**Javascript基础概念以及面向对象基础复习**

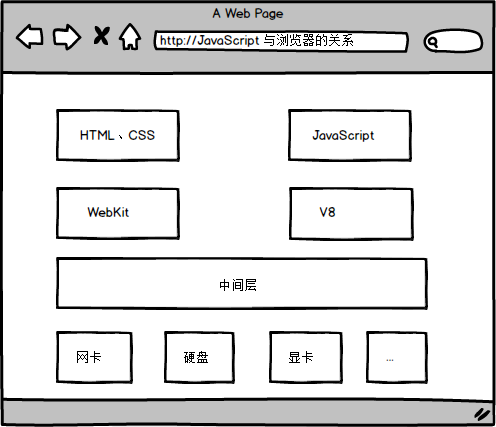
1. **课程介绍：**
   1. 课程大纲
      1. 在线地址：[JavaScript 高级](http://naotu.baidu.com/file/5bcd79bc4f1eaf83f96d1ad23baab345?token=d22135c63546f5ee)
   2. 目标：
      1. 理解面向对象编程思想
      2. 掌握面向对象开发相关模式
      3. 掌握javascript中的正则表达式的应用
   3. 综合案例演示

[[贪吃蛇](https://zagcool.github.io/snake/)]

1. **基本概念复习**

前言-------由于 JavaScript 高级还是针对 JavaScript 语言本身的一个进阶学习，所以在开始之前我们先对以前所学过的 JavaScript 相关知识点做一个快速复习总结。

* 1. javascript是什么？
     1. 运行在浏览器客户端的解释型脚本语言 大前端（web前端（h5）+ios+安卓）
     2. 特点：
        1. 解释执行：不是预先编译，而是边解释边执行，遇到错误则停止
        2. 动态性：在程序运行过程当中可以改变已有和产生新的函数，函数在javascript中是一等公民 js是函数式编程
        3. 执行环境（也叫宿主环境）即程序在当前能够起作用的环境，浏览器是常见的宿主环境，但是除了浏览器我们后面学的node平台也可以运行js(as flash)了html+css+js + node(express+ mongodb)
        4. 编程范式 基于原型可以实现面向对象，所以本质上来讲js是一门基于对象而不是面向对象的语言，采用函数式的编程风格，语法较松散，语句简单
  2. Js和浏览器的关系



* 1. javascript的组成

|  |  |
| --- | --- |
| 组成部分 | 说明 |
| Ecmascript(语法核心) | 描述了该语言的语法和基本对象 |
| DOM | 描述了处理网页内容的方法和接口 |
| BOM | 描述了与浏览器进行交互的方法和接口 |

* 1. Any application that can be written in JavaScript, will eventually be written in JavaScript.

凡是能用 JavaScript 写出来的，最终都会用 JavaScript 写出来

Js能做什么？

Pc端网站（pc偏电商类网站）

移动web

混合开发(代替原生Android和ios开发)

桌面程序或者系统平台：electron Firefox OS

nodeJs做后端（数据库 服务器 网络通讯等）

浏览器插件等 12306抢票插件.js

Js可能一统江湖

Html5 和css3 canvas(替代flash)

Javascript 原生以及基于js的一系列组件和库和框架

学习：练习 理解 重复练习 理解

* 1. Js的发展历史
     1. [[JavaScript 标准参考教程 - JavaScript 语言的历史](http://javascript.ruanyifeng.com/introduction/history.html)]
     2. - JavaScript 的诞生
        1. 1995 网景公司 布兰登艾奇
     3. - JavaScript 与 Ecmascript 的关系

