[**Javascript闭包(closure)** 1](#_Toc302552543)

[**一、** **什么是闭包？** 1](#_Toc302552544)

[**二、** **闭包有什么作用？** 2](#_Toc302552545)

[**三、** **闭包内的微观世界** 2](#_Toc302552546)

[**四、** **闭包的应用场景** 2](#_Toc302552547)

[**五、** **Javascript的垃圾回收机制** 3](#_Toc302552548)

[**Javascript继承机制** 4](#_Toc302552549)

[**一、** **设计思想** 4](#_Toc302552550)

[**二、** **Javascript面向对象编程：封装** 6](#_Toc302552551)

[**三、** **Javascript面向对象编程：构造函数的继承** 10](#_Toc302552552)

[**四、** **Javascript面向对象编程：非构造函数的继承** 14](#_Toc302552553)

[**Javascript的设计缺陷** 17](#_Toc302552554)

[**一、** **为什么Javascript有设计缺陷？** 17](#_Toc302552555)

[**二、** **Javascript的10个设计缺陷** 17](#_Toc302552556)

[**三、** **如何看待Javascript的设计缺陷？** 20](#_Toc302552557)

[**Javascript的this用法** 22](#_Toc302552558)

[**一、** **纯粹的函数调用** 22](#_Toc302552559)

[**二、** **作为对象方法的调用** 22](#_Toc302552560)

[**三、** **作为构造函数调用** 22](#_Toc302552561)

[**四、** **apply调用** 23](#_Toc302552562)

[**Javascript与多线程** 24](#_Toc302552563)

[**问题** 24](#_Toc302552564)

[**解答** 24](#_Toc302552565)

[**相关资料** 24](#_Toc302552566)

**Javascript闭包(closure)**

最近在网上查阅了不少javascript闭包(closure)相关的资料，写的大多是非常的学术和专业。对于初学者来说别说理解闭包了，就连文字叙述都很难看懂。撰写此文的目的就是用最通俗的文字揭开Javascript闭包的真实面目。

1. **什么是闭包？**

官方的解释是：所谓闭包，指的是一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。

相信很少有人能直接看懂这句话，因为他描述的太学术。我想用如何在Javascript中创建一个闭包来告诉你什么是闭包，因为跳过闭包的创建过程直接理解闭包的定义是非常困难的。看下面这段代码：

function a(){

var i=0;

function b(){

　　alert(++i);

}

return b;

}

var c = a();

c();

这段代码有两个特点：

1、函数b嵌套在函数a内部；

2、函数a返回函数b。

这样在执行完var c=a()后，变量c实际上是指向了函数b，再执行c()后就会弹出一个窗口显示i的值(第一次为1)。这段代码其实就创建了一个闭包，为什么？因为函数a外的变量c引用了函数a内的函数b，就是说：当函数a的内部函数b被函数a外的一个变量引用的时候，就创建了一个闭包。

我猜想你一定还是不理解闭包，因为你不知道闭包有什么作用，下面让我们继续探索。

1. **闭包有什么作用？**

简而言之，闭包的作用就是在a执行完并返回后，闭包使得Javascript的垃圾回收机制GC不会收回a所占用的资源，因为a的内部函数b的执行需要依赖a中的变量。这是对闭包作用的非常直白的描述，不专业也不严谨，但大概意思就是这样，理解闭包需要循序渐进的过程。

在上面的例子中，由于闭包的存在使得函数a返回后，a中的i始终存在，这样每次执行c()，i都是自加1后alert出i的值。

那么我们来想象另一种情况，如果a返回的不是函数b，情况就完全不同了。因为a执行完后，b没有被返回给a的外界，只是被a所引用，而此时a也只会被b引用，因此函数a和b互相引用但又不被外界打扰(被外界引用)，函数a和b就会被GC回收。(关于Javascript的垃圾回收机制将在后面详细介绍)

1. **闭包内的微观世界**

如果要更加深入的了解闭包以及函数a和嵌套函数b的关系，我们需要引入另外几个概念：函数的执行环境(excution context)、活动对象(call object)、作用域(scope)、作用域链(scope chain)。以函数a从定义到执行的过程为例阐述这几个概念。

1、当定义函数a的时候，js解释器会将函数a的作用域链(scope chain)设置为定义a时a所在的环境，如果a是一个全局函数，则scope chain中只有window对象。

2、当函数a执行的时候，a会进入相应的执行环境(excution context)。

3、在创建执行环境的过程中，首先会为a添加一个scope属性，即a的作用域，其值就为第1步中的scope chain。即a.scope=a的作用域链。

4、然后执行环境会创建一个活动对象(call object)。活动对象也是一个拥有属性的对象，但它不具有原型而且不能通过JavaScript代码直接访问。创建完活动对象后，把活动对象添加到a的作用域链的最顶端。此时a的作用域链包含了两个对象：a的活动对象和window对象。

5、下一步是在活动对象上添加一个arguments属性，它保存着调用函数a时所传递的参数。

6、最后把所有函数a的形参和内部的函数b的引用也添加到a的活动对象上。在这一步中，完成了函数b的的定义，因此如同第3步，函数b的作用域链被设置为b所被定义的环境，即a的作用域。

