

# JS精华部分理论 与编程实践

郁盛杰

JS和PHP是地球上  
“最好”的编程语言！

# 自我介绍

- ◎ 2009年毕业于东北大学
- ◎ 曾就职于挖财，网易杭研院，携程技术研发中心
- ◎ 曾经负责挖财网，网易云阅读的手机/平板客户端，携程网酒店子站的前端架构与开发

# 关于本次分享

- ◎ 目标听众：JS基础薄弱但对她有兴趣的同学
- ◎ 练习：后续可以下载示例代码，在DevTools中执行，加强理解，更好的方式是看书并练习
- ◎ 期望成果：目标听众基本掌握JS

# 历史回顾

- ◎ 1995年，网景雇了[Brendan Eich](#)，希望他迁移[Scheme](#)编程语言到网景浏览器以增强浏览web的交互体验
- ◎ 仅用了10天，JS就被创造出来了，它是函数式语言和面向对象语言的杂交语言
- ◎ 1996年，JavaScript捐给[ECMA](#)国际组织

- ◎ 1999年， ECMAScript3发布

- ◎ 2009年， ECMAScript5发布

- ◎ 2015年， ECMAScript6发布

- ◎ 。 。 。

- ◎ 借鉴C的语法
- ◎ 借鉴Java的数据类型和内存管理
- ◎ 借鉴Scheme的函数
- ◎ 借鉴Self，使用原型（prototype）继承
- ◎ 借鉴Perl的正则表达式



“JS优秀之处并非原创，原创之处并不优秀”  
---- Brendan Eich

# JS精华部分

- ◎ 对象字面量和数组字面量
- ◎ 函数是头等对象
- ◎ 基于原型继承的动态对象

# 对象/数组字面量

- ◎ JS中除了number, string, boolean, null, undefined五种基本类型，其他所有的值都是对象，数组、函数、正则、对象等都是对象
- ◎ 对象是键值对的集合，键可以是任意字符串，值可以是任意值
- ◎ JS没有传统的数组，但拥有类数组的对象，元素可以是任何值，慢，但是使用方便

# 对象/数组字面量

- ◎ 对象字面量也是JSON作者的灵感来源 (JavaScript Object Notation)
- ◎ 对象字面量的值，数组字面量的元素可以是任意值，包括函数，所以两者有非常强大的表现力

# 对象/数组字面量

```
var hotel = {  
  name: "Hilton",  
  rate: 5,  
  location: "HangZhou",  
  roomTypes: [  
    {  
      standard: {  
        price: 800,  
        area: "30 m²",  
        floor: 5  
      }  
    }  
  ],  
  contact: {  
    tel: "0571-34553434",  
    fax: "0571-343434",  
    weixin: "hz-hilton"  
  }  
};
```

对象字面量

```
hotel.name; // Hilton  
hotel["my prop"] = "prop";  
delete hotel.name;  
hotel.contact.weixin; // "hz-hilton"
```

```
var hotels = [  
  {  
    name: "Hilton",  
    rate: 5  
  },  
  {  
    name: "youhao",  
    rate: 4  
  },  
  {  
    name: "jinjiang",  
    rate: 3  
  },  
  {  
    name: 'rujia',  
    rate: 2  
  },  
  null,  
  "string"  
];
```

数组字面量

```
hotels.push({ name: "7days", rate: 2 });  
hotels.push({ name: "hanting", rate: 3 });  
hotels.pop(); //{name:"hanting"}
```

# 函数是头等对象

- ◎ 函数是JS中最好的特性，用于代码复用、信息隐藏和组合调用
- ◎ 一般来说，所谓JS编程就是将需求分解成一组函数和一套数据结构，再把代码模块化组织起来
- ◎ 在JS中，**函数是对象**，它可以存放在变量、对象和数组中，可以被当做参数传递，可以当做返回值，可以拥有方法。她是一等公民

# 函数是头等对象

- ◎ 函数可以嵌套，可以匿名
- ◎ JS的作用域是静态作用域，但没有块级作用域
- ◎ 函数的参数不能设置类型、默认值，不支持重载，只要函数名相同，JS就认为是同一个函数，后面定义的覆盖前面定义的

# 函数是头等对象

Arguments: 

```
function main() {  
  console.log(arguments.length);  
  console.log(arguments[0]);  
  console.log(arguments[1]);  
}  
main('arg1', 'arg2'); // 2,arg1,arg2  
main(); // 0,undefined,undefined
```

