# JavaScript基础

## 类型与判断

### 内置类型

1. 空值 null
2. 未定义 undefined
3. 布尔值 boolean
4. 数字 Number
5. 字符串 String
6. 对象 Object
7. 符号 symbol

### typeof

typeof操作符返回一个字符串，表示未经计算的操作数的类型。

```
typeof operand
typeof(operand)

typeof undefined        ===  'undefined'
typeof undeclared       ===  'undefined'
typeof true             ===  'boolean'
typeof 22               ===  'number'
typeof NaN              ===  'number'
typeof '22'             ===  'string'
typeof []               ===  'object'
typeof {}               ===  'object'
typeof null             ===  'object'
typeof /regex/          ===  'object'
typeof new Date()       ===  'object'
typeof new String()     ===  'object'
...
typeof new Function()   ===  'function'
typeof function a(){}   ===  'function'
```

**除了Function之外的所有构造函数的类型都是'object'。**
undefined是值的一种，为未定义，而undeclared则表示变量还没有被声明过。在我们试图访问一个undeclared变量时会这样报错：`‘ReferenceError：a is not defined`。通过typeof对undefined和undeclared变量都返回"undefined"

**注意：变量没有类型，只有值才有。类型定义了值的行为和特征。**

## instanceof

`instanceof`运算数是用于检测`contructor.prototype`属性是否出现在某个实例对象的原型链上。

```
object instanceof constructor

22 instanceof Number                                    => false
'22' instanceof String                                  => false
[] instanceof Object                                        => true
{} instanceof Object                                        => true
undefined instanceof Object    => false
null instanceof Object                      => false
null instanceof null                        => Uncaught TypeError: Right-hand side of 'instanceof' is not an object

new String('22') instanceof String   => true
new Number(22) instanceof Number     => true
```

**instanceof的主要实现原理就是只要右边变量的prototype在左边变量的原型链上即可。**因此，instanceof在查找过程中会遍历左边变量的原型链，知道找到右边变量的prototype，如果查找失败就会返回false。

## Object.prototype.toString.call()

```
Object.prototype.toString.call(22)              =>  "[object Number]"
Object.prototype.toString.call('22')            =>  "[object String]"
Object.prototype.toString.call({})              =>  "[object Object]"
Object.prototype.toString.call([])              =>  "[object Array]"
Object.prototype.toString.call(true)            =>  "[object Boolean]"
Object.prototype.toString.call(Math)            =>  "[object Math]"
Object.prototype.toString.call(new Date)        =>  "[object Date]"
Object.prototype.toString.call(Symbol(22))      =>  "[object Symbol]"
Object.prototype.toString.call(() => {})        =>  "[object Function]"
Object.prototype.toString.call(null)            =>  "[object Null]"
```

```
11 Object.prototype.toString.call(undefined)        =>  "[object Undefined]"
```

### 类型判断

用 `typeof` 来判断变量类型的时候，我们需要注意，最好是用 `typeof` 来判断基本数据类型（包括 `symbol`），避免对 null 的判断。不过需要注意当用`typeof`来判断null类型时的问题，如果想要判断一个对象的具体类型可以考虑使用instanceof，但是很多时候它的判断有写不准确。所以当我们在要**准确的判断对象实例的类型时，可以使用`Object.prototype.toString.call()`进行判断。因为`Object.prototype.toString.call()`**是引擎内部的方式。

## 作用域

### 编译原理

JavaScript的编译过程分为三个过程：**词法分析**、**语法分析**、**代码生成**。
**词法分析**阶段主要是将代码分解成词法单元。**语法分析**是将这些词法单元流转换成一个有元素逐级嵌套所组成的代表了程序语法结构的树，抽象语法树AST。**代码生成**是将AST转换为可执行代码的过程。