Js语言是es的标准规范基础之上实现的扩展，es提供的基础的语法核心

* + 1. - JavaScript 与 Java 的关系

js与Java的关系不大，除了名字上类似以外，那就是借鉴了java语言的一些特点

* + 1. - JavaScript 的版本

Es3 es5 es6的关系

* + 1. - JavaScript 周边大事记（了解）
  1. Js的语法：
     1. 标识符的命名规则
        1. 可以由字母数字 下划线以及$组成 不能以数字开头
        2. 严格区分大小写
     2. 标识符的规范：
        1. 标识符的起名要有意义
        2. 遵循驼峰式命名法
     3. 关键字和保留字
     4. 注释：单行注释和多行注释
     5. 严格模式：
        1. 是一种将更好的错误检查引入代码中的方法，使js的语法更加规范，更加严谨 运行更加安全 提高编译器运行效率，为未来的javascript新版本做好铺垫
        2. 声明语法：’use strict’;
     6. 变量 内存中用来存储数据的标识符
     7. 数据类型
        1. number string boolean undefined null object
        2. 判断数据类型的操作符 typeof
     8. 操作符
        1. 一元操作符 ++ --

var k=-2;

alert(++k + k++ + ++k + k);

* + - 1. （）==》算数运算符==》比较运算符=》 （相等运算符）==》逻辑运算符 赋值运算符 以及运算符的优先级
    1. 流程控制语句
       1. 条件判断语句
          1. If相关语句
          2. Switch语句
          3. 三元运算
       2. 循环语句
          1. While
          2. Do……while
          3. for循环
    2. 函数
  1. Javascript中的数据类型
     1. 有五种简单数据类型：undefined null boolean Number String
     2. 一种复杂数据类型 object（数组属于对象）

简单数据类型可以理解为值类型

复杂数据类型可以理解为引用类型

在js中除了Object对象以外还有其他的一些内置对象如

Array Date RegExp Function

基本包装类型 Boolean Number String

单体内置对象：Math （在js中是内置的独立存在的 不是由构造函数构造出来的）

Js无法直接操作内存，它的实现是基于浏览器封装的，所以我们通过模拟内存堆栈来进行理解

值类型和引用类型的存储方式

一种是按值存储

一种是按引用存储（址）

画图演示

值类型复制和引用类型复制

值类型传递和引用类型的传递

* + 1. 值类型与引用类型的差别
       1. 基本类型在内存中占据固定大小的空间，因此被保存在栈内存中
       2. 从一个变量向另一个变量复制基本类型的值，复制的是值的副本
       3. 引用类型的值是对象，保存在堆内存
       4. 包含引用类型值的变量实际上包含的并不是对象本身，而是一个指向该对象的指针
       5. 从一个变量向另一个变量复制引用类型的值的时候，复制是引用指针，因此两个变量最终都指向同一个对象
    2. 补充：属性的访问方式 delete关键字的使用
  1. Javascript的执行过程
     1. Js的运行过程分为两个阶段
        1. 预解析

全局预解析（把声明的变量和声明的函数提前，同名函数覆盖变量名，后者覆盖前者）

函数内部预解析:（所有的变量 函数和形参都参与预解析）

函数

形参

普通变量

* + - 1. 执行

先预解析全局作用域 然后执行全局作用域中的代码

在执行全局代码的过程中遇到函数调用就会先进行函数预解析，然后再执行函数内代码

var a = 25;

function abc (){

alert(a);//undefined

var a = 10;

}

abc();

// 如果变量和函数同名的话，函数优先

console.log(a);

function a() {

console.log('aaaaa');

}

var a = 1;

console.log(a);

面试题：

alert(a);

var a;

alert(a);

function a() {

alert(1);

}

alert(a);

var a = 2;

alert(a);

function a() {

alert(3);

}

alert(a);

function b() {

alert(4);

}

alert(a);

i) 异常处理

1、防止某些程序出错时影响后面代码的执行

2、屏蔽低层的报错细节，从而给用户一个友好提示

try{

// 执行的代码，其中可能有异常，一旦发现异常，

立即跳到catch执行，否则不会执行catch语句

}catch(e){

// 除非try里面执行代码发生了异常，否则这里的代码不会执行

}finally{

// 不管什么情况都会执行，包括try catch里面用了return，可以理解为只要执行了try或者catch，

就一定会执行finally

}

1. **javascript面向对象编程基础**



* 1. 面向对象介绍
     1. 什么是面向对象？

Everything is object （万物皆对象）



世界万物都是由各种各样具有自己的运动规律和内部状态对象所组成

不同对象之间的相互联系和通讯构成了这个完整的现实世界

* + 1. 对象就是对单个事物的抽象
       1. 对象具有特征，对其抽象的一种描述 即属性
       2. 对象具有行为, 对其抽象的一种描述 就是方法

