[**JavaScript继承详解（一）**](http://www.cnblogs.com/jenry/archive/2012/01/19/2327609.html)

**面向对象与基于对象**

几乎每个开发人员都有面向对象语言（比如C++、C#、Java）的开发经验。 在传统面向对象的语言中，有两个非常重要的概念 - 类和实例。 类定义了一类事物公共的行为和方法；而实例则是类的一个具体实现。 我们还知道，面向对象编程有三个重要的概念 - 封装、继承和多态。

但是在JavaScript的世界中，所有的这一切特性似乎都不存在。 因为JavaScript本身不是面向对象的语言，而是基于对象的语言。 这里面就有一些有趣的特性，比如JavaScript中所有事物都是对象， 包括字符串、数组、日期、数字，甚至是函数，比如下面这个例子：

[?](http://www.cnblogs.com/jenry/archive/2012/01/19/2327609.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // 定义一个函数 - add  function add(a, b) {      add.invokeTimes++;      return a + b;  }  // 因为函数本身也是对象，这里为函数add定义一个属性，用来记录此函数被调用的次数  add.invokeTimes = 0;  add(1 + 1);  add(2 + 3);  console.log(add.invokeTimes); // 2 |

**模拟JavaScript中类和继承**

在面向对象的语言中，我们使用类来创建一个自定义对象。然而JavaScript中所有事物都是对象，**那么用什么办法来创建自定义对象呢？**

这就需要引入另外一个概念 - 原型（prototype），我们可以简单的把prototype看做是一个模版，新创建的自定义对象都是这个模版（prototype）的一个拷贝 （实际上不是拷贝而是链接，只不过这种链接是不可见，给人们的感觉好像是拷贝）。

让我们看一下通过prototype创建自定义对象的一个例子：

[?](http://www.cnblogs.com/jenry/archive/2012/01/19/2327609.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // 构造函数     function Person(name, sex) {         this.name = name;         this.sex = sex;     }     // 定义Person的原型，原型中的属性可以被自定义对象引用     Person.prototype = {         getName: function() {             return this.name;         },         getSex: function() {             return this.sex;         }     } |

这里我们把函数Person称为构造函数，也就是创建自定义对象的函数。可以看出，JavaScript通过构造函数和原型的方式模拟实现了类的功能。 创建自定义对象（实例化类）的代码：

[?](http://www.cnblogs.com/jenry/archive/2012/01/19/2327609.html)

|  |  |
| --- | --- |
| 1  2  3  4 | var zhang = new Person("ZhangSan", "man");  console.log(zhang.getName()); // "ZhangSan"  var chun = new Person("ChunHua", "woman");  console.log(chun.getName()); // "ChunHua" |

当代码var zhang = new Person("ZhangSan", "man")执行时，其实内部做了如下几件事情：

* 创建一个空白对象（new Object()）。
* 拷贝Person.prototype中的属性（键值对）到这个空对象中（我们前面提到，内部实现时不是拷贝而是一个隐藏的链接）。
* 将这个对象通过this关键字传递到构造函数中并执行构造函数。
* 将这个对象赋值给变量zhang。

为了证明prototype模版并不是被拷贝到实例化的对象中，而是一种链接的方式，请看如下代码：

[?](http://www.cnblogs.com/jenry/archive/2012/01/19/2327609.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | function Person(name, sex) {      this.name = name;      this.sex = sex;  }  Person.prototype.age = 20;  var zhang = new Person("ZhangSan", "man");  console.log(zhang.age); // 20  // 覆盖prototype中的age属性  zhang.age = 19;  console.log(zhang.age); // 19  delete zhang.age;  // 在删除实例属性age后，此属性值又从prototype中获取  console.log(zhang.age); // 20 |

这种在JavaScript内部实现的隐藏的prototype链接，是JavaScript赖以生存的温润土壤， 也是模拟实现继承的基础。

**如何在JavaScript中实现简单的继承？** 下面的例子将创建一个雇员类Employee，它从Person继承了原型prototype中的所有属性。

[?](http://www.cnblogs.com/jenry/archive/2012/01/19/2327609.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | function Employee(name, sex, employeeID) {      this.name = name;      this.sex = sex;      this.employeeID = employeeID;  }  // 将Employee的原型指向Person的一个实例  // 因为Person的实例可以调用Person原型中的方法, 所以Employee的实例也可以调用Person原型中的所有属性。  Employee.prototype = new Person();  Employee.prototype.getEmployeeID = function() {      return this.employeeID;  };  var zhang = new Employee("ZhangSan", "man", "1234");  console.log(zhang.getName()); // "ZhangSan |

上面关于继承的实现很粗糙，并且存在很多问题：

* 在创建Employee构造函数和原型（以后简称类）时，就对Person进行了实例化，这是不合适的。
* Employee的构造函数没法调用父类Person的构造函数，导致在Employee构造函数中对name和sex属性的重复赋值。
* Employee中的函数会覆盖Person中的同名函数，没有重载的机制（和上一条是一个类型的问题）。
* 创建JavaScript类的语法过于零散，不如C#/Java中的语法优雅。
* 实现中有constructor属性的指向错误，这个会在第二篇文章中讨论。

我们会在第三章完善这个例子。

**JavaScript继承的实现**

正因为JavaScript本身没有完整的类和继承的实现，并且我们也看到通过手工实现的方式存在很多问题， 因此对于这个富有挑战性的任务网上已经有很多实现了：

* Douglas Crockford - [Prototypal Inheritance in JavaScript](http://javascript.crockford.com/prototypal.html)
* Douglas Crockford - [Classical Inheritance in JavaScript](http://javascript.crockford.com/inheritance.html)
* John Resig - [Simple JavaScript Inheritance](http://ejohn.org/blog/simple-javascript-inheritance/)
* Dean Edwards - [A Base Class for JavaScript Inheritance](http://dean.edwards.name/weblog/2006/03/base/)
* [Prototype](http://www.prototypejs.org/learn/class-inheritance)
* [Mootools](http://mootools.net/docs/core/Class/Class)
* [Extjs](http://extjs.com/)

这个系列的文章将会逐一深入分析这些实现，最终达到对JavaScript中如何实现类和继承有一个深入的了解。

# [JavaScript继承详解（二）](http://www.cnblogs.com/jenry/archive/2012/01/19/2327612.html)

这一章我们将会重点介绍JavaScript中几个重要的属性（this、constructor、prototype）， 这些属性对于我们理解如何实现JavaScript中的类和继承起着至关重要的作用。