到此，整个函数a从定义到执行的步骤就完成了。此时a返回函数b的引用给c，又函数b的作用域链包含了对函数a的活动对象的引用，也就是说b可以访问到a中定义的所有变量和函数。函数b被c引用，函数b又依赖函数a，因此函数a在返回后不会被GC回收。

当函数b执行的时候亦会像以上步骤一样。因此，执行时b的作用域链包含了3个对象：b的活动对象、a的活动对象和window对象。

当在函数b中访问一个变量的时候，搜索顺序是先搜索自身的活动对象，如果存在则返回，如果不存在将继续搜索函数a的活动对象，依次查找，直到找到为止。如果整个作用域链上都无法找到，则返回undefined。如果函数b存在Prototype原型对象，则在查找完自身的活动对象后先查找自身的原型对象，再继续查找。这就是Javascript中的变量查找机制。

1. **闭包的应用场景**

1、保护函数内的变量安全。以最开始的例子为例，函数a中i只有函数b才能访问，而无法通过其他途径访问到，因此保护了i的安全性。

2、在内存中维持一个变量。依然如前例，由于闭包，函数a中i的一直存在于内存中，因此每次执行c()，都会给i自加1。以上两点是闭包最基本的应用场景，很多经典案例都源于此。

1. **Javascript的垃圾回收机制**

在Javascript中，如果一个对象不再被引用，那么这个对象就会被GC回收。如果两个对象互相引用，而不再被第3者所引用，那么这两个互相引用的对象也会被回收。因为函数a被b引用，b又被a外的c引用，这就是为什么函数a执行后不会被回收的原因。

**Javascript继承机制**

我一直很难理解Javascript语言的继承机制。它没有"子类"和"父类"的概念，也没有"类"（class）和"实例"（instance）的区分，全靠一种很奇特的"原型链"（prototype chain）模式，来实现继承。

我花了很多时间，学习这个部分，还做了很多笔记。但是都属于强行记忆，无法从根本上理解。直到昨天，我读到法国程序员[Vjeux](http://blog.vjeux.com/2011/javascript/how-prototypal-inheritance-really-works.html" \t "_blank)的解释，才恍然大悟，完全明白了Javascript为什么这样设计。下面，我尝试用自己的语言，来解释它的设计思想。彻底说明白prototype对象到底是怎么回事。其实根本就没那么复杂，真相非常简单。

1. **设计思想**
2. **缘起**

要理解Javascript的设计思想，必须从它的诞生说起。

1994年，网景公司（Netscape）发布了Navigator浏览器0.9版。这是历史上第一个比较成熟的网络浏览器，轰动一时。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。比如，如果网页上有一栏"用户名"要求填写，浏览器就无法判断访问者是否真的填写了，只有让服务器端判断。如果没有填写，服务器端就返回错误，要求用户重新填写，这太浪费时间和服务器资源了。

因此，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](http://brendaneich.com/" \t "_blank)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。

1994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，而Java语言的1.0版即将于第二年推出，Sun公司正在大肆造势。

Brendan Eich无疑受到了影响，Javascript里面所有的数据类型都是对象（object），这一点与Java非常相似。但是，他随即就遇到了一个难题，到底要不要设计"继承"机制呢？

1. **Brendan Eich的选择**

如果真的是一种简易的脚本语言，其实不需要有"继承"机制。但是，Javascript里面都是对象，必须有一种机制，将所有对象联系起来。所以，Brendan Eich最后还是设计了"继承"。

但是，他不打算引入"类"（class）的概念，因为一旦有了"类"，Javascript就是一种完整的面向对象编程语言了，这好像有点太正式了，而且增加了初学者的入门难度。

他考虑到，C++和Java语言都使用new命令，生成实例。

C++的写法是：

ClassName \*object = new ClassName(param);

Java的写法是：

Foo foo = new Foo();

因此，他就把new命令引入了Javascript，用来从原型对象生成一个实例对象。但是，Javascript没有"类"，怎么来表示原型对象呢？

这时，他想到C++和Java使用new命令时，都会调用"类"的构造函数（constructor）。他就做了一个简化的设计，在Javascript语言中，new命令后面跟的不是类，而是构造函数。

举例来说，现在有一个叫做DOG的构造函数，表示狗对象的原型。

function DOG(name){

　　　　this.name = name;

　　}

对这个构造函数使用new，就会生成一个狗对象的实例。

**var dogA = new DOG('大毛');**

alert(dogA.name); // 大毛

注意构造函数中的[this关键字](http://www.ruanyifeng.com/blog/2010/04/using_this_keyword_in_javascript.html" \t "_blank)，它就代表了新创建的实例对象。

1. **new运算符的缺点**

　　用构造函数生成实例对象，有一个缺点，那就是无法共享属性和方法。

　　比如，在DOG对象的构造函数中，设置一个实例对象的共有属性species。

function DOG(name){

　　this.name = name;

**this.species = '犬科';**

}

　　然后，生成两个实例对象：

var dogA = new DOG('大毛');

var dogB = new DOG('二毛');

