

# 潜力无限的JavaScript

## 三、面向对象的JavaScript

周志明 @ 开发二部  
2013年6月

# 潜力无限

- 越来越多的应用使用JavaScript来完成
  - ✓ Window 8后支持原生程序开发。
  - ✓ HTML5下的程序与游戏体验。
  - ✓ Node.js支持下的服务端应用。
- 钱力无限
  - ✓ 不受OS限制，原生态无污染的天然跨平台语言。
  - ✓ 会JavaScript的人很多，精JavaScript的人极少。
  - ✓ Web时代的银弹，随着Web流行而发展。

不是《零基础学会JavaScript》  
不是《21天精通JavaScript语言》  
不是《论前端开发人员的修养》

只讨论一些JavaScript令人又爱又恨的特性

# 目录

## 1. 浏览器渲染与JavaScript执行原理

- ✓ 浏览器渲染和执行引擎、阻塞、异步、定时器、事件流

## 2. JavaScript进阶基础知识

- ✓ 执行上下文、变量对象、this指针、作用域链

## 3. 面向对象的JavaScript

- ✓ 函数对象、构造器、原型、原型链、继承、闭包

## 4. 性能与JavaScript

- ✓ 性能陷阱、“看起来快些”的技巧、HTTP头优化、问题排查工具

ECMA-262-3 , 第七章 :

ECMAScript is an object-oriented  
programming language supporting  
delegating inheritance based on prototypes.

# 引言1：什么是面向对象？

什么是面向对象编程？

面向对象的程序有哪些基本特征？

# 引言1：什么是面向对象？

- 面向对象编程的三个基本特征：
  - ✓ 封装：数据与行为的统一。
  - ✓ 继承：子类能扩展细化父类行为。
  - ✓ 多态：对象行为能根据调用环境改变（重写和覆盖）。

