**1.单例模式（Singleton）**

Singleton模式要求一个类只有一个实例，且提供了一个全局的访问点。单例模式分三种：懒汉式单例、饿汉式单例、登记式单例。

单例模式特点：

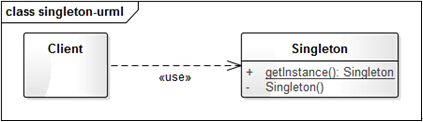
1.单例模式只能有一个实例；

2.单例类必须自己创建自己的唯一实例；

3.单例类必须给所有其他对象提供这一实例。

因为客户程序在调用某一个类时，是不会考虑这个类是否只能有一个实例的，所以这应该是类设计者的责任而不是类使用者的责任。

UML图：



单例模式的经典例子：

public class Singleton {

private static Singleton uniqueInstance = null;

private Singleton() {

// Exists only to defeat instantiation.

}

public static Singleton getInstance() {

if (uniqueInstance == null) {

uniqueInstance = new Singleton();

}

return uniqueInstance;

}

// Other methods...

1.构造方法为private；2.返回一个类的实例。

单例模式的应用场景一般发生在以下条件：

1.资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如日志文件，应用配置；

2.控制资源的情况下，方便资源之间相互通信。如线程池。

应用：Windows的任务管理器，回收站；应用程序的应用日志；数据库连接池；多线程的线程池；操作系统的文件系统。

**2.简单工厂模式（Simple Factory）**

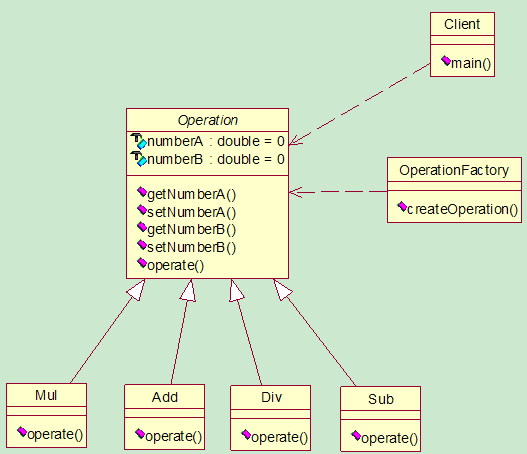
简单工厂模式根据提供给它的数据返回一个类的实例。通常它返回的类都有一个公共的父类（或者接口对象）。

简单工厂模式的作用是实例化对象，而不需要这个对象属于哪个具体的子类。简单工厂模式实例化的类具有相同的接口或者鸡类，在子类比较固定并不需要扩展时，可以使用简单工厂模式。

优点：可以使用户根据参数获得对应类的实例，避免类的直接实例化，降低了耦合性；

缺点：可实例化的类在编译期间已经确定，如果新增加类型，需要修改工厂，不符合OCP（开闭原则）。简单工厂模式需要知道所有要生成的类型，当子类过多或者子类层次过多时不适合使用。

UML图：



Java实现：

Operation类

public abstract class Operation {

double numberA = 0;

double numberB = 0;

public double getNumberA() {

return numberA;

}

public void setNumberA(double numberA) {

this.numberA = numberA;

}

public double getNumberB() {

return numberB;

}

public void setNumberB(double numberB) {

this.numberB = numberB;

}

public abstract double operate();

}

加类

public class Add extends Operation{

@Override

public double operate() {

return numberA + numberB;

}

}

减类

public class Sub extends Operation{

@Override

public double operate(){

return numberA - numberB;

}

}

乘类

public class Mul extends Operation{

@Override

public double operate(){

return numberA \* numberB;

}

}

除类

public class Div extends Operation{

@Override

public double operate(){

return numberA / numberB;

}

}

工厂类

public class OperationFactory {

public static Operation createOperation(char operate) {

Operation operation = null;

switch (operate) {

case '+':

operation = new Add();

break;

case '-':

operation = new Sub();

break;

case '\*':

operation = new Mul();

break;

case '/':

operation = new Div();

break;

default:

break;

}

return operation;

}

}

测试类

public class Test {

public static void main(String[] args) {

Operation operation = OperationFactory.createOperation('\*');

operation.setNumberA(3);

operation.setNumberA(4);

System.out.println(operation.operate());

}

}

**1.立即调用表达式**

(function(window, undefined) {

var jQuery = function() {}

// ...

window.jQuery = window.$ = jQuery;

})(window);

1）window和undefined都是为了减少变量查找所经过的scope作用域。当window通过传递给闭包内部之后，在闭包内部使用它的时候，可以把它当成一个局部变量，显然比原先在window scope下查找的时候要快一些。

2）undefined也是同样的道理，其实这个undefined并不是JavaScript数据类型的undefined，而是一个普普通通的变量名。只是因为没给它传递值，它的值就是undefined，undefined并不是JavaScript的保留字。

jQuery使用()将匿名函数括起来，然后后面再加一对小括号（包含参数列表），那么这小括号能把我们的表达式组合分块，并且每一块（也就是每一对小括号），都有一个返回值。这个返回值实际上也就是小括号中表达式的返回值。所以，当我们用一对小括号把匿名函数括起来的时候，实际上小括号返回的，就是一个匿名函数的Function对象。因此，小括号对加上匿名函数就如同有名字的函数般被我们取得它的引用位置了。所以如果在这个引用变量后面再加上参数列表，就会实现普通函数的调用形式。

这样写的好处：全局变量是魔鬼, 匿名函数可以有效的保证在页面上写入JavaScript，而不会造成全局变量的污染，通过小括号，让其加载的时候立即初始化，这样就形成了一个单例模式的效果从而只会执行一次。

**2.类数组对象**

两个条件：

它们都有一个合法的 length 属性(0到2^32-1之间的正整数)。

length属性的值大于它们的最大索引(index)。

jQuery内部采用了“类数组对象”的方式作为存储结构，所以我们既可以像对象一样处理jQuery操作，也能像数组一样可以使用push、pop、shift、unshift、sort、each、map等类数组的方法操作jQuery对象了。

**3.jQuery中ready与load事件**

DOM文档加载步骤：

1）解析HTML结构；

2）加载外部脚本和样式表文件；

3）解析并执行脚本代码；

4）构造HTML DOM模型；//ready事件

5）加载图片等外部文件；

6）页面加载完毕。//load

<script type="text/javascript">

show('观察脚本加载的顺序')

document.addEventListener("DOMContentLoaded", function() {

show('DOMContentLoaded回调')

}, false);

window.addEventListener("load", function() {

show('load事件回调')

}, false);

show('脚本解析一')

//测试加载

$(function(){

show('脚本解析二')

})

show('脚本解析三')

</script>

观察脚本加载的顺序

脚本解析一

脚本解析三

脚本解析二

DOMContentLoaded回调

load事件回调

**4.jQuery多库共存处理（即无冲突处理）**

在使用jQuery时会遇到两种情况：1.jQuery采用$作为命名空间，不免会与别的库框架或者插件相冲突；2.jQuery版本更新太快，插件跟不上，导致不同版本对插件支持度不一样。

出于以上原因，jQuery给出了解决方案：noConflict函数。

引入jQuery运行这个noConflict函数将变量$的控制权让给第一个实现它的那个库，确保jQuery不会与其他库的$对象发生冲突。在运行这个函数后，就只能使用jQuery变量访问jQuery对象。例如，在要用到$("aaron")的地方，就必须换成jQuery("aaron")，因为$的控制权已经让出去了。

jQuery.noConflict();

// 使用 jQuery

jQuery("aaron").show();

// 使用其他库的 $()

$("aaron").style.display = ‘block’;

这个函数必须在你导入jQuery文件之后，并且在导入另一个导致冲突的库之前使用。当然也应当在其他冲突的库被使用之前，除非jQuery是最后一个导入的。

Var \_jQuery = window.jQuery,

\_$ = window.$;

jQuery.noConflict = function( deep ) {

if ( window.$ === jQuery ) {

window.$ = \_$;

}

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

window.jQuery = \_jQuery;

}

return jQuery;

};

如果我们需要同时使用jQuery和其他JavaScript库，我们可以使用$.noConflict()把$的控制权交给其他库。旧引用的$ 被保存在jQuery的初始化; noConflict() 简单的恢复它们。

通过类似swap交换的概念，先把之前的存在的命名空间给缓存起来，通过对比当前的命名空间达到交换的目的，首先，我们先判断下当前的的$空间是不是被jQuery接管了，如果是则让出控制权给之前的\_$引用的库，如果传入deep为true的话等于是把jQuery的控制权也让出去了。

**5.对象的构建**

在JavaScript世界中函数作为“一等公民”，它不仅拥有一切传统函数的使用方式（声明和调用），而且可以做到像简单值一样赋值、传参、返回，这样的函数也称之为第一级函数。不仅如此，而且还可以通过操作符new来充当类的构造器。

函数在充当类的构造器时，原型prototype是一个重要的概念。prototype是构造函数的一个属性, 该属性指向一个对象。而这个对象将作为该构造函数所创建的所有实例的基引用(base reference), 可以把对象的基引用想像成一个自动创建的隐藏属性。 当访问对象的一个属性时, 首先查找对象本身, 找到则返回；若不, 则查找基引用指向的对象的属性(如果还找不到实际上还会沿着原型链向上查找, 直至到根)。 只要没有被覆盖的话, 对象原型的属性就能在所有的实例中找到。

类一：

function ajQuery() {

this.name = 'jQuery';

this.sayName = function(){

return this.name

}

var a = new ajQuery()

var b = new ajQuery()

var c = new ajQuery()

类二:

function ajQuery() {

this.name = 'jQuery'

}

ajQuery.prototype = {

sayName: function() {

return this.name

}

}

var a = new ajQuery()

var b = new ajQuery()

var c = new ajQuery()

类一与类二产生的结构几乎是一样的，而本质区别就是：类二new产生的a、b、c三个实例对象共享了原型的sayName方法，这样的好处节省了内存空间，类一则是要为每一个实例复制sayName方法，每个方法属性都占用一定的内存的空间，所以如果把所有属性方法都声明在构造函数中，就会无形的增大很多开销，这些实例化的对象的属性一模一样，都是对this的引用来处理。除此之外类一的所有方法都是拷贝到当前实例对象上。类二则是要通过scope连接到原型链上查找，这样就无形之中要多一层作用域链的查找了。

jQuery对象的构建如果在性能上考虑，所以就必须采用原型式的结构：

// jQuery的写法

var $jQuery = function(selector, context) {

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

}

$jQuery.fn = $jQuery.prototype = {

init: function() {

this.name = 'aaron'

return this;

},

constructor: $jQuery

}

var $a = $jQuery();

show('$jQuery的调用')

show($a);

使用原型结构，性能上是得到了优化，但是ajQuery类这个结构与目标jQuery的结构的还是有很大不一致：

☑ 没有采用new操作符；

☑ return返回的是一个通过new出来的的对象 。

**6.分离构造器**

通过new操作符构建一个对象，一般经过四步：

A.创建一个新对象

B.将构造函数的作用域赋给新对象（所以this就指向了这个新对象）

C.执行构造函数中的代码

D.返回这个新对象

最后一点就说明了，我们只要返回一个新对象即可。其实new操作符主要是把原型链跟实例的this关联起来，这才是最关键的一点，所以我们如果需要原型链就必须要new操作符来进行处理。否则this则变成window对象了。

我们来剖析下jQuery的这个结构,以下是我们常见的类式写法:

var $$ = ajQuery = function(selector) {

this.selector = selector;

return this

}

ajQuery.fn = ajQuery.prototype = {

selectorName: function(){

return this.selector;

},

constructor: ajQuery

}

var a = new $$('aaa'); //实例化

a.selectorName() //aaa //得到选择器名字

首先改造jQuery无new的格式，我们可以通过instanceof判断this是否为当前实例：

var $$ = ajQuery = function(selector) {

if(!(this instanceof ajQuery)){

return new ajQuery(selector);

}

this.selector = selector;

return this

}

jQuery为了避免出现这种死循环的问题，采取的手段是把原型上的一个init方法作为构造器

var $$ = ajQuery = function(selector) {

//把原型上的init作为构造器

return new ajQuery.fn.init( selector );

}

ajQuery.fn = ajQuery.prototype = {

name: 'aaron',

init: function() {

console.log(this)

},

constructor: ajQuery

}

这样确实解决了循环递归的问题，但是又问题来了，init是ajQuery原型上作为构造器的一个方法，那么其this就不是ajQuery了，所以this就完全引用不到ajQuery的原型了，所以这里通过new把init方法与ajQuery给分离成2个独立的构造器。

补充知识点：静态方法与实例方法

静态方法，属于类的方法，即类可以直接调用的方法。为类所有实例化对象所共用（但不能用实例对象之间调用），所以静态成员只在内存中占一块区域；

实例方法，属于实例化类后对象的方法，即实例对象调用的方法。每创建一个类的实例，都会在内存中为非静态成员分配一块存储；

静态方法在一启动时就实例化了，因而静态内存是连续的，且静态内存是有限制的；而非静态方法是在程序运行中生成内存的，申请的是离散的空间。

public class HasStaticMethod {

public static void callMe() {

System.out.println("This is a static method.");

}

public void callYou() {

System.out.println("This is a instance method.");

}

}

public class InvokeMethod {

public static void main(String[] args) {

//调用静态方法有两种方式：不创建对象直接调用，利用对象调用

HasStaticMethod.callMe();

HasStaticMethod hasStaticMethod = new HasStaticMethod();

hasStaticMethod.callMe();

//调用非静态方法只能通过创建对象来调用

hasStaticMethod.callYou();

}

}

静态方法只能访问静态成员，实例方法可以访问静态和实例成员。之所以不允许静态方法访问实例成员变量，是因为实例成员变量是属于某个对象的，而静态方法在执行时，并不一定存在对象。同样，因为实例方法可以访问实例成员变量，如果允许静态方法调用实例方法，将间接地允许它使用实例成员变量，所以它也不能调用实例方法。基于同样的道理，静态方法中也不能使用关键字this。

main()方法是一个典型的静态方法，它同样遵循一般静态方法的规则，所以它可以由系统在创建对象之前就调用。

总结：大家对这个问题都有一个共识：那就是实例化方法更多被使用和稳妥，静态方法少使用。

有时候我们对静态方法和实例化方法会有一些误解：

1、大家都以为“ 静态方法常驻内存，实例方法不是，所以静态方法效率高但占内存。”

事实上，他们都是一样的，在加载时机和占用内存上，静态方法和实例方法是一样的，在类型第一次被使用时加载。调用的速度基本上没有差别。

2、大家都以为“ 静态方法在堆上分配内存，实例方法在堆栈上”

事实上所有的方法都不可能在堆或者堆栈上分配内存，方法作为代码是被加载到特殊的代码内存区域，这个内存区域是不可写的。

方法占不占用更多内存，和它是不是static没什么关系。

因为字段是用来存储每个实例对象的信息的，所以字段会占有内存，并且因为每个实例对象的状态都不一致（至少不能认为它们是一致的），所以每个实例对象的所有字段都会在内存中有一分拷贝，也因为这样你才能用它们来区分你现在操作的是哪个对象。

但方法不一样，不论有多少个实例对象，它的方法的代码都是一样的，所以只要有一份代码就够了。因此无论是static还是non-static的方法，都只存在一份代码，也就是只占用一份内存空间。

同样的代码，为什么运行起来表现却不一样？这就依赖于方法所用的数据了。主要有两种数据来源，一种就是通过方法的参数传进来，另一种就是使用class的成员变量的值……

3、大家都以为“实例方法需要先创建实例才可以调用，比较麻烦，静态方法不用，比较简单”

事实上如果一个方法与他所在类的实例对象无关，那么它就应该是静态的，而不应该把它写成实例方法。所以所有的实例方法都与实例有关，既然与实例有关，那么创建实例就是必然的步骤，没有麻烦简单一说。

当然你完全可以把所有的实例方法都写成静态的，将实例作为参数传入即可，一般情况下可能不会出什么问题。

从面向对象的角度上来说，在抉择使用实例化方法或静态方法时，应该根据是否该方法和实例化对象具有逻辑上的相关性，如果是就应该使用实例化对象 反之使用静态方法。这只是从面向对象角度上来说的。

如果从线程安全、性能、兼容性上来看 也是选用实例化方法为宜。

我们为什么要把方法区分为：静态方法和实例化方法 ？

如果我们继续深入研究的话，就要脱离技术谈理论了。早期的结构化编程，几乎所有的方法都是“静态方法”，引入实例化方法概念是面向对象概念出现以后的事情了，区分静态方法和实例化方法不能单单从性能上去理解，创建c++,java,c#这样面向对象语言的大师引入实例化方法一定不是要解决什么性能、内存的问题，而是为了让开发更加模式化、面向对象化。这样说的话，静态方法和实例化方式的区分是为了解决模式的问题。

