面向对象

取消默认行为

Rerurn false 浏览器默认支持

e.preventDefault(false) w3c支持

概念

对象关注，使用的目标，使用面向对象编程

对象的行为称为方法

对象存储数据的变量称为对象的属性

## 创建对象

// 创建一个 Person 类型的对象, 它有 name, age, 和 gender 属性, 还有一个说话的方法 sayHello

class Person {

// 固定的写法

// constructor 这个单词是 构造器 的 意思

// 在 写代码的时候 使用 var p = new Person()

constructor ( name, age, gender ) {

this.name = name;

this.age = age;

this.gender = gender;

}

sayHello () {

// 打印一句话, 自我介绍

console.log( '你好, 我是 ' + this.name + ', 我今年 ' + this.age + ' 岁了, 我是 ' + this.gender + '.' );

}

}

// 此时还是在定义类, 这个类中定义了 一个 Person 的类型, 同时具有一个 sayHello 的 方法

// 还有 name, age 与 gender 属性

var p = new Person( 'jim', 19, '男' );

## js方法创建对象

1. 字面量

A .var a = { a:1,b:2,...}

B . 动态创建对象 var a = {} a.name = "xiaoming"

C. 关联创建对象

function id ( idName ) {

return document.getElementById( idName );

}

var key = id( 'key' );

var value = id( 'value' );

var btn = id( 'btn' );

var o = {};

btn.onclick = function () {

/ / 给全局对象 o 添加成员

o[ key.value ] = value.value;

}

2) 构建器

var a = new Object()

3）工程函数 （可以重复使用）

4）构造器创建对象

Obj.prototype.tostring

也可以检验数据类型

Window.getComputedStyle 可以获取整个document里的所有style

## 面向对象的基本特征

A抽象性：只考虑对象的目标数据，不考虑对象，可以面向各种对象使用

B封装性：将具体的操作步骤打包

C继承性：表示的是扩展，在原来的基础上添加一些新的元素，获取新的东西 继承原来的元素属性方法

D多态性：一个变量调用一个方法得到不同的结果

## 面向对象编程

就是利用对象编程，如果没有这样的对象，就需要自己去写

## 使用new创建对象（创建对象的过程）

1 先new一个空的对象

2 this绑定/原型绑定

3 执行函数体

4 返回一个对象

## Error对象

1--运行是触发时出发

2--如何处理异常

try { // 尝试

可能出现异常的代码

这个过程如果没有发生异常catch是不会执行的

} catch ( e ) { // exception

如果出现了异常就执行

}

3--异常后的代码不在执行

## 值类型和引用类型

### 数据类型

1> 基本类型( 简单类型, 值类型 ): 数字 number, 字符串 string, 布尔 boolean

2> 复杂类型( 复合类型, 引用类型 ): 对象( 数组, Object, 正则, ... ), 函数

3> 空类型: undefined, null

### 存储模型( 重点中的 重点 )

基本类型的变量存储数据本身, 比如 var num = 123; 在内存中开辟一段内存, 这个内存就是 num, 里面存储的 就是 数字 123

引用类型就不一样了, 因为引用类型内的数据个数可以很大也可以很小. 因此不太容易说清楚要开辟多少内存存储数据. 引用类型的变量只是存储对象的引用( 地址 ), 真正的数据存储在一个单独的内存中

### 深拷贝与浅拷贝

--所谓的浅拷贝就是只拷贝当前对象, 而不拷贝对象中引用的其他对象( 不拷贝所有的 对象 )

--所谓的深拷贝就是将所有涉及的对象全部拷贝一份, 在内存中两个对象是完全独立的

## 原型prototype

我们创建的每个函数都有一个prototype（原型）属性，这个属性是一个指针，指向一个对象，而这个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。如果按照字面意思来理解，那么prototype就是通过调用构造函数而创建的那个对象实例的原型对象。使用原受对象的好处是可以让所有对象实例共享它所包含的属性和方法。换句话说，不必在构造函数中定义对象实例的信息，而是可以将这些信息直接添加到原型对象中，如下面的例子所示

