JS高级

## 浅拷贝 与 深拷贝

浅拷贝：将对象中的各个属性依次进行复制，浅拷贝只复制了一层对象的属性。如果对象属性中还有对象，那么赋值的仅仅是地址。还是会相互影响。

深拷贝：深拷贝会递归赋值所有层对象的属性。如果对象属性中还有对象，会继续拷贝，这样拷贝出来的对象完全独立。

**深浅拷贝 只针对于复杂数据类型**

### 什么是对象??

1. **从视觉角度 : 对象是单个事物的抽象。**

**(2) 从编程角度 : 对象是无序键值对的集合，其属性可以包含基本值、对象或者函数**等等

### 什么面向对象 ?

**面向对象编程** —— Object Oriented Programming，简称 **OOP** ，是一种编程开发思想。

面向对象的特性：

**1.- 封装性**

- 将功能的具体实现，全部封装到对象的内部，外界使用对象时，只需要关注对象提供的方法如何使用，而不需要关心对象对象的内部具体实现，这就是封装。

**2.- 继承性**

- 在js中，继承的概念很简单，一个对象没有的一些属性和方法，另外一个对象有，<!-- 拿过来用，就实现了继承。-->

- 注意：在其他语言里面，继承是类与类之间的关系，在js中，是对象与对象之间的关系。

**3.- [多态性]**

- 多态是在强类型的语言中才有的。js是弱类型语言，所以JS不支持多态。

## 创建对象的方式:

### **字面量创建对象 =====** 缺点 : 不能很方便的批量创建

1. **构造函数 Object 创建对象**

缺点 : 属性要一个一个的添加,也是 不能很方便的批量创建,

1. **工厂函数 创建对象**

缺点 : 但却没有解决对象识别的问题，创建出来的对象都是Object类型的。

1. **自定义构造函数创建**

**缺点：**使用构造函数带来的最大的好处就是创建对象更方便了，但是其本身也存在一个浪费内存的问题： 缺点：会暴漏很多的函数，容易造成全局变量污染。

**注意点：**

1. 构造函数 => 函数 + 首字母大写

2. 构造函数要配合new操作赋一起使用才有意义

3. new的作用 : (牢记)

- 创建一个新对象

- this指向了这个新对象

- 执行构造函数, 给对象添加属性和方法

- 返回新对象

4. 构造函数的作用 : 实例化对象 即:给对象赋值,添加属性和方法

原型 ：

### 原型的基本概念：

Javascript 规定，每一个构造函数都有一个 prototype 属性，指向另一个对象。

这个对象的所有属性和方法，都会被构造函数( Person() )的实例 ( p、p1等 )继承 。 这个对象就是原型,也叫原型对象。

这也就意味着，我们可以把所有对象实例需要共享的属性和方法直接定义在 prototype 对象上。

**原型的作用 : 共享数据** (也解决了构造函数资源浪费的问题)

**原型:**

1. 在js中, 任何一个函数,都会自带一个属性 prototype, 这个属性指向了一个对象

2. 这个prototype属性,也就是这个对象,我们叫做原型 ; (原型对象) 这个原型属性是一个对象

3. 通过构造函数创建的对象,可以直接访问这个构造函数的原型中的所有内容(属性和方法)

4. 最常用的就是 : 给构造函数的原型添加一个方法

### **构造函数、实例、原型三者之间的关系**

**1.构造函数 (Person)**：构造函数就是一个函数，配合new可以新建对象。

**2.实例 (p1, p2)**：通过构造函数实例化出来的对象我们把它叫做构造函数的实例。一个构造函数可以有很多实例。

**3.原型 (Person.prototype)** ：每一个构造函数都有一个属性prototype,这个属性就叫做原型对象。通过构造函数创建出来的实例能够直接使用原型上的属性和方法。

三者关系的图：

### 继承关系: **实例对象p, 从Person.prototype里访问属性和方法,,这就叫继承 ;**

1. **\_\_proto\_\_**

通过构造函数创建的对象，自带一个\_\_proro\_\_属性，这个属性指向了构造函数的prototype属性，也就是原型对象。

**获取原型对象：**

1. 通过构造函数.prototype可以获取
2. 通过实例.\_\_proto\_\_可以获取（隐式原型
3. 它们指向了同一个对象构造函数.prototype === 实例.\_\_proto\_\_
4. 又因为是浅白色,,这种是私有属性,,不可遍历的,,,,,不要用它来添加属性和方法,,只负责来**检测它的原型**即可