### 作用域

作用域分为**静态作用域**和**动态作用域**。
静态作用域是在定义的时候就确定了变量的值，而动态作用域是在执行的时候才确定变量的值。
**词法作用域**就是定义在词法阶段的作用域。也就是说词法作用域是在写代码时将变量和块作用域在哪里决定，所以说一般情况下词法分析器处理代码时会保持词法作用域不变。
作用域又分为**函数作用域**和**块级作用域**。
函数作用域就是说与这个函数的全部变量都可以在整个函数的范围内使用及复用，包括它内部的嵌套作用域。函数的作用域是在定义的时候就确定的。
块级作用域是是一个用来对最小授权原则进行扩展的工具，将代码从在函数中隐藏信息扩展为在块中隐藏信息。最常见的两个花括号就是一个块作用域。
ES5实现块级作用域的方式有：**eval**、**with**、**try/catch的catch分句**、**闭包**。
ES6就是let和const，块级作用域的好处是：垃圾回收和let循环。

### 作用域链

**作用域链的产生是因为作用域发生嵌套**。它是当前环境与上层环境的一系列变量对象组成的，它保证了当前执行环境对符合访问权限的变量和函数的有序访问。当查找变量的时候，会先从当前的上下文变量对象中查找，如果没有找到，就会从父级的上下文变量对象中查找，直到找到全局上下文中的变量即全局对象。这个由多个执行上下文的变量对象构成的链表叫做作用域链。
**作用域链是在函数定义的时候就已经决定的。**
这是因为函数有一个内部属性[[scope]]，当函数创建的时候，就会保存所有父变量对象到其中，[[scope]]就是所有父变量对象的层级链，但是注意，[[scope]]并不代表一个函数完成的作用域链。[[scope]]是在函数创

建阶段的，当函数执行后，会将函数本身的AO和[[scope]]合并，这才是这个函数完整的作用域链。

```
ScopeChain = AO.concat([[scope]])
```

### 相关

作用域是 JavaScript中一个相当重要的概念，它与许多知识点都息息相关，也可以说由它衍生出的。比如说**变量提升**、**立即执行函数**、**闭包**、**ES5模块**等相关知识点。

## 闭包

> 当函数记住并访问所在的词法作用域，即使函数是在当前词法作用域之外执行，这时就产生了闭包。达成自己私有作用域的函数体就是闭包。

### 概念

> MDN：那些能够访问自由变量的函数
> JavaScript高级程序设计：有权访问另一个函数作用域中的变量的函数
> JavaScript权威指南：从技术的角度，所有的JavaScript函数都是闭包

**从理论上讲：**所有的函数都是闭包，因为闭包的定义是指那些能访问自由变量的函数，而自由变量是指在函数中使用，但既不是函数参数，也不是函数内部声明的局部变量。
**从实际上讲：**即使创建它的上下文已经被销毁，它依然存在（比如内部函数从父级函数中返回），在代码中引用了自由变量。
函数的执行和定义不在同一个作用域，它就是闭包。

### 使用

闭包是在函数被调用的时候才会被确定创建的，闭包的形成与作用域链的访问顺序有直接的关系，只有内部函数访问上层作用域链中的变量对象才会形成闭包。
通过作用域链来看一下：

```
var scope = "global scope";
function checkscope(){
    var scope = "local scope";
    function f(){
        return scope;
    }
    return f();
}
checkscope();
```

```
// fContext的作用域链
ScopeChain = [AO, checkscopeContext.AO,globalContext.VO]
```

当`checkscopeContext`已经被回收了，但是`fContext`依然引用这`checkscopeContext.AO`，保留着`checkscopeContext`中的变量，并进行使用，这就形成了闭包。