拿别人一个例子说事：

比如说“人”这个类，每个人都有姓名、年龄、性别、身高等，这些属性就应该是非静态的，因为每个人都的这些属性都不相同；但人在生物学上属于哪个门哪个纲哪个目等，这个属性是属于整个人类，所以就应该是静态的——它不依赖与某个特定的人，不会有某个人是“脊椎动物门哺乳动物纲灵长目”而某个人却是“偶蹄目”的。

**7.静态方法与实例方法共享**

function A(){

}

A.staticMethof = function(){

alert('静态方法');

}

A.prototype.instaceMethod = function(){

alert('实例方法');

}

A.staticMethof(); //类A直接调用

var instace = new A();

instace.instaceMethod();//A的实例对象instace调用

拿jQuery框架来看，它的方法都是实例方法，它的工具函数（提供的接口）都是静态方法。静态方法$.each(); 实例方法$('body').each();

我们来看看jQuery的源码：

jQuery.prototype = {

each: function( callback, args ) {

return jQuery.each( this, callback, args );

}

}

实例方法取于静态方法，即实例方法与静态方法共享设计。静态方法挂在jQuery构造器上，原型方法挂在哪里呢？

我们上节不是讲了内部会划分一个新的构造器init吗？jQuery通过new原型prototype上的init方法当作构造器，那么init的原型链方法就是实例的方法了，所以jQuery通过2个构造器划分2种不同的调用方式一种是静态，一种是原型。

方法是共享的，并且实例方法取于静态方法，2个构造器是完全隔离的,这个要如何处理？

看看jQuery给的方案：

画龙点睛的一处init.prototype = jQuery.fn，把jQuery.prototype原型的引用赋给jQuery.fn.init.prototype的原型，这样就把2个构造器的原型给关联起来了。

ajQuery.fn = ajQuery.prototype = {

name: 'aaron',

init: function(selector) {

this.selector = selector;

return this;

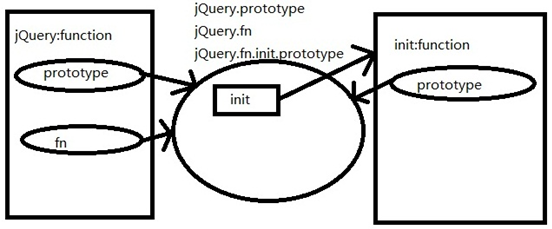
},

constructor: ajQuery

}

ajQuery.fn.init.prototype = ajQuery.fn

init的构造图：



通过原型传递解决问题，把jQuery的原型传递给jQuery.prototype.init.prototype。换句话说jQuery的原型对象覆盖了init构造器的原型对象，因为是引用传递所以不需要担心这个循环引用的性能问题。

补充知识点：理解jQuery的$.extend()和$.fn.extend()

jQuery为开发插件提供；了两个方法，分别是$.extend()和$.fn.extend()。

jQuery.fn = jQuery.prototype =

{

init: function( selector, context ) {//…. //…… };

}

$.fn是指jquery的命名空间，加上fn上的方法及属性，会对jquery实例每一个有效。如扩展$.fn.abc(),即$.fn.abc()是对jquery扩展了一个abc方法,那么后面你的每一个jquery实例都可以引用这个方法了。

可以把jQuery看做一个封装好的类。

$.extend()为jQuery添加类方法，可以理解为添加静态方法：

jQuery.extend({

min: function(a, b) { return a < b ? a : b; },

max: function(a, b) { return a > b ? a : b; }

});

jQuery.min(2,3); // 2

jQuery.max(4,5); // 5

$.fn.extend()对jQuery.prototype进行扩展，就是为jQuery类添加成员函数（实例方法），jQuery类的实例可以使用这个成员函数。

$.fn.extend({

alertWhileClick:function() {

$(this).click(function(){

alert($(this).val());

});

}

});

$("#input1").alertWhileClick();

jQuery.extend() 的调用并不会把方法扩展到对象的实例上，引用它的方法也需要通过jQuery类来实现，如jQuery.init()，而 jQuery.fn.extend()的调用把方法扩展到了对象的prototype上，所以实例化一个jQuery对象的时候，它就具有了这些方法。

直接写在类里面的方法是静态方法，加在prototype上的方法是实例方法。（？）

**面向对象编程（OOP）**

类：具有相同的属性和功能的对象的抽象的集合。

对象：是一个自包含的实体，用一组可识别的特性和行为来标识。

实例就是一个真实的对象，实例化就是创建对象的过程，使用new关键字来创建。

构造方法：又叫构造函数，其实就是对类进行初始化。构造方法与类同名，无返回值，也不需要void关键字，在new的时候调用。所有类都有构造方法，不编码则系统默认生成空的构造方法，如果定义了构造方法，则默认的构造方法失效。

方法重载：提供了创建同名的多个方法的能力，但这些方法需要使用不同的参数类型。方法重载可以在不改变原方法的基础上添加新功能。

字段：是一个成员，它表示与类或对象相关联的变量。

属性：是一个或一对方法，但在调用它的代码看来，它是一个字段，即属性适合以字段的方式使用方法调用的场合。

private int num = 3;

public int num {

get {

return num;

}

set {

num = value;

}

}

属性有两个方法get和set。get访问器返回与生命的属性相同的数据类型，表示的意思是调用时可以得到内部字段的值或是引用；set访问器没有显式设置参数，但它有一个隐式参数，用关键字value表示，它的作用是调用属性时可以给内部的字段或引用赋值。

**OOP三大基本特性，五大基本原则：**

三大特性是：封装,继承,多态

所谓封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。封装是面向对象的特征之一，是对象和类概念的主要特性。 简单的说，一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

所谓继承是指可以让某个类型的对象获得另一个类型的对象的属性的方法。它支持按级分类的概念。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。 通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用基类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力。

如果子类继承父类，1.子类拥有父类非private的属性和功能；2.子类具有自己的属性和功能，即子类可以扩展父类没有的属性和功能；3.子类还可以以自己的方式实现父类的功能（方法重写）。

所谓多态就是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

五大基本原则

单一职责原则SRP(Single Responsibility Principle)

是指一个类的功能要单一，不能包罗万象。如同一个人一样，分配的工作不能太多，否则一天到晚虽然忙忙碌碌的，但效率却高不起来。

开放封闭原则OCP(Open－Close Principle)

一个模块在扩展性方面应该是开放的而在更改性方面应该是封闭的。比如：一个网络模块，原来只服务端功能，而现在要加入客户端功能，

那么应当在不用修改服务端功能代码的前提下，就能够增加客户端功能的实现代码，这要求在设计之初，就应当将服务端和客户端分开，公共部分抽象出来。

替换原则(the Liskov Substitution Principle LSP)

子类应当可以替换父类并出现在父类能够出现的任何地方。比如：公司搞年度晚会，所有员工可以参加抽奖，那么不管是老员工还是新员工，

也不管是总部员工还是外派员工，都应当可以参加抽奖，否则这公司就不和谐了。

依赖原则(the Dependency Inversion Principle DIP) 具体依赖抽象，上层依赖下层。假设B是较A低的模块，但B需要使用到A的功能，

这个时候，B不应当直接使用A中的具体类： 而应当由B定义一抽象接口，并由A来实现这个抽象接口，B只使用这个抽象接口：这样就达到

了依赖倒置的目的，B也解除了对A的依赖，反过来是A依赖于B定义的抽象接口。通过上层模块难以避免依赖下层模块，假如B也直接依赖A的实现，那么就可能造成循环依赖。一个常见的问题就是编译A模块时需要直接包含到B模块的cpp文件，而编译B时同样要直接包含到A的cpp文件。

接口分离原则(the Interface Segregation Principle ISP)

模块间要通过抽象接口隔离开，而不是通过具体的类强耦合起来

**虚函数、虚函数表、多态**

虚函数

虚函数其实是C++中的讲法，就是说将一个函数实现声明为虚函数，这样就为子类提供了一个默认的实现，子类可以选择进行重新实现，或者不进行实现。这样对应到java里，其实就是java里的普通函数。虚函数是可以由子类进行无限覆盖的函数，另外自己也可以进行实现，而java的普通成员函数就可以做到，C++的普通函数就不行，否则就报重复定义error了，必须得加上virsual关键字声明。Java中其实没有虚函数的概念，它的普通函数就相当于C++的虚函数，动态绑定是Java的默认行为。如果Java中不希望某个函数具有虚函数特性，可以加上final关键字变成非虚函数

纯虚函数

也是C++的讲法，与虚函数的区别就是声明的地方不能有实现，完全交给子类，相当于定义保留了一个接口。

C++中纯虚函数形式为：virtual void print() = 0;

对应到java中，就是抽象方法的概念，也就是abstract声明的函数。

JAVA中抽象方法定义形式：abstract void print();

抽象类

C++与JAVA中都有这个概念，在C++中，只要一个类中包含了一个纯虚函数，那么这个类就是一个抽象类;而在JAVA中，如果一个类中包含有一个抽象方法，由于JAVA有规定，这种情况下，这个类必须要用abstract来修饰，也就是一个抽象类了。

C++与JAVA中的抽象类有个共同点，就是抽象类不能直接拿来实例化，因为其中有抽象的函数，只能通过实现了这些函数的子类实例化。

接口

C++中没有独立的接口的概念，其实就是成员函数全部为纯虚函数的类，子类必须对他们进行全部实现。

JAVA中有独立的接口概念，就是interface修饰的类，跟C++中的纯虚基类换汤不换药，代码看起来样式不一样而已，另外，java中的接口在声明的时候必须都定义为public的。

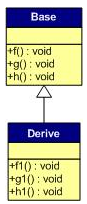
虚函数表（V-Table）

一个类的虚函数地址表，解决了继承、覆盖问题。在有虚函数的类的实例中这个表被分配到了这个实例的内存中，所以，当我们用父类的指针来操作一个子类的时候，这张表就像一个地图一样，指明了实际应该调用的函数。

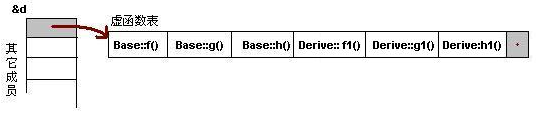
C++的编译器应该是保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证取到虚函数表的有最高的性能——如果有多层继承或是多重继承的情况下）。 这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。

**一般继承（无虚函数覆盖）**

继承关系：

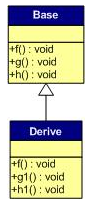


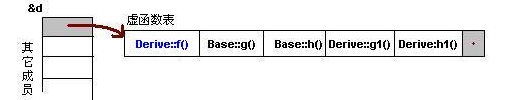
在这个继承关系中，子类没有重载任何父类的函数。在派生类的实例中，其虚函数表如图所示：



1）.虚函数按照其声明顺序放于表中；2）.父类的虚函数在子类虚函数前面。

**2.一般继承（有虚函数覆盖）**

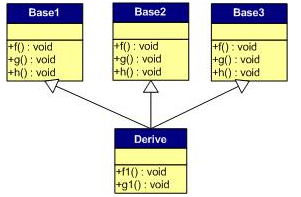


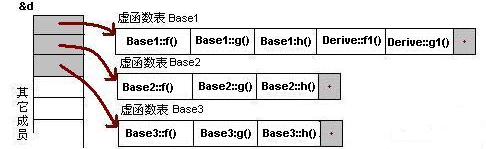


1）覆盖的f()被放到了虚函数表中原来父类虚函数的位置；2）其余函数不变。

由b所指的内存中的虚函数表的f()的位置已经被Derive::f()函数地址所取代，于是在实际调用发生时，是Derive::f()被调用了。这就实现了多态。

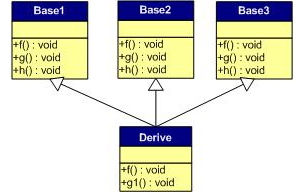
**3.多重继承（无虚函数覆盖）**





1）每个父类都有自己的虚函数表；2）子类的成员被放到了第一个父类表中（所谓第一个父类是按照声明顺序来判断的）。

**4.多重继承（有虚函数覆盖）**





三个父类虚函数表中的f()的位置被替换成了子类的函数指针。这样，我们就可以任一静态类型的父类来指向子类，并调用子类的f()了。

**多态**

父类型的引用可以指向子类型的对象。

多态的原理是：当方法被调用时，无论对象是否被转换成其父类，都只有位于对象继承链最末端的方法实现会被调用。也就是说，虚函数是按照其运行时类型而非其编译时类型动态绑定调用的。

多态的三个条件：要有继承；要有重写；父类引用指向子类对象。

实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

多态的作用：消除类型之间强耦合关系。

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再去调用子类的该同名方法。

如果想要调用子类中有而父类中没有的方法，需要进行强制类型转换。因为当用父类的引用指向子类的对象，用父类引用调用方法时，找不到父类中不存在的方法。这时候需要进行向下的类型转换，将父类引用转换为子类引用。

**多态示例代码**

public class PolyTest

{

public static void main(String[] args)

{

//向上类型转换

Cat cat = new Cat();

Animal animal = cat;

animal.sing();

//向下类型转换

Animal a = new Cat();

Cat c = (Cat)a;

c.sing();

c.eat();

//编译错误

//用父类引用调用父类不存在的方法

//Animal a1 = new Cat();

//a1.eat();

//编译错误

//向下类型转换时只能转向指向的对象类型

//Animal a2 = new Cat();

//Cat c2 = (Dog)a2;

}

}

class Animal

{

public void sing()

{

System.out.println("Animal is singing!");

}

}

class Dog extends Animal

{

public void sing()

{

System.out.println("Dog is singing!");

}

}

class Cat extends Animal

{

public void sing()

{

System.out.println("Cat is singing!");

}

public void eat()

{

System.out.println("Cat is eating!");

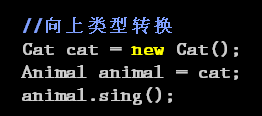
}

}

两种类型的类型转换：

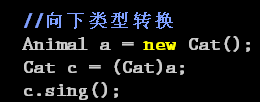
（1）向上类型转换（Upcast）：将子类型转换为父类型。

　　对于向上的类型转换，不需要显示指定，即不需要加上前面的小括号和父类类型名。



（2）向下类型转换（Downcast）：将父类型转换为子类型。

对于向下的类型转换，必须要显式指定，即必须要使用强制类型转换。



并且父类型的引用必须指向子类的对象，即指向谁才能转换成谁。

多态的好处：

1.可替换性（substitutability）。多态对已存在代码具有可替换性。例如，多态对圆Circle类工作，对其他任何圆形几何体，如圆环，也同样工作。

2.可扩充性（extensibility）。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。

3.接口性（interface-ability）。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。

4.灵活性（flexibility）。它在应用中体现了灵活多样的操作，提高了使用效率。

5.简化性（simplicity）。多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

**抽象类**

在上一个例子中，Animal类是不可能被实例化的，因为实例化Dog或是Cat有意义，实例化Animal是没有意义的。可以把实例化没有任何意义的类改为抽象类。

抽象类的特点：

1）抽象类不能实例化；

2）抽象方法是必须被子类重写的方法，抽象方法可以看做是没有实现体的虚方法。

3）如果累中包含抽象方法，那么这个类必须被定义为抽象类，不管这个类中 是否存在其他一般方法。

4）抽象类中包含尽可能多的共同代码和尽可能少的数据。

抽象类代表一个抽象的概念，他提供一个继承的出发点，当设计一个新的抽象类时，一定是用来继承的，所以，在一个以继承关系形成的等级结构里面，树叶节点应该是具体类，而树枝节点都应当是抽象类。所以具体类不是用来继承的。

我们知道HTTP协议是通过HTTP请求和HTTP响应来实现双向通信的。 HTTP状态码（HTTP Status Code）是用以表示Web服务器HTTP响应状态的3位数字代码，由RFC 2616规范定义。 合理的状态码不仅可以让用户或者浏览器做出更加合适的进一步操作，而且可以让客户端代码更加易于理解和维护。

HTTP状态码分为5类：1xx表示继续发送请求；2xx表示请求成功；3xx表示资源已找到但需要继续进行其他操作； 4xx表示客户端错误；5xx表示服务器错误。下面就具体详述常见状态码的语义，及其使用方式。

关于HTTP协议规范以及状态码在其中的作用，参见：读 HTTP 协议。 参考链接：http://zh.wikipedia.org/wiki/HTTP%E7%8A%B6%E6%80%81%E7%A0%81

1xx

1xx表示请求已被接受，但需要后续处理。例如：

100（Continue）

客户端应继续发送请求。

101（Switching Protocols）

需要切换协议，服务器通过的Upgrade响应头字段通知客户端。

HTML5引入的WebSocket便是这样工作的。首先客户端请求websocket所在的URL，服务器返回101，然后便建立了全双工的TCP连接。 注意Upgrade和Connection头字段属于Hop-by-hop字段，设置Websocket代理时需要继续设置这两个字段，而不是简单地转发请求。