## this

this表示当前对象，如果在全局作用范围内使用this，则指代当前页面对象window； 如果在函数中使用this，则this指代什么是根据运行时此函数在什么对象上被调用。 我们还可以使用apply和call两个全局方法来改变函数中this的具体指向。

先看一个在全局作用范围内使用this的例子：

<script type="text/javascript">

console.log(this === window); // true

console.log(window.alert === this.alert); // true

console.log(this.parseInt("021", 10)); // 10

</script>

函数中的this是在运行时决定的，而不是函数定义时，如下：

// 定义一个全局函数

function foo() {

console.log(this.fruit);

}

// 定义一个全局变量，等价于window.fruit = "apple";

var fruit = "apple";

// 此时函数foo中this指向window对象

// 这种调用方式和window.foo();是完全等价的

foo(); // "apple"

// 自定义一个对象，并将此对象的属性foo指向全局函数foo

var pack = {

fruit: "orange",

foo: foo

};

// 此时函数foo中this指向window.pack对象

pack.foo(); // "orange"

全局函数apply和call可以用来改变函数中this的指向，如下：

// 定义一个全局函数

function foo() {

console.log(this.fruit);

}

// 定义一个全局变量

var fruit = "apple";

// 自定义一个对象

var pack = {

fruit: "orange"

};

// 等价于window.foo();

foo.apply(window); // "apple"

// 此时foo中的this === pack

foo.apply(pack); // "orange"

注：apply和call两个函数的作用相同，唯一的区别是两个函数的参数定义不同。

因为在JavaScript中函数也是对象，所以我们可以看到如下有趣的例子：

// 定义一个全局函数

function foo() {

if (this === window) {

console.log("this is window.");

}

}

// 函数foo也是对象，所以可以定义foo的属性boo为一个函数

foo.boo = function() {

if (this === foo) {

console.log("this is foo.");

} else if (this === window) {

console.log("this is window.");

}

};

// 等价于window.foo();

foo(); // this is window.

// 可以看到函数中this的指向调用函数的对象

foo.boo(); // this is foo.

// 使用apply改变函数中this的指向

foo.boo.apply(window); // this is window.

## prototype

我们已经在第一章中使用prototype模拟类和继承的实现。 prototype本质上还是一个JavaScript对象。 并且每个函数都有一个默认的prototype属性。 如果这个函数被用在创建自定义对象的场景中，我们称这个函数为构造函数。 比如下面一个简单的场景：

// 构造函数

function Person(name) {

this.name = name;

}

// 定义Person的原型，原型中的属性可以被自定义对象引用

Person.prototype = {

getName: function() {

return this.name;

}

}

var zhang = new Person("ZhangSan");

console.log(zhang.getName()); // "ZhangSan"

作为类比，我们考虑下JavaScript中的数据类型 - 字符串（String）、数字（Number）、数组（Array）、对象（Object）、日期（Date）等。 我们有理由相信，在JavaScript内部这些类型都是作为构造函数来实现的，比如：

// 定义数组的构造函数，作为JavaScript的一种预定义类型

function Array() {

// ...

}

// 初始化数组的实例

var arr1 = new Array(1, 56, 34, 12);

// 但是，我们更倾向于如下的语法定义：

var arr2 = [1, 56, 34, 12];

同时对数组操作的很多方法（比如concat、join、push）应该也是在prototype属性中定义的。 实际上，JavaScript所有的固有数据类型都具有只读的prototype属性（这是可以理解的：因为如果修改了这些类型的prototype属性，则哪些预定义的方法就消失了）， 但是我们可以向其中添加自己的扩展方法。

// 向JavaScript固有类型Array扩展一个获取最小值的方法

Array.prototype.min = function() {

var min = this[0];

for (var i = 1; i < this.length; i++) {

if (this[i] < min) {

min = this[i];

}

}

return min;

};

// 在任意Array的实例上调用min方法

console.log([1, 56, 34, 12].min()); // 1

注意：这里有一个陷阱，向Array的原型中添加扩展方法后，当使用for-in循环数组时，这个扩展方法也会被循环出来。 下面的代码说明这一点（假设已经向Array的原型中扩展了min方法）：

var arr = [1, 56, 34, 12];

var total = 0;

for (var i in arr) {

total += parseInt(arr[i], 10);

}

console.log(total); // NaN

解决方法也很简单：

var arr = [1, 56, 34, 12];

var total = 0;

for (var i in arr) {

if (arr.hasOwnProperty(i)) {

total += parseInt(arr[i], 10);

}

}

console.log(total); // 103

## constructor

constructor始终指向创建当前对象的构造函数。比如下面例子：

// 等价于 var foo = new Array(1, 56, 34, 12);

var arr = [1, 56, 34, 12];

console.log(arr.constructor === Array); // true

// 等价于 var foo = new Function();

var Foo = function() { };

console.log(Foo.constructor === Function); // true

// 由构造函数实例化一个obj对象

var obj = new Foo();

console.log(obj.constructor === Foo); // true

// 将上面两段代码合起来，就得到下面的结论

console.log(obj.constructor.constructor === Function); // true

但是当constructor遇到prototype时，有趣的事情就发生了。 我们知道每个函数都有一个默认的属性prototype，而这个prototype的constructor默认指向这个函数。如下例所示：

function Person(name) {

this.name = name;

};

Person.prototype.getName = function() {

return this.name;

};

var p = new Person("ZhangSan");

console.log(p.constructor === Person); // true

console.log(Person.prototype.constructor === Person); // true

// 将上两行代码合并就得到如下结果

console.log(p.constructor.prototype.constructor === Person); // true

当时当我们重新定义函数的prototype时（注意：和上例的区别，这里不是修改而是覆盖）， constructor的行为就有点奇怪了，如下示例：

function Person(name) {

this.name = name;

};

Person.prototype = {

getName: function() {

return this.name;

}

};

var p = new Person("ZhangSan");

console.log(p.constructor === Person); // false

console.log(Person.prototype.constructor === Person); // false

console.log(p.constructor.prototype.constructor === Person); // false

为什么呢？ 原来是因为覆盖Person.prototype时，等价于进行如下代码操作：

Person.prototype = new Object({

getName: function() {

return this.name;

}

});

而constructor始终指向创建自身的构造函数，所以此时Person.prototype.constructor === Object，即是：

function Person(name) {

this.name = name;

};

Person.prototype = {

getName: function() {

return this.name;

}

};

var p = new Person("ZhangSan");

console.log(p.constructor === Object); // true

console.log(Person.prototype.constructor === Object); // true

console.log(p.constructor.prototype.constructor === Object); // true

怎么修正这种问题呢？方法也很简单，重新覆盖Person.prototype.constructor即可：

function Person(name) {

this.name = name;

