1. 类和对象

ECMAScript 中没有类的概念。

ECMA-262 中对对象的定义： “**无序**属性的集合，其属性可以包含基本值、对象或者函数”。

每个对象都是基于一个引用类型创建的，可以是原生的 Object 也可以是用户自定义的类型。

var person = {

name: “Nicholas”,

age: 29,

job: “Software Engineer”,

sayName: function(){

alert(this.name)

}

};

ECMAScript 中有两种属性：数据属性和访问器属性。

1. 数据属性

Configurable：能否通过 delete 删除属性，能否修改属性的特性。默认为 true 。

Enumerable： 能否用 for – in 循环返回属性。默认为 true 。

Writable：能否修改属性的值。默认为 true 。

Value：属性的数据值，默认值为 undefined

Object.defineProperty(Person, “name”, {

configurable: true,

writable: false,

Value: “Gray”

});

delete Person.name;

注意：一旦把属性定义为不可配置的，就不能再把它变回可配置了。严格模式下会报错。

在调用 Object.defineProperty() 方法时，如果不指定，configurable, enumerable 和 writable 特性的默认值都是 false。

1. 访问器属性

Configurable: 能否调用 delete 删除， 能否修改属性，默认为 true.

Enumerable: 能否使用 for – in 循环访问属性，默认为 true.

Get: 读取属性时调用的函数， 默认为 undefined.

Set: 写入属性时调用的函数， 默认为 undefined.

var book = {

\_year: 2004,

edition: 1

};

Object.defineProperty(book, “year”, {

get: function(){

return this.\_year;

},

set: function(){

if(newValue > 2004) {

this.\_year = newValue;

this.edition += newValue – 2004;

}

}

});

读取属性的方法： Object.getOwnPropertyDescriptor(book, “\_year”)

1. 创建对象
2. 工厂函数

考虑到 ECMAScript 中无法创建类，开发人员发明了 一种函数，用函数来封装以特定接口创建对象的细节。

function createPerson(name, age, job){

var o = new Object();

o.name = name;

o.age = age;

o.job = job;

o.sayName = function(){

alert(this.name);

};

return o;

}

var person1 = createPerson(“Nicholas”, 29, “Software Engineer”);

优点：解决了创建多个相似对象的问题。

缺点：无法知道对象的类型。

2）构造函数

ECMAScript 中的构造函数可以用来创建特定类型的对象，如 Object，Array 等。因而也可以创建自定义的构造函数。

function Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

this.sayName = function(){

alert(this.name);

};

}

var person1 = new Person(“Nicholas”, 29, “Software Engineer”);

var person2 = new Person(“Grey”, 27, “Doctor”);

要创建 Person 的新实例，必须使用 new 操作符，实际经历4个步骤：

1. **创建一个新对象**
2. **将构造函数的作用域赋给新对象（this 指向新对象）**
3. **执行构造函数中的代码**
4. **返回新对象**

alert(person1.constructor == Person); // true

alert(person1 instanceof Object); // true

alert(person1 instanceof Person); // true

以这种方式定义的构造函数实际是定义在 Global 对象中的，在浏览器中即定义在 window 对象中。

构造函数与其他函数的唯一区别就在于构造函数必须使用 new 操作符调用，否则它就是普通函数。

如：

var o = new Object();

Person.call(o, “Kristen”, 25, “Nurse”);

o.sayName(); // Kristen

call/Apply的区别：call以一个一个的形式传递，apply以数组的形式传递.