　　这两个对象的species属性是独立的，修改其中一个，不会影响到另一个。

**dogA.species = '猫科';**

alert(dogB.species); // 显示"犬科"，不受dogA的影响

　　每一个实例对象，都有自己的属性和方法的副本。这不仅无法做到数据共享，也是极大的资源浪费。

1. **prototype属性的引入**

　　考虑到这一点，Brendan Eich决定为构造函数设置一个prototype属性。

　　这个属性包含一个对象（以下简称"prototype对象"），所有实例对象需要共享的属性和方法，都放在这个对象里面；那些不需要共享的属性和方法，就放在构造函数里面。

　　实例对象一旦创建，将自动引用prototype对象的属性和方法。也就是说，实例对象的属性和方法，分成两种，一种是本地的，另一种是引用的。

　　还是以DOG构造函数为例，现在用prototype属性进行改写：

function DOG(name){

　　this.name = name;

}

**DOG.prototype = { species : '犬科' };**

var dogA = new DOG('大毛');

var dogB = new DOG('二毛');

alert(dogA.species); // 犬科

alert(dogB.species); // 犬科

　　现在，species属性放在prototype对象里，是两个实例对象共享的。只要修改了prototype对象，就会同时影响到两个实例对象。

**DOG.prototype.species = '猫科';**

alert(dogA.species); // 猫科

alert(dogB.species); // 猫科

　　由于所有的实例对象共享同一个prototype对象，那么从外界看起来，prototype对象就好像是实例对象的原型，而实例对象则好像"继承"了prototype对象一样。

1. **Javascript面向对象编程：封装**

学习Javascript，最难的地方是什么？我觉得，Object（对象）最难。因为Javascript的Object模型很独特，和其他语言都不一样，初学者不容易掌握。

下面就是我的学习笔记，希望对大家学习这个部分有所帮助。我主要参考了以下两本书籍：[《面向对象的Javascript》](https://www.packtpub.com/object-oriented-javascript-applications-libraries/book" \t "_blank)（Object-Oriented JavaScript）[《Javascript高级程序设计（第二版）》](http://www.wrox.com/WileyCDA/WroxTitle/Professional-JavaScript-for-Web-Developers-2nd-Edition.productCd-047022780X.html)（Professional JavaScript for Web Developers, 2nd Edition)它们都是非常优秀的Javascript读物，推荐阅读。

Javascript是一种基于对象（object-based）的语言，你遇到的所有东西几乎都是对象。但是，它又不是一种真正的面向对象编程（OOP）语言，因为它的语法中没有class（类）。

那么，如果我们要把"属性"（property）和"方法"（method），封装成一个对象，甚至要从原型对象生成一个实例对象，我们应该怎么做呢？

**1. 生成对象的原始模式**

假定我们把猫看成一个对象，它有"名字"和"颜色"两个属性。

　　var Cat = {

　　　　name : '',

　　　　color : ''

　　}

现在，我们需要根据这个原型对象，生成两个实例对象。

　　var cat1 = {}; // 创建一个空对象

　　　　cat1.name = "大毛"; // 按照原型对象的属性赋值

　　　　cat1.color = "黄色";

　　var cat2 = {};

　　　　cat2.name = "二毛";

　　　　cat2.color = "黑色";

好了，这就是最简单的封装了。但是，这样的写法有两个缺点，一是如果多生成几个实例，写起来就非常麻烦；二是实例与原型之间，没有任何办法，可以看出有什么联系。

**2. 原始模式的改进**

我们可以写一个函数，解决代码重复的问题。

　　function Cat(name,color){

　　　　return {

　　　　　　name:name,

　　　　　　color:color

　　　　}

　　}

然后生成实例对象，就等于是在调用函数：

　　var cat1 = Cat("大毛","黄色");

　　var cat2 = Cat("二毛","黑色");

这种方法的问题依然是，cat1和cat2之间没有内在的联系，不能反映出它们是同一个原型对象的实例。

**3. 构造函数模式**

为了解决从原型对象生成实例的问题，Javascript提供了一个构造函数（Constructor）模式。

所谓"构造函数"，其实就是一个普通函数，但是内部使用了[this变量](http://www.ruanyifeng.com/blog/2010/04/using_this_keyword_in_javascript.html" \t "_blank)。对构造函数使用new运算符，就能生成实例，并且this变量会绑定在实例对象上。

比如，猫的原型对象现在可以这样写，

　　function Cat(name,color){

　　　　this.name=name;

　　　　this.color=color;

　　}

我们现在就可以生成实例对象了。

　　var cat1 = new Cat("大毛","黄色");

　　var cat2 = new Cat("二毛","黑色");

　　alert(cat1.name); // 大毛

　　alert(cat1.color); // 黄色

这时cat1和cat2会自动含有一个constructor属性，指向它们的构造函数。

　　alert(cat1.constructor == Cat); //true

　　alert(cat2.constructor == Cat); //true

Javascript还提供了一个instanceof运算符，验证原型对象与实例对象之间的关系。

　　alert(cat1 instanceof Cat); //true

　　alert(cat2 instanceof Cat); //true

**4. 构造函数模式的问题**

构造函数方法很好用，但是存在一个浪费内存的问题。

请看，我们现在为Cat对象添加一个不变的属性"type"（种类），再添加一个方法eat（吃老鼠）。那么，原型对象Cat就变成了下面这样：

　　function Cat(name,color){

　　　　this.name = name;

　　　　this.color = color;

　　　　this.type = "猫科动物";

　　　　this.eat = function(){alert("吃老鼠");};