};

Person.prototype = new Object({

getName: function() {

return this.name;

}

});

Person.prototype.constructor = Person;

var p = new Person("ZhangSan");

console.log(p.constructor === Person); // true

console.log(Person.prototype.constructor === Person); // true

console.log(p.constructor.prototype.constructor === Person); // true

下一章我们将会对第一章提到的Person-Employee类和继承的实现进行完善。

# [JavaScript继承详解（三）](http://www.cnblogs.com/jenry/archive/2012/01/19/2327616.html)

在第一章中，我们使用构造函数和原型的方式在JavaScript的世界中实现了类和继承， 但是存在很多问题。这一章我们将会逐一分析这些问题，并给出解决方案。

        注：本章中的jClass的实现参考了[Simple            JavaScript Inheritance](http://ejohn.org/blog/simple-javascript-inheritance/)的做法。

首先让我们来回顾一下第一章中介绍的例子：

function Person(name) {

this.name = name;

}

Person.prototype = {

getName: function() {

return this.name;

}

}

function Employee(name, employeeID) {

this.name = name;

this.employeeID = employeeID;

}

Employee.prototype = new Person();

Employee.prototype.getEmployeeID = function() {

return this.employeeID;

};

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "ZhangSan"

## 修正constructor的指向错误

从上一篇文章中关于constructor的描述，我们知道Employee实例的constructor会有一个指向错误，如下所示：

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.constructor === Employee); // false

console.log(zhang.constructor === Object); // true

我们需要简单的修正：

function Employee(name, employeeID) {

this.name = name;

this.employeeID = employeeID;

}

Employee.prototype = new Person();

Employee.prototype.constructor = Employee;

Employee.prototype.getEmployeeID = function() {

return this.employeeID;

};

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.constructor === Employee); // true

console.log(zhang.constructor === Object); // false

## 创建Employee类时实例化Person是不合适的

但另一方面，我们又必须依赖于这种机制来实现继承。 解决办法是不在构造函数中初始化数据，而是提供一个原型方法（比如init）来初始化数据。

// 空的构造函数

function Person() {

}

Person.prototype = {

init: function(name) {

this.name = name;

},

getName: function() {

return this.name;

}

}

// 空的构造函数

function Employee() {

}

// 创建类的阶段不会初始化父类的数据，因为Person是一个空的构造函数

Employee.prototype = new Person();

Employee.prototype.constructor = Employee;

Employee.prototype.init = function(name, employeeID) {

this.name = name;

this.employeeID = employeeID;

};

Employee.prototype.getEmployeeID = function() {

return this.employeeID;

};

这种方式下，必须在实例化一个对象后手工调用init函数，如下：

var zhang = new Employee();

zhang.init("ZhangSan", "1234");

console.log(zhang.getName()); // "ZhangSan"

## 如何自动调用init函数？

必须达到两个效果，构造类时不要调用init函数和实例化对象时自动调用init函数。看来我们需要在调用空的构造函数时有一个状态标示。

// 创建一个全局的状态标示 - 当前是否处于类的构造阶段

var initializing = false;

function Person() {

if (!initializing) {

this.init.apply(this, arguments);

}

}

Person.prototype = {

init: function(name) {

this.name = name;

},

getName: function() {

return this.name;

}

}

function Employee() {

if (!initializing) {

this.init.apply(this, arguments);

}

}

// 标示当前进入类的创建阶段，不会调用init函数

initializing = true;

Employee.prototype = new Person();

Employee.prototype.constructor = Employee;

initializing = false;

Employee.prototype.init = function(name, employeeID) {

this.name = name;

this.employeeID = employeeID;

};

Employee.prototype.getEmployeeID = function() {

return this.employeeID;

};

// 初始化类实例时，自动调用类的原型函数init，并向init中传递参数

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "ZhangSan"

但是这样就必须引入全局变量，这是一个不好的信号。

## 如何避免引入全局变量initializing？

我们需要引入一个全局的函数来简化类的创建过程，同时封装内部细节避免引入全局变量。

// 当前是否处于创建类的阶段

var initializing = false;

function jClass(baseClass, prop) {

// 只接受一个参数的情况 - jClass(prop)

if (typeof (baseClass) === "object") {

prop = baseClass;

baseClass = null;

}

// 本次调用所创建的类（构造函数）

function F() {

// 如果当前处于实例化类的阶段，则调用init原型函数

if (!initializing) {

this.init.apply(this, arguments);

}

}

// 如果此类需要从其它类扩展

if (baseClass) {

initializing = true;

F.prototype = new baseClass();

F.prototype.constructor = F;

initializing = false;

}

// 覆盖父类的同名函数

for (var name in prop) {

if (prop.hasOwnProperty(name)) {

F.prototype[name] = prop[name];

}

}

return F;

};

使用jClass函数来创建类和继承类的方法：

var Person = jClass({

init: function(name) {

this.name = name;

},

getName: function() {

return this.name;

}

});

var Employee = jClass(Person, {

init: function(name, employeeID) {

this.name = name;

this.employeeID = employeeID;

},

getEmployeeID: function() {

return this.employeeID;

}

});

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "ZhangSan"

OK，现在创建类和实例化类的方式看起来优雅多了。 但是这里面还存在明显的瑕疵，Employee的初始化函数init无法调用父类的同名方法。

## 如何调用父类的同名方法？

我们可以通过为实例化对象提供一个base的属性，来指向父类（构造函数）的原型，如下：

// 当前是否处于创建类的阶段

var initializing = false;

function jClass(baseClass, prop) {

// 只接受一个参数的情况 - jClass(prop)

if (typeof (baseClass) === "object") {

prop = baseClass;

baseClass = null;

}

// 本次调用所创建的类（构造函数）

function F() {

// 如果当前处于实例化类的阶段，则调用init原型函数

if (!initializing) {

// 如果父类存在，则实例对象的base指向父类的原型

// 这就提供了在实例对象中调用父类方法的途径

if (baseClass) {

this.base = baseClass.prototype;

}

this.init.apply(this, arguments);

}

}

// 如果此类需要从其它类扩展

if (baseClass) {

initializing = true;

F.prototype = new baseClass();

F.prototype.constructor = F;

initializing = false;

}

// 覆盖父类的同名函数

for (var name in prop) {

if (prop.hasOwnProperty(name)) {

F.prototype[name] = prop[name];

}

}

return F;

};

调用方式：

var Person = jClass({

init: function(name) {

this.name = name;

},

getName: function() {

return this.name;