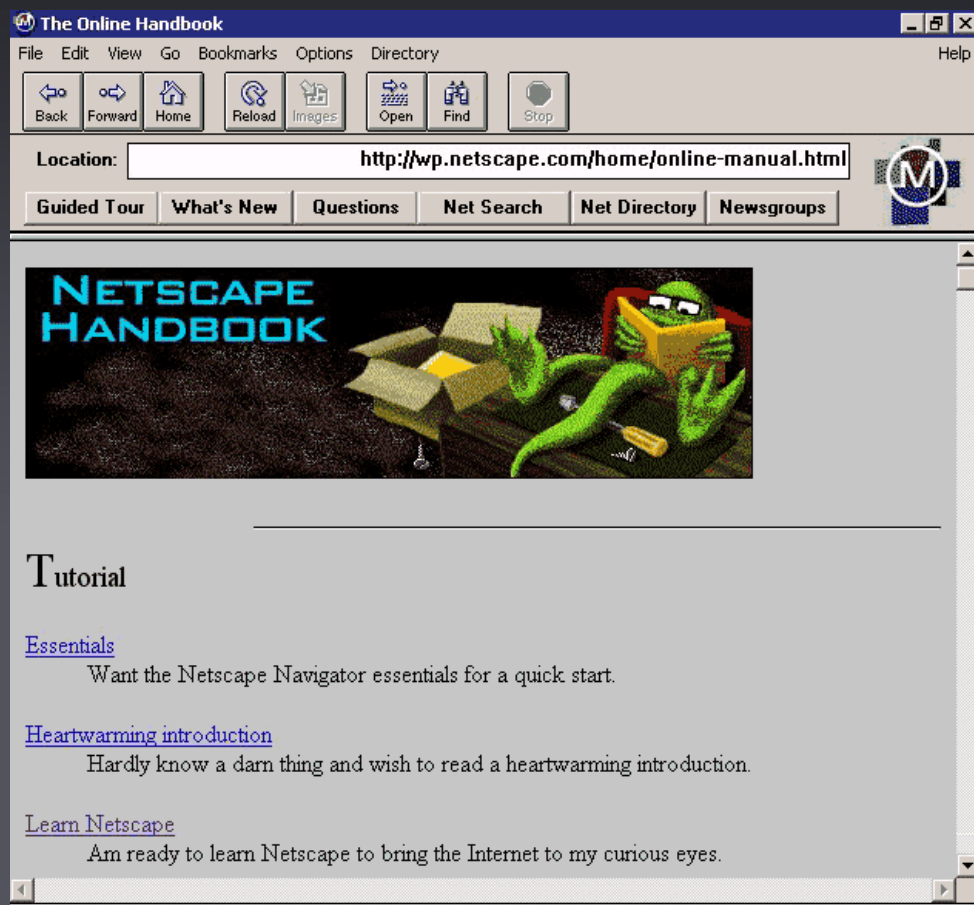


没回答出来的同学，  
大学计算机课是体育  
老师教的吧？



# 引言2：JavaScript与OOP的一些历史

- 1994年，网景公司（Netscape）发布了Navigator浏览器0.9版。但是，这个版本的浏览器只能用来浏览，不具备与访问者互动的能力。



## 引言2：JavaScript与OOP的一些历史

- 网景公司急需一种网页脚本语言，使得浏览器可以与网页互动。工程师Brendan Eich负责开发这种新语言。他觉得，**没必要设计得很复杂**，这种语言只要能够完成一些简单操作就够了，比如判断用户有没有填写表单。
- 另一方面，1994-1996年正是面向对象编程（Object-Oriented Programming）最兴盛的时期，C++是当时最流行的语言，Java也在1995年推出随后风靡全球。Brendan Eich觉得**有必要按照OOP的思路设计JavaScript**。



## 引言2：JavaScript与OOP的一些历史

- 那不是很复杂，又OOP的JavaScript是怎样的？
  - ✓ 所有对象数据类型都从Object派生出来。并且借鉴了Java的toString、valueOf等默认行为。
  - ✓ 函数调用仅通过名称进行静态分派，通过函数的arguments对象实现重载。
  - ✓ 没有给class关键字赋予含义，不能定义类。
  - ✓ 因为没有类，需要引入原型的概念来实现继承。
  - ✓ 因为没有类，“function”被赋予许多类的多重含义

# OOP via JS

- 编写基于JavaScript的面向对象程序需要考虑的问题：
  - ✓ 如何创建对象？
  - ✓ 如何把数据和行为封装在对象中？
  - ✓ 如何设定数据和方法的可访问性？
  - ✓ 如何继承、细化父类的行为？
  - ✓ 如何重写、重载父类的方法？
  - ✓ .....

# OOP via JS ( 尝试1 )

- 第一个 “面向对象” 的JavaScript程序

```
<script type="text/javascript">
  var person = new Object();
  person.name = 'zzm';
  person.age = 29;
  person.company = 'ygsoft';

  person.getName = function(){
    alert(this.name);
  };

  person.getName();
</script>
```

# OOP via JS ( 尝试1 )

- 第一个 “面向对象” 的JavaScript程序
  - ✓ 有什么缺陷？
    - 没有真正进行封装，创建多个对象，会产生大量的重复代码。
  - ✓ 怎么解决？
    - 通过工厂方法封装
    - 通过构造函数封装

# OOP via JS ( 尝试2 )

- 第二个 “面向对象” 的JavaScript程序

```
<script type="text/javascript">
  function Persion (name,age,company) {
    this.name = name;
    this.age = age;
    this.company = company;
    this.getName = function(){
      alert(this.name);
    }
  }

  var persion = new Persion('zzm',29,'ygsoft');
  persion.getName();
</script>
```