构造函数的缺点：每个实例都实例化了一个 sayName 对象，因为 function 本质上也是对象。new Function(“alert(this.name)”)

因此，不同实例上的同名函数是不相等的，但功能却相同。

1. 原型模式

每个函数都有一个 prototype 属性，该属性是一个指针，指向一个对象，该对象的用途是包含可以由特定类型的所有实例共享的属性和方法，可以理解为通过调用构造函数而创建的那个**对象实例的原型对象**。因此不必再在构造函数中定义对象实例的信息，而是直接添加到原型对象中。

function Person(){}

Person.prototype.name = “Nicholas”;

Person.prototype.age = 29;

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

person1.sayName(); // “Nicholas”

var person2 = new Person();

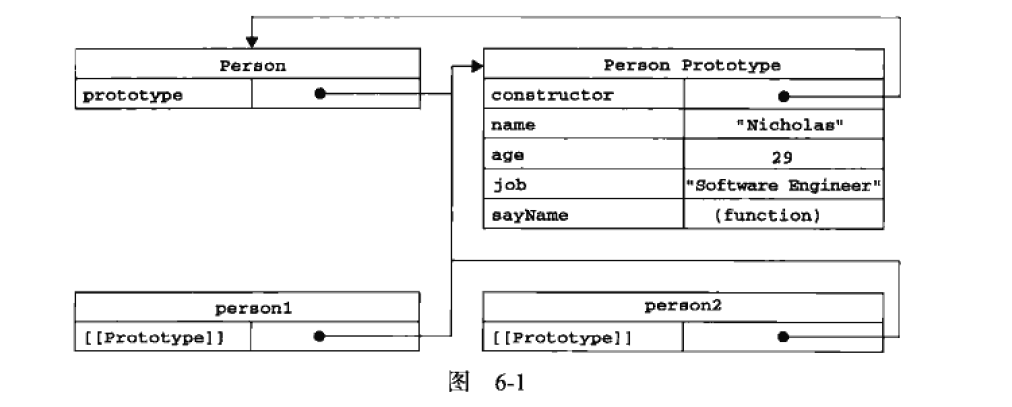
alert(person1.sayName == person2.sayName); //true

属性和方法由所有实例共享！

任何函数都会有一个 prototype 属性，该属性指向函数的原型对象。所有原型对象都会自动获得一个 constructor （构造函数） 属性，这个属性包含一个指向 prototype 属性所在**函数的指针**。

当调用构造函数创建一个新实例后，该实例的内部将包含一个指针，指向构造函数的原型对象。ECMA-262 第 5 版中管这个指针叫 [[Prototype]] 。虽然在脚本中没有标准的方式访问 [Prototype] ，但在 Firefox、Safari 和 Chrome 在每个对象上都支持一个属性 \_proto\_；

注意：这种链接关系存在于实例与构造函数的原型对象之间，而不是存在于实例与构造函数之间。



虽然在所有实现中都无法访问到 [[Prototype]] ，但可以通过 isPrototypeOf() 方法来确定对象之间是否存在这种关系。

如： alert(Person.prototype.isPrototypeOf(person1)); // true

在 ECMAScript 5 中，还可以 Object.getPrototypeOf() ，可以返回 [[Prototype]] 的值，即 alert(Object.getPrototypeOf(person1).name); // “Nicholas”

虽然可以通过对象实例访问保存在原型中的值，但却不能通过对象实例重写原型中的值。如果我们在实例中添加了一个属性，而该属性与原实例原型中的一个属性同名，该属性则会屏蔽原型中的属性，但不会修改原型对象中的属性。使用 delete 可以删除实例中的属性，从而可以继续访问原型对象中的属性。