属性是对象的状态，方法是对象的行为（完成某种任务）。比如，我们可以把动物抽象为animal对象，使用“属性”记录具体是那一种动物特征，使用“方法”表示动物的某种行为（奔跑、捕猎、休息等等）

在实际开发中，对象是一个抽象的概念，可以将其简单理解为：数据集或功能集。

ECMAScript-262 把对象定义为：无序属性的集合，其属性可以包含基本值、对象或者函数。

* 1. 在程序中 （在咱们的js中）如何理解面向对象呢？

用程序的方式描述对象：属性+行为

1、属性

2、行为

男朋友（抽象的模板 --- 构造函数）

属性：性别、身高、体重、外貌

行为：做饭、开车、写代码、洗衣服。唱歌

面向对象不是新的东西，它只是过程式代码的一种高度封装，目的在于提高代码的开发效率和可维护性。



面向对象编程 oop，在程序开发过程当中，每个对象都是功能中心，具有明确分工，可以完成接收信息，处理数据，发出信息的任务

相比面向过程而言：面向对象具有灵活，可复用，高度模块化，容易开发和维护，相比传统一系列复杂函数和指令的过程式编程

**面向对象特性：**

**继承性**

一个对象没有某些属性和方法，另外一个对象有，拿过来用，就是继承！

**封装性**

封装就是将功能封装整合在对象内，只对外界暴露指定的接口，外界在使用的时候，只需要考虑接口怎么用，而不需要考虑内部的具体实现

**你看电视 你只需要会开关电视和调频道就可以了，你不需要关系电视如何造的 接口（学遥控器）**

**多态性**

JS中没有多态！

有继承关系，父类引用指向子类的对象，就是多态！（不用强行理解！）

在 JavaScript 中，所有数据类型都可以视为对象，当然也可以自定义对象。

自定义的对象数据类型就是面向对象中的类（ Class ）的概念。

var std1 = { name: 'Michael', score: 98 }

var std2 = { name: 'Bob', score: 81 }

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

function printScore (student) {

console.log('姓名：' + student.name + ' ' + '成绩：' + student.score)

}

分析思路：

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，

而是 Student 这种数据类型应该被视为一个对象，这个对象拥有 name 和 score 这两个属性（Property）。

如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个 printScore 消息，让对象自己把自己的数据打印出来。

function Student (name, score) {

this.name = name;

this.score = score;

this.printScore = function () {

console.log('姓名：' + this.name + ' ' + '成绩：' + this.score)

}

}

根据模板创建具体实例对象（Instance）：

var std1 = new Student('Michael', 98)

var std2 = new Student('Bob', 81)

实例对象具有自己的具体行为（给对象发消息）：

std1.printScore() // => 姓名：Michael 成绩：98

std2.printScore() // => 姓名：Bob 成绩 81

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。

Class 是一种抽象概念，比如我们定义的 Class——Student ，是指学生这个概念，

而实例（Instance）则是一个个具体的 Student ，比如， Michael 和 Bob 是两个具体的 Student 。

其实在js中（es5）没有类的概念，我们通过构造函数去实现类

所以，面向对象的设计思想是：

- 抽象出 Class

- 根据 Class 创建 Instance

- 指挥 Instance 得结果

c) 如何创建对象

通过前面的基础学习，我们可以通过标准声明方式和字面量声明方式创建

var person = new Object();

Person.name = “lisi”;

Person.age = 25;

Person.sayName = function() {

console.log(this.name)

}

var person = {

name: 'Jack',

age: 18,

sayName: function () {

console.log(this.name)

}

}

上面的写法固然没问题，但是我们想生成两个不同的实例呢

var person1 = {

name: 'Jack',

age: 18,

sayName: function () {

console.log(this.name)