### 好处

1. 立即执行函数
2. 类库封装，隔离作用域，避免变量污染
3. 实现类和继承

### 弊端

1. 内存泄露
2. this指向问题
3. 引用的外部变量改变不会在内部不会生效
4. for循环形成闭包（函数工厂、匿名闭包、let）

**注：造成内存泄露的原因：**

1. 全局变量
2. 闭包
3. dom删除或者清空时绑定的事件未清除

浅谈 instanceof 和 typeof 的实现原理

## this指向

this指向的问题归总一下，可分为6种类型：

1. 全局环境、普通函数（非严格模式）this都指向window
2. 普通函数（严格模式）指向undefined
3. 函数作为对象方法及原型链指向的都是上一级对象
4. 构造函数指向构造的对象
5. DOM事件中指向触发事件的元素
6. 箭头函数指向它父级的环境

### 1.全局环境

全局环境下，this始终指向window，无论是否严格模式。

### 2.函数环境

### 2.1普通函数

2.1.1 严格模式下，this指向undefined。
2.1.2 非严格模式下，没有被上一级的对象调用，this默认指向全局对象window

### 2.2函数作为对象的方法被调用

2.2.1 函数由上一级对象所调用，那么this指向的就是上一级的对象

```
var obj = {
    a:22,
    fn:function(){
        return this.a;
    }
}
obj.fn()    // => 22
```

2.2.2 多层嵌套的对象，内部方法的this指向**距离**被调用函数最近的对象（window也是对象，其内部函数调用方法的this指向内部对象，而非window）

```
var obj = {
    a:22,
    b:{
        a:10,
        fn:function(){
            console.log(this.a)
            console.log(this)
        }
    }
}
obj.b.fn()   // => 10
var f = o.b.fn;
f()  // => undefined
```

this始终指向的是**最后调用**它的对象，也可就是说谁调用它，它就指向谁。当使用`obj.b.fn()`调用**执行**函数时，离它最近的对象是b，所以this指向b。第二种方式调用时，虽然函数fn被b所引用，但是却将fn赋值给了变量f，并没有执行，所以this指向了window，而window中没有a属性，故得到undefined。

### 2.3 setTimeOut或setInterval

2.3.1 对于延迟函数内部的回调函数的this指向全局对象window
2.3.2 可以通过bind方法改变内部函数this指向，详见3.4。

### 3.原型链调用

3.1 如果该方法存在于一个对象的原型链上，当通过该对象调用这个方法时，this指向的是调用这个方法的对象。
3.2 call和apply：当函数通过Function对象的原型中继承的方法call()和apply()方法调用时，其中函数内部的this值可绑定到call()或apply()方法指定的第一个对象上，如果第一个参数不是对象，js会尝试将它转成对象然后指向它。
3.4 bind()：有ES5引入，存在于Function的原型链上，Function.propotype.bind()。通过bind绑定后，函数永远绑定到其第一个参数对象上，无论什么时候调用。

```
function f(){
    console.log(this.a)
}
var g = f.bind({a:'ghh'})
g(); // => ghh
var o = {
    a:'指向o',
    f:f,
    g:g
}
o.f() // => 指向o
o.g() // => ghh
```

### 4.构造函数

当一个函数用做构造函数时（通过new创建），它的this指向正在构造的新对象上。构造器默认返回的是this指向的对象，也可以手动返回其他的对象。

```
function c(){
    this.a = 1111;
    return {c:"2222"}
}
var o = new c();
console.log(o.a)  // => undefined
// 注意：这里的构造器中  如果没有return返回值或者返回的为非对象（包括null和undefined）时，通过new构造出来的实例对象，this指向构造函数；如果有return返回对象时，通过new构造出来的实例对象，this指向返回的对象。
```

### 5.在DOM事件中

#### 5.1 作为一个DOM事件处理函数

当函数被用作事件处理函数时，它的this指向触发事件的元素。

#### 5.2 作为一个内联事件处理函数

5.2.1 当代码被内联处理函数直接调用时，他的this指向覆盖监听器所在的DOM元素
5.2.2 当代码被函数包裹内联调用时，等同于普通函数调用，非严格模式下指向window，严格模式下指向undefined，详见2.1。