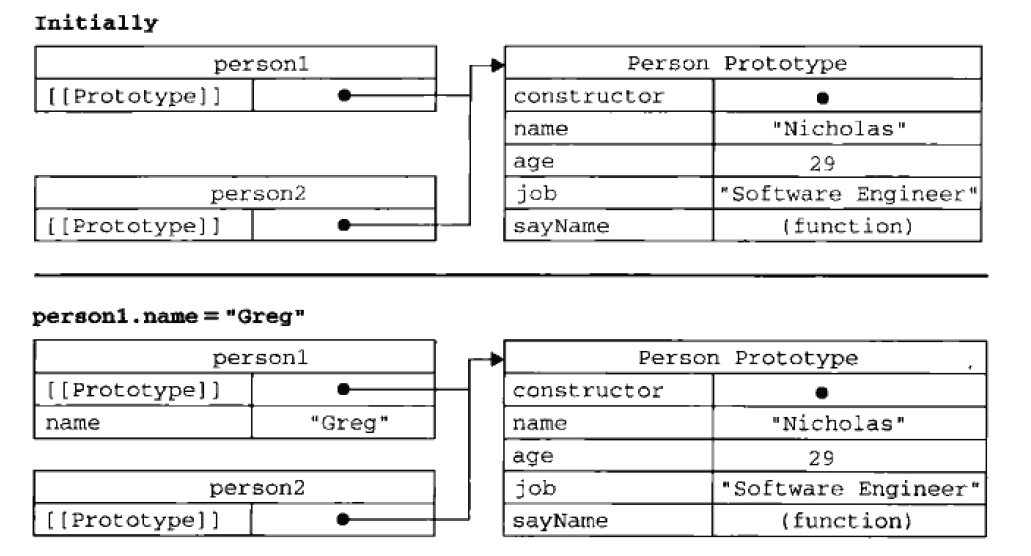
hasOwnProperty() 方法继承自 Object , 在给定属性存在于对象实例中时返回 true ，r**如何**可以检测一个属性是存在于实例中，还是存在于原型中？

例：

alert(person1.hasOwnProperty(“name”)); // false

person1.name = “gray”;

alert(person1.hasOwnProperty(“name”)); // true



更简单的原型语法：

function Person(){}

Person.prototype={

// constructor: Person,

name: “Nicholas”,

age: 29,

sayName: function(){

return this.name;

}

}

将 prototype 设置为以字面量对象形式创建的新对象，区别： constructor 属性**不再指向 Person** 了。这种定义方式本质上完全重写了 prototype 对象，因此 constructor 属性也发生变化，指向 Object 构造函数。

注意： 这种手动赋值的方式会将 constructor 属性的 Enumerable 属性设为 true , 原生的默认值是 false 。 可以尝试 Object.defineProperty() 定义。

原型动态性

var friend = new Person();

Person.prototype.sayHi = function(){

alert(“hi”);

};

friend.sayHi(); //没问题

原生对象的原型

Array.prototype.sort

String.prototype.substring

还可以为原生对象添加新的方法，但最好别这样做。

1. 继承

接口继承和实现继承

ECMAScript 只支持实现继承，而且其实现继承主要是依靠原型链来实现的。

让一个函数的原型对象等于另一个类型的实例，此时，该原型对象将包含一个指向另一个原型的指针，相应地，另一个原型中也包含着一个指向它自己的构造函数的指针。多继承以此类推。

function SuperType(){

this.property = true;

}

SuperType.prototype.getSuperValue = function(){

return this.property;

};

function SubType(){

this.subproperty = false;

}

SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function(){

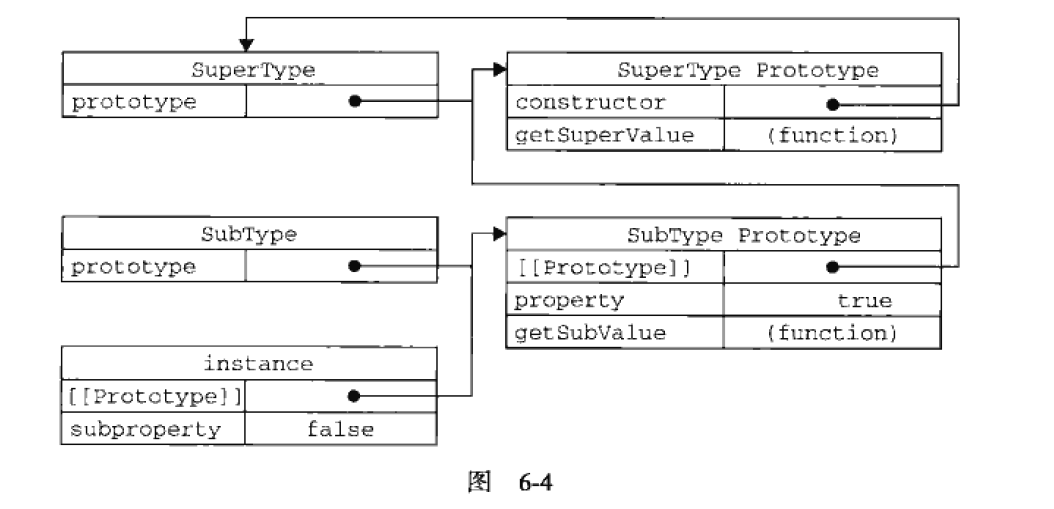
return this.subproperty;