2xx

请求已成功被服务器接收、理解、并接受。

200（OK）

请求已成功，请求所希望的响应头或数据体将随此响应返回。

201（Created）

请求已经被实现，而且有一个新的资源已经依据请求的需要而创建。在RESTFul风格的URL设计中，通常用来响应POST请求。

202（Accepted）

服务器已接受请求，但尚未处理。比如POST一个资源应当返回201，但由于性能原因未能立即创建，可以返回202。

204（No Content）

服务器成功处理了请求，但不需要返回任何实体内容，204响应禁止包含任何消息体。浏览器收到该响应后不应产生文档视图的变化。

205（Reset Content）

服务器成功处理了请求，但不需要返回任何实体内容，205响应禁止包含任何消息体。 与204不同的是，返回此状态码的响应要求请求者重置文档视图。比如用户刚刚提交一个表单，返回205后页面重置，用户可以立即填写下一个表单。

206（Partial Content）

HTTP协议允许分片传输。请求头中包含Range字段时，响应需要只返回Range指定的那一段。响应中应包含Content-Range来指示返回内容的范围。

其他

203（Non-Authoritative Information）

207（Multi-Status）

3xx

这类状态码代表需要客户端采取进一步的操作才能完成请求。通常，这些状态码用来重定向， 重定向目标在本次响应的Location头字段中指明。

301（Moved Permanently）

被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个URI之一。如果该请求不是GET/HEAD， 浏览器通常会要求用户确认重定向。

301通常用于网站迁移时，服务器对旧的URL进行301重定向到新的URL。这样搜索引擎可以正确地更新原有的页面排名等信息。

302（Found）

请求的资源现在临时从不同的URI响应请求。除非指定了Cache-Control或Expires，否则该响应不可缓存。 如果当前请求非HEAD或GET，浏览器需取得用户确认，再进行重定向。

这很好理解，因为上下文发生了变化，比如POST请求不是幂等的。

303（See Other）

对应当前请求的响应可以在另一个URI上被找到，而且客户端应当采用GET的方式访问那个资源。 这个方法的存在主要是为了允许由脚本激活的POST请求输出重定向到一个新的资源。 303响应禁止被缓存。

303会使得浏览器直接GET那个资源，不需用户同意。这是Web应用中最常见的重定向方式。

304（Not Modified）

如果客户端发送了一个带条件的GET请求且该请求已被允许，而文档的内容（自上次访问以来或者根据请求的条件）并没有改变。 304响应禁止包含消息体。

304响应也是一种缓存机制。Web服务器对静态资源文件通常会采取缓存，因此在Web开发中你可以看到大量的304响应。 服务器给出的相应中通常会包含Etag来标识资源ID，比如：

ETag: "686897696a7c876b7e"

客户端在下次访问同一URL时会设置头字段If-None-Match（这是一个请求条件）：

If-None-Match: "686897696a7c876b7e"

服务器返回资源前会判断Etag是否与客户端提供的If-None-Match匹配，如果匹配则说明资源未发生改变，此时应返回304.

其他

300（Multiple Choices）

305（Use Proxy）

307（Temporary Redirect）

4xx

这类的状态码代表了客户端看起来可能发生了错误，妨碍了服务器的处理。 除非响应的是一个HEAD请求，否则服务器就应该返回一个解释当前错误状况的实体。

400（Bad Request）

由于包含语法错误，当前请求无法被服务器理解。400通常在服务器端表单验证失败时返回。

401（Unauthorized）

当前请求需要用户验证，响应中会包含一个WWW-Authenticate字段来询问用户的授权信息。 而客户端的下次请求需要提供包含Authorization头的请求。

HTTP Basic Auth就是这样实现的。当服务器返回401时浏览器会弹出窗口：

![][basic-auth]

输入验证信息并点击确定，浏览器会根据你的输入填写Authentication头并重新发送请求。对于Basic Auth看起来是这样的：

Authorization:Basic eWFuZ2p2bjp5YW5nanZuaGFydA==

403（Forbidden）

服务器已经理解请求，但是拒绝执行它。与401响应不同的是，身份验证并不能提供任何帮助。

403和401一样，需要在相应消息体中需要给出原因。除非是一个HEAD请求。

通常用于服务器已经知道用户的身份的情况。比如从请求的Cookie得到的Session中可以得知当前用户无权进行该操作。 通常的Web应用中，对于401的情况应当重定向至登录页面，403的情况应当直接告知错误（这属于前端Bug）。

404（Not Found）

这太常见了。就是请求所希望得到的资源未被在服务器上发现。

当通常用于当服务器不想揭示到底为何请求被拒绝时，比如应当返回500时服务器不愿透露自己的错误。

405（Method Not Allowed）

请求行中指定的请求方法不能被用于请求相应的资源。

在Web开发中通常是因为客户端和服务器的方法不一致，比如客户端通过PUT来修改一个资源，而服务器把它实现为POST方法。 开发中统一规范就好了。

413（Request Entity Too Large）

服务器拒绝处理当前请求，因为该请求提交的实体数据大小超过了服务器愿意或者能够处理的范围。

一般的服务器都会设置HTTP请求消息体的最大长度，当然这是一种阻挡攻击的手段。 例如你在使用HTTP方式来访问Git仓库，如果你在仓库中加入了大的二进制文件（通常为目标文件或多媒体文件）， 在Push时服务器很可能会返回413错误。如果切换为ssh协议就不会有这样的问题了，服务器只能限制整个仓库的大小。

414（Request-URI Too Large）

当URI太长时，服务器可以返回414. 当HTTP协议并未规定URI应当有多长。这取决于浏览器和服务器的设置， 在服务器中当然你想设置多长都可以，但是浏览器是你决定不了的，而且不同的厂商在采用不同的长度限制，可以认为最短的是2K：

The limit is in MSIE and Safari about 2KB, in Opera about 4KB and in Firefox about 8KB, (255 bytes if we count very old browsers) .

当然URI长度的这一点限制会使得HTTP GET方法的能力有限，不能传输超过2K的数据。另外因为GET方法的所有数据都体现在URI上， 对于用户是可见的。这一点不太安全。

其实HTTP GET传输数据理论上并不比POST安全更多，因为POST的Entity Body也是明文传输的。只是GET的数据用户直接可见而已。

其他

402（Payment Required）

406（Not Acceptable）

407（Proxy Authentication Required）

408（Request Time-out）

409（Conflict）

410（Gone）

411（Length Required）

412（Precondition Failed）

415（Unsupported Media Type）

416（Requested range not satisfiable）

417（Expectation Failed）

5xx

这类状态码代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理。 并且响应消息体中应当给出理由，除非是HEAD请求。

500（Internal Server Error）

通常是代码出错，后台Bug。一般的Web服务器通常会给出抛出异常的调用堆栈。 然而多数服务器即使在生产环境也会打出调用堆栈，这显然是不安全的。

502（Bad Gateway）

作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应。

如果你在用HTTP代理来翻墙，或者你配置了nginx来反向代理你的应用，你可能会常常看到它。

504（Gateway Time-out）

作为网关或者代理工作的服务器尝试执行请求时，未能及时从上游服务器收到响应。

注意与502的区别：502是接收到了无效响应比如Connection Refused； 504是响应超时，通常是被墙了。

很多国外站点都会使用CDN来Serve静态文件，但我大中华墙掉了一些CDN。导致这些资源文件的请求会一直处于Pending状态直到超时。 表现为浏览器显示空白页面，长时间处于等待状态。这时在控制台看到HTML已经载入了就可以点击停止按钮了，停止载入那些资源。 页面会立即显示出来，虽然样式和交互可能有问题。

说到这里想吐槽GFW简直没有行业道德，即使不允许我们访问也返回一个错误嘛，不要让浏览器一直处于等待的状态。

其他

501（Not Implemented）

503（Service Unavailable）

505（HTTP Version not supported）

什么是正则表达式（Regular Expression）？

使用单个字符串来匹配、描述一系列符合某个句法规则的字符串。即按照某种规则去匹配符合条件的字符串。

正则表达式在线工具：http://regexper.com

JavaScript通过内置对象RegExp支持正则表达式。有两种方法实例化RegExp对象：

1.字面量：var reg = /\bis\b/g;

2.构造函数：var reg = new RegExp('\\bis\\b', 'g');//含义同上。其中双反斜线\\是转义字符

\b代表单词边界；^代表开头；$代表结尾；\d代表数字；[-/]表示-或/中的一个；{2}代表有两个，例如\d{4}代表4个数字；末尾加g代表全文匹配。如：

var reg = /\bis\b/;

'He is a boy.This is a dog.'.replace(reg,'IS');

把小写的is替换为大写IS，这里只能匹配第一个is。

var reg = /\bis\b/g;

'He is a boy.This is a dog.'.replace(reg,'IS');

这里可以匹配所有的is。

修饰符：g，global，全文搜索，不添加则搜索到第一个匹配停止；i，ignore case，忽略大小写，默认大小写敏感；m，多行搜索。

正则表达式由两种基本字符类型组成：

原义文本字符：代表它本来意思的字符；

元字符：正则表达式中有特殊含义的非字母字符。如：\* + ? $ % ^ . | \ () {} []

预定义类：



贪婪模式：

在'123456'中匹配\d{3,6}，既可以匹配三次，也可以匹配4次5次6次。

这种情况下正则表达式会尽可能多的匹配，直到匹配出错，这是贪婪模式。

如果希望尽可能少的匹配，即非贪婪模式，只需在量词后加?，如\d{3,6}?

字符类：

一般情况下正则表达式一个字符对应字符串的一个字符。

匹配某类字符时，可以用元字符[]来构建一个简单的类（所谓类是指符合某些特征的对象，一个泛指，而不是指某个字符）。例如表达式[abc]把字符a或b或c归为一类，表达式可以匹配这类的字符。

字符类取反：^

反向类[^abc]意思是不是a或b或c中的内容。

范围类：

可以使用[a-z]来连接两个字符表示从a到z的任意字符。包含a和b本身。

在[]组成的类内部可以连写，如：[a-zA-Z]

边界：



量词：



分组：

匹配字符串Byron重复三次，不能写Byron{3}，这样只能匹配n出现3次。应该写成：(Byron){3}。用()可以实现分组功能，使数字不仅匹配到紧挨着它的那个单词，而是匹配整个分组。

或：

使用“|”可达到效果。如：/Liuya(ying|win)/g

反向引用：

2015-12-25替换为12/25/2015

'2016-12-25'.replace(/(\d{4})-(\d{2})-(\d{2})/,'$2/$3/$1');

三个$分别代表三个分组中的内容。

忽略分组：

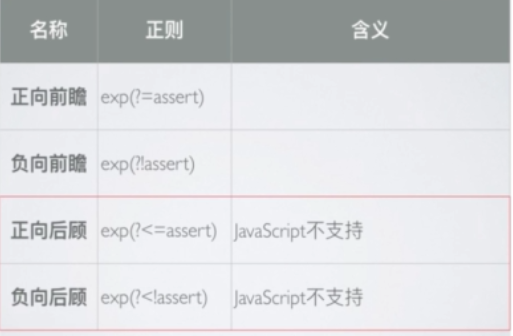
在分组内加上 ?: 即可。如：(?:Byron).(ok)

前瞻：

正则表达式从文本头部向尾部开始解析，文本尾部方向，称为“前”。

前瞻就是在正则表达式匹配到规则的时候，向前检查是否符合断言，后顾/后瞻方向相反。JavaScript不支持后顾。

符合和不符合特定断言称为肯定/正向匹配和否定/负向匹配。



例子：'a2\*34v8'.replace(/\w(?=\d)/g,'A')匹配后面跟数字的单词字符（字母数字下划线）。

对象属性：

global：是否全文搜索，默认为false；ignore case：是否大小写敏感，默认是false；multiline：多行搜索，默认值是false；lastIndex：是当前表达式匹配内容的最后一个字符的下一个位置；sourse：正则表达式的文本字符串。这几个标志位都是只读的。

RegExp.prototype.test(str)

用于测试字符串参数中是否存在匹配正则表达式模式的字符串。存在返回true，否则返回false。

RegExp.prototype.exec(str)

使用正则表达式模式对字符串执行搜索，并将更新全局RegExp对象的属性以反映匹配的结果。没匹配到返回null，匹配到了返回一个结果数组，数组的第一个元素是匹配到的结果，第二个元素是第一个分组匹配的结果，第三个结果是第二个分组匹配到的结果……

String.prototype.search(reg)

用于检索字符串中指定的子字符串，或检索与正则表达式相匹配的子字符串。方法返回第一个匹配结果index，匹配不到返回-1。search()方法不执行全局匹配，它将忽略标志g，并且总是从字符串的开始进行检索。

String.prototype.match(reg)

match()方法将检索字符串，以找到一个或多个与regexp匹配的文本。regexp是否具有标志g对结果影响很大。

如果没有g（非全局调用），那么match()方法就只能在字符串中执行一次匹配，如果没有匹配到任何文本，将返回null；否则它将返回一个数组，其中存放了与它找到的匹配文本有关的信息。返回数组的第一个元素存放的是匹配文本，而其余的元素存放的是与正则表达式的子表达式（分组）匹配的文本。除了常规的数组元素外，返回的数组还有两个属性：index邓明匹配文本的起始字符在字符串中的位置；input声明对stringObject的引用。

如果具有标志g（全局调用）时找到字符串中所有匹配字符串，没有找到任何匹配的子串返回null；如果找到了一个或多个匹配的子串则返回一个数组。数组元素中存放的是字符串中所有的匹配子串，没有分组信息，也没有index和input属性。

String.prototype.split(reg)

用split()方法将字符串分割为数组。"a1b2c3d4e".split(/\d/);

String.prototype.replace(reg,str)

替换："a1b2c3d4e".replace(/\d/g,'A');



最后一个中的function会在每次匹配替换的时候调用，有四个参数：匹配字符串；正则表达式分组内容，没有分组则没有该参数；匹配项在字符串中的index；原字符串。

window.onload()和$(document).ready()的 区别：

1. window.onload()是在网页中所有元素（包括元素的所有关联文件）都加载完毕后才执行的；$(document).ready()在DOM完全就绪就可以调用

2.多次使用时，前面的window.onload()会被后面的覆盖掉；$(document).ready()会根据注册的顺序依次执行。

this，表示当前的上下文对象是一个html对象，可以调用html对象所拥有的属性，方法

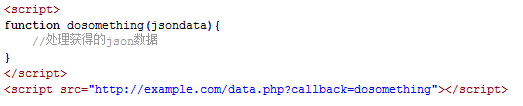
$(this),代表的上下文对象是一个jquery的上下文对象，可以调用jquery的方法和属性值。

只要协议、域名、端口有任一个不同，就被当做是不同的域。

**1.通过jsonp跨域**

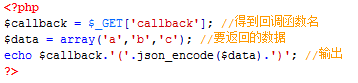
在js中不能通过XMLHttpRequest请求不同域上的数据，但是可以在页面上引入不同域的js文件，jsonp正是利用这个特性来实现的。

比如，有个a.html页面，需要通过ajax来湖区不同域上的json数据，假设这个json数据地址是http://example.com/data.php，那么a.html中的代码可以这样：



我们看到获取数据的地址后面还有一个callback参数，按惯例是用这个参数名，但是你用其他的也一样。当然如果获取数据的jsonp地址页面不是你自己能控制的，就得按照提供数据的那一方的规定格式来操作了。

因为是当做一个js文件来引入的，所以<http://example.com/data.php>返回的必须是一个能执行的js文件，所以这个页面的php代码可能是这样的:



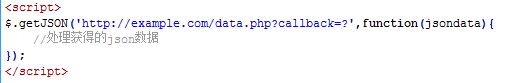
最终那个页面输出的结果是:

http://images.cnitblog.com/blog/130623/201307/15184528-32bbea18197b4e35bb5f67480da1f367.png

所以通过<http://example.com/data.php?callback=dosomething>得到的js文件，就是我们之前定义的dosomething函数,并且它的参数就是我们需要的json数据，这样我们就跨域获得了我们需要的数据。

这样jsonp的原理就很清楚了，通过script标签引入一个js文件，这个js文件载入成功后会执行我们在url参数中指定的函数，并且会把我们需要的json数据作为参数传入。所以jsonp是需要服务器端的页面进行相应的配合的。

知道jsonp跨域的原理后我们就可以用js动态生成script标签来进行跨域操作了，而不用特意的手动的书写那些script标签。如果你的页面使用jquery，那么通过它封装的方法就能很方便的来进行jsonp操作了。



原理是一样的，只不过我们不需要手动的插入script标签以及定义回调函数。jquery会自动生成一个全局函数来替换callback=?中的问号，之后获取到数据后又会自动销毁，实际上就是起一个临时代理函数的作用。$.getJSON方法会自动判断是否跨域，不跨域的话，就调用普通的ajax方法；跨域的话，则会以异步加载js文件的形式来调用jsonp的回调函数。