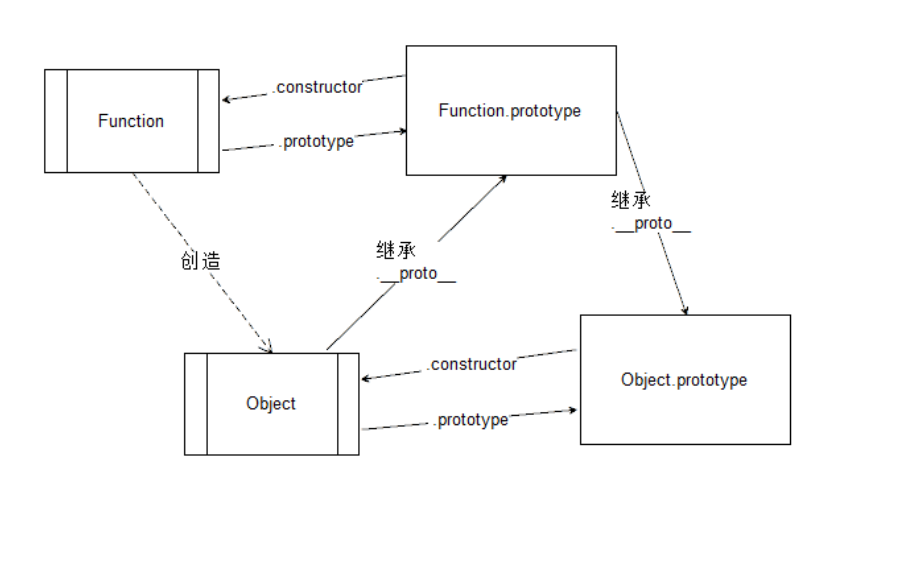
.prototype  .constructor

Function a()---------------->原型----------------------------->Funtion a()

.\_\_proto\_\_ .constructor

New a() -------------------->原型----------------------------->Function a()

New P().\_\_proto\_\_ = P.constructor.prototype



Object.cteare( )

原型继承

一般继承

function Person ( name, age, gender ) {

this.name = name;

this.age = age;

this.gender = gender;

}

Person.prototype.sayHello = function () {

console.log( '你好, 我是 ' + this.name );

};

Person.prototype.run = function () {};

替换原型继承

function Person () {}

var p1 = new Person();

Person.prototype.sayHello = function () {

console.log( '原始原型的 sayHello' );

}

var p2 = new Person();

Person.prototype = {

sayHello: function () {

console.log( '替换对象的 sayHello' );

}

}

var p3 = new Person();

p1.sayHello();

p2.sayHello();

p3.sayHello();

Object.cteare( )

var o = {

sayHello: function () {

console.log( '我是一个带有 sayHello 方法的对象' );

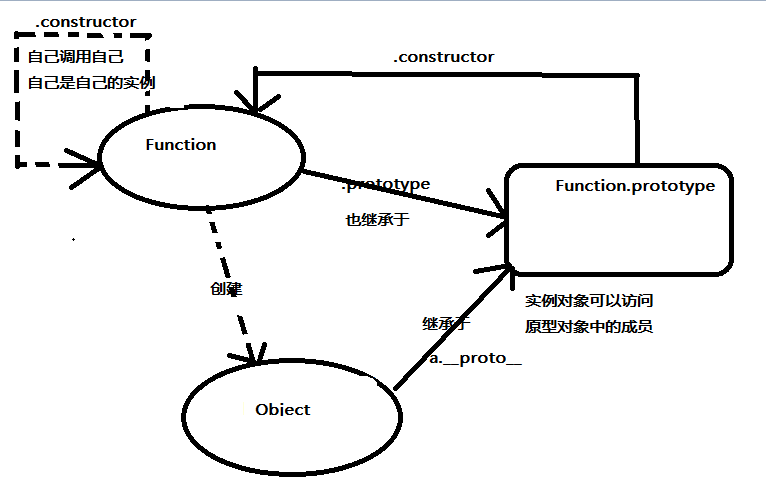
}

}

var newObj = Object.create( o );