}

}

var person2 = {

name: 'Mike',

age: 16,

sayName: function () {

console.log(this.name)

}

}

通过上面的代码我们不难看出，这样写的代码太过冗余，重复性太高

简单方式的改进：工厂函数：

function createPerson (name, age) {

var obj = {

name: name,

age: age,

sayName: function () {

console.log(this.name)

}

}

return obj;

}

然后生成实例对象：

var p1 = createPerson('Jack', 18)

var p2 = createPerson('Mike', 18)

这样封装确实爽多了，通过工厂模式我们解决了创建多个相似对象代码冗余的问题，

但却没有解决对象识别的问题（即怎样知道一个对象的类型）

这种方式返回的是同一个obj对象，无法做识别

d) 更优雅的工厂函数：构造函数

function Person (name, age) {

this.name = name

this.age = age

this.sayName = function () {

console.log(this.name)

}

}

var p1 = new Person('Jack', 18)

p1.sayName() // => Jack

var p2 = new Person('Mike', 23)

p2.sayName() // => Mike

我们注意到，Person() 中的代码与 createPerson() 有以下几点不同之处：

- 没有显示的创建对象

- 直接将属性和方法赋给了 this 对象

- 没有 return 语句

- 函数名使用的是大写的 Person

而要创建 Person 实例，则必须使用 new 操作符。

以这种方式调用构造函数会经历以下 4 个步骤：

1. 创建一个新的隐式对象

2. 将构造函数的作用域赋给新对象（因此 this 就指向了这个隐式对象）

3. 执行构造函数中的代码

4. 返回这个新的隐式对象

function Person (name, age) {

// 当使用 new 操作符调用 Person() 的时候，实际上这里会先创建一个对象

// var instance = {}

// 然后让内部的 this 指向 instance 对象

// this = instance

// 接下来所有针对 this 的操作实际上操作的就是 instance

this.name = name

this.age = age

this.sayName = function () {

console.log(this.name)

}

// 在函数的结尾处会将 this 返回，也就是 instance

// return this

}

注意点：

1 如果主动从构造函数内部return数据，数据为非对象类型则函数依然返回隐式对象，如果返回的数据是一个对象（除了null），那么则返回对应对象

2 所谓构造函数就是用来构造对象的函数，不成文规定构造函数首字母大写

3 使用构造函数的好处不仅仅在于代码的简洁性，更重要的是我们可以识别对象的具体类型了。

在每一个实例对象中同时有一个 constructor 属性，该属性指向创建该实例的构造函数：

console.log(p1.constructor === Person) // => true

console.log(p2.constructor === Person) // => true

console.log(p1.constructor === p2.constructor) // => true

在该示例中，从表面上好像没什么问题，但是实际上这样做，有一个很大的弊端。

那就是对于每一个实例对象，type 和 sayHello 都是一模一样的内容，

每一次生成一个实例，都必须为重复的内容，多占用一些内存，如果实例对象很多，会造成极大的内存浪费。

console.log(p1.sayHello === p2.sayHello) // => false

对于这种问题我们可以把需要共享的函数定义到构造函数外部

function sayHello = function () {

console.log('hello ' + this.name)

}

function Person (name, age) {

this.name = name

this.age = age

this.type = 'human'

this.sayHello = sayHello

}

var p1 = new Person('lpz', 18)

var p2 = new Person('Jack', 16)

console.log(p1.sayHello === p2.sayHello) // => true

这样确实可以了，但是如果有多个需要共享的函数的话就会造成全局命名空间冲突的问题。

你肯定想到了可以把多个函数放到一个对象中用来避免全局命名空间冲突的问题

var fns = {

sayHello: function () {

console.log('hello ' + this.name)

},

sayAge: function () {

console.log(this.age)

}

}

function Person (name, age) {

this.name = name

this.age = age

this.type = 'human'

this.sayHello = fns.sayHello

this.sayAge = fns.sayAge

}

var p1 = new Person('lpz', 18)

var p2 = new Person('Jack', 16)

console.log(p1.sayHello === p2.sayHello) // => true

console.log(p1.sayAge === p2.sayAge) // => true