = a + b; } **var** s = {}; fn2.call(s,3,4); *//这个方法调用时，this指向s* console.log(s.result); *//apply 和 call 是一样的用法，只不过apply第二个参数用数组进行传递* |