　　}

还是采用同样的方法，生成实例：

　　var cat1 = new Cat("大毛","黄色");

　　var cat2 = new Cat ("二毛","黑色");

　　alert(cat1.type); // 猫科动物

　　cat1.eat(); // 吃老鼠

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属性和eat()方法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。这样既不环保，也缺乏效率。

　　alert(cat1.eat == cat2.eat); //false

能不能让type属性和eat()方法在内存中只生成一次，然后所有实例都指向那个内存地址呢？回答是可以的。

**5. Prototype模式**

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。

这意味着，我们可以把那些不变的属性和方法，直接定义在prototype对象上。

　　function Cat(name,color){

　　　　this.name = name;

　　　　this.color = color;

　　}

　　Cat.prototype.type = "猫科动物";

　　Cat.prototype.eat = function(){alert("吃老鼠")};

然后，生成实例。

　　var cat1 = new Cat("大毛","黄色");

　　var cat2 = new Cat("二毛","黑色");

　　alert(cat1.type); // 猫科动物

　　cat1.eat(); // 吃老鼠

这时所有实例的type属性和eat()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。

　　alert(cat1.eat == cat2.eat); //true

**6. Prototype模式的验证方法**

**6.1 isPrototypeOf()**

这个方法用来判断，某个proptotype对象和某个实例之间的关系。

　　alert(Cat.prototype.isPrototypeOf(cat1)); //true

　　alert(Cat.prototype.isPrototypeOf(cat2)); //true

**6.2 hasOwnProperty()**

每个实例对象都有一个hasOwnProperty()方法，用来判断某一个属性到底是本地属性，还是继承自prototype对象的属性。

　　alert(cat1.hasOwnProperty("name")); // true

　　alert(cat1.hasOwnProperty("type")); // false

**6.3 in运算符**

in运算符可以用来判断，某个实例是否含有某个属性，不管是不是本地属性。

　　alert("name" in cat1); // true

　　alert("type" in cat1); // true

in运算符还可以用来遍历某个对象的所有属性。

　　for(var prop in cat1) { alert("cat1["+prop+"]="+cat1[prop]); }

1. **Javascript面向对象编程：构造函数的继承**

上部分主要介绍了如何"封装"数据和方法，以及如何从原型对象生成实例。今天要介绍的是，如何生成一个"继承"多个对象的实例。

比如，现在有一个"动物"对象的构造函数，

　　function Animal(){

　　　　this.species = "动物";

　　}

还有一个"猫"对象的构造函数，

　　function Cat(name,color){

　　　　this.name = name;

　　　　this.color = color;

　　}

怎样才能使"猫"继承"动物"呢？

**1. 构造函数绑定**

最简单的方法，大概就是使用call或apply方法，将父对象的构造函数绑定在子对象上，也就是在子对象构造函数中加一行：

　　function Cat(name,color){

**Animal.apply(this, arguments);**

　　　　this.name = name;

　　　　this.color = color;

　　}

　　var cat1 = new Cat("大毛","黄色");

　　alert(cat1.species); // 动物

**2. prototype模式**

更常见的做法，则是使用prototype属性。

如果"猫"的prototype对象，指向一个Animal的实例，那么所有"猫"的实例，就能继承Animal了。

　　Cat.prototype = new Animal();

　　Cat.prototype.constructor = Cat;

　　var cat1 = new Cat("大毛","黄色");

　　alert(cat1.species); // 动物

代码的第一行，我们将Cat的prototype对象指向一个Animal的实例。

　　Cat.prototype = new Animal();

它相当于完全删除了prototype 对象原先的值，然后赋予一个新值。但是，第二行又是什么意思呢？

　　Cat.prototype.constructor = Cat;

原来，任何一个prototype对象都有一个constructor属性，指向它的构造函数。也就是说，Cat.prototype 这个对象的constructor属性，是指向Cat的。

我们在前一步已经删除了这个prototype对象原来的值，所以新的prototype对象没有constructor属性，所以我们必须手动加上去，否则后面的"继承链"会出问题。这就是第二行的意思。

总之，这是很重要的一点，编程时务必要遵守。下文都遵循这一点，即如果替换了prototype对象，

　　o.prototype = {};

那么，下一步必然是为新的prototype对象加上constructor属性，并将这个属性指回原来的构造函数。

　　o.prototype.constructor = o;

**3. 直接继承prototype**

由于Animal对象中，不变的属性都可以直接写入Animal.prototype。所以，我们也可以让Cat()跳过 Animal()，直接继承Animal.prototype。

现在，我们先将Animal对象改写：

　　function Animal(){ }

　　Animal.prototype.species = "动物";

然后，将Cat的prototype对象，然后指向Animal的prototype对象，这样就完成了继承。

　　Cat.prototype = Animal.prototype;