补充知识点：JSON和JSONP

JSON是一种数据交换的格式，JSONP是一种非官方跨域数据交互协议。

**JSON的优点：**

1、基于纯文本，跨平台传递极其简单；

2、Javascript原生支持，后台语言几乎全部支持；

3、轻量级数据格式，占用字符数量极少，特别适合互联网传递；

4、可读性较强，虽然比不上XML那么一目了然，但在合理的依次缩进之后还是很容易识别的；

5、容易编写和解析，当然前提是你要知道数据结构；

**缺点**：

格式要求较为严格

var person = {

"Name": "Bob",

"Age": 32,

"Company": "IBM",

"Engineer": true

}

// 获取这个人的信息

var personAge = person.Age;

var members = [

{

"Name": "Bob",

"Age": 32,

"Company": "IBM",

"Engineer": true

},

{

"Name": "Henry",

"Age": 45,

"Company": "Microsoft",

"Engineer": false

}

]

// 读取其中John的公司名称

var johnsCompany = members[1].Company;

var conference = {

"Conference": "Future Marketing",

"Date": "2012-6-1",

"Address": "Beijing",

"Members":

[

{

"Name": "Bob",

"Age": 32,

"Company": "IBM",

"Engineer": true

},

{

"Name": "John",

"Age": 20,

"Company": "Oracle",

"Engineer": false

},

{

"Name": "Henry",

"Age": 45,

"Company": "Microsoft",

"Engineer": false

}

]

}

// 读取参会者Henry是否工程师

var henryIsAnEngineer = conference.Members[2].Engineer;

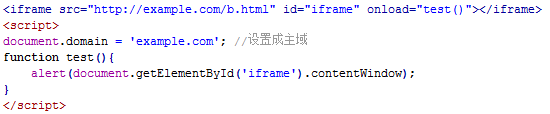
**2.通过修改document.domain来跨子域**

同源策略中的限制之一是不能通过ajax的方法去请求不同域中的文档，其另一个限制是浏览器中不同域的框架之间是不能进行js交互操作的。不同框架之间（父子或同辈）是能够获取到彼此的window对象的，但却不能使用获取到的对象的属性或方法。比如，有一个页面地址是<http://www.example.com/a.html>  ， 在这个页面里面有一个iframe，它的src是<http://example.com/b.html>, 很显然，这个页面与它里面的iframe框架是不同域的，所以我们是无法通过在页面中书写js代码来获取iframe中的东西的：



这时只要把<http://www.example.com/a.html> 和 <http://example.com/b.html>这两个页面的document.domain都设成相同的域名就可以了。但要注意的是，document.domain的设置是有限制的，我们只能把document.domain设置成自身或更高一级的父域，且主域必须相同。例如：a.b.example.com 中某个文档的document.domain 可以设成a.b.example.com、b.example.com 、example.com中的任意一个，但是不可以设成 c.a.b.example.com,因为这是当前域的子域，也不可以设成baidu.com,因为主域已经不相同了。

在页面 <http://www.example.com/a.html> 中设置document.domain:



在页面 <http://example.com/b.html> 中也设置document.domain，而且这也是必须的，虽然这个文档的domain就是example.com,但是还是必须显示的设置document.domain的值：

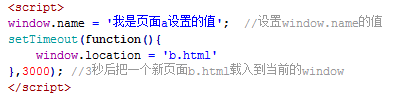
15184533-aaa37d1a446f4ff2871d9e2b3e9477da

这样我们就可以通过js访问到iframe中的各种属性和对象了。

**3.使用window.name来进行跨域**

window对象有个name属性，该属性有个特征：即在一个窗口(window)的生命周期内,窗口载入的所有的页面都是共享一个window.name的，每个页面对window.name都有读写的权限，window.name是持久存在一个窗口载入过的所有页面中的，并不会因新页面的载入而进行重置。

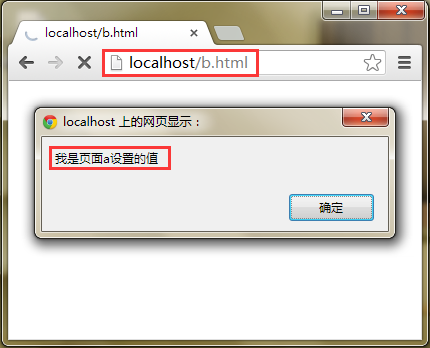
比如：有一个页面a.html,它里面有这样的代码：



再看看b.html页面的代码：

http://images.cnitblog.com/blog/130623/201307/15184535-fd6a9a0604754241a8f8947ced97b8e9.png

a.html页面载入后3秒，跳转到了b.html页面，结果为：



上面的例子中，我们用到的页面a.html和b.html是处于同一个域的，但是即使a.html与b.html处于不同的域中，上述结论同样是适用的，这也正是利用window.name进行跨域的原理。

比如有一个[www.example.com/a.html](http://www.example.com/a.html)页面,需要通过a.html页面里的js来获取另一个位于不同域上的页面[www.cnblogs.com/data.html](http://www.cnblogs.com/data.html)里的数据。

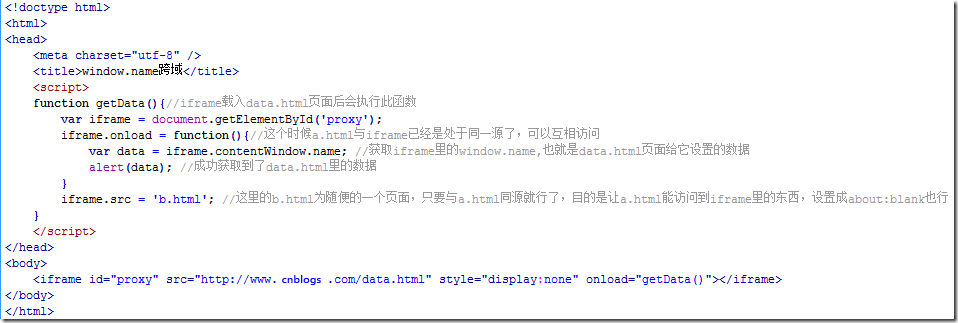
data.html页面里的代码很简单，就是给当前的window.name设置一个a.html页面想要得到的数据值。data.html里的代码：

QQ截图20130613230631

那么在a.html页面中，我们怎么把data.html页面载入进来呢？显然我们不能直接在a.html页面中通过改变window.location来载入data.html页面，因为我们想要即使a.html页面不跳转也能得到data.html里的数据。答案就是在a.html页面中使用一个隐藏的iframe来充当一个中间人角色，由iframe去获取data.html的数据，然后a.html再去得到iframe获取到的数据。

充当中间人的iframe想要获取到data.html的通过window.name设置的数据，只需要把这个iframe的src设为[www.cnblogs.com/data.html](http://www.cnblogs.com/data.html)就行了。然后a.html想要得到iframe所获取到的数据，也就是想要得到iframe的window.name的值，还必须把这个iframe的src设成跟a.html页面同一个域才行，不然根据前面讲的同源策略，a.html是不能访问到iframe里的window.name属性的。这就是整个跨域过程。

看一下a.html的代码：



上面的代码只是最简单的原理演示代码，你可以对使用js封装上面的过程，比如动态的创建iframe,动态的注册各种事件等等，当然为了安全，获取完数据后，还可以销毁作为代理的iframe。

**4.使用HTML5中新引进的window.postMessage方法来跨域传送数据**

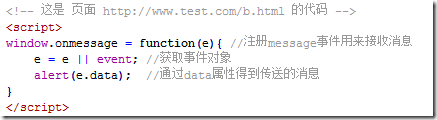
window.postMessage(message,targetOrigin) 方法是html5新引进的特性，可以使用它来向其它的window对象发送消息，无论这个window对象是属于同源或不同源，目前IE8+、FireFox、Chrome、Opera等浏览器都已经支持window.postMessage方法。

调用postMessage方法的window对象是指要接收消息的那一个window对象，该方法的第一个参数message为要发送的消息，类型只能为字符串；第二个参数targetOrigin用来限定接收消息的那个window对象所在的域，如果不想限定域，可以使用通配符 \* 。

需要接收消息的window对象，可是通过监听自身的message事件来获取传过来的消息，消息内容储存在该事件对象的data属性中。

上面所说的向其他window对象发送消息，其实就是指一个页面有几个框架的那种情况，因为每一个框架都有一个window对象。在讨论第二种方法的时候，我们说过，不同域的框架间是可以获取到对方的window对象的，而且也可以使用window.postMessage这个方法。下面看一个简单的示例，有两个页面：





运行a页面之后看到效果：



注意到a页面是没有alert的，所以我们看到b页面成功的收到了消息。

路由

能够从页面的一个视图跳转到另外一个视图，对单页面应用来讲是至关重要的。当应用变得越来越复杂时，我们需要一个合理的方式来管理用户在使用过程中看到的界面。AngularJS的做法是将视图分解成布局和模板视图，并且根据用户当前访问的URL来展示对应的视图。

安装：使用angular的路由功能需要引入routing模块，即引入angular-route.js。

定义：在应用的main模块里面注入ngRoute依赖：

angular.module('myapp',['ngRoute']).config(function($routeProvide){});

1.页面布局

引用scripts：

<script src="../Scripts/jquery-1.9.1.min.js"></script>

<script src="../Scripts/angular.min.js"></script>

<script src="../Scripts/angular-route.min.js"></script>

页面布局：

<div>

<ul>

<li><a href="#page1">go page 1</a></li>

<li><a href="#page2">go page 2</a></li>

<li><a href="#other">to other page</a></li>

</ul>

</div>

<div ng-view></div>

ng-view是由ngRoute模块提供的一个特殊指令，其告诉AngularJS把模板渲染到何处。这个例子中，我们将需要渲染的内容放到下面的div中。上面的三个a链接分别指向了三个视图view。

2.页面模板

创建两模板页面，分别叫 Subpage\_1.html 和 Subpage\_2.html。

3.路由规则 routing config

angular.module("myRouteApp", ["ngRoute"])

.config(['$routeProvider', function ($routeProvider) {

$routeProvider

.when("/page1", {

templateUrl: "Subpage\_1.html"

})

.when("/page2", {

templateUrl: "Subpage\_2.html"

})

.otherwise({

redirectTo: "/"

});

}]);

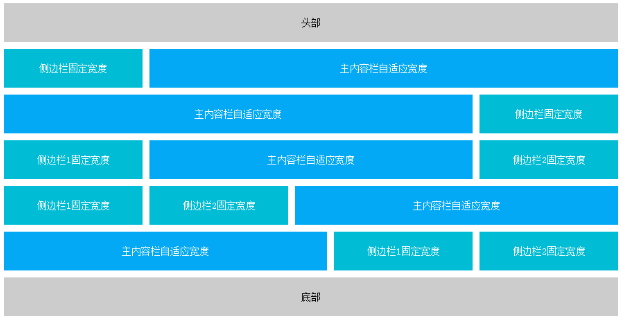
把ngRoute模块在我们的应用中当作依赖加载进来。用config函数在模块或应用中定义路由，使用AngularJS提供的when和otherwise两个方法来定义应用的路由。

templateUrl：应用会根据templateUrl属性所指定的路径通过XHR读取视图（或者从$templateCache中读取）。如果能够找到并读取这个模板，AngularJS将模板的内容渲染到具有ng-view指令的DOM元素中。

redirectTo：如果redirectTo属性的值是一个字符串，那么路径会被替换成这个值，并根据这个目标路径触发路由变化。如果redirectTo属性的值是一个函数，那么路径会被替换成函数的返回值，并根据这个目标路径触发路由变化。

**圣杯布局**

经典三列布局，也叫做圣杯布局【Holy Grail of Layouts】是Kevin Cornell在2006年提出的一个布局模型概念，在国内最早是由淘宝UED的工程师传播开来，在中国也有叫法是双飞翼布局，它的布局要求有几点： 1、三列布局，中间宽度自适应，两边定宽； 2、中间栏要在浏览器中优先展示渲染； 3、允许任意列的高度最高； 4、要求只用一个额外的DIV标签； 5、要求用最简单的CSS、最少的HACK语句；



html代码中  middle部分首先要放在container的最前部分。然后是left,right

**1.**将三者都 float:left , 再加上一个position:relative (因为相对定位后面会用到）

**2.**middle部分 width:100%占满

**3.**此时middle占满了，所以要把left拉到最左边，使用margin-left:-100%

**4.**这时left拉回来了，但会覆盖middle内容的左端，要把middle内容拉出来，所以在外围container加上 padding:0 220px 0 200px

**5.**middle内容拉回来了，但left也跟着过来了，所以要还原，就对left使用相对定位 left:-200px  同理，right也要相对定位还原 right:-220px

**6.**到这里大概就自适应好了。如果想container高度保持一致可以给left middle right都加上min-height:130px

实现代码：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">

<title>圣杯布局</title>

<style type="text/css">

body {background-color: #ffffff; font-size:14px;}

#hd, #ft {padding:20px 3px; background-color: #cccccc; text-align: center;}

.bd-lft, .bd-rgt, .bd-3-lr, .bd-3-ll, .bd-3-rr {margin:10px 0; min-width:400px;}

.main {background-color: #03a9f4; color:#ffffff;}

.aside, .aside-1, .aside-2 {background-color: #00bcd4; color:#ffffff;}

p {margin:0; padding:20px; text-align: center;}

/\* 左侧栏固定宽度，右侧自适应 \*/

.bd-lft {

zoom:1;

overflow:hidden;

padding-left:210px;

}

.bd-lft .aside {

float:left;

width:200px;

margin-left:-100%; /\*= -100%\*/

position:relative;

left:-210px; /\* = -parantNode.paddingLeft \*/

\_left: 0; /\*IE6 hack\*/

}

.bd-lft .main {

float:left;

width:100%;

}

/\* 右侧栏固定宽度，左侧自适应 \*/

.bd-rgt {

zoom:1;

overflow:hidden;

padding-right:210px;

}

.bd-rgt .aside {

float:left;

width:200px;

margin-left:-200px; /\* = -this.width \*/

position:relative;

right:-210px; /\* = -parantNode.paddingRight \*/

}

.bd-rgt .main {

float:left;

width:100%;

}

/\* 左中右 三栏自适应 \*/

.bd-3-lr {

zoom:1;

overflow:hidden;

padding-left:210px;

padding-right:210px;

}

.bd-3-lr .main {

float:left;

width:100%;

}

.bd-3-lr .aside-1 {

float: left;

width:200px;

margin-left: -100%;

position:relative;

left: -210px;

\_left: 210px; /\*IE6 hack\*/

}

.bd-3-lr .aside-2 {

float: left;

width:200px;

margin-left: -200px;

position:relative;

right: -210px;

}

/\* 都在左边，右侧自适应 \*/

.bd-3-ll {

zoom:1;

overflow:hidden;

padding-left:420px;

}

.bd-3-ll .main {

float:left;

width:100%;

}

.bd-3-ll .aside-1 {

float: left;

width:200px;

margin-left: -100%;

position:relative;

left: -420px;

\_left: 0px; /\*IE6 hack\*/

}

.bd-3-ll .aside-2 {

float: left;

width:200px;

margin-left: -100%;

position:relative;

left: -210px;

\_left: 210px; /\*IE6 hack\*/

}

/\* 都在右边，左侧自适应 \*/

.bd-3-rr {

zoom:1;

overflow:hidden;

padding-right:420px;

}

.bd-3-rr .main {

float:left;

width:100%;

}

.bd-3-rr .aside-1 {

float: left;

width:200px;

margin-left: -200px;

position:relative;

right: -210px;

}

.bd-3-rr .aside-2 {

float: left;

width:200px;

margin-left: -200px;

position:relative;

right: -420px;

}

</style>

</head>

<body>

<div id="hd">头部</div>

<div class="bd-lft">

<div class="main">

<p>主内容栏自适应宽度</p>

</div>

<div class="aside">

<p>侧边栏固定宽度</p>

</div>

</div>

<div class="bd-rgt">

<div class="main">

<p>主内容栏自适应宽度</p>

</div>

<div class="aside">

<p>侧边栏固定宽度</p>

</div>

</div>

<div class="bd-3-lr">

<div class="main">

<p>主内容栏自适应宽度</p>

</div>

<div class="aside-1">

<p>侧边栏1固定宽度</p>

</div>

<div class="aside-2">

<p>侧边栏2固定宽度</p>

</div>

</div>

<div class="bd-3-ll">

<div class="main">

<p>主内容栏自适应宽度</p>

</div>

<div class="aside-1">

<p>侧边栏1固定宽度</p>

</div>

<div class="aside-2">

<p>侧边栏2固定宽度</p>

</div>

</div>

<div class="bd-3-rr">

<div class="main">

<p>主内容栏自适应宽度</p>

</div>

<div class="aside-1">

<p>侧边栏1固定宽度</p>

</div>

<div class="aside-2">

<p>侧边栏2固定宽度</p>

</div>

</div>

<div id="ft">底部</div>

</body>

</html>

**双飞翼布局**

**1.**html代码中，main要放最前边，sub  extra

**2.**将main  sub  extra 都float:left

**3.**将main占满 width:100%

**4.**此时main占满了，所以要把sub拉到最左边，使用margin-left:-100%  同理 extra使用margin-left:-220px

（这时可以直接继续上边圣杯布局的步骤，也可以有所改动）

**5.**main内容被覆盖了吧，除了使用外围的padding，还可以考虑使用margin。

给main增加一个内层div-- main-inner, 然后margin:0 220px 0 200px

**6.**main正确展示

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>双飞翼布局</title>

<style type="text/css">

\* {

margin: 0;

padding: 0;