Arguments是一个类数组的对象，没有数组的方法，可以通过[].slice.call(arguments)方便的转成数组



# 函数是头等对象

This:



```
var x = 9;
var module = {
  x: 81,
  getX: function() { return this.x; }
};

var otherModule = {
  x: 100
};

module.getX(); // 81

var getX = module.getX;
//在浏览器中，相当于 window.getX()
getX(); // 9, this 指向全局对象

var boundGetX = getX.bind(otherModule);
boundGetX(); // 100

getX.call(otherModule); // 100
```

bind只是改变函数绑定的对象，不立即执行

call/apply不仅改变，而且立即执行

# 函数的调用方式

## 1. 作为对象的方法调用：

```
var myObject = {  
  value: 0,  
  increment: function () {  
    this.value++;  
  }  
};  
// increment作为myObject对象的方法调用  
myObject.increment();  
myObject.value; // 1
```

也是匿名函数

# 函数的调用方式

## 2. 函数直接调用：

```
function add(a, b) {  
    return a + b;  
}  
add(1, 2); //add函数调用  
  
myObject.double = function () {  
    var that = this;  
    var helper = function () {  
        that.value = add(that.value, that.value);  
    };  
    helper(); // helper函数调用  
};  
myObject.double();  
myObject.value; // 2
```

直接调用

# 函数的调用方式

## 3. 作为构造函数调用：

```
function Con(param) {  
    this.name = param;  
    this.getName = function () {  
        console.log(this.name);  
    }  
}  
// Con作为构造函数被调用  
var obj = new Con("constructor");  
obj.getName(); // constructor
```

Con构造函数调用

# 函数的调用方式

## 4. apply/call方式调用（会改变this指向）：

```
function Con(param) {  
    this.name = param;  
    this.getName = function () {  
        console.log(this.name);  
    }  
}  
  
// Con作为构造函数被调用  
var obj = new Con("constructor");  
obj.getName(); // constructor  
  
var newObj = {  
    name: "apply/call"  
};  
  
//getName用apply的方式调用，此时this已指向newObj  
obj.getName.apply(newObj); // apply/call
```



getName函数通过apply方式调用

# 函数的编程模式

- ◎ 嵌套（略）
- ◎ 递归（略）
- ◎ 闭包
- ◎ 回调
- ◎ 模块
- ◎ 级联
- ◎ 柯里化

# 函数的编程模式

**匿名函数**，也就是lambda函数，在JS中非常常用：

函数的两种声明方式

```
var f = function(x) {  
    console.log( x * x );  
};  
f(); //100
```

函数表达式

函数声明

C++11， Java8已支持匿名函数

```
function f(x) {  
    console.log( x * x );  
};
```

# 函数的编程模式

自执行函数，有多种自执行方式：

```
(function(x) {  
    console.log( x * x );  
})(10); // 100
```

```
(function(x) {  
    console.log( x * x );  
})(10)); // 100
```

```
!function(x) {  
    console.log( x * x );  
}(10); // 100
```

```
+function(x) {  
    console.log( x * x );  
}(10); // 100
```



闭包

闭包很难理解，也很难用语言解释。在JS中，闭包非常强大和实用，示例：

```
function outer(x) {  
  var y = 1;  
  function inner(z) {  
    console.log( x + y + z );  
  }  
  return inner;  
}
```

```
var closure1 = outer(1);  
var closure2 = outer(2);  
console.log(closure1(3)); // 5  
console.log(closure2(4)); // 7
```

闭包

闭包，是词法闭包的简称，是引用了非局部变量([non-local variable](#))的函数与其相关的引用环境组合而成的实体。

上述例子中，x,y对于inner函数来说就是即不是全局，也不是局部变量，它们就是所谓的非局部变量。x,y变量与inner函数会共同存在，闭包可以有多个实例，互不干扰，如上述closure1和closure2。