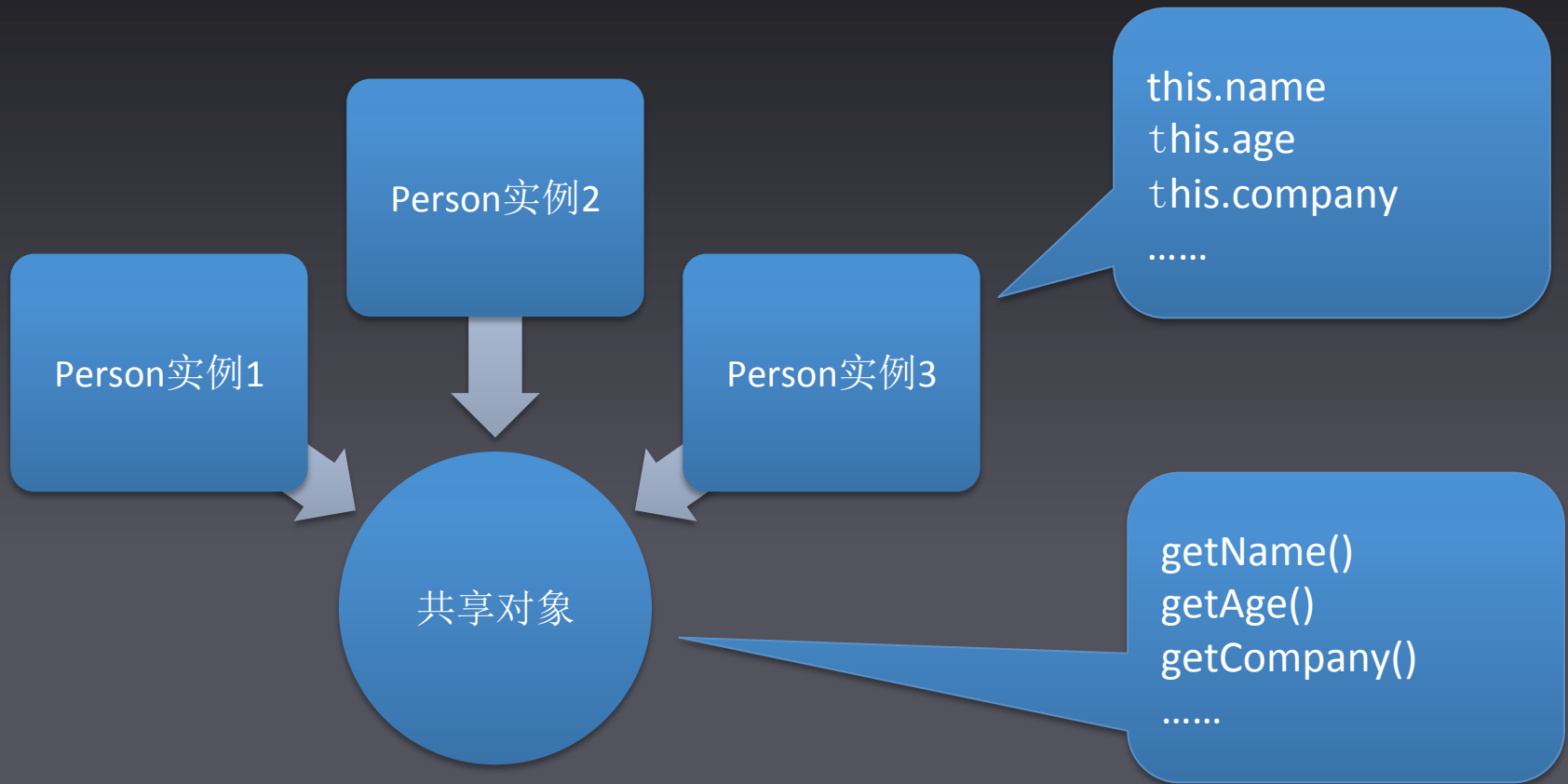
# OOP via JS ( 尝试2 )

- 第二个 “面向对象” 的JavaScript程序
  - ✓ 有什么缺陷？
    - 全部数据都是实例属性，没有类属性存在
    - 方法也是实例级的，没有哪门语言这样整的，这个问题很大呀
  - ✓ 怎么解决？
    - 让同一个类型的所有对象共享一个对象，把类属性和方法放在这个对象上。访问类属性和方法，就转为访问该对象。