### **constructor属性**

默认情况下，原型对象中值包含了一个属性：constructor，constructor属性指向了当前的构造函数。

## **原型链**

### 属性查找原则 ：构造函数实例化的对象 p , 能够访问原型对象里的全部属性和方法

1. **如果是获取操作 =====** 沿着 \_\_proto\_\_ 一直往上找
   1. 1. 会先在自身上查找，如果没有
   2. 2. 则根据\_\_proto\_\_对应的原型去找，如果没有
   3. 3. 一直找到Object.prototype，如果没有，那就找不到了。
2. **如果是修改操作 ====**

1. 只会修改对象自身的属性，

2. 如果自身没有这个属性，那么就会添加这个属性，并不会修改原型中的属性。

### **原型链概念**

**任何一个对象，都有原型对象**，**原型对象**本身又是一个**对象\***\*，所以**原型对象也有自己的原型对象**，这样一环扣一环就形成了一个**链式**结构，我们把这个链式结构称为**：原型链。 \_\_ proto \_\_**

1. var s = new Strudent();

2. var o = new Object();

3. var arr = new Array();;

4. var date = new Date();

5. Math

**总结：Object.prototype是原型链的尽头，Object.prototype的原型是null。**

**Day02**

# **Object.prototype成员介绍**

constructor: 返回对创建此对象的构造函数

hasOwnProperty:

isPrototypeOf:

propertyIsEnumerable:

toLocaleString:

toString:

valueOf:

## **1.hasOwnProperty ===** hasOwnProperty() 方法会返回一个布尔值，判断某个属性是否是对象自己的属性。

**参数 : 属性字符串 返回值 :true或者false**

如果 **是** 自己的属性，会返回true。

如果 **不是** 自己的属性,会返回false。( 原型链上的属性也不算自己的 )，

**hasOwnProperty 与 in 的区别？**

**（1）in 操作符**：如果属性 **是自己的 + 原型上继承**来的，==> true

**in 能判断的属性**

**1. 自己的属性 OK**

**2. 原型链上的属性也是**

1. ****hasOwnProperty**: 该属性必须是**自己的属性**，== > true，否则返回false。**

## propertyIsEnumerable 　可枚举／可遍历

propertyIsEnumerable() 方法返回一个布尔值，。 判断该属性是否是当前对象**可枚举**的**自身属性**

可枚举 ( 可以 for in 的 => 自身属性 + 原型链新加的属性)

自身属性

## Object.defineProperty() - 定义一个新属性（高级第二天）

## valueOf/toString()/toLocaleString()

1. **toString() + toLocalString()**

Object toLocaleString 返回调用 toString() 的结果。

基本上都是一样的结果：　　比如两个数组

不一样的就是关于ｄａｔｅ的时候不一样

1. **valueOf() + toString()**

toString() 转化为字符串 ========== 作用 : 以字符串的形式表示

valueOf ========== 作用 : 返回object本身

## isPrototypeOf() 和 instanceOf

* 1. **isPrototypeOf()**

作用 : 用于测试一个**原型对象** 是否存在**对象的原型链上**

语法结构 :  **A.isPrototypeOf(B):**

判断Ａ　是否存在　Ｂ　的原型链上。 说白了就是判断A是否是B的祖先

1. **Instanceof**

作用 : 作用和isPrototypeOf类似，用于判断 **构造函数** 的prototype属性是否在**对象的原型链上**。如果是，就返回true，如果不在，就返回false。