## 应用闭包，改变DOM的表现：

```
// 网页背景颜色渐变
var fade = function (node) {
  var level = 1;
  var step = function () {
    var hex = level.toString(16);
    node.style.backgroundColor = '#FFF' + hex + hex;
    if (level < 15) {
      level += 1;
      setTimeout(step, 100);
    }
  };
  setTimeout(step, 100);
};
fade(document.body);
```



step函数执行期间始终保持着对level变量的引用

连setTimeout也不能阻止闭包的使用！

闭包在JS事件中的应用:

```
var clickHandlers = function (nodes) {
    var i;
    for (i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = function (e) {
            alert(i); // i引用onclick外部定义的i    弹出6, 6, 6, 6, 6, 6..
        };
    }
};

var clickHandlers = function (nodes) {
    var i;
    for (i = 0; i < nodes.length; i += 1) {
        // 用闭包来处理
        nodes[i].onclick = function (i_nonlocal) {
            return function (e) {
                alert(i_nonlocal); // i_nonlocal引用onclick函数传进来的参数
            };
        }(i);    弹出1, 2, 3, 4, 5, 6...
    }
};

clickHandlers(document.querySelectorAll('p'));
```

# 回调

回调对应着异步执行，示例：

```
// 异步函数，例如ajax请求，Node.js读取文件
var sendRequestAsync = function (request, callback) {
  // do something, create data
  callback('created data');
};
sendRequestAsync('load', function(data){
  console.log('callback invoke, data:', data);
});
```



sendRequestAsync函数的执行不会阻塞后面代码的执行

模块

在JS中，模块是提供对外接口，但隐藏了状态与实现细节的函数或者对象。

```
var Module = function () {  
  var privateVar = 1;  
  var privateFunc = function (p) {  
    console.log('private func call', p);  
  };  
  
  return {  
    method1: function () {  
      privateVar++;  
      console.log(privateVar);  
    },  
    method2: function () {  
      privateFunc(privateVar);  
    }  
  };  
}();  
Module.method1(); //2  
Module.method2(); //private func call, 2  
Module.privateVar; //undefined
```

```
var Module = {  
  _privateVar: 1,  
  _privateFunc: function (p) {  
    console.log('private func call', p);  
  },  
  
  method1: function () {  
    this._privateVar++;  
    console.log(this._privateVar);  
  },  
  method2: function () {  
    this._privateFunc(this._privateVar);  
  }  
};  
Module.method1(); //2  
Module.method2(); //private func call, 2  
Module._privateVar; //2  
Module._privateFunc(100); //100
```

区别在于用函数实现可以真正的实现信息隐藏，用对象字面量则是约定了信息隐藏的格式，加“\_”前缀，但其实外部还是可以访问的



级联

getElement().  
move().  
append().  
height(). —————> jQuery  
on().  
fade();

有兴趣的同学可以读一下前端组基础库psky的代码实现

柯里化

```
var curry = function(a) {  
  return function(b) {  
    return a * a + b * b;  
  }  
};  
  
var foo = curry(3);  
  
var bar1 = foo(1); // 10  
var bar2 = foo(2); // 13
```

也叫部分执行，借用需输入两个参数的函数，输出为有一个固定参数的函数

# 函数式编程

```
var sum = function(x,y) { return x + y; };
var square = function(x) { return x * x; };

var data = [1,1,3,5,5,-2,-1,-4];
var result = data.filter(function(x){ return x<0; }).
    map(square).
    reduce(sum);
console.log(result); //21
```

```
function not(f) {
    return function() {
        var result = f.apply(this, arguments);
        return !result;
    };
}

var even = function(x) { return x % 2 === 0; };
var odd = not(even);
[1,1,3,5,5].every(odd); //true, 数组全是奇数
```

晕了，保持简单！

光有函数还不行，还需要  
对象才能构建web应用程序

# 基于原型继承的动态对象

- ◎ 对象是JS最基本的数据类型，是JS的根基
- ◎ JS中，没有真实的类，但是可以通过构造函数来模拟
- ◎ 我们可以随意给一个对象增加成员，对象也可以从其他对象继承成员
- ◎ 通过原型（prototype）来实现继承



# 对象的创建方式

- 对象字面量

```
var o = {key: "value"}
```

- new 构造函数

```
var o = new Object()
```

```
var date = new Date()
```

```
var type = new Type() // Type是自定义构造函数
```

- Object.create()

```
var child = Object.create({key: "value"})
```

- Prototype对象

```
var f = function() {};
```

```
var protoObj = f.prototype;
```

# 原型对象及构造器

```
function Type() {  
}
```

函数对象创建时，会产生类似这样的一行代码：  
`this.prototype = {constructor : this};`

验证：

```
Type.prototype.constructor === Type; // true  
var type = new Type();  
type.constructor === Type; // true
```