# OOP via JS ( 尝试2 )

- 共享对象



# OOP via JS ( 尝试3 )

- 第三个 “面向对象” 的JavaScript程序

```
<script type="text/javascript">
  function Persion (name,age) {
    this.name = name;
    this.age = age;
  }
  Persion.prototype = {
    constructor : Persion,
    company : 'ygsoft',
    getName : function(){
      alert(this.name);
    }
  }

  var persion = new Persion('zzm',29);
  persion.getName();
</script>
```

# OOP via JS ( 尝试3 )

- 第三个 “面向对象” 的JavaScript程序
  - ✓ 换一种写法：

```
<script type="text/javascript">
  function Persion (name,age) {
    this.name = name;
    this.age = age;
    Persion.prototype.company = 'ygsoft';

    if(typeof this.getName != "function"){
      Persion.prototype.getName = function(){
        alert(this.name);
      }
    }
  }

  var persion = new Persion('zzm',29);
  persion.getName();
</script>
```

# 函数类型、原型和原型链

- 函数类型

- ✓ 定义：凡是typeof XXX返回结果为“function”的类型就是函数类型。

- 以下为典型的函数类型：Object、String、RegExp、parseInt、document.getElementById、new Function()

- 以下为典型的非函数类型：window、document、'zzm'、true、new function(){}

- ✓ 函数类型事实上承载了class的许多含义。

# 函数类型、原型和原型链

- 原型

- ✓ 任何一个函数类型，都包含了一个名为“prototype”的属性。这个属性称为该函数的原型。
- ✓ 在许多JS引擎实现中，函数生成的实例对象可以通过名为“\_\_proto\_\_”的属性访问到该函数的原型。
- ✓ 原型对象可以是任何一种基本类型（可以设置为123、true这样的值类型，但是没有意义）。如果没有被人为设置，原型默认为object类型的对象，并包含了一些默认属性。
- ✓ 原型对象的作用是：当查找属性或者方法时，如果对象上找不到指定属性或方法，将继续去原型对象上查找。

# 函数类型、原型和原型链

- 原型链

- ✓ 原型链是特指当某个函数对象的原型对象仍然存在原型的情况。JavaScript的对象体系中，就是通过原型链来完成继承的。
- ✓ 原型链查找：仍然符合原型查找的一般规则，当在一个对象上查找属性时，先在对象上查找，没有找到就去原型找，如果原型还有原型，则递归该规则。

# 函数类型、原型和原型链

- 原型链例子

```
<script type="text/javascript">
  function Foo() {
    this.value = 42;
  }
  Foo.prototype = {
    method: function() {}
  };
  function Bar() {}

  // 设置Bar的prototype属性为Foo的实例对象
  Bar.prototype = new Foo();
  Bar.prototype.foo = 'Hello World';

  var test = new Bar() // 创建Bar的一个新实例
</script>
```

# 函数类型、原型和原型链

- 原型链例子

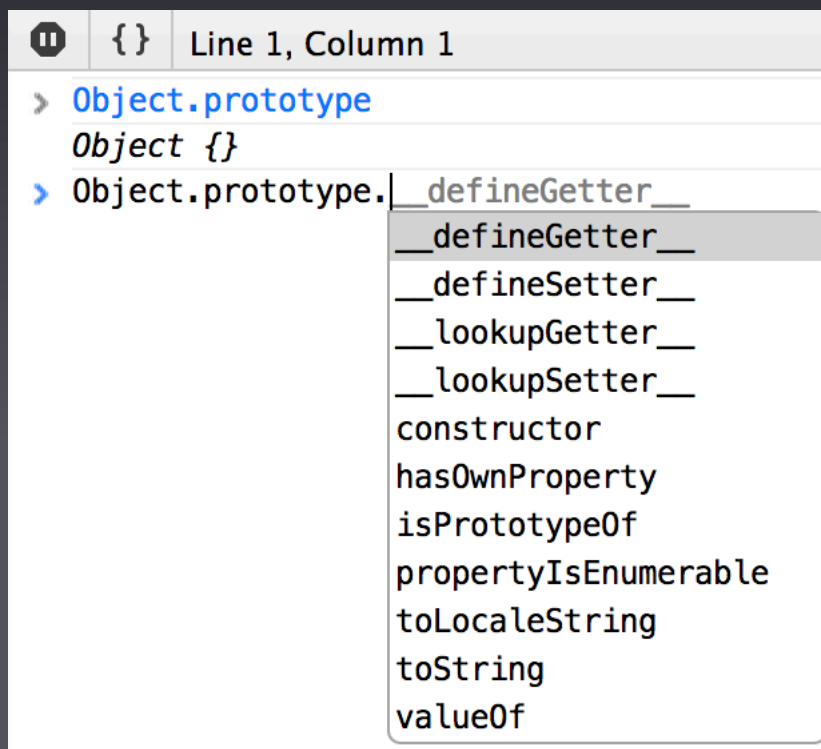
```
// 原型链
test [Bar的实例]
  Bar.prototype [Foo的实例]
    { foo: 'Hello World', value: 42 }
    Foo.prototype
      {method: ...};
      Object.prototype
        {toString: ... /* etc. */};
```