}

body {

min-width: 700px;

}

.header,

.footer {

border: 1px solid #333;

background: #aaa;

text-align: center;

}

.sub,

.main,

.extra {

float: left;

min-height: 130px;

}

.sub {

margin-left: -100%;

width: 200px;

background: red;

}

.extra {

margin-left: -220px;

width: 220px;

background: blue;

}

.main {

width: 100%;

}

.main-inner {

margin-left: 200px;

margin-right: 220px;

min-height: 130px;

background: green;

word-break: break-all;

}

.footer {

clear: both;

}

</style>

</head>

<body>

<div class="header">

<h4>header</h4>

</div>

<div class="main">

<div class="main-inner">

<h4>main</h4>

<p>HHHHHHHHHHHHHHHHHHHHHH hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh HHHHHHHHHHHHHHHHHHHHHH hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh

</p>

</div>

</div>

<div class="sub">

<h4>sub</h4>

<p>oooooooooooooo 0000000000000000 00000000000000000 ooooooooooooooo ooooooooooooooo 000000000000000

</p>

</div>

<div class="extra">

<h4>extra</h4>

<p>BBBBBBBBBBBBBB 888888888888888888 BBBBBBBBBBBBBBBBBB 88888888888888888888

</p>

</div>

<div class="footer">

<h4>footer</h4>

</div>

</body>

</html>

**1.text-align: 属性规定元素中的文本的水平对齐方式; 该属性通过指定行框与哪个点对齐，从而设置块级元素内文本的水平对齐方式; 一般情况下设置文本对齐方式的时使用此属性。支持值 justify。**

　　Example: div { text-align: left; }  //文本居左对齐

　　注释：所有浏览器都支持 text-align 属性；任何的版本的 Internet Explorer （包括 IE8）都不支持属性值 "inherit"。

**2.margin 是设置对象四边的外延边距，被称为外补丁或外边距。**

　　Example: div { margin: 20px 10px 30px 40px; }  // 表示对象外边距，顶20px、右10px、下30px、左40px

**区别如下:**

1.text-align:center 设置文本或img标签等一些内联对象（或与之类似的元素）的居中。

2.margin:0 auto 设置块元素（或与之类似的元素）的居中。

这两个属性IE与FF的理解也有所不同。**我们设置一个段落P，在段落内存在一个图片img标签。**

**注意：**

1.当设置body{text-align:center;}。 在IE中，段落P，图片img同时实现了居中对齐，也就是说text-align:center;同时作用于元素p与元素img。 在FF中，段落P，没有能实现居中对齐，而图片img实现了居中对齐，也就是说text-align:center;作用于img标签，而段落p标签没有起到居中的作用。

2.当设置段落 p {margin:0 auto;}。 在IE与FF中，段落P均实现了居中对齐。图片img由于不是作用对象，所以不会居中对齐。

有三种情况需要说明： 1.margin:0 auto;的选择器是作用对象，如div，p，而不是body。如果设置：body { margin:0 auto; }将不会达到任何效果，除非你定义body的宽度，那将会让body内的元素产生位置变化。如我们设置body宽度为500px。对p段落不作任何设置， 我们最大化窗口将会看到段落并非处于窗口的最左上角。 2.设置段落 p {text-align:center;} 将要实现的并不是段落本身的对齐方式，而是段落内元素居中对齐。 3. 设置图片标签img {margin:0 auto;} ，就犯了一个小错误，img类于内联对象，不可以设置图片img标签的margin属性，如果一定要设置，那么先将它的属性转变为块元素，如下面的代 码：img {display:block; margin:0 auto;}

vertical-align只对inline和inline-block元素有效。（？）

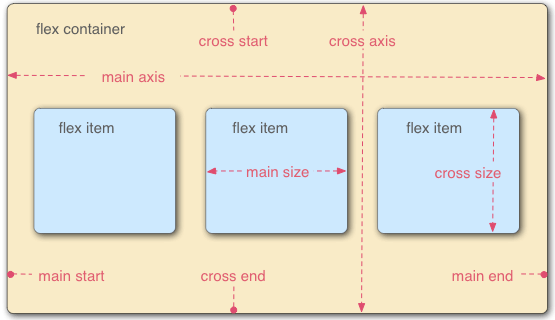
**Flex布局**

**基本概念**

Flex是Flexible Box的缩写，意味“弹性布局”，用来为盒状模型提供最大的灵活性。

设置为Flex 布局后，元素的float、clear和vertical-align将失效。

采用Flex布局的元素，称为Flex容器（flex container），简称“容器”，它的所有子元素自动成为容器成员（flex item），称为Flex项目，简称“项目”。



容器默认存在两根轴：水平的主轴（main axis）和垂直的交叉轴（cross axis）。主轴的开始位置（与边框的交叉点）叫做main start，结束位置叫做main end；交叉轴的开始位置叫做cross start，结束位置叫做cross end。

项目默认沿主轴排列。单个项目占据的主轴空间叫做main size，占据的交叉轴空间叫做cross size。

**容器的属性**

* flex-direction
* flex-wrap
* flex-flow
* justify-content
* align-items
* align-content

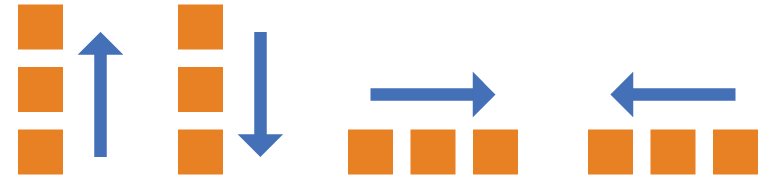
1.x-direction属性决定主轴的方向（即项目的排列方向）。

.box {

flex-direction: row | row-reverse | column | column-reverse;

}

* row（默认值）：主轴为水平方向，起点在左端。
* row-reverse：主轴为水平方向，起点在右端。
* column：主轴为垂直方向，起点在上沿。
* column-reverse：主轴为垂直方向，起点在下沿。



2.flex-wrap属性

默认情况下，项目都排在轴线上。flex-wrap属性定义如果一条轴线排不下该如何换行。

.box{

flex-wrap: nowrap | wrap | wrap-reverse;

}

nowrap：不换行；

wrap：换行，第一行在上方；

wrap-reverse：换行，第一行在下方。

3.flex-flow属性

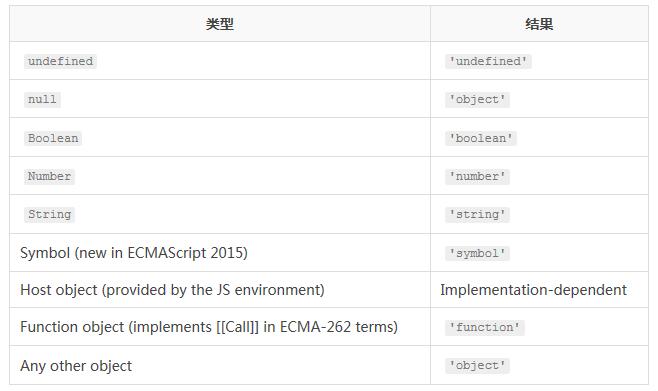
**问题1**

(function(){

 return typeof arguments;//求这里的返回值

})();

arguments是一个Array-like对象，对应的就是传入函数的参数列表。你可以在任何函数中直接使用该变量。typeof操作符只会返回string类型的结果。参照如下列表可知对应不同数据，typeof返回的值都是什么：



由此我们推断出，typeof arguments是object 。

**问题2**

var f = function g(){ return 23; };

typeof g();

这是一个名字是g的function expression，然后又被赋值给了变量f。  
 这里的函数名g和被其赋值的变量f有如下差异：  
 函数名g不能变动，而变量f可以被重新赋值  
 函数名g只能在函数体内部被使用，试图在函数外部使用g会报错的

综上，此处会报错。

考虑一下typeof f 和 typeof f()的值（Function和Number）

**问题3**

(function(x){

 delete x;

 return x;//1

})(1);

delete操作符可以从对象中删除属性，正确用法如下：  
delete object.property  
delete object['property']   
delete操作符只能作用在对象的属性上，对变量和函数名无效。也就是说delete x是没有意义的。

你最好也知道，delete是不会直接释放内存的，她只是间接的中断对象引用 。

**问题4**

var y = 1, x = y = typeof x;

console.log(x);

我们试图分解上述代码成下面两步：  
var y = 1; //step 1  
var x = y = typeof x; //step 2   
第一步应该没有异议，我们直接看第二步  
 1.赋值表达式从右向左执行  
 2.y被重新赋值为typeof x的结果，也就是undefined  
 3.x被赋值为右边表达式(y = typeof x)的结果，也就是undefined 。

**问题5**

(function f(f){

 return typeof f();

})(function(){ return 1; });

返回 “number”

**问题6**

var foo = {

bar: function() {

return this.baz;

},

baz: 1

};

(function() {

console.log(typeof arguments[0]()) ;

})(foo.bar);

这里你可能会误以为最终结果是number。向函数中传递参数可以看作是一种赋值，所以arguments[0]得到的是真正的bar函数的值，而不是foo.bar这个引用，那么自然this也就不会指向foo，而是window了。 所以打印出来的是undefined。

**问题7**

var foo = {

 bar: function(){ return this.baz; },

 baz: 1

}

typeof (f = foo.bar)();//"undefined"

这和上一题是一样的问题，(f = foo.bar)返回的就是bar的值，而不是其引用，那么this也就指的不是foo了。

**问题8**

var f = (function f(){ return '1'; }, function g(){ return 2; })();

typeof f;//"number"

逗号操作符对它的每个操作对象求值（从左至右），然后返回最后一个操作对象的值。

所以(function f(){ return '1'; }, function g(){ return 2; })的返回值就是函数g，然后执行她，那么结果是2；最后再typeof 2，根据问题一的表格，结果自然是number

**问题9**

var x = 1;

if (function f(){}) {

 x += typeof f;

}

x;//"1undefined"

1.除null、undefined、0、-0、""、NaN转换成bool值是false，其余全为true

2.函数表达式中的函数名f是不能在函数体外部访问的。

**问题10**

var x = [typeof x, typeof y][1];

typeof typeof x;

1.因为没有声明过变量y，所以typeof y返回"undefined"

2.将typeof y的结果赋值给x，也就是说x现在是"undefined"  
3.然后typeof x当然是"string"  
4.最后typeof "string"的结果自然还是"string"

**问题11**

(function(foo){

 return typeof foo.bar;//"undefined"

})({ foo: { bar: 1 } });

解析看注释：

(function(foo){

 //这里的foo，是{ foo: { bar: 1 } }，并没有bar属性哦。

 //bar属性是在foo.foo下面

 //所以这里结果是"undefined"

 return typeof foo.bar;

})({ foo: { bar: 1 } });

**问题12**

(function f(){

 function f(){ return 1; }

 return f();//2

 function f(){ return 2; }

})();

通过function declaration声明的函数甚至可以在声明之前使用，这种特性我们称之为hoisting。于是上述代码其实是这样被运行环境解释的：

(function f(){

 function f(){ return 1; }

 function f(){ return 2; }

 return f();

})();

**问题13**

function f(){ return f; }

new f() instanceof f;//false

当代码new f()执行时，下面事情将会发生：  
 1.一个新对象被创建。它继承自f.prototype  
 2.构造函数f被执行。执行的时候，相应的传参会被传入，同时上下文(this)会被指定为这个新实例。new f等同于new f()，只能用在不传递任何参数的情况。  
 3.如果构造函数返回了一个“对象”，那么这个对象会取代整个new出来的结果。如果构造函数没有返回对象，那么new出来的结果为步骤1创建的对象，

ps：一般情况下构造函数不返回任何值，不过用户如果想覆盖这个返回值，可以自己选择返回一个普通对象来覆盖。当然，返回数组也会覆盖，因为数组也是对象。

于是，我们这里的new f()返回的仍然是函数f本身，而并非他的实例

**问题十四** with (function(x, undefined){}) length;//2   
with语句将某个对象添加的作用域链的顶部，如果在statement中有某個未使用命名空间的变量，跟作用域链中的某個属性同名，則這個变量将指向這個属性值。如果沒有同名的属性，则将拋出ReferenceError异常。

1.词法作用域

词法作用域又称静态作用域，与之相对应的是动态作用域。

词法作用域是根据你写代码（定义函数）时变量和块作用域写在哪里来决定的，和使用/调用的位置无关。当定义了一个函数后，当前的作用域就会保存下来，成为函数内部状态的一部分。

词法作用域的函数中遇到既不是形参也不是函数内部定义的局部变量的变量时，去函数定义时的环境中查询。是在程序编译时决定的作用域。

动态域的函数中遇到既不是形参也不是函数内部定义的局部变量的变量时，到函数调用时的环境中查。是在程序运行时决定的作用域。

采用静态作用域的语言中，基本都是最内嵌套作用域规则：由一个声明引进的标识符在这个声明所在的作用域里可见，而且在其内部嵌套的每个作用域里也可见，除非它被嵌套于内部的对同名标识符的另一个声明所掩盖。为了找到某个给定的标识符所引用的对象，应该在当前最内层作用域里查找。如果找到了一个声明，也就可以找到该标识符所引用的对象。否则我们就到直接的外层作用域里去查找，并继续向外顺序地检查外层作用域，直到到达程序的最外嵌套层次，也就是全局对象声明所在的作用域。如果在所有层次上都没有找到有关声明，那么这个程序就有错误。（跟原型链比较记忆）

Javascript 则是在运行时编译，边编译边执行，这个机制涉及到三样东西：

1. 引擎

2. 编译器

3. 作用域

编译器首先对代码进行编译，然后将生成的代码提供给引擎执行。引擎和编译器工作的时候，都会用到作用域。编译器编译时，会在作用域中查找标示符，如果没有找到的话则会在对应的作用域创建一个标示符。引擎在执行时不断的在作用域中查找标示符，如果一直到全局对象(Global object)都没有找到的话则抛出一个“Reference Error” 并停止代码的执行。

例子：

var foo=1;

function static(){

alert(foo);

}

!function(){

var foo=2;

static();

}();

这里会弹出2而不是1。

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（除开异常的情况）。

在上面的例子中，函数foo创建时的上下文中，x=10，这个作用域在函数创建的时候就会保存下来：

var x = 10;

function foo() {

alert(x);

}

// foo是闭包

foo: <FunctionObject> = {

[[Call]]: <code block of foo>,

[[Scope]]: [

global: {

x: 10

}

],

... // 其它属性

};

2.所有内部函数都共享一个父作用域[[scope]]

var firstClosure;

var secondClosure;

function foo() {

var x = 1;

firstClosure = function () { return ++x; };

secondClosure = function () { return --x; };

x = 2; // 影响 AO["x"], 在2个闭包公有的[[Scope]]中

alert(firstClosure()); // 3, 通过第一个闭包的[[Scope]]

}

foo();

alert(firstClosure()); // 4

alert(secondClosure()); // 3

又一个例子：

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

上述例子就证明了同一个上下文中创建的闭包是共用一个[[Scope]]属性的。因此上层上下文中的变量“k”是可以很容易就被改变的：

activeContext.Scope = [

... // 其它变量对象

{data: [...], k: 3} // 活动对象

];

data[0].[[Scope]] === Scope;

data[1].[[Scope]] === Scope;

data[2].[[Scope]] === Scope;

联想到js的事件代理。

正确代码：

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

让我们来看看上述代码都发生了什么？函数“\_helper”创建出来之后，通过传入参数“k”激活。其返回值也是个函数，该函数保存在对应的数组元素中。这种技术产生了如下效果： 在函数激活时，每次“\_helper”都会创建一个新的变量对象，其中含有参数“x”，“x”的值就是传递进来的“k”的值。这样一来，返回的函数的[[Scope]]就成了如下所示：

data[0].[[Scope]] === [

... // 其它变量对象

父级上下文中的活动对象AO: {data: [...], k: 3},

\_helper上下文中的活动对象AO: {x: 0}

];

data[1].[[Scope]] === [

... // 其它变量对象

父级上下文中的活动对象AO: {data: [...], k: 3},

\_helper上下文中的活动对象AO: {x: 1}

];

data[2].[[Scope]] === [

... // 其它变量对象

父级上下文中的活动对象AO: {data: [...], k: 3},

\_helper上下文中的活动对象AO: {x: 2}

];

我们看到，这时函数的[[Scope]]属性就有了真正想要的值了，为了达到这样的目的，我们不得不在[[Scope]]中创建额外的变量对象。要注意的是，在返回的函数中，如果要获取“k”的值，那么该值还是会是3。

然而，上述提到的方法并不是唯一的方法。通过其他方式也可以获得正确的“k”的值，如下所示：

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

全局对象

JavaScript通过函数管理作用域。在函数内部声明的变量只在这个函数内部可用，而在函数外面不可用。另一方面，全局变量就是在任何函数外面声明的或是未声明直接简单使用的。

每个JavaScript环境有一个全局对象，当你在任意的函数外面使用this的时候可以访问到。你创建的每一个全局变量都成了这个全局对象的属性。在浏览器中，方便起见，该全局对象有个附加属性叫做window，此window(通常)指向该全局对象本身。下面的代码片段显示了如何在浏览器环境中创建和访问的全局变量：

myglobal = "nowamagic"; // 不推荐写法

console.log(myglobal); // "hello"

console.log(window.myglobal); // "hello"

console.log(window["myglobal"]); // "hello"

console.log(this.myglobal); // "hello"

危害

全局变量的问题在于，你的JavaScript应用程序和web页面上的所有代码都共享了这些全局变量，他们住在同一个全局命名空间，所以当程序的两个不同部分定义同名但不同作用的全局变量的时候，命名冲突在所难免。

可能创建全局变量的情况：

1）声明变量不加var关键字

function sum(x, y) {

// 不推荐写法: 隐式全局变量

result = x + y;

return result;

}

2）使用任务链进行部分var声明。下面的片段中，a是本地变量但是b确实全局变量：

function foo() {

var a = b = 0;

// ...