语法结构 : object instanceof constructor

- 对象 instanceof 构造函数

**返回值：检测构造函数的prototype属性是否在实例对象的原型链上**。

**区别 :A.isPrototypeOf(B)**

**判断A是否在B的原型链上 A： 是一个原型对象**

**例如 : Person.prototype.isPrototypeOf(p)**

**- Ｃ instanceof Ｄ**

**判断Ｄ的prototype是否在Ａ的原型链上 (Ｄ是Ｃ的构造函数) Ｄ：是一个构造函数**

**例如 p instanceof Person**

**贪吃蛇案例**

**Day03**

# 沙箱模式

沙箱其实就是一个独立的环境，这个环境中任何的改变，都不会对外部环境产生影响。

函数自调用一样，在自调用函数内部的变量是不会影响到外部的，因此函数自调用模式也叫沙箱模式。

# **继承** 继承的目的：让一个对象可以使用另一个对象的属性和方法。

**JS常见的几种继承模式：( 第三天 )**

## **混入式继承（mixin）===** 把一个对象中的属性和方法拷贝到另一个对象中。

缺点 : 只能给一个对象继承, 而且给自身属性添加了新的方法

## **原型链继承**

一个对象可以访问构造函数的原型中的属性和方法，那么如果想要让一个对象增加某些属性和方法，

只需要把这些属性和方法放到原型对象中即可。这样就实现了继承, 称之为原型链继承

1. 直接给原型增加属性和方法
2. 原型替换（注意：constructor）

- 缺点 : 只能继承一个对象

## **3.mixin+原型替换**

混入继承的缺点 : 引入多的方法 原型继承的缺点 : 只能继承1个

## 4.**Object.create**

ES5中新增了一个方法Object.create(),方法会使用指定的原型对象及其属性去创建一个新的对象。

**参数：proto 一个对象**

**返回值：obj 新对象，新对象的原型就是proto**

## **call**

## 构造函数.call (对象A) === 翻译 : 构造函数里的this ,指向了 对象A

**Day04**

## 1.定义函数的三种方式

### **1.函数声明**

### **2.函数表达式**

### **3.构造函数Function**

**try...catch**

**eval函数--了解**\*\*

## **2.**函数的四种调用模式

### **1.函数调用模式**

如果一个函数不是一个对象的属性时，就是被当做一个函数来进行调用的。此时this指向了window

### **2.方法调用模式**

1.当一个函数被保存为对象的一个属性时，我们称之为一个方法。当一个方法被调用时，this被绑定到当前对象

2.事件中的this指向的是当前的元素，在事件触发的时候，浏览器让当前元素调用了function

### **构造函数调用模式**

如果函数是通过new关键字进行调用的，此时this被绑定到创建出来的新对象上。

****定时器中的this指向了window，因为定时器的function最终是由window来调用的**。**

### **上下文调用模式 (方法借用)**

上下文调用模式也叫方法借用模式，分为apply与call

使用方法： 函数.call() 或者 函数.apply()

#### **1.call方法 =====** call方法可以调用一个函数，并且可以指定这个函数的this指向

1. 所有的函数都可以使用call进行调用

**1. 没有参数**

**2. 有参数的 ==== call(参数1,参数2,参数3..)**

**参数1 : this 指向了第一个参数, (如果第一个参数为null,默认指向windw)**

**2.伪数组与数组 =========** 伪数组也叫类数组

1.伪数组其实就是一个对象，但是跟数组一样，伪数组也会有length属性，也有0,1,2,3等属性。

2.arguments 的**proto** 和 var arr = [] 的 \_\_ proto \_\_ 比较

3.伪数组并没有数组的方法，不能使用push/pop等方法

4.伪数组可以跟数组一样进行遍历，通过下标操作。

5.常见的伪数组：arguments、document.getElementsByTagName的返回值、jQuery对象

**伪数组借用数组的方法**

Array.prototype.push.call(obj, "赵六");

[].push.call(obj,'田七'); 也是可以的

var str = Array.prototype.join.call(obj, "-");

#### **将伪数组转换成真数组 == var arr = Array.prototype.slice.call(obj);**

#### **apply方法 参数1：指定this 参数2：数组 ，需要把所有的形参都放到一个数组中**

apply()方法接受的是**一个包含多个参数的数组**。而call()方法接受的是**若干个参数的列表**

**call和apply的使用场景：**

1. 如果参数比较少，使用call会更加简洁
2. 如果参数存放在数组中，此时需要使用apply
3. 函数.call(this的指向目标, 参数1,参数2) 参数列表
4. 函数.apply(this的指向目标, [参数1,参数2]) 参数素组

#### 4.**bind方法**

**bind()**方法创建一个新的函数, 可以绑定新的函数的this指向

返回值：新的函数

参数：新函数的this指向，当绑定了新函数的this指向后，无论使用何种调用模式，this都不会改变

函数也是对象

函数是由new Function创建出来的，因此函数也是一个对象， 所有的函数都是new Function的实例。

### 函数的原型链结构(画图)

### **Function.prototype成员**

### arguments：获取函数的实参，被函数内部的arguments替代了。 (废弃)

### length：获取形参的长度

### 3.name：获取函数的名字，此属性不允许修改

4.caller:用于获取当前在函数是在哪个函数中调用的，已经被废弃了。(废弃)

5.constructor：指向当前构造函数，Function

6.call：调用函数，重新指向this

7.apply：调用函数，重新指向this

8.bind：重新指向this，返回一个新的函数，不调用。

### **完整版原型链（day04）**

**总结：**

1. 所有函数都是new Function创建出来的，因此所有函数.\_\_proto\_\_都是Function.prototype
2. 所有对象都是new Object创建出来的，因此所有对象.\_\_proto\_\_都是Object.prototype

## 预解析与作用域

预解析 **js执行代码分为两个过程：** (记忆)