### 6.箭头函数

#### 6.1 全局环境下

在全局代码中，箭头函数被设置为全局对象

#### 6.2 this捕捉上下文

箭头函数没有自己的this，而是使用箭头函数所在作用域的this。当在setTimeOut或setInterval中使用箭头函数时，this指向构造函数新生成的对象，情况如同2.3.2。

#### 6.3 箭头函数作为对象方法

箭头函数作为对象方法时，this指向全局对象，而普通函数指向该对象。

#### 6.4 箭头函数中，call()、apply()和bind()方法无效

#### 6.5 this指向固定化

箭头函数的this指向不会根据调用它的方式决定，**箭头函数的this永远指向它父级的上下文环境**，这种特性有利于封装回调函数。

#### 6.6 箭头函数不适合场景

6.6.1 箭头函数不适合定义对象的方法，因此指向window
6.6.2 需要动态this的时候

## 异步事件循环

### 浏览器渲染进程

JavaScript是单线程的，这里的单线程指的是：浏览器中负责解释和执行JavaScript代码的只有一个线程，也就是JS引擎线程，但是浏览器的渲染进程是提供多个线程的：

1. **GUI渲染线程**
   a. 负责渲染浏览器界面，解析HTML、CSS、构建DOM树和RenderObject树、布局和绘制等；
   b. 当界面需要重绘（Repaint）或由于某种操作引发回流（reflow）时，该线程就会执行；
   c. 注意：**GUI渲染线程和JS引擎线程是互斥的，JS引擎线程优先级高于GUI渲染线程**，当JS引擎执行时GUI线程会被挂起，GUI更新会保存在一个队列中等到JS引擎空闲时间立即被执行。

2. **JS引擎线程**
   a. 也成为JS内核，负责处理和解析JavaScript脚本程序，运行代码，如V8引擎；
   b. JS引擎一直等待这任务队列中的任务，然后加以处理，render进程中永远只有一个JS线程在运行js程序
   c. 同样注意：**GUI渲染线程和JS引擎线程是互斥的，**JS执行时间过长，就会造成页面的渲染不连贯，导致页面渲染加载阻塞。
3. **事件触发线程**
   a. 归属于浏览器而不是JS引擎，用来控制事件循环；
   b. 当JS引擎执行代码块如：setTimeout、鼠标点击、Ajax异步请求，会将对应的任务添加到事件线程中；
   c. 当对应的时间服务触发条件时，该线程会把事件添加到待处理队列的队尾，等到JS引擎来处理。
4. **定时处理线程**
   a. setInterval与setTimeout所在的线程；
   b. 浏览器定时计数器并不是有JavaScript引擎计数的，它是通过单独线程来计时并触发定时；
   c. 注意：W3C在HTML标准中规定，要求setTimeout中低于4ms的时间间隔算为4ms。
5. **异步http请求线程**
   a. 在XMLHttpRequest在连接后通过浏览器新开一个线程请求
   b. 将检测到状态变更时，如果设置有回调函数，异步线程就会产生状态变更事件，将这个回调再放入事件队列中，再由JavaScript引擎执行。

### 浏览器渲染流程

这里从浏览器内核拿到内容开始，大概经历如下几个步骤：
1. 解析html，建立dom树
2. 解析css，构建render树，将css代码解析树型的数据结构，然后结合DOM合并成render树
3. 布局render树（Layout和reflow），负责各元素尺寸和位置的计算
4. 绘制render树（paint），绘制页面像素信息
5. 浏览器会将各层信息发送给GPU，GPU会将各层合成（composite），显示在屏幕上