}

此现象发生的原因在于这个从右到左的赋值，首先，是赋值表达式b = 0，此情况下b是未声明的。这个表达式的返回值是0，然后这个0就分配给了通过var定义的这个局部变量a。

var关键字的副作用：

隐式全局变量和明确定义的全局变量间有些小的差异，就是通过delete操作符让变量未定义的能力。

通过var创建的全局变量（任何函数之外的程序中创建）是不能被删除的。

无var创建的隐式全局变量（无视是否在函数中创建）是能被删除的。

// 定义三个全局变量

var global\_var = 1;

global\_novar = 2; // 反面教材

(function () {

global\_fromfunc = 3; // 反面教材

}());

// 试图删除

delete global\_var; // false

delete global\_novar; // true

delete global\_fromfunc; // true

// 测试该删除

typeof global\_var; // "number"

typeof global\_novar; // "undefined"

typeof global\_fromfunc; // "undefined"

在ES5严格模式下，未声明的变量工作时会抛出一个错误。

函数声明与函数表达式

函数声明必须带有标识符（即函数名称），而函数表达式可以省略这个标识符：

函数声明：function 函数名称 (参数：可选){函数体}

函数表达式：function 函数名称（可选） (参数：可选){函数体}

可以看出，如果不声明函数名称，则必为函数表达式。如果声明函数名称时，如果function foo(){}是作为赋值表达式的一部分的话，那它就是一个函数表达式；如果function foo(){}被包含在一个函数体内或是位于程序最顶部的话，那他就是一个函数声明。

function foo(){} // 声明，因为它是程序的一部分

var bar = function foo(){}; // 表达式，因为它是赋值表达式的一部分

new function bar(){}; // 表达式，因为它是new表达式

(function(){

function bar(){} // 声明，因为它是函数体的一部分

})();

还有一种函数表达式不太常见，就是被括号括住的(function foo(){})，他是表达式的原因是因为括号 ()是一个分组操作符，它的内部只能包含表达式，我们来看几个例子：

function foo(){} // 函数声明

(function foo(){}); // 函数表达式：包含在分组操作符内

try {

(var x = 5); // 分组操作符，只能包含表达式而不能包含语句：这里的var就是语句

} catch(err) {

// SyntaxError

}

表达式和声明存在着十分微妙的差别，首先，函数声明会在任何表达式被解析和求值之前先被解析和求值，即使你的声明在代码的最后一行，它也会在同作用域内第一个表达式之前被解析/求值，参考如下例子，函数fn是在alert之后声明的，但是在alert执行的时候，fn已经有定义了（函数声明提前）：

alert(fn());

function fn() {

return 'Hello world!';

}

var foo = {

bar: function() {

return this.baz;

},

baz: 1

};

(function() {

console.log(typeof arguments[0]()) ; //"undefined"

})(foo.bar);

**构造函数**

除了创建对象，构造函数(constructor) 还做了另一件有用的事情：自动为创建的新对象设置了原型对象(prototype object) 。原型对象存放于 ConstructorFunction.prototype 属性中。

// 构造函数

function Foo(y) {

// 构造函数将会以特定模式创建对象：被创建的对象都会有"y"属性

this.y = y;

}

// "Foo.prototype"存放了新建对象的原型引用

// 所以我们可以将之用于定义继承和共享属性或方法

// 所以，和上例一样，我们有了如下代码：

// 继承属性"x"

Foo.prototype.x = 10;

// 继承方法"calculate"

Foo.prototype.calculate = function (z) {

return this.x + this.y + z;

};

// 使用foo模式创建 "b" and "c"

var b = new Foo(20);

var c = new Foo(30);

// 调用继承的方法

b.calculate(30); // 60

c.calculate(40); // 80

// 让我们看看是否使用了预期的属性

console.log(

b.\_\_proto\_\_ === Foo.prototype, // true

c.\_\_proto\_\_ === Foo.prototype, // true

// "Foo.prototype"自动创建了一个特殊的属性"constructor"

// 指向a的构造函数本身

// 实例"b"和"c"可以通过授权找到它并用以检测自己的构造函数

b.constructor === Foo, // true

c.constructor === Foo, // true

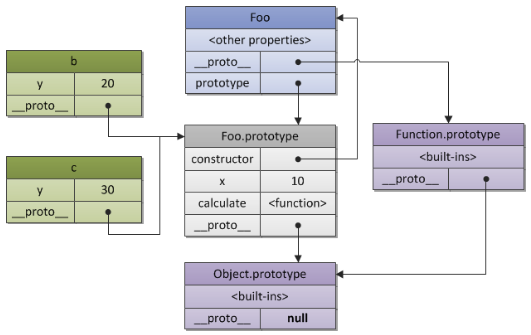
Foo.prototype.constructor === Foo // true

b.calculate === b.\_\_proto\_\_.calculate, // true

b.\_\_proto\_\_.calculate === Foo.prototype.calculate // true

);

上述代码可表示为如下的关系：



上述图示可以看出，每一个object都有一个prototype. 构造函数Foo也拥有自己的\_\_proto\_\_, 也就是Function.prototype, 而Function.prototype的\_\_proto\_\_指向了Object.prototype. 重申一遍，Foo.prototype只是一个显式的属性，也就是b和c的\_\_proto\_\_属性。

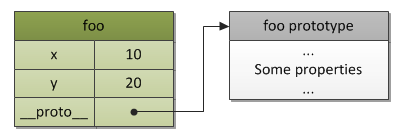
var foo = {

x: 10,

y: 20

};

上述代码foo对象有两个显式的属性[explicit own properties]和一个自带隐式的 \_\_proto\_\_ 属性[implicit \_\_proto\_\_ property]，指向foo的原型。



原型对象也是普通的对象，并且也有可能有自己的原型，如果一个原型对象的原型不为null的话，我们就称之为原型链（prototype chain）。

原型继承：

var a = {

x: 10,

calculate: function (z) {

return this.x + this.y + z

}

};

var b = {

y: 20,

\_\_proto\_\_: a

};

var c = {

y: 30,

\_\_proto\_\_: a

};

// 调用继承过来的方法

b.calculate(30); // 60

c.calculate(40); // 80

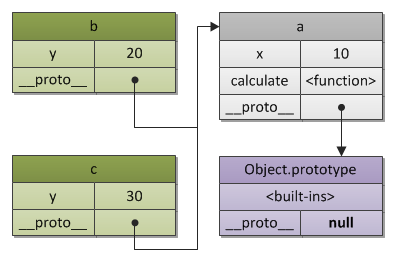
b和c可以使用a中定义的calculate方法，这就是有原型链来[prototype chain]实现的。

原理很简单:如果在对象b中找不到calculate方法(也就是对象b中没有这个calculate属性), 那么就会沿着原型链开始找。如果这个calculate方法在b的prototype中没有找到，那么就会沿着原型链找到a的prototype，一直遍历完整个原型链。记住，一旦找到，就返回第一个找到的属性或者方法。因此，第一个找到的属性成为继承属性。如果遍历完整个原型链，仍然没有找到，那么就会返回undefined。

注意一点，this这个值在一个继承机制中，仍然是指向它原本属于的对象，而不是从原型链上找到它时，它所属于的对象。例如，以上的例子，this.y是从b和c中获取的，而不是a。当然，你也发现了this.x是从a取的，因为是通过原型链机制找到的。

如果一个对象的prototype没有显示的声明过或定义过，那么\_\_prototype\_\_的默认值就是object.prototype, 而object.prototype也会有一个\_\_prototype\_\_, 这个就是原型链的终点了，被设置为null。

下面的图示就是表示了上述a,b,c的继承关系：



**简单粗暴的理解原型链**

不要纠结于那一堆术语了，那除了让你脑筋拧成麻花，真的不能帮你什么。简单粗暴点看原型链吧，想点与代码无关的事，比如人、妖以及人妖。

1）人是人他妈生的，妖是妖他妈生的。人和妖都是对象实例，而人他妈和妖他妈就是原型。原型也是对象，叫原型对象。



2）人他妈和人他爸啪啪啪能生出一堆人宝宝、妖他妈和妖他爸啪啪啪能生出一堆妖宝宝，啪啪啪就是构造函数，俗称造人。



3）人他妈会记录啪啪啪的信息，所以可以通过人他妈找到啪啪啪的信息，也就是说能通过原型对象找到构造函数。

4）人他妈可以生很多宝宝，但这些宝宝只有一个妈妈，这就是原型的唯一性。

5）人他妈也是由人他妈他妈生的，通过人他妈找到人他妈他妈，再通过人他妈他妈找到人他妈他妈……，这个关系叫做原型链。

6）原型链并不是无限的，当你通过人他妈一直往上找，最后发现你会发现人他妈他妈他妈……的他妈都不是人，也就是原型链最终指向null。

7）人他妈生的人会有人的样子，妖他妈生的妖会有妖的丑陋，这叫继承。



8）你继承了你妈的肤色，你妈继承了你妈他妈的肤色，你妈他妈……，这就是原型链的继承。

9）你没有家，那你家指的就是你妈家；你妈也没有家，那你家指的就是你妈他妈家……这就是原型链的向上搜索。

10）你会继承你妈的样子，但是你也可以去染发洗剪吹，就是说对象的属性可以自定义，会覆盖继承得到的属性。

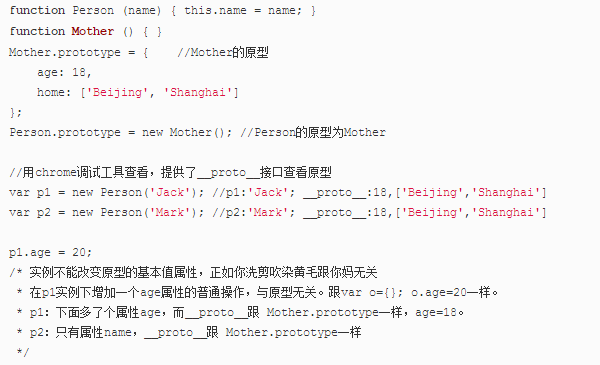
11）虽然你洗剪吹了染成黄毛了，但你不能改变你妈的样子，你妈生的弟弟妹妹跟你的黄毛洗剪吹没一点关系，就是说对象实例不能改动原型的属性。

12）但是你家被你玩火烧了的话，那就是说你家你妈家你弟们家都被烧了，这就是原型属性的共享。

13）你妈外号阿珍，邻居大娘都叫你阿珍儿，但你妈头发从飘柔做成了金毛狮王后，隔壁大婶都改口叫你金毛狮王子，这叫原型的动态性。

14）你妈爱美，又跑到韩国整形，整到你妈他妈都认不出来，即使你妈头发换回飘柔了，但隔壁邻居还是叫你金毛狮王子。因为没人认出你妈，整形后的你妈已经回炉再造了，这就是原型的整体重写。

上代码









现在再来说说 p1.age = 20、p1.home = ['Hangzhou', 'Guangzhou'] 和  p1.home[0] = ‘Shenzhen’ 的区别。 p1.home[0] = ‘Shenzhen’;  总结一下是 p1.object.method，p1.object.property 这样的形式。

p1.age = 20;  p1.home = ['Hangzhou', 'Guangzhou'];这两句还是比较好理解的，先忘掉原型吧，想想我们是怎么为一个普通对象增加属性的：



那为什么 p1.home[0] = ‘Shenzhen’ 不会在 p1 下创建一个 home 数组属性，然后将其首位设为 ’Shenzhen’呢？ 我们还是先忘了这个，想想上面的obj对象，如果写成这样： var obj.name = ‘xxx’, obj.num = [100, 200]，能得到你要的结果吗？ 显然，除了报错你什么都得不到。因为obj还未定义，又怎么能往里面加入东西呢？同理，p1.home[0]中的 home 在 p1 下并未被定义，所以也不能直接一步定义 home[0] 了。如果要在p1下创建一个 home 数组，当然是这么写了：



而之所以 p1.home[0] = ‘Shenzhen’ 不直接报错，是因为在原型链中有一个搜索机制。当我们输入 p1.object 的时候，原型链的搜索机制是先在实例中搜索相应的值，找不到就在原型中找，还找不到就再往上一级原型中搜索……一直到了原型链的终点，就是到null还没找到的话，就返回一个 undefined。当我们输入 p1.home[0] 的时候，也是同样的搜索机制，先搜索 p1 看有没有名为 home 的属性和方法，然后逐级向上查找。最后我们在Mother的原型里面找到了，所以修改他就相当于修改了 Mother 的原型啊。

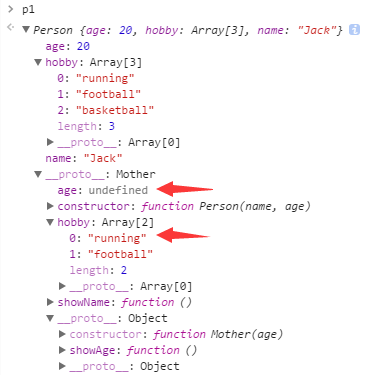
一句话概括：p1.home[0] = ‘Shenzhen’  等同于  Mother.prototype.home[0] = ’Shenzhen’。

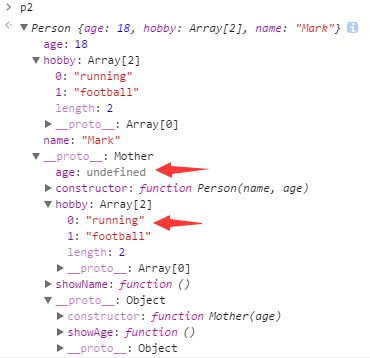
由上面的分析可以知道，原型链继承的主要问题在于属性的共享，很多时候我们只想共享方法而并不想要共享属性，理想中每个实例应该有独立的属性。因此，原型继承就有了下面的两种改良方式：

**1）组合继承**



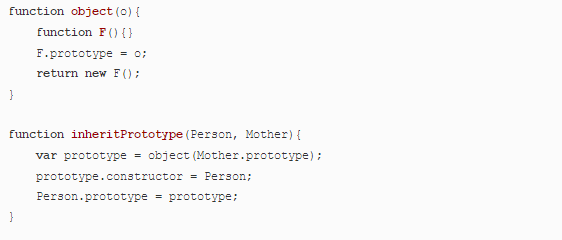
结果是酱紫的：





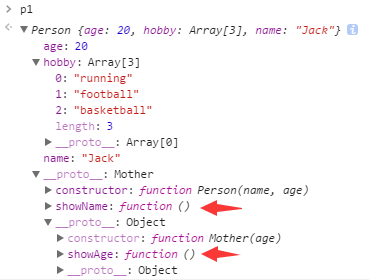
这里第一次执行的时候，得到 Person.prototype.age = undefined, Person.prototype.hobby = ['running','football']，第二次执行也就是 var p1 = new Person(‘Jack’, 20) 的时候，得到 p1.age =20, p1.hobby = ['running','football']，push后就变成了 p1.hobby = ['running','football', 'basketball']。其实分辨好 this 的变化，理解起来也是比较简单的，把 this 简单替换一下就能得到这个结果了。 如果感觉理解起来比较绕的话，试着把脑子里面的概念扔掉吧，把自己当浏览器从上到下执行一遍代码，结果是不是就出来了呢？