// 返回的对象 newObj 的 原型对象 就是 o, 即 newObj 继承自 o

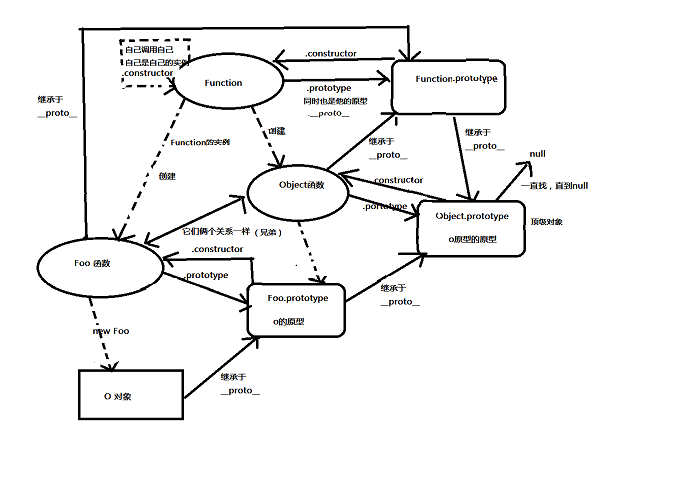
newObj.sayHello();



### 静态成团与实例成员

凡是定以后由实例对象所使用的成员就是实例成员

凡是由构造方法所使用的成员就是静态成员



## 混入和混入继承

混入：一个对象加入另一个对象中，那么就可以完成对象的一个扩展

Min: mix

使用 mix 可以将一个对象混入到另一个对象中

function mix ( obj1, obj2 ) {

for ( var k in obj2 ) {

obj1[ k ] = obj2[ k ];

}

}

Extend

extend

这个单词的含义原意就是扩展, 所谓的混入就是将对象加入到一个对象中, 扩展该对象.

function extend ( obj1, obj2 ) {

for ( var k in obj2 ) {

obj1[ k ] = obj2[ k ];

}

}

混合继承：也就是多种继承方式用到一起

## 将字符串作为代码执行

eval 函数

该函数可以将一个字符串作为一段 js 代码来执行

eval( 'var num = 123;' ); // 就好像此时代码中写了 var num = 123 一样

Function 函数

语法 var a = new Function("参数1","参数2","return 函数内容"）

HMWEB01@bj.itcast.cn

## Instanceof运算符

用来判断某一个对象是否为某个构造函数的实例

判断一个的对象的原型链中是否含有这个构造函数的prototype所表示的对象

判断函数prototype在不在对象\_\_proto\_\_的原型链为上 在为true 不在 false

function Func() {}

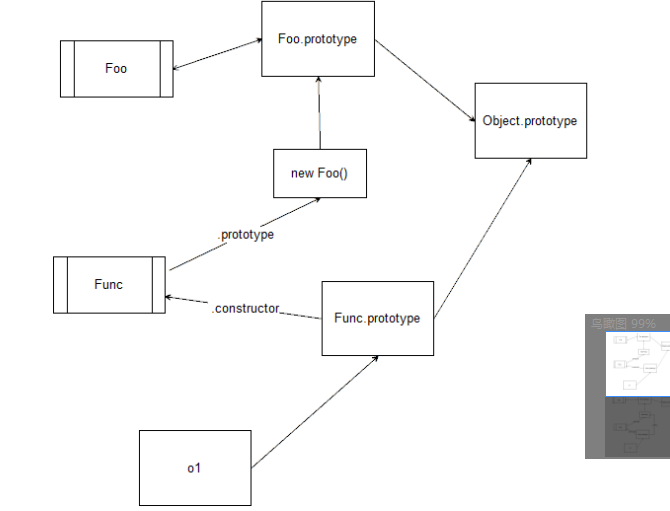
var o1 = new Func();

function Foo() {}

Func.prototype = new Foo();

var o2 = new Func();