　　Cat.prototype.constructor = Cat;

　　var cat1 = new Cat("大毛","黄色");

　　alert(cat1.species); // 动物

与前一种方法相比，这样做的优点是效率比较高（不用执行和建立Animal的实例了），比较省内存。缺点是 Cat.prototype和Animal.prototype现在指向了同一个对象，那么任何对Cat.prototype的修改，都会反映到 Animal.prototype。

所以，上面这一段代码其实是有问题的。请看第二行

　　Cat.prototype.constructor = Cat;

这一句实际上把Animal.prototype对象的constructor属性也改掉了！

　　alert(Animal.prototype.constructor); // Cat

**4. 利用空对象作为中介**

由于"直接继承prototype"存在上述的缺点，所以可以利用一个空对象作为中介。

　　var F = function(){};

　　F.prototype = Animal.prototype;

　　Cat.prototype = new F();

　　Cat.prototype.constructor = Cat;

F是空对象，所以几乎不占内存。这时，修改Cat的prototype对象，就不会影响到Animal的prototype对象。

　　alert(Animal.prototype.constructor); // Animal

**5. prototype模式的封装函数**

我们将上面的方法，封装成一个函数，便于使用。

　　function extend(Child, Parent) {  
  
　　　　var F = function(){};

　　　　F.prototype = Parent.prototype;

　　　　Child.prototype = new F();

　　　　Child.prototype.constructor = Child;

　　　　Child.uber = Parent.prototype;

　　}

使用的时候，方法如下

　　extend(Cat,Animal);

　　var cat1 = new Cat("大毛","黄色");

　　alert(cat1.species); // 动物

这个extend函数，就是YUI库如何实现继承的方法。

另外，说明一点。函数体最后一行

　　Child.uber = Parent.prototype;

意思是为子对象设一个uber属性，这个属性直接指向父对象的prototype属性。这等于是在子对象上打开一条通道，可以直接调用父对象的方法。这一行放在这里，只是为了实现继承的完备性，纯属备用性质。

**6. 拷贝继承**

上面是采用prototype对象，实现继承。我们也可以换一种思路，纯粹采用"拷贝"方法实现继承。简单说，如果把父对象的所有属性和方法，拷贝进子对象，不也能够实现继承吗？

首先，还是把Animal的所有不变属性，都放到它的prototype对象上。

　　function Animal(){}

　　Animal.prototype.species = "动物";

然后，再写一个函数，实现属性拷贝的目的。

　　function extend2(Child, Parent) {

　　　　var p = Parent.prototype;

　　　　var c = Child.prototype;

　　　　for (var i in p) {

　　　　　　c[i] = p[i];

　　　　　　}

　　　　c.uber = p;

　　}

这个函数的作用，就是将父对象的prototype对象中的属性，一一拷贝给Child对象的prototype对象。

使用的时候，这样写：

　　extend2(Cat, Animal);

　　var cat1 = new Cat("大毛","黄色");

　　alert(cat1.species); // 动物

1. **Javascript面向对象编程：非构造函数的继承**

**1、什么是"非构造函数"的继承？**

比如，现在有一个对象，叫做"中国人"。

　　var Chinese = {  
　　　　nation:'中国'  
　　};

还有一个对象，叫做"医生"。

　　var Doctor ={  
　　　　career:'医生'  
　　}

请问怎样才能让"医生"去继承"中国人"，也就是说，我怎样才能生成一个"中国医生"的对象？

这里要注意，这两个对象都是普通对象，不是构造函数，无法使用构造函数方法实现"继承"。

**2、object()方法**

json格式的发明人Douglas Crockford，提出了一个object()函数，可以做到这一点。

　　function object(o) {

　　　　function F() {}

　　　　F.prototype = o;

　　　　return new F();

　　}

这个object()函数，其实只做一件事，就是把子对象的prototype属性，指向父对象，从而使得子对象与父对象连在一起。

使用的时候，第一步先在父对象的基础上，生成子对象：

　　var Doctor = object(Chinese);

然后，再加上子对象本身的属性：

　　Doctor.career = '医生';

这时，子对象已经继承了父对象的属性了。

　　alert(Doctor.nation); //中国

**3、浅拷贝**

除了使用"prototype链"以外，还有另一种思路：把父对象的属性，全部拷贝给子对象，也能实现继承。

下面这个函数，就是在做拷贝：

　　function extendCopy(p) {

　　　　var c = {};

　　　　for (var i in p) {

　　　　　　c[i] = p[i];

　　　　}

　　　　c.uber = p;

　　　　return c;

　　}

使用的时候，这样写：

　　var Doctor = extendCopy(Chinese);

　　Doctor.career = '医生';

　　alert(Doctor.nation); // 中国

但是，这样的拷贝有一个问题。那就是，如果父对象的属性等于数组或另一个对象，那么实际上，子对象获得的只是一个内存地址，而不是真正拷贝，因此存在父对象被篡改的可能。

请看，现在给Chinese添加一个"出生地"属性，它的值是一个数组。

　　Chinese.birthPlaces = ['北京','上海','香港'];

通过extendCopy()函数，Doctor继承了Chinese。

　　var Doctor = extendCopy(Chinese);

然后，我们为Doctor的"出生地"添加一个城市：

　　Doctor.birthPlaces.push('厦门');