};

var instance = new SubType();

alert(instance.getSuperValue()); // true

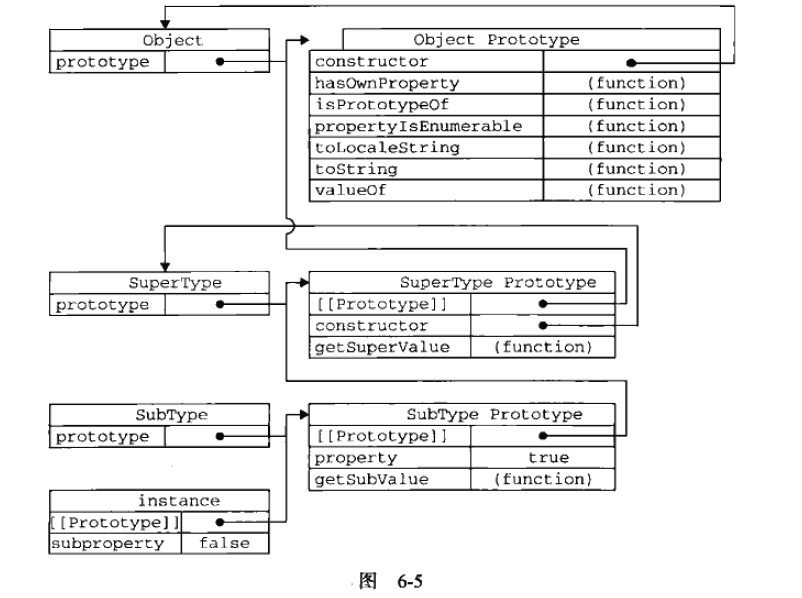
实现的本质是**重写原型对象**



注意：instance.constructor 现在指向的是 SuperType ， 这是因为原来的 SubType 的原型指向了 SuperType 的原型，而 SuperType 的原型对象的 constructor 属性指向的是 SuperType。

原型搜索机制：当以读取模式访问一个实例属性时，首先会在实例中搜索该属性。如果没有找到，则会继续搜索实例的原型。搜索过程会一直沿着原型链向上。

所有引用类型默认都通过原型链的方式继承了 Object ，所有函数的默认原型都是 Object 的实例。因此默认原型都会包含一个内部指针，指向 Object.prototype 。这也正是所有自定义类型都会继承 toString（）、valueOf（） 等默认方法的根本原因。



确定原型和实例的关系

1. Instance Instanceof Supertype
2. SuperType.prototype.isPrototypeOf(instance);

4. JQuery源码初探

（function(window, undefined) {

var

rootjQuery,

location = window.location,

document = window.document,

docElem = document.documentElement,

\_jQuery = window.jQuery,

\_$ = window.$,

**jQuery** = function(selector, context) {

return new jQuery.fn.init(selector, context, rootjQuery);

},

compelted = function(){//…},

detach = function(){//…};

**jQuery.fn = jQuery.prototype** = {

constructor: jQuery,

pushStack: function(elem){

var ret = jQuery.merge(this.constructor(), elems);

        // 给返回的新 jQuery 对象添加属性 prevObject

        // 所以也就是为什么通过 prevObject 能取到上一个合集的引用了

        ret.prevObject = this;

        ret.context = this.context;

        // Return the newly-formed element set

        return ret;

},

end: function(){

return this.prevObject || this.constructor(null);

},

eq: function(i){

 // jQuery 对象集合的长度

        var len = this.length,

            j = +i + (i < 0 ? len : 0);

        // 利用 pushStack 返回

        return this.pushStack(j >= 0 && j < len ? [this[j]] : []);

