1 JavaScript基础数据类型

JavaScript数据类型包括原始类型和引用类型，原始类型有五个:

Number（数值） String（字符串） Boolean（布尔） Null（空） Undefined（未定义）

引用类型有一个：

Object（对象）

通过typeof(x)可以返回一个变量x的数据类型“number”、“string”、“boolean”、“undefined”、"object"，这里要注意一点：typeof运算符对于null类型返回的是object。

《JavaScript高级程序设计》：

这实际上是JavaScript最初实现中的一个错误，后来被ECMAScript沿用了。现在null被认为是对象的占位符，从而解释了这一矛盾。但是从技术上来说，它仍然是原始值。

2 谈一谈JavaScript作用域链

当执行一段JavaScript代码（全局代码或函数）时，JavaScript引擎会创建为其创建一个作用域又称为执行上下文（Execution Context），在页面加载后会首先创建一个全局的作用域，然后每执行一个函数，会建立一个对应的作用域，从而形成了一条作用域链。每个作用域都有一条对应的作用域链，链头是全局作用域，链尾是当前函数作用域。

作用域链的作用是用于解析标识符，当函数被创建时（不是执行），会将this、arguments、命名参数和该函数中的所有局部变量添加到该当前作用域中，当JavaScript需要查找变量X的时候（这个过程称为变量解析），它首先会从作用域链中的链尾也就是当前作用域进行查找是否有X属性，如果没有找到就顺着作用域链继续查找，直到查找到链头，也就是全局作用域链，仍未找到该变量的话，就认为这段代码的作用域链上不存在x变量，并抛出一个引用错误（ReferenceError）的异常。

3 如何理解JavaScript原型链

JavaScript中的每个对象都有一个prototype属性，我们称之为原型，而原型的值也是一个对象，因此它也有自己的原型，这样就串联起来了一条原型链，原型链的链头是object,它的prototype比较特殊，值为null。

原型链的作用是用于对象继承，函数A的原型属性(prototype property)是一个对象，当这个函数被用作构造函数来创建实例时，该函数的原型属性将被作为原型赋值给所有对象实例，比如我们新建一个数组，数组的方法便从数组的原型上继承而来。

当访问对象的一个属性时, 首先查找对象本身, 找到则返回; 若未找到, 则继续查找其原型对象的属性(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，若整个原型链未找到则返回undefined；

4 JavaScript变量声明提前

《JavaScript权威指南》中是这样解释的：JavaScript变量在声明之前已经可用，JavaScript的这个特性被非正式的称为声明提前（hoisting），即JavaScript函数中声明的所有变量（但不涉及赋值）都被“提前”至函数的顶部。

从一个例子来看：

var scope = "global";

function myFunc(){

console.log(scope);

var scope = "local";

}

控制台打印出来的不是“global”而是“undefined”，这是因为在myFunc这个函数的作用域中，局部变量scope声明被提前至函数顶部，而此时，scope仅声明，未赋值，因此输出undefined。实际上，上面的代码和下面的效果是一样的：

var scope = "global";

function myFunc(){

var scope;

console.log(scope);

scope = "local";

}

5 如何理解和应用JavaScript闭包

关于闭包具体的定义文献中给的概念很抽象，我认为闭包是一种使函数能够都去其它函数的局部变量的语法机制。

举个例子：

function outFunc(){

var name = "Vicfeel";

function inFunc(){

console.log(name);

}

return inFunc();

}

outFunc(); //控制台显示"Vicfeel"

这这个例子我们可以看出，在函数inFunc中依然可以访问outFunc的局部变量name。

闭包应用举例，模拟类的私有属性,利用闭包的性质，局部变量只有在sayAge方法中才可以访问，而name在外部也访问，从而实现了类的私有属性。

function User(){

this.name = "Vicfeel"; //共有属性

var age = 23; //私有属性

this.sayAge=function(){

console.log("my age is " + age);

}

}

var user = new User();

console.log(user.name); //"Vicfeel"

console.log(user.age); //"undefined"

user.sayAge(); //"my age is 23"

要了解详细的闭包，推荐一下 阮一峰的网络日志-学习Javascript闭包（Closure）。

6 new构建对象的本质

function User(){

this.name = "Vicfeel";

this.age = 23;

}

var user = new User();

通过new操作符，实际上在构造函数User中完成了如下操作：

•创建一个新的对象，这个对象的类型是object；

•设置这个新的对象的内部、可访问性和prototype属性为构造函数（指prototype.construtor所指向的构造函数）中设置的；

•执行构造函数；

•返回新创建的对象。

function User(){

//this = {};

//this.constructor = User；

this.name = "Vicfeel";

this.age = 23;

//return this;

}

var user = new User();

如果构造函数默认返回的新创建的this对象，如果手动return 一个变量的话，如果该变量是原始类型则无效，如果是对象，则返回该对象。

7 JavaScript代理

当我们需要对很多元素添加事件的时候，可以通过将事件添加到它们的父节点而将事件委托给父节点来触发处理函数。

比如我们需要向一个ul中动态添加很多个li，需要遍历li逐个添加点击事件

<ul id='list'></ul>

var count = 100;

var ulList = document.getElementById("list");

//动态构建节点

for(var i = count;i--;){

var liDom = document.createElement('li');

ulList.appendChild(liDom);

}

//绑定点击事件

var liNode = ulList.getElementByTagName("li");

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

liNode[i].onClick = function(){

//li点击事件

}

}

众所周知，DOM操作是十分消耗性能的。所以重复的事件绑定简直是性能杀手。而事件代理的核心思想，就是通过尽量少的绑定，去监听尽量多的事件。如何做呢？答案是利用事件冒泡机制,对其父节点ul进行事件绑定（Event Bubble),然后通过event.target来判断是哪个节点触发的事件，从而减少很多EventHandler的绑定。

var count = 100;

var ulList = document.getElementById("list");

//动态构建节点

for(var i = count;i--;){

var liDom = document.createElement('li');

ulList.appendChild(liDom);

}

//绑定点击事件

var liNode = ulList.getElementByTagName("li");

liNode.onClick = function(e){

if(e.target && e.target.nodeName.toUpperCase == "LI") {

// li点击事件

}

}