}

});

var Employee = jClass(Person, {

init: function(name, employeeID) {

// 调用父类的原型函数init，注意使用apply函数修改init的this指向

this.base.init.apply(this, [name]);

this.employeeID = employeeID;

},

getEmployeeID: function() {

return this.employeeID;

},

getName: function() {

// 调用父类的原型函数getName

return "Employee name: " + this.base.getName.apply(this);

}

});

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

目前为止，我们已经修正了在第一章手工实现继承的种种弊端。 通过我们自定义的jClass函数来创建类和子类，通过原型方法init初始化数据， 通过实例属性base来调用父类的原型函数。

唯一的缺憾是调用父类的代码太长，并且不好理解， 如果能够按照如下的方式调用岂不是更妙：

var Employee = jClass(Person, {

init: function(name, employeeID) {

// 如果能够这样调用，就再好不过了

this.base(name);

this.employeeID = employeeID;

}

});

## 优化jClass函数

// 当前是否处于创建类的阶段

var initializing = false;

function jClass(baseClass, prop) {

// 只接受一个参数的情况 - jClass(prop)

if (typeof (baseClass) === "object") {

prop = baseClass;

baseClass = null;

}

// 本次调用所创建的类（构造函数）

function F() {

// 如果当前处于实例化类的阶段，则调用init原型函数

if (!initializing) {

// 如果父类存在，则实例对象的baseprototype指向父类的原型

// 这就提供了在实例对象中调用父类方法的途径

if (baseClass) {

this.baseprototype = baseClass.prototype;

}

this.init.apply(this, arguments);

}

}

// 如果此类需要从其它类扩展

if (baseClass) {

initializing = true;

F.prototype = new baseClass();

F.prototype.constructor = F;

initializing = false;

}

// 覆盖父类的同名函数

for (var name in prop) {

if (prop.hasOwnProperty(name)) {

// 如果此类继承自父类baseClass并且父类原型中存在同名函数name

if (baseClass &&

typeof (prop[name]) === "function" &&

typeof (F.prototype[name]) === "function") {

// 重定义函数name -

// 首先在函数上下文设置this.base指向父类原型中的同名函数

// 然后调用函数prop[name]，返回函数结果

// 注意：这里的自执行函数创建了一个上下文，这个上下文返回另一个函数，

// 此函数中可以应用此上下文中的变量，这就是闭包（Closure）。

// 这是JavaScript框架开发中常用的技巧。

F.prototype[name] = (function(name, fn) {

return function() {

this.base = baseClass.prototype[name];

return fn.apply(this, arguments);

};

})(name, prop[name]);

} else {

F.prototype[name] = prop[name];

}

}

}

return F;

};

此时，创建类与子类以及调用方式都显得非常优雅，请看：

var Person = jClass({

init: function(name) {

this.name = name;

},

getName: function() {

return this.name;

}

});

var Employee = jClass(Person, {

init: function(name, employeeID) {

this.base(name);

this.employeeID = employeeID;

},

getEmployeeID: function() {

return this.employeeID;

},

getName: function() {

return "Employee name: " + this.base();

}

});

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

至此，我们已经创建了一个完善的函数jClass， 帮助我们在JavaScript中以比较优雅的方式实现类和继承。

在以后的章节中，我们会陆续分析网上一些比较流行的JavaScript类和继承的实现。 不过万变不离其宗，那些实现也无非把我们这章中提到的概念颠来簸去的“炒作”， 为的就是一种更优雅的调用方式。

[**JavaScript继承详解（四）**](http://www.cnblogs.com/jenry/archive/2012/01/19/2327622.html)

在本章中，我们将分析Douglas Crockford关于JavaScript继承的一个实现 - [Classical Inheritance in JavaScript](http://javascript.crockford.com/inheritance.html)。 Crockford是JavaScript开发社区最知名的权威，是[JSON](http://www.json.org/)、[JSLint](http://www.jslint.com/)、[JSMin](http://www.crockford.com/javascript/jsmin.html)和[ADSafe](http://www.adsafe.org/)之父，是《JavaScript: The Good Parts》的作者。 现在是Yahoo的资深JavaScript架构师，参与YUI的设计开发。 这里有一篇[文章](http://dancewithnet.com/2009/03/29/douglas-crockford/)详细介绍了Crockford的生平和著作。 当然Crockford也是我等小辈崇拜的对象。

**调用方式**

首先让我们看下使用Crockford式继承的调用方式： 注意：代码中的method、inherits、uber都是自定义的对象，我们会在后面的代码分析中详解。

// 定义Person类

function Person(name) {

this.name = name;

}

// 定义Person的原型方法

Person.method("getName", function() {

return this.name;

});

// 定义Employee类

function Employee(name, employeeID) {

this.name = name;

this.employeeID = employeeID;

}

// 指定Employee类从Person类继承

Employee.inherits(Person);

// 定义Employee的原型方法

Employee.method("getEmployeeID", function() {

return this.employeeID;

});

Employee.method("getName", function() {

// 注意，可以在子类中调用父类的原型方法

return "Employee name: " + this.uber("getName");

});

// 实例化子类

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

这里面有几处不得不提的硬伤：

* 子类从父类继承的代码必须在子类和父类都定义好之后进行，并且必须在子类原型方法定义之前进行。
* 虽然子类方法体中可以调用父类的方法，但是子类的构造函数无法调用父类的构造函数。
* 代码的书写不够优雅，比如原型方法的定义以及调用父类的方法（不直观）。

当然Crockford的实现还支持子类中的方法调用带参数的父类方法，如下例子：

function Person(name) {

this.name = name;

}

Person.method("getName", function(prefix) {

return prefix + this.name;

});

function Employee(name, employeeID) {

this.name = name;

this.employeeID = employeeID;

}

Employee.inherits(Person);

Employee.method("getName", function() {

// 注意，uber的第一个参数是要调用父类的函数名称，后面的参数都是此函数的参数

// 个人觉得这样方式不如这样调用来的直观：this.uber("Employee name: ")

return this.uber("getName", "Employee name: ");

});

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

**代码分析**

首先method函数的定义就很简单了：

Function.prototype.method = function(name, func) {

// this指向当前函数，也即是typeof(this) === "function"

this.prototype[name] = func;

return this;

};

要特别注意这里this的指向。当我们看到this时，不能仅仅关注于当前函数，而应该想到当前函数的调用方式。 比如这个例子中的method我们不会通过new的方式调用，所以method中的this指向的是当前函数。

inherits函数的定义有点复杂：

Function.method('inherits', function (parent) {

// 关键是这一段：this.prototype = new parent()，这里实现了原型的引用

var d = {}, p = (this.prototype = new parent());