},

init: function(selector, context, rootjQuery) {

//…

},

selector: “”,

length: 0,

//…

};

**jQuery.fn.init.prototype = jQuery.fn;**

**jQuery.extend = jQuery.fn.extend** = function() {

//扩展合并函数

};

**jQuery.extend({**

noConflict: function(deep){

// 判断全局 $ 变量是否等于 jQuery 变量

// 如果等于，则重新还原全局变量 $ 为 jQuery 运行之前的变量（存储在内部变量 \_$ 中）

            if (window.$ === jQuery) {

                // 此时 jQuery 别名 $ 失效

                window.$ = \_$;

            }

     // 当开启深度冲突处理并且全局变量 jQuery 等于内部 jQuery，则把局部l jQuery 还原成之前的状况

            if (deep && window.jQuery === jQuery) {

                // 如果 deep 为 true，此时 jQuery 失效

                window.jQuery = \_jQuery;

            }

            // 这里返回的是 jQuery 库内部的 jQuery 构造函数（new jQuery.fn.init()）

            // 像使用 $ 一样尽情使用它吧

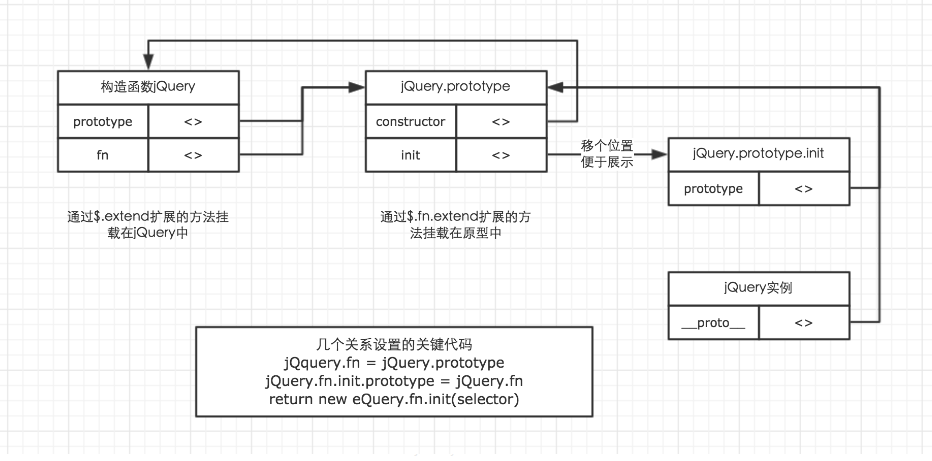
            return jQuery;

},

ready: function(){//…},

});

}）(window);



jQuery: 在封闭空间内定义的一个对象， 指向一个匿名函数

jQuery.fn: jQuery 对象内定义的一个属性，指向 jQuery 原型对象

1）**jQuery.extend(object)** 为扩展 jQuery 类本身，为类添加新的静态方法；

2）**jQuery.fn.extend(object)** 给 jQuery 对象添加实例方法，也就是通过这个 extend 添加的新方法，实例化的 jQuery 对象都能使用，因为它是挂载在 jQuery.fn 上的方法（上文有提到，jQuery.fn = jQuery.prototype ）。

它们的官方解释是：

1）jQuery.extend(): 把两个或者更多的对象合并到第一个当中，

2）jQuery.fn.extend()：把对象挂载到 jQuery 的 prototype 属性，来扩展一个新的 jQuery 实例方法。

也就是说，使用 jQuery.extend() 拓展的**静态方法**，我们可以直接使用 $.xxx 进行调用（xxx是拓展的方法名），

而使用 jQuery.fn.extend() 拓展的**实例方法**，需要使用 $().xxx 调用。

$.trim

$().css();$().attr();等都是通过 jQuery.fn.entend() 扩展的。

参考文献：

<http://www.cnblogs.com/coco1s/p/5261646.html>

<http://www.jianshu.com/p/3f97570d22b4>

<http://www.jianshu.com/p/15ac7393bc1f>