1.预解析过程（变量与函数提升） 2.代码一行一行执行

预解析过程：JavaScript解析器在执行代码前，会把所有变量的声明和函数的声明提升到当前作用域的顶部。例如var a = 11;其实会分为var a; 和a = 11两部分，其中var a;会被提升。

### **预解析规则：**

### 把var 声明的变量提升,,赋值不提升;

1. 函数声明(不是表达式), 整体提升
2. 函数同名,后者覆盖前者
3. 函数与变量同名,,函数覆盖变量

**推荐：不要在一个作用域内重复的声明相同的变量和函数**

### **作用域** 变量起作用的区域，作用域决定了一个变量被定义在哪里，以及该如何被查找。

**全局作用域** : 函数外部的区域;

**函数作用域** : 函数内部的区域;

**全局变量**：在函数外定义的变量就叫全局变量，全局变量在任何地方都能访问到。

**局部变量**：在函数内定义的变量就叫局部变量，局部变量只有在当前函数内才能访问到。

**隐式全局变量** : 只赋值未声明的变量, 隐式全局也是全局;

**编程语言中，作用域规则分为两种**

**词法作用域**（静态作用域）===(变量在函数声明的时候，它的作用域就定下来了，与函数的调用无关。)

**动态作用域**

### 作用域链

只要是函数，就会形成一个作用域，如果这个函数被嵌套在其他函数中，那么外部函数也有自己的作用域，这个一直往上到全局环境，就形成了一个条作用域链。

变量的搜索原则：

1. 从当前作用域开始搜索变量，如果存在，那么就直接返回这个变量的值。 2.如果不存在，就会往上一层作用域查询，如果存在，就返回。

3.如果不存在，一直查询到全局作用域，如果存在，就返回。如果不存在说明该变量是不存在的。

4.如果一个变量不存在

(1)获取这个变量的值会报错xxx is not defined;，

(2)给这个变量设置值，那么设置变量就是隐式全局变量。

Day05

# 函数闭包

在JavaScript中，在外部函数中嵌套另一个子函数时，如果内部的函数引用了外部的函数的变量，则可能产生闭包。

**闭包中包含了内部函数的代码，以及所需外部函数中的变量的引用**

### **产生闭包的条件()**

## 闭包的应用

### (1)计数器闭包的应用 需求：统计一个函数的调用次数

### **闭包的作用:**

### 把变量保护起来 ()

### 这些变量的值始终保持在内存中 ==> 应用

### **私有变量: ==** 使用闭包实现私有变量的读取和设置

## **闭包存在的问题**

闭包占用的内存是不会被释放的，因此，如果滥用闭包，会造成内存泄漏的问题。闭包很强大，但是只有在必须使用闭包的时候才使用。

### js的垃圾回收机制 (day 05)

# 正则表达式

用于匹配规律规则的表达式，正则表达式最初是科学家对人类神经系统的工作原理的早期研究，现在在编程语言中有广泛的应用，经常用于表单校验，高级搜索等

## **创建正则表达式**

1. **构造函数的方式**  == var regExp = new RegExp(/\d/);
2. 正则字面量 ===== var regExp = /\d/;
3. 正则的使用 ====== /\d/.test("aaa1");

## 元字符(常见元字符)（day 05）

### **优先级====** |`表示或，优先级最低 ()优先级最高，表示分组

### **字符类的元字符**

1.[]在正则表达式中表示一个字符的位置，[]里面写这个位置可以出现的字符

2.[^]在中扩号中的^表示非的意思。 是否包括除了XX以外的字符串

3.[a-z] [1-9]表示范围

### **边界类元字符**

1. ^表示开头   []里面的^表示取反
2. $表示结尾

### **量词类元字符**

### \*表示能够出现0次或者更多次，x>=0;

### +表示能够出现1次或者多次，x>=1

1. ?表示能够出现0次或者1次，x=0或者x=1
2. {n}表示能够出现n次
3. {n,}表示能够出现n次或者n次以上
4. {n,m}表示能够出现n-m次

## **正则替换 replace 正则提取 exec => 数组**

## **正则匹配 match**

## **字符串总结:**

正则表达式只有：2个： test:测试 reg.test()

exec:提取 reg.exec()

字符串有两个方法： replace match: 支持参数传正则

str.replace() str.match();

正则的使用：

1. 字符串的replace: 正则的替换

2. 字符串的匹配：match: 匹配某个字符串中所有符合规律的字符串。

3. 正则的测试：test: 表单校验，判断某个字符串是否符合正则的规律

4. 正则的提取： 提取匹配的字符串的每一个部分。 ()进行分组