// 只为子类的原型增加uber方法，这里的Closure是为了在调用uber函数时知道当前类的父类的原型（也即是变量 - v）

this.method('uber', function uber(name) {

// 这里考虑到如果name是存在于Object.prototype中的函数名的情况

// 比如 "toString" in {} === true

if (!(name in d)) {

// 通过d[name]计数，不理解具体的含义

d[name] = 0;

}

var f, r, t = d[name], v = parent.prototype;

if (t) {

while (t) {

v = v.constructor.prototype;

t -= 1;

}

f = v[name];

} else {

// 个人觉得这段代码有点繁琐，既然uber的含义就是父类的函数，那么f直接指向v[name]就可以了

f = p[name];

if (f == this[name]) {

f = v[name];

}

}

d[name] += 1;

// 执行父类中的函数name，但是函数中this指向当前对象

// 同时注意使用Array.prototype.slice.apply的方式对arguments进行截断（因为arguments不是标准的数组，没有slice方法）

r = f.apply(this, Array.prototype.slice.apply(arguments, [1]));

d[name] -= 1;

return r;

});

return this;

});

注意，在inherits函数中还有一个小小的BUG，那就是没有重定义constructor的指向，所以会发生如下的错误：

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

console.log(zhang.constructor === Employee); // false

console.log(zhang.constructor === Person); // true

**改进建议**

根据前面的分析，个人觉得method函数必要性不大，反而容易混淆视线。 而inherits方法可以做一些瘦身（因为Crockford可能考虑更多的情况，原文中介绍了好几种使用inherits的方式，而我们只关注其中的一种）， 并修正了constructor的指向错误。

Function.prototype.inherits = function(parent) {

this.prototype = new parent();

this.prototype.constructor = this;

this.prototype.uber = function(name) {

f = parent.prototype[name];

return f.apply(this, Array.prototype.slice.call(arguments, 1));

};

};

调用方式：

function Person(name) {

this.name = name;

}

Person.prototype.getName = function(prefix) {

return prefix + this.name;

};

function Employee(name, employeeID) {

this.name = name;

this.employeeID = employeeID;

}

Employee.inherits(Person);

Employee.prototype.getName = function() {

return this.uber("getName", "Employee name: ");

};

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

console.log(zhang.constructor === Employee); // true

**有点意思**

在文章的结尾，Crockford居然放出了这样的话：

I have been writing JavaScript for 8 years now, and I have never once found need to use an uber function. The super idea is fairly important in the classical pattern, but it appears to be unnecessary in the prototypal and functional patterns. I now see my early attempts to support the classical model in JavaScript as a mistake.

可见Crockford对在JavaScript中实现面向对象的编程不赞成，并且声称JavaScript应该按照原型和函数的模式（the prototypal and functional patterns）进行编程。 不过就我个人而言，在复杂的场景中如果有面向对象的机制会方便的多。 但谁有能担保呢，即使像jQuery UI这样的项目也没用到继承，而另一方面，像Extjs、Qooxdoo则极力倡导一种面向对象的JavaScript。 甚至Cappuccino项目还发明一种Objective-J语言来实践面向对象的JavaScript。

# [JavaScript继承详解（五）](http://www.cnblogs.com/jenry/archive/2012/01/19/2327623.html)

在本章中，我们将分析John Resig关于JavaScript继承的一个实现 - [Simple JavaScript Inheritance](http://ejohn.org/blog/simple-javascript-inheritance/)。 John Resig作为[jQuery](http://jquery.com/)的创始人而声名在外。是《Pro JavaScript Techniques》的作者，而且Resig将会在今年秋天推出一本书《JavaScript Secrets》，非常期待。

## 调用方式

调用方式非常优雅： 注意：代码中的Class、extend、\_super都是自定义的对象，我们会在后面的代码分析中详解。

var Person = Class.extend({

// init是构造函数

init: function(name) {

this.name = name;

},

getName: function() {

return this.name;

}

});

// Employee类从Person类继承

var Employee = Person.extend({

// init是构造函数

init: function(name, employeeID) {

// 在构造函数中调用父类的构造函数

this.\_super(name);

this.employeeID = employeeID;

},

getEmployeeID: function() {

return this.employeeID;

},

getName: function() {

// 调用父类的方法

return "Employee name: " + this.\_super();

}

});

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

说实话，对于完成本系列文章的目标-继承-而言，真找不到什么缺点。方法一如jQuery一样简洁明了。

## 代码分析

为了一个漂亮的调用方式，内部实现的确复杂了很多，不过这些也是值得的 - 一个人的思考带给了无数程序员快乐的微笑 - 嘿嘿，有点肉麻。 不过其中的一段代码的确迷惑我一段时间：

fnTest = /xyz/.test(function(){xyz;}) ? /\b\_super\b/ : /.\*/;

我曾在几天前的博客中写过一篇文章专门阐述这个问题，有兴趣可以向前翻一翻。

// 自执行的匿名函数创建一个上下文，避免引入全局变量

(function() {

// initializing变量用来标示当前是否处于类的创建阶段，

// - 在类的创建阶段是不能调用原型方法init的

// - 我们曾在本系列的第三篇文章中详细阐述了这个问题

// fnTest是一个正则表达式，可能的取值为（/\b\_super\b/ 或 /.\*/）

// - 对 /xyz/.test(function() { xyz; }) 的测试是为了检测浏览器是否支持test参数为函数的情况

// - 不过我对IE7.0,Chrome2.0,FF3.5进行了测试，此测试都返回true。

// - 所以我想这样对fnTest赋值大部分情况下也是对的：fnTest = /\b\_super\b/;

var initializing = false, fnTest = /xyz/.test(function() { xyz; }) ? /\b\_super\b/ : /.\*/;

// 基类构造函数

// 这里的this是window，所以这整段代码就向外界开辟了一扇窗户 - window.Class

this.Class = function() { };

// 继承方法定义

Class.extend = function(prop) {

// 这个地方很是迷惑人，还记得我在本系列的第二篇文章中提到的么

// - this具体指向什么不是定义时能决定的，而是要看此函数是怎么被调用的

// - 我们已经知道extend肯定是作为方法调用的，而不是作为构造函数

// - 所以这里this指向的不是Object，而是Function（即是Class），那么this.prototype就是父类的原型对象

// - 注意：\_super指向父类的原型对象，我们会在后面的代码中多次碰见这个变量

var \_super = this.prototype;

// 通过将子类的原型指向父类的一个实例对象来完成继承

// - 注意：this是基类构造函数（即是Class）

initializing = true;

var prototype = new this();

initializing = false;

// 我觉得这段代码是经过作者优化过的，所以读起来非常生硬，我会在后面详解

for (var name in prop) {

prototype[name] = typeof prop[name] == "function" &&

typeof \_super[name] == "function" && fnTest.test(prop[name]) ?