- ✓ test 对象从 Bar.prototype 和 Foo.prototype 继承下来；因此，它能访问 Foo 的原型方法 method。同时，它也能够访问那个定义在原型上的 Foo 实例属性 value。需要注意的是 new Bar() 不会创建出一个新的 Foo 实例，而是重复使用它原型上的那个实例；因此，所有的 Bar 实例都会共享相同的 value 属性。



# 与原型相关的方法

- 在Object.prototype中，默认有1个属性和6个方法，JS中所有对象都派生至Object，这意味着所有对象有这6个方法



The screenshot shows a JavaScript console interface. At the top, there is a status bar with a pause icon, a curly brace icon, and the text "Line 1, Column 1". Below this, the console displays the following:

```
> Object.prototype
Object {}
> Object.prototype.
  __defineGetter__
  __defineSetter__
  __lookupGetter__
  __lookupSetter__
  constructor
  hasOwnProperty
  isPrototypeOf
  propertyIsEnumerable
  toLocaleString
  toString
  valueOf
```

The methods listed in the dropdown menu are: `__defineGetter__`, `__defineSetter__`, `__lookupGetter__`, `__lookupSetter__`, `constructor`, `hasOwnProperty`, `isPrototypeOf`, `propertyIsEnumerable`, `toLocaleString`, `toString`, and `valueOf`.

# 与原型相关的方法

- `hasOwnProperty`：判断一个对象是否包含自定义属性而不是原型链上的属性，它是JavaScript中唯一一个处理属性但是不查找原型链的函数。

```
<script type="text/javascript">
  Object.prototype.bar = 1;
  var foo = {moo: 2};

  for(var i in foo) {
    console.log(i); // 输出两个属性：bar 和 moo
  }
  for(var i in foo) {
    if (foo.hasOwnProperty(i)) {
      console.log(i); // 输出一个属性：moo
    }
  }
</script>
```

# 与原型相关的方法

- `isPrototypeOf` : 返回一个布尔值，指出对象是否存在于另一个对象的原型链中。

```
<script type="text/javascript">
  function Foo() {
    this.value = 42;
  }
  Foo.prototype = {
    method: function() {}
  };
  function Bar() {}

  Bar.prototype = new Foo();
  Bar.prototype.foo = 'Hello World';
  var test = new Bar();
  alert(Bar.prototype.isPrototypeOf(test));
  alert(Foo.prototype.isPrototypeOf(test));
  alert(Object.prototype.isPrototypeOf(test));
</script>
```

# 思考

在JavaScript语言中有没有办法定义出  
一个constant属性？

# 再说回封装

- ECMA-262-5
  - ✓ Object.defineProperty()

```
<script type="text/javascript">
  var foo = new Object();
  Object.defineProperty(foo, "x", {
    value: 10,
    writable: false, // 是否可写
    enumerable: false, // 是否可被in操作符枚举
    configurable: false // 是否禁止删除
  });

  alert(foo.x); // 输出10
  foo.x = 20;
  alert(foo.x); // 输出10
  delete foo.x;
  alert(foo.x); // 输出10
</script>
```