# 面向对象

传统面向对象语言C++/Java的四种重要成员：

- ◎ 实例字段
- ◎ 实例方法
- ◎ 类字段
- ◎ 类方法

类

## JS模拟传统OO语言

```
function Animal(name) {  
  this.name = name;  
}  
Animal.prototype.getName = function() {  
  return this.name;  
};  
Animal.prototype.say = function() {  
  Animal._log(this.saying || '');  
};  
Animal.VER = '1.0.0';  
Animal._log = function(x) {  
  console.debug(Animal.VER, 'Animal:', x);  
}
```

Animal类

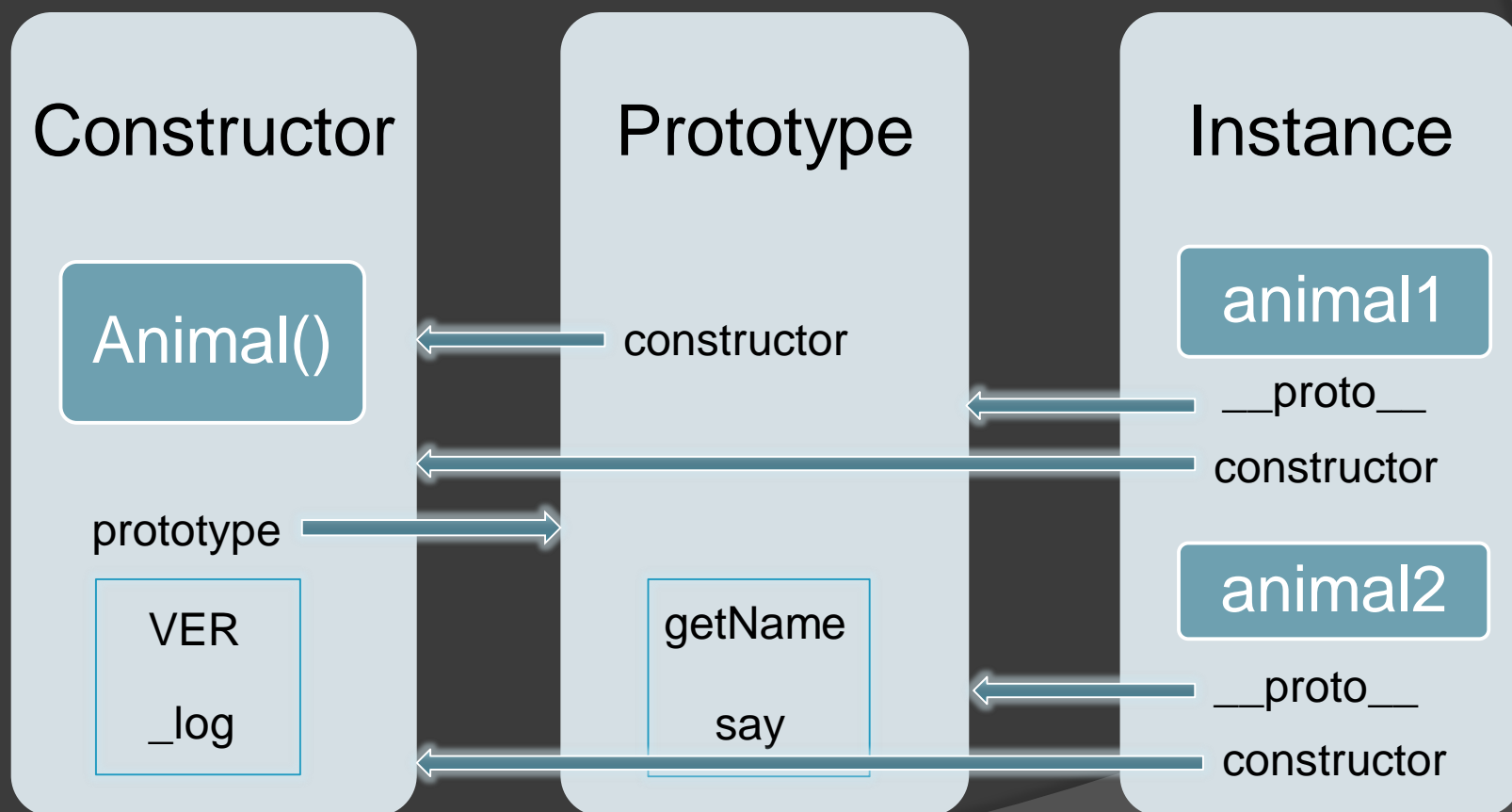
- 实例字段: name
- 实例方法: getName, say
- 类字段: VER
- 类方法: \_log

```
var animal1 = new Animal('a1');  
animal1.getName(); //a1  
animal1.say(); // 1.0.0 Animal:  
var animal2 = new Animal('a2');  
animal2.saying = 'say something';  
animal2.getName(); //a2  
animal2.say(); //1.0.0 Animal: say something
```