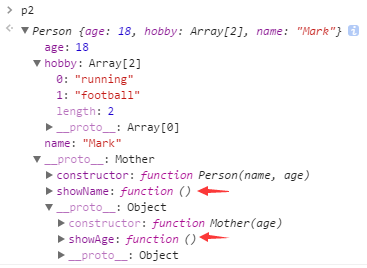
通过第二次执行原型的构造函数 Mother()，我们在对象实例中复制了一份原型的属性，这样就做到了与原型属性的分离独立。细心的你会发现，我们第一次调用 Mother()，好像什么用都没有呢，能不调用他吗？可以，就有了下面的寄生组合式继承。





结果是酱紫的：





原型中不再有 age 和 hobby 属性了，只有两个方法，正是我们想要的结果！

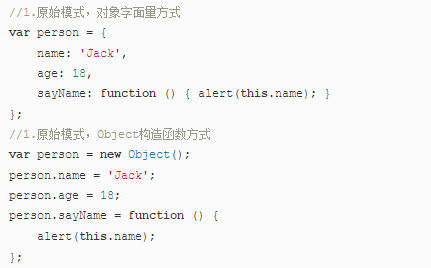
关键点在于 object(o) 里面，这里借用了一个临时对象来巧妙避免了调用new Mother()，然后将原型为 o 的新对象实例返回，从而完成了原型链的设置。很绕，对吧，那是因为我们不能直接设置 Person.prototype = Mother.prototype 啊。

## 小结

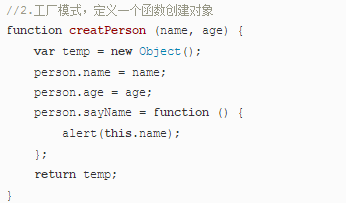
说了这么多，其实核心只有一个：属性共享和独立的控制，当你的对象实例需要独立的属性，所有做法的本质都是在对象实例里面创建属性。若不考虑太多，你大可以在Person里面直接定义你所需要独立的属性来覆盖掉原型的属性。总之，使用原型继承的时候，要对于原型中的属性要特别注意，因为他们都是牵一发而动全身的存在。

下面简单罗列下js中创建对象的各种方法，现在最常用的方法是组合模式。

1）原始模式

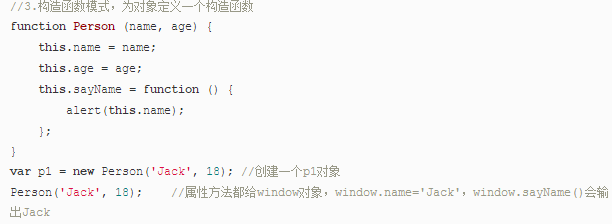


2）工厂模式



工厂模式就是批量化生产，简单调用就可以进入造人模式（啪啪啪……）。指定姓名年龄就可以造一堆小宝宝啦，解放双手。但是由于是工厂暗箱操作的，所以你不能识别这个对象到底是什么类型、是人还是狗傻傻分不清（instanceof 测试为 Object），另外每次造人时都要创建一个独立的temp对象，代码臃肿，雅蠛蝶啊。

3）构造函数



构造函数与C++、JAVA中类的构造函数类似，易于理解，另外Person可以作为类型识别（instanceof 测试为 Person 、Object）。但是所有实例依然是独立的，不同实例的方法其实是不同的函数。这里把函数两个字忘了吧，把sayName当做一个对象就好理解了，就是说张三的 sayName 和李四的 sayName是不同的存在，但显然我们期望的是共用一个 sayName 以节省内存。

4）原型模式



这里需要注意的是原型属性和方法的共享，即所有实例中都只是引用原型中的属性方法，任何一个地方产生的改动会引起其他实例的变化。

5）混合模式（构造+原型）



做法是将需要独立的属性方法放入构造函数中，而可以共享的部分则放入原型中，这样做可以最大限度节省内存而又保留对象实例的独立性。

**原型**

刚开始学JavaScript是这样写代码：

var decimalDigits = 2,

tax = 5;

function add(x, y) {

return x + y;

}

function subtract(x, y) {

return x - y;

}

alert(add(1, 3));

后来这样写：

//原型使用方式1

var Calculator = function (decimalDigits, tax) {

this.decimalDigits = decimalDigits;

this.tax = tax;

};

Calculator.prototype = {

add: function (x, y) {

return x + y;

},

subtract: function (x, y) {

return x - y;

}

};

alert((new Calculator()).add(1, 3));

//原型使用方式2

var Calculator = function (decimalDigits, tax) {

this.decimalDigits = decimalDigits;

this.tax = tax;

};

Calculator.prototype = function () {

add = function (x, y) {

return x + y;

},

subtract = function (x, y) {

return x - y;

}

return {

add: add,

subtract: subtract

}

} ();

alert((new Calculator()).add(11, 3));

第二种方式是，在赋值原型prototype的时候使用function立即执行的表达式来赋值，即如下格式：

Calculator.prototype = function () { } ();

它的好处在前面的帖子里已经知道了，就是可以封装私有的function，通过return的形式暴露出简单的使用名称，以达到public/private的效果。

**分步声明**

上述使用原型的时候，有一个限制就是一次性设置了原型对象，我们再来说一下如何分来设置原型的每个属性吧。

var BaseCalculator = function() {

this.decimalDigits = 2;

};

BaseCalculator.prototype = {

add: function(x, y) {

return x + y;

},

subtract: function(x, y) {

return x - y;

}

};

var Calculator = function () {

//为每个实例都声明一个税收数字

this.tax = 5;

};

Calculator.prototype = new BaseCalculator();

我们可以看到Calculator的原型是指向到BaseCalculator的一个实例上，目的是让Calculator集成它的add(x,y)和subtract(x,y)这2个function，还有一点要说的是，由于它的原型是BaseCalculator的一个实例，所以不管你创建多少个Calculator对象实例，他们的原型指向的都是同一个实例。

var calc = new Calculator();

alert(calc.add(1, 1));

//BaseCalculator 里声明的decimalDigits属性，在 Calculator里是可以访问到的

alert(calc.decimalDigits);

上面的代码，运行以后，我们可以看到因为Calculator的原型是指向BaseCalculator的实例上的，所以可以访问他的decimalDigits属性值，那如果我不想让Calculator访问BaseCalculator的构造函数里声明的属性值，应该这么写：

var Calculator = function () {

this.tax= 5;

};

Calculator.prototype = BaseCalculator.prototype;

通过将BaseCalculator的原型赋给Calculator的原型，这样你在Calculator的实例上就访问不到那个decimalDigits值了，如果你访问如下代码，那将会提示出错。

var calc = new Calculator();

alert(calc.add(1, 1));

alert(calc.decimalDigits);//undefined

**重写原型**

在使用第三方JS类库的时候，往往有时候他们定义的原型方法是不能满足我们的需要，但是又离不开这个类库，所以这时候我们就需要重写他们的原型中的一个或者多个属性或function，我们可以通过继续声明的同样的add代码的形式来达到覆盖重写前面的add功能，代码如下：

//覆盖前面Calculator的add() function

Calculator.prototype.add = function (x, y) {

return x + y + this.tax;

};

var calc = new Calculator();

alert(calc.add(1, 1));

**原型链**

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';

// 修正Bar.prototype.constructor为Bar本身

Bar.prototype.constructor = Bar;

var test = new Bar() // 创建Bar的一个新实例

// 原型链

test [Bar的实例]

Bar.prototype [Foo的实例]

{ foo: 'Hello World' }

Foo.prototype

{method: ...};

Object.prototype

{toString: ... /\* etc. \*/};

**属性查找**

当查找一个对象的属性时，JavaScript 会向上遍历原型链，直到找到给定名称的属性为止，到查找到达原型链的顶部 - 也就是 Object.prototype - 但是仍然没有找到指定的属性，就会返回 undefined，我们来看一个例子：

function foo() {

this.add = function (x, y) {

return x + y;

}

}

foo.prototype.add = function (x, y) {

return x + y + 10;

}

Object.prototype.subtract = function (x, y) {

return x - y;

}

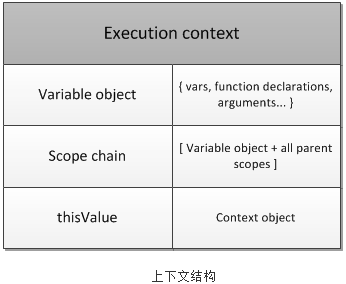
var f = new foo();

alert(f.add(1, 2)); //结果是3，而不是13

alert(f.subtract(1, 2)); //结果是-1

**执行上下文**

一个执行上下文可以抽象的理解成object。没一个执行的上下文都有一系列的属性（上下文状态），用来追踪关联代码的执行进度。下图是一个context的结构：



除了这三个属性（变量对象variable object、this指针this value、作用域链scope chain），执行上下文根据其具体实现还可以具有任意额外属性。一下详细介绍这三个属性：

**变量对象**

变量对象是一个抽象的概念，不同的上下文中，它表示使用不同的object。例如，在gloable全局上下文中，变量对象也是全局对象自身[global object]，这就是为什么我们可以通过全局变量的属性来指向全局变量。

var foo = 10;

function bar() {} // // 函数声明

(function baz() {}); // 函数表达式

console.log(

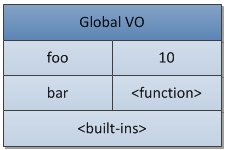
this.foo == foo, // true

window.bar == bar // true

);

console.log(baz); // 引用错误，baz没有被定义

全局上下文中的变量对象（VO）有如下属性：



如上所示，函数“baz”作为函数表达式则不被包含于变量对象。这就是在函数外部尝试访问产生引用错误(ReferenceError) 的原因。请注意，ECMAScript和其他语言相比(比如C/C++)，仅有函数能够创建新的作用域。在函数内部定义的变量与内部函数，在外部非直接可见并且不污染全局对象。使用 eval 的时候，我们同样会使用一个新的(eval创建)执行上下文。eval会使用全局变量对象或调用者的变量对象(eval的调用来源)。

那函数以及自身的变量对象又是怎样的呢?在一个函数上下文中，变量对象被表示为活动对象(activation object)。

**活动对象**

当函数被调用者激活，这个特殊的活动对象(activation object) 就被创建了。它包含普通参数(formal parameters) 与特殊参数(arguments)对象(具有索引属性的参数映射表)。活动对象在函数上下文中作为变量对象使用。

即：函数的变量对象保持不变，但除去存储变量与函数声明之外，还包含特殊对象arguments 。考虑下面的情况：

function foo(x, y) {

var z = 30;

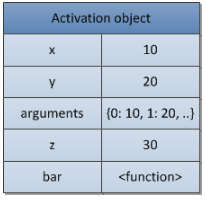
function bar() {} // 函数声明

(function baz() {}); // 函数表达式

}

foo(10, 20);

“foo”函数上下文的下一个激活对象(AO)如下图所示：



同样道理，function expression（函数表达式）不在AO的行列。

**作用域链**

作用域链是一个对象列表，用于检索上下文代码中出现的标识符（identifiers）（变量名称、函数声明和普通参数）。其原理与原型链类似，如果变量在自己的作用域中没有，那么它会寻找父级的，直至最顶层。

在一般情况下，一个作用域链包括父级变量对象（variable object）（作用域链的顶部）、函数自身变量VO和活动对象（activation object）。不过，有些情况下也会包含其它的对象，例如在执行期间，动态加入作用域链中的—例如with或者catch语句。[译注：with-objects指的是with语句，产生的临时作用域对象；catch-clauses指的是catch从句，如catch(e)，这会产生异常对象，导致作用域变更]。

当查找标识符的时候，会从作用域链的活动对象部分开始查找，然后(如果标识符没有在活动对象中找到)查找作用域链的顶部，循环往复，就像作用域链那样。

var x = 10;

(function foo() {

var y = 20;

(function bar() {

var z = 30;

// "x"和"y"是自由变量

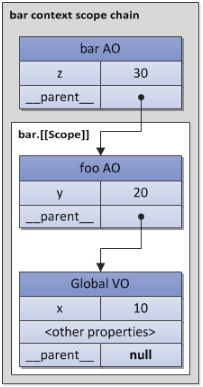
// 会在作用域链的下一个对象中找到（函数”bar”的互动对象之后）

console.log(x + y + z);

})();

})();

我们假设作用域链的对象联动是通过一个叫做\_\_parent\_\_的属性，它是指向作用域链的下一个对象。这可以在Rhino Code中测试一下这种流程，这种技术也确实在ES5环境中实现了(有一个称为outer链接)。当然也可以用一个简单的数据来模拟这个模型。使用\_\_parent\_\_的概念，我们可以把上面的代码演示成如下的情况。（因此，父级变量是被存在函数的[[Scope]]属性中的）。



在代码执行过程中，如果使用with或者catch语句就会改变作用域链。而这些对象都是一些简单对象，他们也会有原型链。这样的话，作用域链会从两个维度来搜寻。

1）首先在原本的作用域链

2）每一个链接点的作用域的链（如果这个链接点是有prototype的话）

看下面的例子：

Object.prototype.x = 10;

var w = 20;

var y = 30;

// 在SpiderMonkey全局对象里

// 例如，全局上下文的变量对象是从"Object.prototype"继承到的

// 所以我们可以得到“没有声明的全局变量”

// 因为可以从原型链中获取

console.log(x); // 10

(function foo() {

// "foo" 是局部变量

var w = 40;

var x = 100;

// "x" 可以从"Object.prototype"得到，注意值是10哦

// 因为{z: 50}是从它那里继承的

with ({z: 50}) {

console.log(w, x, y , z); // 40, 10, 30, 50

}

// 在"with"对象从作用域链删除之后

// x又可以从foo的上下文中得到了，注意这次值又回到了100哦

// "w" 也是局部变量

console.log(x, w); // 100, 40

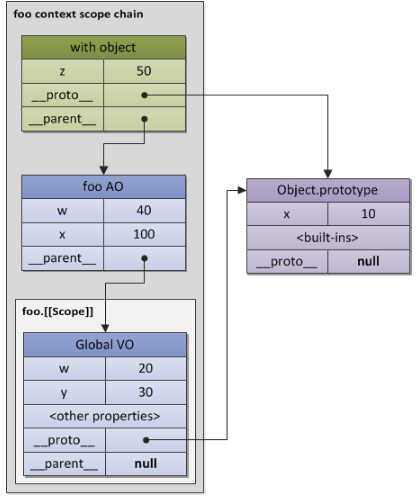
// 在浏览器里

// 我们可以通过如下语句来得到全局的w值

console.log(window.w); // 20

})();

我们就会有如下结构图示。这表示，在我们去搜寻\_\_parent\_\_之前，首先会去\_\_proto\_\_的链接中。



注意，不是所有的全局对象都是由Object.prototype继承而来的。

**This指针**

this是执行上下文环境的一个属性，而不是某个变量对象的属性。和变量不同，this没有一个类似搜寻变量的过程。当在代码中使用了this，这个this的值就直接从执行的上下文中获取了，而不会从作用域链中搜寻。this的值只取决于进入上下文时的情况。

在global context（全局上下文）中，this的值就是指全局这个对象：

var x = 10;

console.log(

x, // 10

this.x, // 10

window.x // 10

);

在函数上下文[function context]中，this会可能会根据每次的函数调用而成为不同的值.this会由每一次caller提供,caller是通过调用表达式[call expression]产生的（也就是这个函数如何被激活调用的）。例如，下面的例子中foo就是一个callee，在全局上下文中被激活。下面的例子就表明了不同的caller引起this的不同。

// "foo"函数里的alert没有改变

// 但每次激活调用的时候this是不同的

function foo() {

alert(this);

}

// caller 激活 "foo"这个callee，

// 并且提供"this"给这个 callee

foo(); // 全局对象

foo.prototype.constructor(); // foo.prototype

var bar = {

baz: foo

};

bar.baz(); // bar

(bar.baz)(); // also bar

(bar.baz = bar.baz)(); // 这是一个全局对象

(bar.baz, bar.baz)(); // 也是全局对象

(false || bar.baz)(); // 也是全局对象

var otherFoo = bar.baz;

otherFoo(); // 还是全局对象

**VO的声明**

当前ECMAScript规范指出独立的作用域只能通过函数代码类型的执行上下文创建：

var a = 10; // 全局上下文中的变量

(function () {

var b = 20; // function上下文中的局部变量

})();

alert(a); // 10

alert(b); // 全局变量 "b" 没有声明

for (var k in {a: 1, b: 2}) {

alert(k);

}

alert(k); // 尽管循环已经结束但变量k依然在当前作用域

如果变量与执行上下文相关，那变量自己应该知道它的数据存储在哪里，并且知道如何访问。这种机制称为变量对象(variable object)。

变量对象(缩写为VO)是一个与执行上下文相关的特殊对象，它存储着在上下文中声明的以下内容：

1）变量 (var, 变量声明);

2）函数声明 (FunctionDeclaration, 缩写为FD);

3）函数的形参

举例来说，我们可以用普通的ECMAScript对象来表示一个变量对象：VO = {};

就像我们所说的, VO就是执行上下文的属性(property)：

activeExecutionContext = {

VO: {

// 上下文数据（var, FD, function arguments)

}

};

只有全局上下文的变量对象允许通过VO的属性名称来间接访问(因为在全局上下文里，全局对象自身就是变量对象，稍后会详细介绍)，在其它上下文中是不能直接访问VO对象的，因为它只是内部机制的一个实现。

当我们声明一个变量或一个函数的时候，和我们创建VO新属性的时候一样没有别的区别（即：有名称以及对应的值）。例如：

var a = 10;

function test(x) {

var b = 20;

};

test(30);

对应的变量对象是：

// 全局上下文的变量对象

VO(globalContext) = {

a: 10,

test: <reference to function>

};

// test函数上下文的变量对象

VO(test functionContext) = {

x: 30,

b: 20

};

在具体实现层面(以及规范中)变量对象只是一个抽象概念。(从本质上说，在具体执行上下文中，VO名称是不一样的，并且初始结构也不一样。

注：只需清楚函数作用域，以及全局上下文变量对象和函数上下文变量对象。全局上下文变量对象允许通过VO的属性名称来间接访问，而后者不行。

对于所有类型的执行上下文来说，变量对象的一些操作(如变量初始化)和行为都是共通的。从这个角度来看，把变量对象作为抽象的基本事物来理解更为容易。同样在函数上下文中也定义和变量对象相关的额外内容。

抽象变量对象VO (变量初始化过程的一般行为)

║

╠══> 全局上下文变量对象GlobalContextVO

║ (VO === this === global)

║

╚══> 函数上下文变量对象FunctionContextVO

(VO === AO, 并且添加了<arguments>和<formal parameters>)

全局对象(Global object)是在进入任何执行上下文之前就已经创建了的对象。这个对象只存在一份，它的属性在程序中任何地方都可以访问，全局对象的生命周期终止于程序退出那一刻。

**全局上下文中的变量对象：**

全局对象(Global object) 是在进入任何执行上下文之前就已经创建了的对象。这个对象只存在一份，它的属性在程序中任何地方都可以访问，全局对象的生命周期终止于程序退出那一刻。

全局对象初始创建阶段将Math、String、Date、parseInt作为自身属性，等属性初始化，同样也可以有额外创建的其它对象作为属性（其可以指向到全局对象自身）。例如，在DOM中，全局对象的window属性就可以引用全局对象自身(当然，并不是所有的具体实现都是这样)：

global = {

Math: <...>,

String: <...>

...