注意：具体流程及优化方式将在**前端性能优化**篇章进行详细讲解。

### 事件循环

事件循环机制和异步事件队列（消息队列）的维护是由事件触发线程控制的。
事件触发线程同样是浏览器渲染引擎提供的，它会维护一个异步事件队列。
JS引擎线程遇到异步，就会交给相应的线程单独去维护异步任务，等异步任务执行完成，然后由**事件触发线程**将异步对应的回调函数加入到**异步事件队列**中。
JS引擎线程会维护一个同步执行栈，同步代码会依次加入执行栈进行执行，执行结束后出栈。如果执行栈中的任务执行完成，即同步执行栈空闲，**事件触发线程**就会从异步事件队列中取出最先加入的异步回调函数进行执行，提取规则遵循**先入后出FIFO**规则，异步事件队列类似**队列的数据结构**。

### 微任务宏任务

所有任务分为**微任务**`microtask`和**宏任务**`macrotask`。

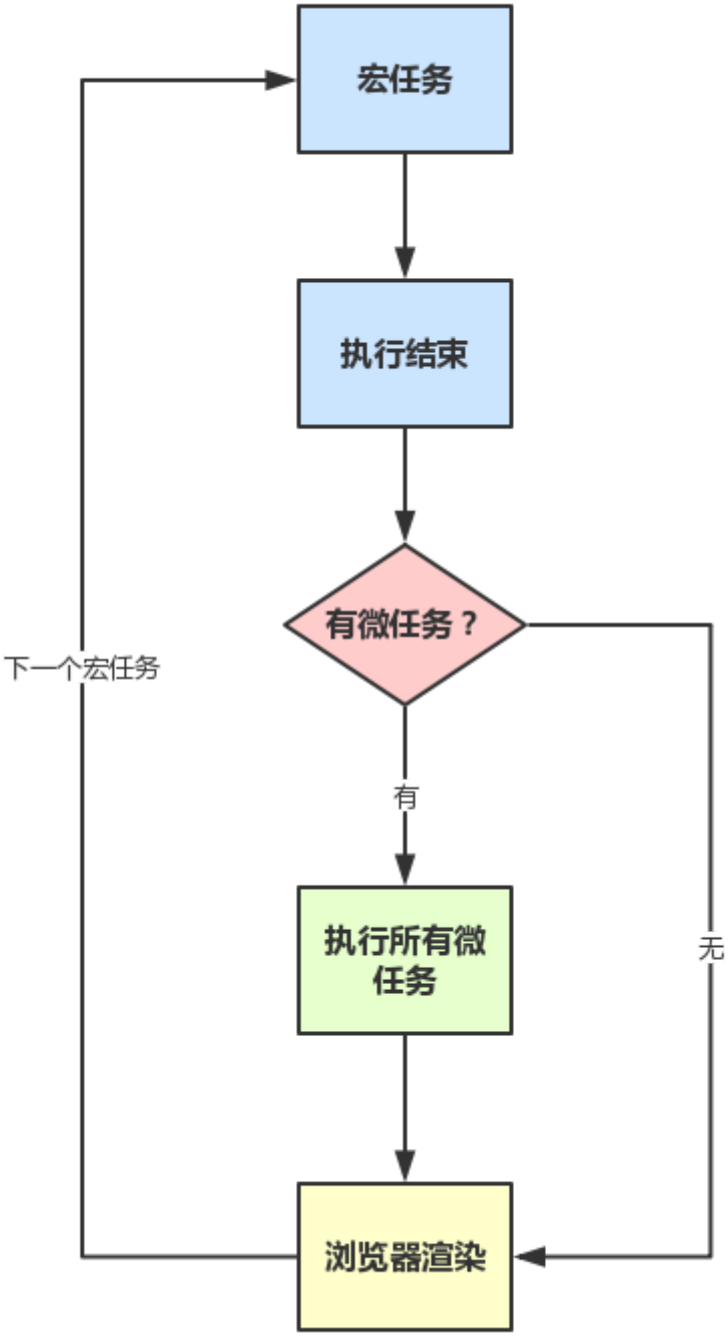
**macrotask:**

- 主代码块
- setTimeout
- setInterval
- I/O（ajax）
- UI rendering
- setImmediate(nodejs)
- 可以看到，事件队列中的每一个事件都是一个 macrotask，现在称之为宏任务队列