牵扯到的三种JS对象，分别是：

- 构造器对象Animal
- 原型对象Animal.prototype
- 实例对象animal1, animal2

# 构造函数/原型/实例间关系



# 原型链继承

```
function Cat(name) {  
  this.name = name;  
  this.saying = 'miao';  
}  
  
Cat.prototype = new Animal();  
Cat.prototype.sleep = function () {  
  console.log('sleep ing');  
};  
  
var cat = new Cat('cat1');  
cat.getName(); // cat1  
cat.say(); //miao  
cat.sleep(); //sleep ing  
cat instanceof Cat; //true  
cat instanceof Animal; //true  
Cat.prototype.isPrototypeOf(cat); //true  
Animal.prototype.isPrototypeOf(cat); //true  
Object.getPrototypeOf(cat) === Cat.prototype; //true  
cat.hasOwnProperty('saying'); //true
```

Cat类继承自Animal类

- Cat类并且拥有自己的原型方法sleep
- cat对象有实例属性name, saying, 继承而来的方法getName, say, sleep

# 原型链查找

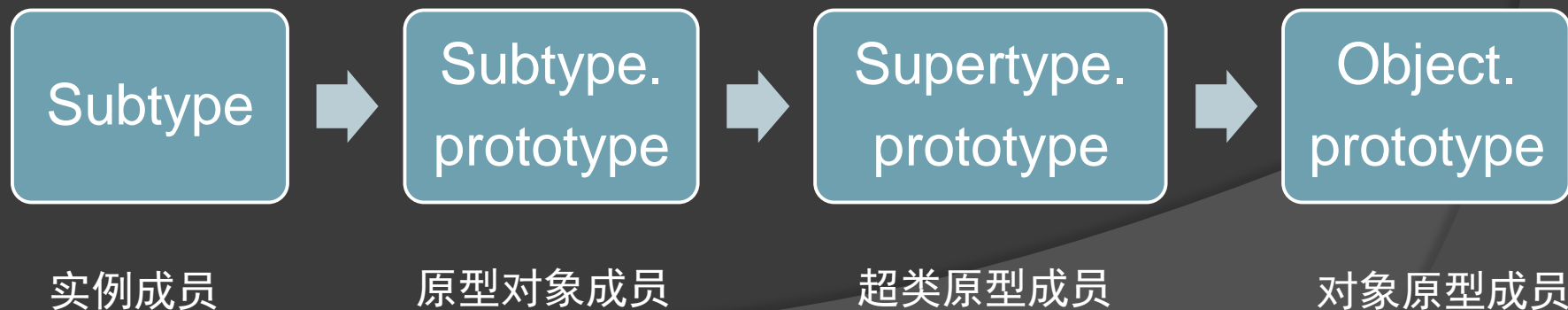
```
function SuperType () {}
```

```
function SubType () {}
```

```
SubType.prototype = new SuperType();
```

```
var subInstance = new SubType();
```

调用subInstance的属性或方法，查找过程如下：





# 原型链继承的问题

```
function SuperType () {  
  this.name = name;  
  this.arr = ['1','2','3'];  
}  
function SubType () {  
}  
SubType.prototype = new SuperType();  
  
var subInstance = new SubType();  
subInstance.arr.push('4'); // [1,2,3,4]  
var otherInstance = new SubType();  
otherInstance.arr; // [1,2,3,4]
```