发生了什么事？Chinese的"出生地"也被改掉了！

　　alert(Doctor.birthPlaces); //北京, 上海, 香港, 厦门

　　alert(Chinese.birthPlaces); //北京, 上海, 香港, 厦门

所以，extendCopy()只是拷贝基本类型的数据，我们把这种拷贝叫做"浅拷贝"。这是早期jQuery实现继承的方式。

**4、深拷贝**

所谓"深拷贝"，就是能够实现真正意义上的数组和对象的拷贝。它的实现并不难，只要递归调用"浅拷贝"就行了。

　　function deepCopy(p, c) {

　　　　var c = c || {};

　　　　for (var i in p) {

　　　　　　if (typeof p[i] === 'object') {

　　　　　　　　c[i] = (p[i].constructor === Array) ? [] : {};

　　　　　　　　deepCopy(p[i], c[i]);

　　　　　　} else {

　　　　　　　　　c[i] = p[i];

　　　　　　}  
　　　　}

　　　　return c;  
　　}

使用的时候这样写：

　　var Doctor = deepCopy(Chinese);

现在，给父对象加一个属性，值为数组。然后，在子对象上修改这个属性：

　　Chinese.birthPlaces = ['北京','上海','香港'];

　　Doctor.birthPlaces.push('厦门');

这时，父对象就不会受到影响了。

　　alert(Doctor.birthPlaces); //北京, 上海, 香港, 厦门

　　alert(Chinese.birthPlaces); //北京, 上海, 香港

目前，jQuery库使用的就是这种继承方法。

**Javascript的设计缺陷**

1. **为什么Javascript有设计缺陷？**

　　这里有三个客观原因，导致Javascript的设计不够完善。

**1. 设计阶段过于仓促**

　　Javascript的设计，其实只用了十天。而且，设计师是为了向公司交差，本人并不愿意这样设计（参见[《Javascript诞生记》](http://www.ruanyifeng.com/blog/2011/06/birth_of_javascript.html" \t "_blank)）。

　　另一方面，这种语言的设计初衷，是为了解决一些简单的网页互动（比如，检查"用户名"是否填写），并没有考虑复杂应用的需要。设计者做梦也想不到，Javascript将来可以写出像Gmail这种极其庞大复杂的网页。

**2. 没有先例**

　　Javascript同时结合了函数式编程和面向对象编程的特点，这很可能是历史上的第一例。而且直到今天为止，Javascript仍然是世界上唯一使用[Prototype继承模型](http://www.ruanyifeng.com/blog/2011/06/designing_ideas_of_inheritance_mechanism_in_javascript.html" \t "_blank)的主要语言。这使得它没有设计先例可以参考。

**3. 过早的标准化**

　　Javascript的发展非常快，根本没有时间调整设计。

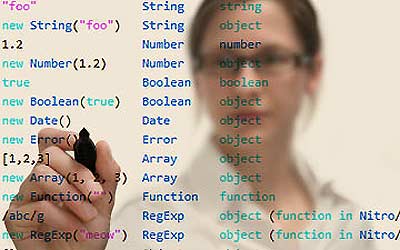
　　1995年5月，设计方案定稿；10月，解释器开发成功；12月，向市场推出，立刻被广泛接受，全世界的用户大量使用。Javascript缺乏一个从小到大、慢慢积累用户的过程，而是连续的爆炸式扩散增长。大量的既成网页和业余网页设计者的参与，使得调整语言规格困难重重。

　　更糟的是，Javascript的规格还没来及调整，就固化了。

　　1996年8月，微软公司强势介入，宣布推出自己的脚本语言Jscript；11月，为了压制微软，网景公司决定申请Javascript的国际标准；1997年6月，第一个国际标准ECMA-262正式颁布。

　　也就是说，Javascript推出一年半之后，国际标准就问世了。设计缺陷还没有充分暴露就成了标准。相比之下，C语言问世将近20年之后，国际标准才颁布。

1. **Javascript的10个设计缺陷**



**1. 不适合开发大型程序**

　　Javascript没有名称空间（namespace），很难模块化；没有如何将代码分布在多个文件的规范；允许同名函数的重复定义，后面的定义可以覆盖前面的定义，很不利于模块化加载。

**2. 非常小的标准库**

　　Javascript提供的标准函数库非常小，只能完成一些基本操作，很多功能都不具备。

**3. null和undefined**

　　null属于对象（object）的一种，意思是该对象为空；undefined则是一种数据类型，表示未定义。

　　typeof null; // object

　　typeof undefined; // undefined

　　两者非常容易混淆，但是含义完全不同。

　　var foo;

　　alert(foo == null); // true

　　alert(foo == undefined); // true

　　alert(foo === null); // false

　　alert(foo === undefined); // true

　　在编程实践中，null几乎没用，根本不应该设计它。

**4. 全局变量难以控制**

　　Javascript的全局变量，在所有模块中都是可见的；任何一个函数内部都可以生成全局变量，这大大加剧了程序的复杂性。

　　a = 1;

　　(function(){

　　　　b=2;

　　　　alert(a);