# 再说回封装

- ECMA-262-5
  - ✓ Object.getOwnPropertyDescriptor()

```
<script type="text/javascript">  
  var descriptors = Object.getOwnPropertyDescriptor(foo, "x");  
  for(name in descriptors){  
    alert(name+":"+descriptors[name]);  
  }  
</script>
```

# OOP via JS ( 尝试4 )

- 通过原型链实现继承
  - ✓ 思考：有什么问题？

```
<script type="text/javascript">
    function Pet() {
        // 宠物共有的特性：会叫会生娃
        this.children = [];
        this.shout = function(){
            alert("hahahahah!!");
        }
    }

    function Dog(master){
        // 狗的特性：有主人
        this.master = master;
        this.getMaster = function(){
            alert(this.master);
        }
    }

    // 让狗继承于宠物
    Dog.prototype = new Pet();

    var luoluo = new Dog("wk");
    var suisui = new Dog("zzm");
    luoluo.getMaster() // wk
    suisui.getMaster() // zzm
    alert(luoluo instanceof Dog); // true
    alert(luoluo instanceof Pet); // true
</script>
```

# OOP via JS ( 尝试4 )

- 通过原型链实现继承
  - ✓ 思考：有什么问题？

```
<script type="text/javascript">
  var luoluo = new Dog("wk");
  var suisui = new Dog("zzm");

  luoluo.children.push("大白"、"小白"、"大黑"、"小黑");
  suisui.children.push("叉烧");

  alert(luoluo.children);
  alert(suisui.children);
</script>
```



# OOP via JS ( 尝试4 )

- 通过原型链实现继承

- ✓ 有什么缺陷？

- 父类的实例属性，通过prototype继承到子类之后，会被所有子类共享，变成“类属性”。
    - 父类构造函数没有办法传递参数。

- ✓ 怎样解决：

- 想办法在原型之外，以子类对象为“this”，独立执行父类的构造函数。

# OOP via JS ( 尝试5 )

- 通过借用构造器实现继承

```
<script type="text/javascript">
    function Pet() {
        // 宠物共有的特性：会叫会生娃
        this.children = [];
        this.shout = function(){
            alert("hahahahah!!");
        }
    }
    function Dog(master){
        // 让狗继承于宠物
        Pet.call(this);
        this.master = master;
        this.getMaster = function(){
            alert(this.master);
        }
    }

    var luoluo = new Dog("wk");
    var suisui = new Dog("zzm");
    luoluo.children.push("大白", "小白", "大黑", "小");
    suisui.children.push("叉烧");
    alert(luoluo.children);
    alert(suisui.children);
</script>
```

# OOP via JS ( 尝试5 )

- 通过借用构造器实现继承
  - ✓ 解决了对象共用的问题
  - ✓ 缺陷：
    - 放弃了原型，父类方法没有共享。
    - instanceof等操作符无法进行父类判定。

# OOP via JS ( 尝试6 )

- 揉合尝试4、5的方法，实现组合继承

```
<script type="text/javascript">
  function Pet() {
    this.children = [];
  }
  // 把原来父类的方法，挪到父类的原型上
  Pet.prototype.shout = function(){
    alert("hahahahah!!");
  }

  function Dog(master){
    Pet.call(this);
    this.master = master;
    this.getMaster = function(){
      alert(this.master);
    }
  }
  // Dog继承于Pet
  Dog.prototype = new Pet();
</script>
```

# OOP via JS ( 尝试6 )

- 组合继承是目前JavaScript最常用的继承模式之一
  - ✓ 优势：
    - 在原型链上定义方法保证函数复用
    - 在对象实例上定义属性保证属性独立。
    - 能够使用instanceof和isPrototypeOf()进行继承关系判定。
  - ✓ 缺陷？还有什么可抱怨的？