**microtask:**

- Promise
- Object.observe(已经废弃)
- MutationObserver
- process.nextTick(nodejs)

先看一张图：




1. 执行一个宏任务（栈中没有就从事件队列中获取，可以理解为一个script标签）
2. 执行过程中如果遇到微任务，就将它添加到微任务的任务队列中
3. 宏任务执行完毕后，立即执行当前微任务队列中的所有微任务（依次执行）
4. 当前宏任务执行完毕，开始检查渲染，然后GUI线程接管渲染
5. 渲染完毕后，JS线程继续接管，开始下一个宏任务（从事件队列中获取）

**注: 宏任务是js引擎进行处理的，微任务是浏览器的行为。微任务必然是由宏任务执行是创建的。**

例子

```
    <script>
      console.log("1-start");
      setTimeout(() => {
        console.log("1-setTimeout");
      }, 0);
      new Promise((resolve, reject) => {
        console.log("1-Promise");
        resolve();
      }).then(() => {
        console.log("1-Promise.then");
      });
      console.log("1-end");
    </script>
    <script>
       console.log("2-start");
      setTimeout(() => {
        console.log("2-setTimeout");
      }, 0);
      new Promise((resolve, reject) => {
        console.log("2-Promise");
        resolve();
      }).then(() => {
        console.log("2-Promise.then");
      });
      console.log("2-end");
    </script>

// 注意中间连个script标签，有和没有的区别
// 有：输出如下
1 - start;
1 - Promise;
1 - end;
1 - Promise.then;
2 - start;
2 - Promise;
```

```
2 - end;
2 - Promise.then;
1 - setTimeout;
2 - setTimeout;
// 无 输出如下
1 - start;
1 - Promise;
1 - end;
2 - start;
2 - Promise;
2 - end;
1 - Promise.then;
2 - Promise.then;
1 - setTimeout;
2 - setTimeout;
```

## 变量提升

引擎会在解释JavaScript代码之前首先对其进行编译。编译阶段中的一部分工作就是找到所有的声明，并用合适的作用域将他们关联起来。因此，正确的思考思路是，**包括变量和函数在内的所有声明会在任何代码被执行之前首先被处理**。注意，**每个作用域都会进行提升操作**。

### 函数提升

函数声明和变量声明都会被提升。但是一个值得注意的细节是函数会首先被提升，然后才是变量。

### 总结

无论在作用域中的声明在什么地方，都将在代码本身被首先执行前处理。可以将这个过程形象地想象成所有的声明（变量和函数）都会被移动到各自作用域的最顶端，这个过程叫**提升**。
声明本身会被提升，而包括函数表达式的赋值在内的赋值操作并不会被提升。

## JavaScript执行堆栈

这里的堆栈和js存储变量的堆栈意义不同。JavaScript代码执行的时候会将不同的变量存储于内存的不同位置：堆heap和栈stack中来加以区分。其中堆中存放一些对象，而栈中存放基本数据类型和对象的指针。
回到正题。JavaScript在执行可执行脚本的时候，首先会创建一个全局的可执行上下文globalContext，每执行一个函数的时候也会创建一个可执行上下文（executionContext EC）。为了管理这些执行上下文，js引擎创建了**执行上下文栈**（Execution Context Stack）ECS来管理这些执行上下文。当函数调用完成后，就会把

当前函数的执行上下文销毁，回到上一个执行上下文... 这个过程会反复执行，直到执行中的代码执行完毕。
需要注意的是，在ECS中永远会有globalContext垫底，永远存在于栈底。
在这个过程中要注意如下几个变量：

1. **全局可执行上文**：globalContext
2. **可执行上下文**：executionContext EC
3. **执行上下文栈**：Execution Context Stack　ECS
4. **变量对象**：Variable Object VO
5. **活动对象**：Activation Object　AO