(function(name, fn) {

return function() {

var tmp = this.\_super;

this.\_super = \_super[name];

var ret = fn.apply(this, arguments);

this.\_super = tmp;

return ret;

};

})(name, prop[name]) :

prop[name];

}

// 这个地方可以看出，Resig很会伪装哦

// - 使用一个同名的局部变量来覆盖全局变量，很是迷惑人

// - 如果你觉得拗口的话，完全可以使用另外一个名字，比如function F()来代替function Class()

// - 注意：这里的Class不是在最外层定义的那个基类构造函数

function Class() {

// 在类的实例化时，调用原型方法init

if (!initializing && this.init)

this.init.apply(this, arguments);

}

// 子类的prototype指向父类的实例（完成继承的关键）

Class.prototype = prototype;

// 修正constructor指向错误

Class.constructor = Class;

// 子类自动获取extend方法，arguments.callee指向当前正在执行的函数

Class.extend = arguments.callee;

return Class;

};

})();

下面我会对其中的for-in循环进行解读，把自执行的匿名方法用一个局部函数来替换， 这样有利于我们看清真相：

(function() {

var initializing = false, fnTest = /xyz/.test(function() { xyz; }) ? /\b\_super\b/ : /.\*/;

this.Class = function() { };

Class.extend = function(prop) {

var \_super = this.prototype;

initializing = true;

var prototype = new this();

initializing = false;

// 如果父类和子类有同名方法，并且子类中此方法（name）通过\_super调用了父类方法

// - 则重新定义此方法

function fn(name, fn) {

return function() {

// 将实例方法\_super保护起来。

// 个人觉得这个地方没有必要，因为每次调用这样的函数时都会对this.\_super重新定义。

var tmp = this.\_super;

// 在执行子类的实例方法name时，添加另外一个实例方法\_super，此方法指向父类的同名方法

this.\_super = \_super[name];

// 执行子类的方法name，注意在方法体内this.\_super可以调用父类的同名方法

var ret = fn.apply(this, arguments);

this.\_super = tmp;

// 返回执行结果

return ret;

};

}

// 拷贝prop中的所有属性到子类原型中

for (var name in prop) {

// 如果prop和父类中存在同名的函数，并且此函数中使用了\_super方法，则对此方法进行特殊处理 - fn

// 否则将此方法prop[name]直接赋值给子类的原型

if (typeof prop[name] === "function" &&

typeof \_super[name] === "function" && fnTest.test(prop[name])) {

prototype[name] = fn(name, prop[name]);

} else {

prototype[name] = prop[name];

}

}

function Class() {

if (!initializing && this.init) {

this.init.apply(this, arguments);

}

}

Class.prototype = prototype;

Class.constructor = Class;

Class.extend = arguments.callee;

return Class;

};

})();

写到这里，大家是否觉得Resig的实现和我们在第三章一步一步实现的jClass很类似。 其实在写这一系列的文章之前，我已经对prototype、mootools、extjs、 jQuery-Simple-Inheritance、Crockford-Classical-Inheritance这些实现有一定的了解，并且大部分都在实际项目中使用过。 在第三章中实现jClass也参考了Resig的实现，在此向Resig表示感谢。 下来我们就把jClass改造成和这里的Class具有相同的行为。

## 我们的实现

将我们在第三章实现的jClass改造成目前John Resig所写的形式相当简单，只需要修改其中的两三行就行了：

(function() {

// 当前是否处于创建类的阶段

var initializing = false;

jClass = function() { };

jClass.extend = function(prop) {

// 如果调用当前函数的对象（这里是函数）不是Class，则是父类

var baseClass = null;

if (this !== jClass) {

baseClass = this;

}

// 本次调用所创建的类（构造函数）

function F() {

// 如果当前处于实例化类的阶段，则调用init原型函数

if (!initializing) {

// 如果父类存在，则实例对象的baseprototype指向父类的原型

// 这就提供了在实例对象中调用父类方法的途径

if (baseClass) {

this.\_superprototype = baseClass.prototype;

}

this.init.apply(this, arguments);

}

}

// 如果此类需要从其它类扩展

if (baseClass) {

initializing = true;

F.prototype = new baseClass();

F.prototype.constructor = F;

initializing = false;

}

// 新创建的类自动附加extend函数

F.extend = arguments.callee;

// 覆盖父类的同名函数

for (var name in prop) {

if (prop.hasOwnProperty(name)) {

// 如果此类继承自父类baseClass并且父类原型中存在同名函数name

if (baseClass &&

typeof (prop[name]) === "function" &&

typeof (F.prototype[name]) === "function" &&

/\b\_super\b/.test(prop[name])) {

// 重定义函数name -

// 首先在函数上下文设置this.\_super指向父类原型中的同名函数

// 然后调用函数prop[name]，返回函数结果

// 注意：这里的自执行函数创建了一个上下文，这个上下文返回另一个函数，

// 此函数中可以应用此上下文中的变量，这就是闭包（Closure）。

// 这是JavaScript框架开发中常用的技巧。

F.prototype[name] = (function(name, fn) {

return function() {

this.\_super = baseClass.prototype[name];

return fn.apply(this, arguments);

};

})(name, prop[name]);

} else {

F.prototype[name] = prop[name];

}

}

}

return F;

};

})();

// 经过改造的jClass

var Person = jClass.extend({

init: function(name) {

this.name = name;

},

getName: function(prefix) {

return prefix + this.name;

}

});

var Employee = Person.extend({

init: function(name, employeeID) {

// 调用父类的方法

this.\_super(name);

this.employeeID = employeeID;

},

getEmployeeIDName: function() {

// 注意：我们还可以通过这种方式调用父类中的其他函数

var name = this.\_superprototype.getName.call(this, "Employee name: ");

return name + ", Employee ID: " + this.employeeID;

},

getName: function() {

// 调用父类的方法

return this.\_super("Employee name: ");

}

});

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

console.log(zhang.getEmployeeIDName()); // "Employee name: ZhangSan, Employee ID: 1234"

JUST COOL!