- 子类型实例会把父类型实例成员当成它的原型成员
- 无法借用父类型的构造函数

# 改进原型链继承

```
function SuperType (name) {  
  this.name = name;  
  this.arr = ['1','2','3'];  
}  
function SubType (name) {  
  SuperType.call(this, name); //第二次调用SuperType  
}  
SubType.prototype = new SuperType(); //第一次调用SuperType  
SubType.prototype.constructor = SubType;  
  
var subInstance = new SubType();  
subInstance.arr.push('4'); // [1,2,3,4]  
var otherInstance = new SubType('myname');  
otherInstance.arr; //[1,2,3]  
otherInstance.name; //myname
```

借用了SuperType构造函数

这个arr是SubType的实例属性

```
function SuperType (name) {
  this.name = name;
  this.arr = ['1','2','3'];
}
SuperType.prototype.sayHi = function() {
  alert(this.name);
};
function SubType (name) {
  SuperType.call(this, name); //第二次调用SuperType
}
SubType.prototype = new SuperType(); //第一次调用SuperType
SubType.prototype.constructor = SubType;

var subInstance = new SubType();
subInstance.arr.push('4'); // [1,2,3,4]
var otherInstance = new SubType('myname');
otherInstance.arr; // [1,2,3]
otherInstance.name; // myname
otherInstance.sayHi(); // myname
```

只是为了继承SuperType的原型方法，为什么非得初始化一个SuperType实例？

导致SubType拿到了不需要的SuperType实例属性

```
var object = {};  
Object.create(object);
```

//隐藏继承的实现细节

```
function inherit(subType, superType) {  
  var prototype = Object.create(superType.prototype);  
  prototype.constructor = subType;  
  subType.prototype = prototype;  
}
```

```
function SuperType (name) {  
  this.name = name;  
}  
SuperType.prototype.f = function() {  
  console.log('func call');  
}  
function SubType (name, type) {  
  SuperType.call(this, name);  
  this.type = type;  
}
```

```
inherit(SubType, SuperType);  
var instance = new SubType('xx', 3);  
instance.type; //3  
instance.f(); //func call
```

常用的继承模式

//隐藏继承的实现细节

```
function inherit(subType, superType) {  
  var prototype = Object.create(superType.prototype);  
  prototype.super = function(){  
    superType.apply(prototype, arguments);  
  }  
  prototype.constructor = subType;  
  subType.prototype = prototype;  
}
```

super方法

```
function SuperType (name) {  
  this.name = name;  
}  
SuperType.prototype.f = function() {  
  console.log('func call');  
}  
function SubType (name, type) {  
  this.super(name);  
  this.type = type;  
}
```

super

的方式调用父类构造函数

```
inherit(SubType, SuperType);  
var instance = new SubType('xx', 3);  
alert(instance.name); //xx  
instance.f(); //func call
```

# 多继承

```
function SuperType1 (name) {  
  this.name = name;  
}  
SuperType1.prototype.f1 = function() {  
  alert(this.type);  
};  
function SuperType2 (age) {  
  this.age = age;  
}  
SuperType2.prototype.f2 = function() {  
  alert('f2');  
}  
function SubType (name, age, type) {  
  SuperType1.call(this, name);  
  SuperType2.call(this, age);  
  this.type = type;  
}
```

继承属性

```
extend(SubType.prototype, SuperType1.prototype);  
extend(SubType.prototype, SuperType2.prototype);  
var instance = new SubType('xx', 'mytype', 3);  
instance.type; //3  
instance.f1(); //3  
instance.f2(); //f2
```

继承方法



```
function extend(subType, superType) {  
    for(var prop in superType) {  
        subType[prop] = superType[prop];  
    }  
}
```

//定义类


```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  toString() {  
    return '('+this.x+', '+this.y+')';  
  }  
}
```

```
var point = new Point(2,3);  
point.toString() // (2, 3)
```

```
class ColorPoint extends Point {  
  constructor(x, y, color) {  
    super(x, y); // 等同于super.constructor(x, y)  
    this.color = color;  
  }  
  toString() {  
    return this.color+' '+super();  
  }  
}
```

ECMAScript6 --class

# 资源推荐

- ◎ 《DOM编程艺术2》
  - ◎ 《Object –Oriented JavaScript》
  - ◎ 《JavaScript高级程序设计3》
  - ◎ 《JavaScript权威指南6》
  - ◎ 《JavaScript语言精粹》
- 

MDN：系统性学习各种API，范例

联系我：

旺旺： @aoto111

微博： @aotoX

邮件： shengjie.yu@shenma-inc.com  
aoto\_yu@163.com

谢谢

QA