console.log( o1 instanceof Func ); //false



// Func.prototype 在 o1 所表示的原型链上

// 想一个办法 让 newFoo 在 o1 的链上

// 原始的 Func.prototype 就是 o1.\_\_proto\_\_

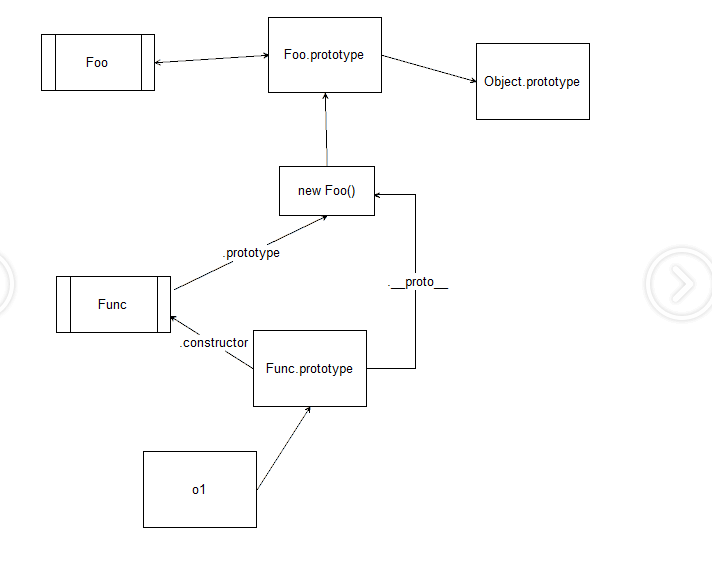
// 将 newFoo 放在 o1.\_\_proto.\_\_ 的原型中

// 找到前面的 newFoo, 注意不能再 new 一个新的

// 原来的那一个就是 Func.prototype

o1.\_\_proto\_\_.\_\_proto\_\_ = Func.prototype;

console.log( o1 instanceof Func ); //true



## 块级作用域

一个变量的作用范围，就是{ }，（）的范围，从变量定义开始，到变量所在的语句块结束，在这样一个范围内可以被作用

Const和let 与 var 一样，但let具有块级作用域，在块级作用域中，子块可以访问父级，父级不能访问子级

{

let num = '一个数字';

{

let str = '子块可以访问父块';

console.log( num ); // 子块访问父块

}

console.log( str ); //父级不能访问子级

}

{

let num = 123;

{ // 在此处, 父块的变量就会被隐藏起来, 那么在这里先打印 就会抱一个错误, 就是未定义

// console.log( num );

let num = 456;

console.log( num ); 456

num = 789;

console.log( num ); 789

}

// 由于是隐藏这个变量, 因此出来以后就可以使用原来的变量了

console.log( num ); 123

}

Var不受{}限制，只有全局作用域和函数作用域

## 词法作用域

// 全局作用域, 0 级作用域

function foo() {

// 子级作用域, 1 级作用域

}

// 0 级作用域

function func() {

// 子级作用域, 1 级作用域

function func1() {

// 2 级作用域

}

function func2() {

// 2 级作用域

function func3() {

// 3 级作用域

}

}

}

// 0 级作用域

## 预解析

1 先要检查基本语法，看看有没有语法错误

2 检查所有变量声明，并提升

3 所有函数声明提升function a（）{ } 函数在（）中无法调用

## 闭包

所谓的闭包就是 函数. 即在函数中定义的变量无法在外界直接被访问. 而函数具有包裹的作用. 因此函数就是闭包.

特点：闭包可以保存函数内部被销毁的变量的值类型（闭包允许函数带有一个缓存结构，也就是只有该函数可以使用的一个私有的独立的内部空间），也可以被函数外部调用；

避免全局污染（沙箱模式）

注意：当没有用的包被遗弃，就是泄露

优缺点

1 由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。

2 闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

## 函数的四中调用模式（this的理解）

函数调用模式

分辨: 函数调用没有任何引导数据, 就只有一个函数名来调用 foo ();

在函数调用模式中 this 表示 全局对象( 在浏览器中 就是 window )

方法调用模式

分辨: 方法调用模式一定要有一个对象, 是对象引导方法来调用的.

在方法模式中 this 表示引导对象( 当前对象 )

构造器调用模式

分辨: 方法在调用的时候 前面有一个 new 关键字引导.

在方法中 this 的含义就是刚刚被new创建出来的 对象

上下文调用模式

允许调用者控制函数中 this 的含义4

上下文, 就是环境, 依托的对象.Call和apply方法

bind 模式

bind 就是在调用之前指定 this 的含义. 提前绑定 函数.bind( 对象 )

## Call和apply方法

Call方法：调用一个对象的一个方法，以另一个对象替换当前对象

如： thisObj本没有slice方法，通过call来代替另一个对象执行函数

[ ].slice.call(thisObj, obj1, obj2...)

This指向 slice的参数

Apply方法：应用某一个对象的一个方法，用另一个替换当前对象

就是参数 不一样

如： [ ].push.(thisObj, [ obj1, obj2...])