[**JavaScript继承详解（六）**](http://www.cnblogs.com/jenry/archive/2012/01/19/2327624.html)

在本章中，我们将分析[Prototypejs](http://prototypejs.org/)中关于JavaScript继承的实现。        Prototypejs是最早的JavaScript类库，可以说是JavaScript类库的鼻祖。 我在几年前接触的第一个JavaScript类库就是这位，因此Prototypejs有着广泛的群众基础。          
        不过当年Prototypejs中的关于继承的实现相当的简单，源代码就寥寥几行，我们来看下。

**早期Prototypejs中继承的实现**

        源码：

var Class = {

// Class.create仅仅返回另外一个函数，此函数执行时将调用原型方法initialize

create: function() {

return function() {

this.initialize.apply(this, arguments);

}

}

};

// 对象的扩展

Object.extend = function(destination, source) {

for (var property in source) {

destination[property] = source[property];

}

return destination;

};

调用方式：

var Person = Class.create();

Person.prototype = {

initialize: function(name) {

this.name = name;

},

getName: function(prefix) {

return prefix + this.name;

}

};

var Employee = Class.create();

Employee.prototype = Object.extend(new Person(), {

initialize: function(name, employeeID) {

this.name = name;

this.employeeID = employeeID;

},

getName: function() {

return "Employee name: " + this.name;

}

});

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

很原始的感觉对吧，在子类函数中没有提供调用父类函数的途径。

**Prototypejs 1.6以后的继承实现**

        首先来看下调用方式：

// 通过Class.create创建一个新类

var Person = Class.create({

// initialize是构造函数

initialize: function(name) {

this.name = name;

},

getName: function(prefix) {

return prefix + this.name;

}

});

// Class.create的第一个参数是要继承的父类

var Employee = Class.create(Person, {

// 通过将子类函数的第一个参数设为$super来引用父类的同名函数

// 比较有创意，不过内部实现应该比较复杂，至少要用一个闭包来设置$super的上下文this指向当前对象

initialize: function($super, name, employeeID) {

$super(name);

this.employeeID = employeeID;

},

getName: function($super) {

return $super("Employee name: ");

}

});

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

这里我们将Prototypejs 1.6.0.3中继承实现单独取出来， 那些不想引用整个prototype库而只想使用prototype式继承的朋友， 可以直接把下面代码拷贝出来保存为JS文件就行了。

var Prototype = {

emptyFunction: function() { }

};

var Class = {

create: function() {

var parent = null, properties = $A(arguments);

if (Object.isFunction(properties[0]))

parent = properties.shift();

function klass() {

this.initialize.apply(this, arguments);

}

Object.extend(klass, Class.Methods);

klass.superclass = parent;

klass.subclasses = [];

if (parent) {

var subclass = function() { };

subclass.prototype = parent.prototype;

klass.prototype = new subclass;

parent.subclasses.push(klass);

}

for (var i = 0; i < properties.length; i++)

klass.addMethods(properties[i]);

if (!klass.prototype.initialize)

klass.prototype.initialize = Prototype.emptyFunction;

klass.prototype.constructor = klass;

return klass;

}

};

Class.Methods = {

addMethods: function(source) {

var ancestor = this.superclass && this.superclass.prototype;

var properties = Object.keys(source);

if (!Object.keys({ toString: true }).length)

properties.push("toString", "valueOf");

for (var i = 0, length = properties.length; i < length; i++) {

var property = properties[i], value = source[property];

if (ancestor && Object.isFunction(value) && value.argumentNames().first() == "$super") {

var method = value;

value = (function(m) {

return function() { return ancestor[m].apply(this, arguments) };

})(property).wrap(method);

value.valueOf = method.valueOf.bind(method);

value.toString = method.toString.bind(method);

}

this.prototype[property] = value;

}

return this;

}

};

Object.extend = function(destination, source) {

for (var property in source)

destination[property] = source[property];

return destination;

};

function $A(iterable) {

if (!iterable) return [];

if (iterable.toArray) return iterable.toArray();

var length = iterable.length || 0, results = new Array(length);

while (length--) results[length] = iterable[length];

return results;

}

Object.extend(Object, {

keys: function(object) {

var keys = [];

for (var property in object)

keys.push(property);

return keys;

},

isFunction: function(object) {

return typeof object == "function";

},

isUndefined: function(object) {

return typeof object == "undefined";

}

});

Object.extend(Function.prototype, {

argumentNames: function() {

var names = this.toString().match(/^[\s\(]\*function[^(]\*\(([^\)]\*)\)/)[1].replace(/\s+/g, '').split(',');

return names.length == 1 && !names[0] ? [] : names;

},

bind: function() {

if (arguments.length < 2 && Object.isUndefined(arguments[0])) return this;

var \_\_method = this, args = $A(arguments), object = args.shift();

return function() {

return \_\_method.apply(object, args.concat($A(arguments)));

}

},

wrap: function(wrapper) {

var \_\_method = this;

return function() {

return wrapper.apply(this, [\_\_method.bind(this)].concat($A(arguments)));

}

}

});

Object.extend(Array.prototype, {

first: function() {

return this[0];

}

});

首先，我们需要先解释下Prototypejs中一些方法的定义。

* argumentNames: 获取函数的参数数组
* function init($super, name, employeeID) {
* }
* console.log(init.argumentNames().join(",")); // "$super,name,employeeID"

* bind: 绑定函数的上下文this到一个新的对象（一般是函数的第一个参数）
* var name = "window";
* var p = {
* name: "Lisi",
* getName: function() {
* return this.name;
* }
* };
* console.log(p.getName()); // "Lisi"
* console.log(p.getName.bind(window)()); // "window"

* wrap: 把当前调用函数作为包裹器wrapper函数的第一个参数
* var name = "window";
* var p = {
* name: "Lisi",
* getName: function() {
* return this.name;
* }
* };
* function wrapper(originalFn) {
* return "Hello: " + originalFn();
* }
* console.log(p.getName()); // "Lisi"
* console.log(p.getName.bind(window)()); // "window"
* console.log(p.getName.wrap(wrapper)()); // "Hello: window"
* console.log(p.getName.wrap(wrapper).bind(p)()); // "Hello: Lisi"

有一点绕口，对吧。这里要注意的是wrap和bind调用返回的都是函数，把握住这个原则，就很容易看清本质了。

对这些函数有了一定的认识之后，我们再来解析Prototypejs继承的核心内容。                这里有两个重要的定义，一个是Class.extend，另一个是Class.Methods.addMethods。

var Class = {

create: function() {

// 如果第一个参数是函数，则作为父类

var parent = null, properties = $A(arguments);

if (Object.isFunction(properties[0]))

parent = properties.shift();

// 子类构造函数的定义

function klass() {

this.initialize.apply(this, arguments);

}

// 为子类添加原型方法Class.Methods.addMethods

Object.extend(klass, Class.Methods);