...

window: global //引用自身

};

当访问全局对象的属性时通常会忽略掉前缀，这是因为全局对象是不能通过名称直接访问的。不过我们依然可以通过全局上下文的this来访问全局对象，同样也可以递归引用自身。例如，DOM中的window。综上所述，代码可以简写为：

String(10); // 就是global.String(10);

// 带有前缀

window.a = 10; // === global.window.a = 10 === global.a = 10;

this.b = 20; // global.b = 20;

因此，回到全局上下文中的变量对象——在这里，变量对象就是全局对象自己：

VO(globalContext) === global;

非常有必要要理解上述结论，基于这个原理，在全局上下文中声明的对应，我们才可以间接通过全局对象的属性来访问它（例如，事先不知道变量名称）。

var a = new String('test');

alert(a); // 直接访问，在VO(globalContext)里找到："test"

alert(window['a']); // 间接通过global访问：global === VO(globalContext): "test"

alert(a === this.a); // true

var aKey = 'a';

alert(window[aKey]); // 间接通过动态属性名称访问："test"

**函数上下文中的变量对象：**

在函数执行上下文中，VO是不能直接访问的，此时由活动对象(activation object,缩写为AO)扮演VO的角色。

VO(functionContext) === AO;

Arguments对象是活动对象的一个属性，它包括如下属性：

1. callee — 指向当前函数的引用
2. length — 真正传递的参数（实参）个数
3. properties-indexes (字符串类型的整数) 属性的值就是函数的参数值(按参数列表从左到右排列)。 properties-indexes内部元素的个数等于arguments.length. properties-indexes 的值和实际传递进来的参数之间是共享的。

function foo(x, y, z) {

// 声明的函数参数数量arguments (x, y, z)

alert(foo.length); // 3

// 真正传进来的参数个数(only x, y)

alert(arguments.length); // 2

// 参数的callee是函数自身

alert(arguments.callee === foo); // true

// 参数共享

alert(x === arguments[0]); // true

alert(x); // 10

arguments[0] = 20;

alert(x); // 20

x = 30;

alert(arguments[0]); // 30

// 不过，没有传进来的参数z，和参数的第3个索引值是不共享的

z = 40;

alert(arguments[2]); // undefined

arguments[2] = 50;

alert(z); // 40

}

foo(10, 20);

这个例子的代码，在当前版本的Google Chrome浏览器里有一个bug — 即使没有传递参数z，z和arguments[2]仍然是共享的。

**执行上下文的两个阶段**

执行上下文的代码被分为两个基本阶段来处理：

1）进入执行上下文；2）执行代码。

**进入执行上下文：**

当进入执行上下文(代码执行之前)时，VO里已经包含了下列属性：

1）函数的所有形参(如果我们是在函数执行上下文中) — 由名称和对应值组成的一个变量对象的属性被创建；没有传递对应参数的话，那么由名称和undefined值组成的一种变量对象的属性也将被创建。

2）所有函数声明(FunctionDeclaration, FD) —由名称和对应值（函数对象(function-object)）组成一个变量对象的属性被创建；如果变量对象已经存在相同名称的属性，则完全替换这个属性。

3）所有变量声明(var, VariableDeclaration) — 由名称和对应值（undefined）组成一个变量对象的属性被创建；如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性。

来个栗子：

function test(a, b) {

var c = 10;

function d() {}

var e = function \_e() {};

(function x() {});

}

test(10); // call

当进入带有参数10的test函数上下文时，AO表现为如下：

AO(test) = {

a: 10,

b: undefined,

c: undefined,

d: <reference to FunctionDeclaration "d">

e: undefined

};

注意，AO里并不包含函数“x”。这是因为“x” 是一个函数表达式(FunctionExpression, 缩写为 FE) 而不是函数声明，函数表达式不会影响VO。 不管怎样，函数“\_e” 同样也是函数表达式，但是就像我们下面将看到的那样，因为它分配给了变量 “e”，所以它可以通过名称“e”来访问。

**执行代码：**

这个周期内，AO/VO已经拥有了属性(不过，并不是所有的属性都有值，大部分属性的值还是系统默认的初始值undefined )。还是前面那个例子，AO/VO在代码解释期间被修改如下：

AO['c'] = 10;

AO['e'] = <reference to FunctionExpression "\_e">;

再次注意，因为FunctionExpression“\_e”保存到了已声明的变量“e”上，所以它仍然存在于内存中。而FunctionExpression “x”却不存在于AO/VO中，也就是说如果我们想尝试调用“x”函数，不管在函数定义之前还是之后，都会出现一个错误“x is not defined”，未保存的函数表达式只有在它自己的定义或递归中才能被调用（第三遍了）。

另一个栗子：

alert(x); // function

var x = 10;

alert(x); // 10

x = 20;

function x() {};

alert(x); // 20

为什么第一个alert “x” 的返回值是function，而且它还是在“x” 声明之前访问的“x” 的？为什么不是10或20呢？因为，根据规范函数声明是在当进入上下文时填入的；同一周期，在进入上下文的时候还有一个变量声明“x”，那么正如我们在上一个阶段所说，变量声明在顺序上跟在函数声明和形式参数声明之后，而且在这个进入上下文阶段，变量声明不会干扰VO中已经存在的同名函数声明或形式参数声明，因此，在进入上下文时，VO的结构如下：

VO = {};

VO['x'] = <reference to FunctionDeclaration "x">

// 找到var x = 10;

// 如果function "x"没有已经声明的话

// 这时候"x"的值应该是undefined

// 但是这个case里变量声明没有影响同名的function的值

VO['x'] = <the value is not disturbed, still function>

紧接着在代码执行阶段，VO作如下修改：

VO['x'] = 10;

VO['x'] = 20;

可以在第二、三个alert中看到这个效果。

在下面的例子里我们可以再次看到，变量是在进入上下文阶段放入VO中的。(因为，虽然else部分代码永远不会执行，但是不管怎样，变量“b”仍然存在于VO中。)

if (true) {

var a = 1;

} else {

var b = 2;

}

alert(a); // 1

alert(b); // undefined,不是b没有声明，而是b的值是undefined

**任何时候，变量只能通过使用var关键字才能声明。**

如下面的赋值语句：a=10;

这句话仅仅是给全局变量创建了一个新的属性，而不是声明了一个变量。

证据

1.var声明的变量用delete操作符无法删除，而用上述方式声明的可以。

2. alert(a); // undefined

alert(b); // "b" 没有声明

b = 10;

var a = 20;

所有根源仍然是VO和进入上下文阶段和代码执行阶段。进入上下文阶段：

VO = {

a: undefined

};

我们可以看到，因为“b”不是一个变量，所以在这个阶段根本就没有“b”，“b”将只在代码执行阶段才会出现(但是在我们这个例子里，还没有到那就已经出错了)。

**函数的生命周期：创建和激活阶段（调用时）**

**函数创建**

var x = 10;

function foo() {

var y = 20;

alert(x + y);

}

foo(); // 30

我们看到变量“y”在函数“foo”中定义（意味着它在foo上下文的AO中），但是变量“x”并未在“foo”上下文中定义，相应地，它也不会添加到“foo”的AO中。乍一看，变量“x”相对于函数“foo”根本就不存在；但正如我们在下面看到的——也仅仅是“一瞥”，我们发现，“foo”上下文的活动对象中仅包含一个属性－－“y”。

fooContext.AO = {

y: undefined // undefined – 进入上下文的时候是20 – at activation

};

函数“foo”如何访问到变量“x”？理论上函数应该能访问一个更高一层上下文的变量对象。实际上它正是这样，这种机制是通过函数内部的[[scope]]属性来实现的。

[[scope]]是所有父变量对象的层级链，处于当前函数上下文之上，在函数创建时存于其中。

注意这重要的一点－－[[scope]]在函数创建时被存储－－静态（不变的），永远永远，直至函数销毁。即：函数可以永不调用，但[[scope]]属性已经写入，并存储在函数对象中。

另外一个需要考虑的是－－与作用域链对比，[[scope]]是函数的一个属性而不是上下文。考虑到上面的例子，函数“foo”的[[scope]]如下：

foo.[[Scope]] = [

globalContext.VO // === Global

];

举例来说，我们用通常的ECMAScript 数组展现作用域和[[scope]]。

继续，我们知道在函数调用时进入上下文，这时候活动对象被创建，this和作用域（作用域链）被确定。让我们详细考虑这一时刻。

**函数激活**

正如在定义中说到的，进入上下文创建AO/VO之后，上下文的Scope属性（变量查找的一个作用域链）作如下定义： Scope = AO|VO + [[Scope]]

上面代码的意思是：活动对象是作用域数组的第一个对象，即添加到作用域的前端。

Scope = [AO].concat([[Scope]]);

这个算法的返回值中，我们总有一个引用类型，它的base组件是相应的变量对象（或若未找到则为null）,属性名组件是向上查找的标示符的名称。引用类型的详细信息在后面已讨论。

标识符解析过程包含与变量名对应属性的查找，即作用域中变量对象的连续查找，从最深的上下文开始，绕过作用域链直到最上层。

这样一来，在向上查找中，一个上下文中的局部变量较之于父作用域的变量拥有较高的优先级。万一两个变量有相同的名称但来自不同的作用域，那么第一个被发现的是在最深作用域中。

一言不合，又来一个栗子：

var x = 10;

function foo() {

var y = 20;

function bar() {

var z = 30;

alert(x + y + z);

}

bar();

}

foo(); // 60

对此，我们有如下的变量/活动对象，函数的的[[scope]]属性以及上下文的作用域链：

全局上下文的变量对象：

globalContext.VO === Global = {

x: 10

foo: <reference to function>

};

在“foo”创建时，“foo”的[[scope]]属性是：

foo.[[Scope]] = [

globalContext.VO

];

在“foo”激活时（进入上下文），“foo”上下文的活动对象是：

fooContext.AO = {

y: 20,

bar: <reference to function>

};

“foo”上下文的作用域链为：

fooContext.Scope = fooContext.AO + foo.[[Scope]] // i.e.:

fooContext.Scope = [

fooContext.AO,

globalContext.VO

];

内部函数“bar”创建时，其[[scope]]为：

bar.[[Scope]] = [

fooContext.AO,

globalContext.VO

];

在“bar”激活时，“bar”上下文的活动对象为：

barContext.AO = {

z: 30

};

“bar”上下文的作用域链为：

barContext.Scope = barContext.AO + bar.[[Scope]] // i.e.:

barContext.Scope = [

barContext.AO,

fooContext.AO,

globalContext.VO

];

对“x”、“y”、“z”的标识符解析如下：

- "x"

-- barContext.AO // not found

-- fooContext.AO // not found

-- globalContext.VO // found - 10

- "y"

-- barContext.AO // not found

-- fooContext.AO // found - 20

- "z"

-- barContext.AO // found - 30

**那些年我们一起清除过的浮动**

1）清除浮动：清除对应的单词是clear，对应的CSS属性是clear:left|right|botn|none；

2）闭合浮动：更确切的含义是使浮动元素闭合，从而减少浮动带来的影响。

CSS的定位机制：普通流、浮动、绝对定位。

**闭合浮动**

1）添加额外的标签

<div class="warp" id="float1">  
 <h2>1）添加额外标签</h2>  
 <div class="main left">.main{float:left;}</div>  
 <div class="side left">.side{float:right;}</div>  
 <div style="clear:both;"></div>  
</div>  
<div class="footer">.footer</div>

通过在浮动元素末尾添加一个空的标签例如<div style="clear:both;"></div>，其他标签</br>也可以。这种方法易于理解和掌握，但是会添加无意义的标签，有违表现与结构相分离的原则，不利于后期维护。

2）使用br标签和其自身的HTML属性

<div class="warp" id="float2">

<h2>2）使用 br标签和其自身的 html属性</h2>

<div class="main left">.main{float:left;}</div>

<div class="side left">.side{float:right;}</div>

<br clear="all" />

</div>  
<div class="footer">.footer</div>

同样有违表现与结构相分离，不推荐使用。

3）父元素overflow:hidden，IE中还需要触发hasLayout，例如zoom:1

优点是代码量小，缺点是内容增多时导致内容不会自动换行而被隐藏掉，无法显示要溢出的元素。不推荐使用。

4）父元素设置overflow:auto，同上

5）父元素也设置浮动。这样会使得与父元素相邻元素的布局也受到影响，不推荐使用

6）父元素display:table。这样改变了盒模型的属性，不推荐使用

7）使用after伪元素（加在父元素上）

.clearfix:after {

content: ".";

display: block;

height: 0;

visibility: hidden;

clear: both;

}

.clearfix {

\*zoom: 1;

}

复用不当会导致代码量增加。

通过对比，我们不难发现，其实以上列举的方法，无非有两类：

　　其一，通过在浮动元素的末尾添加一个空元素，设置 clear：both属性，after伪元素其实也是通过 content 在元素的后面生成了内容为一个点的块级元素；

　　其二，通过设置父元素 overflow 或者display：table 属性来闭合浮动，我们来探讨一下这里面的原理。

**更好的解决方案**

对于使用after伪元素：

1) display:block 使生成的元素以块级元素显示,占满剩余空间;

2) height:0 避免生成内容破坏原有布局的高度。

3) visibility:hidden 使生成的内容不可见，并允许可能被生成内容盖住的内容可以进行点击和交互;

4）通过 content:"."生成内容作为最后一个元素，至于content里面是点还是其他都是可以的，例如oocss里面就有经典的content:"XXXXXXXXX",有些版本可能content 里面内容为空,一丝冰凉是不推荐这样做的,firefox直到7.0 content:”" 仍然会产生额外的空隙；

5）zoom：1 触发IE hasLayout。

分析发现，除了clear:both，其他代码都是为了隐藏掉content生成的内容。

方案一

.clearfix:after {

content: "\200B";

display: block;

height: 0;

clear: both;

}

.clearfix {

\*zoom: 1;

}

其中

\200B为“零宽度空格”，这个字符本身不可见，可以省略掉visibility:hidden 这一句。

方案二

/\* For modern browsers \*/

. clearfix:before,. clearfix:after {

content:"";

display:table;

}

. clearfix:after { clear:both; }/\* For IE 6/7 (trigger hasLayout) \*/

. clearfix{ zoom:1; }

上面的方法用到了 :before 伪元素，很多人对这个有些迷惑，到底我什么时候需要用 before 呢？为什么方案一没有呢？其实它是用来处理 margin 边距重叠的，由于内部元素 float 创建了BFC，导致内部元素的margin-top和 上一个盒子的 margin-bottom 发生叠加。如果这不是你所希望的，那么就可以加上 before，如果只是单纯的闭合浮动，after 就够了！