　　})(); // 1

　　alert(b); //2

**5. 自动插入行尾分号**

　　Javascript的所有语句，都必须以分号结尾。但是，如果你忘记加分号，解释器并不报错，而是为你自动加上分号。有时候，这会导致一些难以发现的错误。

　　比如，下面这个函数根本无法达到预期的结果，返回值不是一个对象，而是undefined。

　　function(){

　　　　return  
　　　　　　{  
　　　　　　　　i=1  
　　　　　　};

　　}

　　原因是解释器自动在return语句后面加上了分号。

　　function(){

**return;**  
　　　　　　{  
　　　　　　　　i=1  
　　　　　　};

　　}

**6. 加号运算符**

　　+号作为运算符，有两个含义，可以表示数字与数字的和，也可以表示字符与字符的连接。

　　alert(1+10); // 11

　　alert("1"+"10"); // 110

　　如果一个操作项是字符，另一个操作项是数字，则数字自动转化为字符。

　　alert(1+"10"); // 110

　　alert("10"+1); // 101

　　这样的设计，不必要地加剧了运算的复杂性，完全可以另行设置一个字符连接的运算符。

**7. NaN**

　　NaN是一种数字，表示超出了解释器的极限。它有一些很奇怪的特性：

　　NaN === NaN; //false

　　NaN !== NaN; //true

　　alert( 1 + NaN ); // NaN

　　与其设计NaN，不如解释器直接报错，反而有利于简化程序。

**8. 数组和对象的区分**

　　由于Javascript的数组也属于对象（object），所以要区分一个对象到底是不是数组，相当麻烦。[Douglas Crockford](http://javascript.crockford.com/remedial.html)的代码是这样的：

　　if ( arr &&   
　　　　typeof arr === 'object' &&  
　　　　typeof arr.length === 'number' &&  
　　　　!arr.propertyIsEnumerable('length')){

　　　　alert("arr is an array");

　　}

**9. == 和 ===**

　　==用来判断两个值是否相等。当两个值类型不同时，会发生自动转换，得到的结果非常不符合直觉。

　　"" == "0" // false

　　0 == "" // true

　　0 == "0" // true

　　false == "false" // false

　　false == "0" // true

　　false == undefined // false

　　false == null // false

　　null == undefined // true

　　" \t\r\n" == 0 // true

　　因此，推荐任何时候都使用"==="（精确判断）比较符。

**10. 基本类型的包装对象**

　　Javascript有三种基本数据类型：字符串、数字和布尔值。它们都有相应的建构函数，可以生成字符串对象、数字对象和布尔值对象。

　　new Boolean(false);

　　new Number(1234);

　　new String("Hello World");

　　与基本数据类型对应的对象类型，作用很小，造成的混淆却很大。

　　alert( typeof 1234); // number

　　alert( typeof new Number(1234)); // object

　　关于Javascript的更多怪异行为，请参见[Javascript Garden](http://bonsaiden.github.com/JavaScript-Garden/zh/" \t "_blank)和[wtfjs.com](http://wtfjs.com/" \t "_blank)。

1. **如何看待Javascript的设计缺陷？**

　　既然Javascript有缺陷，数量还不少，那么它是不是一种很糟糕的语言？有没有前途？

　　回答是Javascript并不算糟糕，相反它的编程能力很强大，前途很光明。

　　首先，如果遵守良好的编程规范，加上第三方函数库的帮助，Javascript的这些缺陷大部分可以回避。

　　其次，Javascript目前是网页编程的唯一语言，只要互联网继续发展，它就必然一起发展。目前，许多新项目大大扩展了它的用途，[node.js](http://nodejs.org/" \t "_blank)使得Javascript可以用于后端的服务器编程，[coffeeScript](http://jashkenas.github.com/coffee-script/" \t "_blank)使你可以用python和ruby的语法，撰写Javascript。

　　最后，只要发布新版本的语言标准（比如 [ECMAscript 5](http://www.ecma-international.org/publications/standards/Ecma-262.htm" \t "_blank)），就可以弥补这些设计缺陷。当然，标准的发布和标准的实现是两回事，上述的很多缺陷也许会一直伴随到Javascript存在的最后一天。

**Javascript的this用法**

this是Javascript语言的一个关键字。

它代表函数运行时，自动生成的一个内部对象，只能在函数内部使用。比如，

　　function test(){

　　　　this.x = 1;

　　}

随着函数使用场合的不同，this的值会发生变化。但是有一个总的原则，那就是this指的是，调用函数的那个对象。

下面分四种情况，详细讨论this的用法。

1. **纯粹的函数调用**

这是函数的最通常用法，属于全局性调用，因此this就代表全局对象Global。

请看下面这段代码，它的运行结果是1。

　　function test(){

　　　　this.x = 1;

　　　　alert(this.x);

　　}

　　test(); // 1

为了证明this就是全局对象，我对代码做一些改变：

　　var x = 1;

　　function test(){

　　　　alert(this.x);

　　}

　　test(); // 1

运行结果还是1。再变一下：

　　var x = 1;

　　function test(){

　　　　this.x = 0;

　　}

　　test();

　　alert(x); //0

1. **作为对象方法的调用**

函数还可以作为某个对象的方法调用，这时this就指这个上级对象。

　　function test(){

　　　　alert(this.x);