// 不仅为当前类保存父类的引用，同时记录了所有子类的引用

klass.superclass = parent;

klass.subclasses = [];

if (parent) {

// 核心代码 - 如果父类存在，则实现原型的继承

// 这里为创建类时不调用父类的构造函数提供了一种新的途径

// - 使用一个中间过渡类，这和我们以前使用全局initializing变量达到相同的目的，

// - 但是代码更优雅一点。

var subclass = function() { };

subclass.prototype = parent.prototype;

klass.prototype = new subclass;

parent.subclasses.push(klass);

}

// 核心代码 - 如果子类拥有父类相同的方法，则特殊处理，将会在后面详解

for (var i = 0; i < properties.length; i++)

klass.addMethods(properties[i]);

if (!klass.prototype.initialize)

klass.prototype.initialize = Prototype.emptyFunction;

// 修正constructor指向错误

klass.prototype.constructor = klass;

return klass;

}

};

再来看addMethods做了哪些事情：

Class.Methods = {

addMethods: function(source) {

// 如果父类存在，ancestor指向父类的原型对象

var ancestor = this.superclass && this.superclass.prototype;

var properties = Object.keys(source);

// Firefox和Chrome返回1，IE8返回0，所以这个地方特殊处理

if (!Object.keys({ toString: true }).length)

properties.push("toString", "valueOf");

// 循环子类原型定义的所有属性，对于那些和父类重名的函数要重新定义

for (var i = 0, length = properties.length; i < length; i++) {

// property为属性名，value为属性体（可能是函数，也可能是对象）

var property = properties[i], value = source[property];

// 如果父类存在，并且当前当前属性是函数，并且此函数的第一个参数为 $super

if (ancestor && Object.isFunction(value) && value.argumentNames().first() == "$super") {

var method = value;

// 下面三行代码是精华之所在，大概的意思：

// - 首先创建一个自执行的匿名函数返回另一个函数，此函数用于执行父类的同名函数

// - （因为这是在循环中，我们曾多次指出循环中的函数引用局部变量的问题）

// - 其次把这个自执行的匿名函数的作为method的第一个参数（也就是对应于形参$super）

// 不过，窃以为这个地方作者有点走火入魔，完全没必要这么复杂，后面我会详细分析这段代码。

value = (function(m) {

return function() { return ancestor[m].apply(this, arguments) };

})(property).wrap(method);

value.valueOf = method.valueOf.bind(method);

// 因为我们改变了函数体，所以重新定义函数的toString方法

// 这样用户调用函数的toString方法时，返回的是原始的函数定义体

value.toString = method.toString.bind(method);

}

this.prototype[property] = value;

}

return this;

}

};

上面的代码中我曾有“走火入魔”的说法，并不是对作者的亵渎，        只是觉得作者对JavaScript中的一个重要准则（通过自执行的匿名函数创建作用域）        运用的有点过头。

value = (function(m) {

return function() { return ancestor[m].apply(this, arguments) };

})(property).wrap(method);

其实这段代码和下面的效果一样：

value = ancestor[property].wrap(method);

我们把wrap函数展开就能看的更清楚了：

value = (function(fn, wrapper) {

var \_\_method = fn;

return function() {

return wrapper.apply(this, [\_\_method.bind(this)].concat($A(arguments)));

}

})(ancestor[property], method);

可以看到，我们其实为父类的函数ancestor[property]通过自执行的匿名函数创建了作用域。        而原作者是为property创建的作用域。两则的最终效果是一致的。

**我们对Prototypejs继承的重实现**

        分析了这么多，其实也不是很难，就那么多概念，大不了换种表现形式。                下面我们就用前几章我们自己实现的jClass来实现Prototypejs形式的继承。

// 注意：这是我们自己实现的类似Prototypejs继承方式的代码，可以直接拷贝下来使用

// 这个方法是借用Prototypejs中的定义

function argumentNames(fn) {

var names = fn.toString().match(/^[\s\(]\*function[^(]\*\(([^\)]\*)\)/)[1].replace(/\s+/g, '').split(',');

return names.length == 1 && !names[0] ? [] : names;

}

function jClass(baseClass, prop) {

// 只接受一个参数的情况 - jClass(prop)

if (typeof (baseClass) === "object") {

prop = baseClass;

baseClass = null;

}

// 本次调用所创建的类（构造函数）

function F() {

// 如果父类存在，则实例对象的baseprototype指向父类的原型

// 这就提供了在实例对象中调用父类方法的途径

if (baseClass) {

this.baseprototype = baseClass.prototype;

}

this.initialize.apply(this, arguments);

}

// 如果此类需要从其它类扩展

if (baseClass) {

var middleClass = function() {};

middleClass.prototype = baseClass.prototype;

F.prototype = new middleClass();

F.prototype.constructor = F;

}

// 覆盖父类的同名函数

for (var name in prop) {

if (prop.hasOwnProperty(name)) {

// 如果此类继承自父类baseClass并且父类原型中存在同名函数name

if (baseClass &&

typeof (prop[name]) === "function" &&

argumentNames(prop[name])[0] === "$super") {

// 重定义子类的原型方法prop[name]

// - 这里面有很多JavaScript方面的技巧，如果阅读有困难的话，可以参阅我前面关于JavaScript Tips and Tricks的系列文章

// - 比如$super封装了父类方法的调用，但是调用时的上下文指针要指向当前子类的实例对象

// - 将$super作为方法调用的第一个参数

F.prototype[name] = (function(name, fn) {

return function() {

var that = this;

$super = function() {

return baseClass.prototype[name].apply(that, arguments);

};

return fn.apply(this, Array.prototype.concat.apply($super, arguments));

};

})(name, prop[name]);

} else {

F.prototype[name] = prop[name];

}

}

}

return F;

};

调用方式和Prototypejs的调用方式保持一致：

var Person = jClass({

initialize: function(name) {

this.name = name;

},

getName: function() {

return this.name;

}

});

var Employee = jClass(Person, {

initialize: function($super, name, employeeID) {

$super(name);

this.employeeID = employeeID;

},

getEmployeeID: function() {

return this.employeeID;

},

getName: function($super) {

return "Employee name: " + $super();

}

});

var zhang = new Employee("ZhangSan", "1234");

console.log(zhang.getName()); // "Employee name: ZhangSan"

经过本章的学习，就更加坚定了我们的信心，像Prototypejs形式的继承我们也能够轻松搞定。                以后的几个章节，我们会逐步分析mootools,Extjs等JavaScript类库中继承的实现，敬请期待。

分类: [Javascript](http://www.cnblogs.com/jenry/category/51530.html)