# OOP via JS ( 尝试6 )

- 组合继承的缺陷

```
<script type="text/javascript">
  function Pet() {
    this.children = [];
  }
  Pet.prototype.shout = function(){
    alert("hahahahah!!");
  }

  function Dog(master){
    Pet.call(this);    // 第一次调用父类构造函数
    this.master = master;
    this.getMaster = function(){
      alert(this.master);
    }
  }
  Dog.prototype = new Pet();    // 第二次调用父类构造函数
</script>
```

# OOP via JS ( 尝试7 )

- 目前开源框架中 “工业级” 的继承实现（原理）：
  - ✓ 组合继承中，子类原型赋值为父类实例，目的是得到一个使用父类原型的独立对象作为原型对象。以下代码也可以完成这项工作。

```
function inheritPrototype (subType,superType) {  
    // 得到以传入对象为原型的空对象  
    function createObject(prototypeObj){  
        function F(){};  
        F.prototype = prototypeObj;  
        return new F();  
    }  
  
    var prototype = createObject(superType.prototype);  
    prototype.constructor = subType;  
    subType.prototype = prototype;  
}
```

# OOP via JS ( 尝试7 )

- 寄生组合集成
  - ✓ YUI的YAHOO.lang.extend()开始广泛使用这种继承方式。

```
<script type="text/javascript">
    function Pet() {
        this.children = [];
    }
    Pet.prototype.shout = function(){
        alert("hahahahah!!");
    }

    function Dog(master){
        Pet.call(this);
        this.master = master;
        this.getMaster = function(){
            alert(this.master);
        }
    }
    inheritPrototype(Dog,Pet);    // 完成继承
</script>
```



# 原型链、构造器与OOP总结

- 原型链、构造器与OOP总结

- ✓ 实例级的属性，通过借用构造器的方式继承至子类。
- ✓ 方法和类级的属性，通过原型链继承至子类。
- ✓ 通过创建原型对象副本，保障子类对象的创建性能。

- 思考：

- ✓ 子类如何调用父类的方法（`super.xxx()`）？
- ✓ `private`、`protected`属性如何处理？
- ✓ ECP和jQuery是如何简化封装对象继承的？

## 谈谈另一种语言风格

ECMAScript除了是OOP语言，它还是  
函数式语言

# 函数式语言的特征

- 基本特征：函数即数据
  - ✓ 可以将函数赋值给变量
  - ✓ 可以将函数作为参数进行传递
  - ✓ 可以将函数作为返回值
  - ✓ 可以接受自身作为参数，接受自身作为返回值

# 函数式语言的特征

- 函数式语言与栈架构的冲突
  - ✓ 外部变量访问问题。

```
<script type="text/javascript">
  function testFn() {
    var localVar = 10;

    function innerFn(innerParam) {
      alert(innerParam + localVar);
    }
    return innerFn;
  }

  var someFn = testFn();
  var localVar = 20;
  someFn(20); // 输出30
</script>
```

# 闭包

- 闭包定义：闭包是代码块和创建该代码块的上下文中数据的结合。
- 常见的闭包形式：
  - ✓ JavaScript中的函数返回函数
  - ✓ Java中的内部类
  - ✓ Lambda表达式
  - ✓ .....

# 闭包

- 在ECMAScript中，**所有的函数都是闭包**，因为它们都是在创建的时候就保存了上层上下文的作用域链，不管这个函数后续是否会激活——[[Scope]]在函数创建的时候就有了。
- 开发人员经常错误将闭包简化理解成从父上下文中返回内部函数，甚至理解成只有匿名函数才能是闭包。
- 只有一种例外，那就是通过Function构造器创建的函数，因为其[[Scope]]只包含全局对象。

# 闭包

- 从实践角度出发，一般而言口语中的闭包是指满足以下两个条件的函数（否则函数和闭包两个名词就基本上是一个东西了）：
  - ✓ 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）
  - ✓ 在代码中引用了自由变量