　　}

　　var o = {};

　　o.x = 1;

　　o.m = test;

　　o.m(); // 1

1. **作为构造函数调用**

所谓构造函数，就是通过这个函数生成一个新对象（object）。这时，this就指这个新对象。

　　function test(){

　　　　this.x = 1;

　　}

　　var o = new test();

　　alert(o.x); // 1

运行结果为1。为了表明这时this不是全局对象，我对代码做一些改变：

　　var x = 2;

　　function test(){

　　　　this.x = 1;

　　}

　　var o = new test();

　　alert(x); //2

运行结果为2，表明全局变量x的值根本没变。

1. **apply调用**

apply()是函数对象的一个方法，它的作用是改变函数的调用对象，它的第一个参数就表示改变后的调用这个函数的对象。因此，this指的就是这第一个参数。

　　var x = 0;

　　function test(){

　　　　alert(this.x);

　　}

　　var o={};

　　o.x = 1;

　　o.m = test;

　　o.m.apply(); //0

apply()的参数为空时，默认调用全局对象。因此，这时的运行结果为0，证明this指的是全局对象。

如果把最后一行代码修改为

　　o.m.apply(o); //1

运行结果就变成了1，证明了这时this代表的是对象o。

**Javascript与多线程**

——Javascript异步通信的实现原理与multi-node

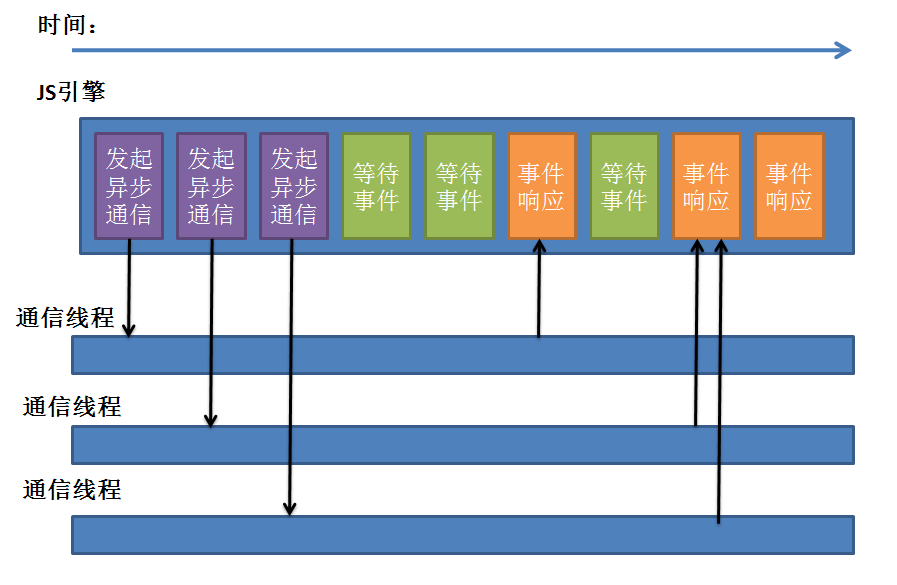
**问题**

1、Node编程中所有通信都是异步执行的，如果某个事件的响应函数需要操作全局变量，则多个回调同时运行时怎么实现同步？

2、群里里面的兄弟“老狗”发现了一个项目：multi-node，即使用可以用其编写多线程的网络服务。 但是node本来就是异步的，为什么还需要多线程？

**解答**

首先说一下JS的执行原理：js引擎执行js代码的时候是单线程的，即同一时刻只会有一个进程执行JS代码，回调函数也是一个一个执行的（按照事件发生的顺序，而不是代码的顺序）。JS中的异步通信和定时是由另外的线程实现的，脱离js线程上下文。以JS定时操作举例，当JS引擎执行setTimeout(callbackFunction, 100)操作时，它会通知定时线程我需要100毫秒的定时，之后JS引擎进入事件循环。100毫秒之后，定时引擎向事件队列中加入一个时间已到的事件。JS引擎从队列中读取时间已到的事件，执行callbackFunction。 如果同一时间有多个事件加入事件队列，JS引擎也只会一个一个的执行callback。对于异步也是同样，JS代码发起通信请求，通信线程执行通信操作，并在操作完成后将完成事件加入事件队列。JS引擎从队列中取出事件并调用回调处理通信结果。JS引擎在执行回调函数的时候，不能同时响应其他事件。



JS引擎示意图

如上所述，第一个问题迎刃而解，单线程自然就不必考虑同步问题了。

对于Node应用而言，它是部分多线程，即非阻塞IO为多线程，但是IO结果的处理还是单线程执行。所以在IO结果处理比较复杂占用很多时间的时候，还是有必要启动多个node线程来响应请求的。不过话又说回来，像node这种编程模型的优点就是可以使用简单的代码写出高性能的应用，如果使用多个node线程，则处理它们之间的数据交换和同步就会大大增加应用编写的复杂程度，是否得不偿失就需要认真斟酌了。

**相关资料**

浏览器中JS执行原理的说明：http://www.phpv.net/html/1700.html

浏览器伪多线程库：http://www.infoq.com/cn/articles/js\_multithread