# 闭包

- 许多优雅的应用

```
<script type="text/javascript">
    // 同样的例子还有，数组的map方法是根据函数中定义的条件将原数组映射到一个新的数组中：
    [1, 2, 3].map(function (element) {
        return element * 2;
    }); // [2, 4, 6]

    // 使用函数式参数，可以很方便的实现一个搜索方法，并且可以支持无限制的搜索条件：
    someCollection.find(function (element) {
        return element.someProperty == 'searchCondition';
    });

    // 还有应用函数，比如常见的forEach方法，将函数应用到每个数组元素：
    [1, 2, 3].forEach(function (element) {
        if (element % 2 != 0) {
            alert(element);
        }
    }); // 1, 3
</script>
```



# 闭包

- 同一个父上下文中创建的闭包的[[Scope]]属性是相同的。也就是说，某个闭包对其中[[Scope]]的变量做修改会影响到其他闭包对其变量的读取。

```
<script type="text/javascript">
  var data = [];
  for (var k = 0; k < 3; k++) {
    data[k] = function () {
      alert(k);
    };
  }

  data[0]();
  data[1]();
  data[2]();
</script>
```

# 闭包

- 同一个父上下文中创建的闭包的[[Scope]]属性是相同的。也就是说，某个闭包对其中[[Scope]]的变量做修改会影响到其他闭包对其变量的读取。

```
<script type="text/javascript">
  var data = [];
  for (var k = 0; k < 3; k++) {
    data[k] = function () {
      alert(k);
    };
  }

  data[0](); // 3, 而不是0
  data[1](); // 3, 而不是1
  data[2](); // 3, 而不是2
</script>
```

# 闭包

- 解决方案

```
<script type="text/javascript">
  var data = [];
  for (var k = 0; k < 3; k++) {
    data[k] = (function _helper(x) {
      return function () {
        alert(x);
      };
    })(k); // 传入"k"值
  }

  // 现在结果是正确的了
  data[0](); // 0
  data[1](); // 1
  data[2](); // 2
</script>
```

为什么？

# 复习：作用域链

- 作用域链的特征

- ✓ 是执行上下文的一个属性

```
activeExecutionContext = {  
  VO: {...},  
  this: thisValue,  
  Scope: [ // Scope chain  
           // 所有变量对象的列表  
        ]  
};
```

- ✓ 逻辑上是一个数组，每个元素都是变量对象

- ✓ 定义为：

Scope = ActiveContext.VO + Function.[[Scope]]

- ✓ 注意，[[Scope]]是函数的属性。

# 闭包

- 另一个解决方案

```
<script type="text/javascript">
  var data = [];
  for (var k = 0; k < 3; k++) {
    (data[k] = function () {
      alert(arguments.callee.x);
    }).x = k; // 将k作为函数的一个属性
  }

  // 结果也是对的
  data[0](); // 0
  data[1](); // 1
  data[2](); // 2
</script>
```

- 自己想想为什么？

# 闭包

- 闭包返回问题

✓ 从闭包函数中返回找到的第一个偶数

```
function getElement() {  
    [1, 2, 3].forEach(function (element) {  
        if (element % 2 == 0) {  
            // 返回给函数"forEach"函数  
            // 而不是返回给getElement函数  
            alert('found: ' + element); // found: 2  
            return element;  
        }  
    });  
    return null;  
}
```

# 闭包

- 闭包返回问题

✓ 可以throw实现，但不推荐这样做

```
<script type="text/javascript">
  function getElement() {
    try {
      [1, 2, 3].forEach(function (element) {
        if (element % 2 == 0) {
          // 从getElement中"返回"
          alert('found: ' + element); // found: 2
          throw element;
        }
      });
    } catch (e) {
      return e;
    }
    return null;
  }
  alert(getElement()); // 2
</script>
```



# References

- 《ECMAScript Language Specification》
  - ✓ <http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf>
- 《深入学习Javascript》
  - ✓ <http://blog.goddyzhao.me/JavaScript-Internal>
- 《深入了解JavaScript系列》
  - ✓ <http://www.cnblogs.com/TomXu/archive/2011/12/15/2288411.html>
- 《Inside-the-Browser》