### 执行栈压栈顺序

一开始执行js代码的时候，就会创建一个全局可执行上下文`global execution context`压入执行上下文栈栈底，当调用了函数时，程序会创建一个新的EC，然后压入ECS中，当该函数中调用了其他的函数时，又会创建一个新的EC，然后压入栈中。一旦EC执行完毕后，就会从ECS中推出。

**变量对象**

变量对象VO是与执行上下文相关的特殊对象，用来存储上线文中的**函数声明**、**形参**和**变量**。

1. 函数声明FD，不包含函数表达式
2. 函数形参function arguments
3. 变量声明

**活动对象**

在函数上下文中，变量对象被激活为活动对象AO。活动对象在函数上下文中作为变量对象使用。活动对象又分为**创建阶段**和**激活/执行阶段**。

1. 创建阶段

   会发生**属性名的定义，而不进行变量赋值**。在此阶段会发生重要的**变量提升**。
   - 在函数执行上下文中VO不能被直接访问，此时活动对象扮演着VO的角色
   - Arguments对象，它包含如下属性：callee、length。在此阶段会对函数的形参进行赋值
   - 内部定义的函数
   - 以及绑定上对应的变量环境
   - 内部定义的变量
   - 不包含函数表达式
   - 在此阶段所包含的内容和VO一样
2. 激活/执行阶段

   一旦创建阶段结束，便进入了激活/执行阶段，那么当前执行上下文就会对AO进行赋值。

### 执行上下文环境

下面来对比一下当前函数执行上下文的变化：

```
function bar(){
    function foo(a,b){
        var c = 'Jokul',
```

```
        var d = function o(){}
        function e (){}
        (function f(){})
    }
   return foo(1,2)
}
// 创建阶段
foo:ExecutionContext={
    // 在此阶段进行变量提升
    AO:{
        arguments:{
            0:'1',
            1:'2',
            length:2
        },
        a:1,
        b:2,
        c:undefined,
        d:undefined,
        e:pointer to function c()
    },
    scopeChain:{Scope},
    VO:{...},
    Scope:[AO,globalContext.VO]
}
// 激活/执行阶段
foo:ExecutionContext={
    AO:{
        arguments:{
            0:'1',
            1:'2',
            length:2
        },
        a:1,
        b:2,
```

```
        c:'Jokul',
        d: pointer to function o(),
        e:pointer to function c()
    },
    scopeChain:{Scope},// 作用域链 即下边的Scope 包含 当前活动变量，父执行上下文的AO...以及全局执行上下文的VO
    VO:{...},
    Scope:[AO,barExecutionContext.AO,globalContext.VO], // 闭包的原理就是当bar环境已经被销毁，但是foo的作用域链中还保存着bar中的变量，这就形成了闭包。
    this:'运行时确认'// this的原理是动态绑定，永远指向ECS的栈顶
}
```

### 总结

闭包的原理是Scope，this的原理是动态绑定，作用域链的原理是`Scope:[AO,barExecutionContext.AO,globalContext.VO]`。变量提升发生在AO的准备阶段，异步队列的原理是ECS。

## 原型链

### 概念

上边说到instanceof的原理就是原型链的查找。那么原型链到底是什么？
**当访问一个对象的属性时，会先从这个对象身上查找。如果没有找到就会去它的`__proto__`隐式原型上查找，即它的构造函数的prototype。如果还未找到就继续去它的构造函数的`__proto__`上找。这样一层一层的向上查找，就成了一个链式结构，称为原型链。**

### Prototype

每个js对象在创建的时候就会有一个与之关联的另一个对象，这个对象就是原型，每个对象就会从原型上继承属性。称为**显示原型**。

### __proto__

每个js对象都具有一个属性，这个属性指向该对象的原型。称为**隐式原型**。

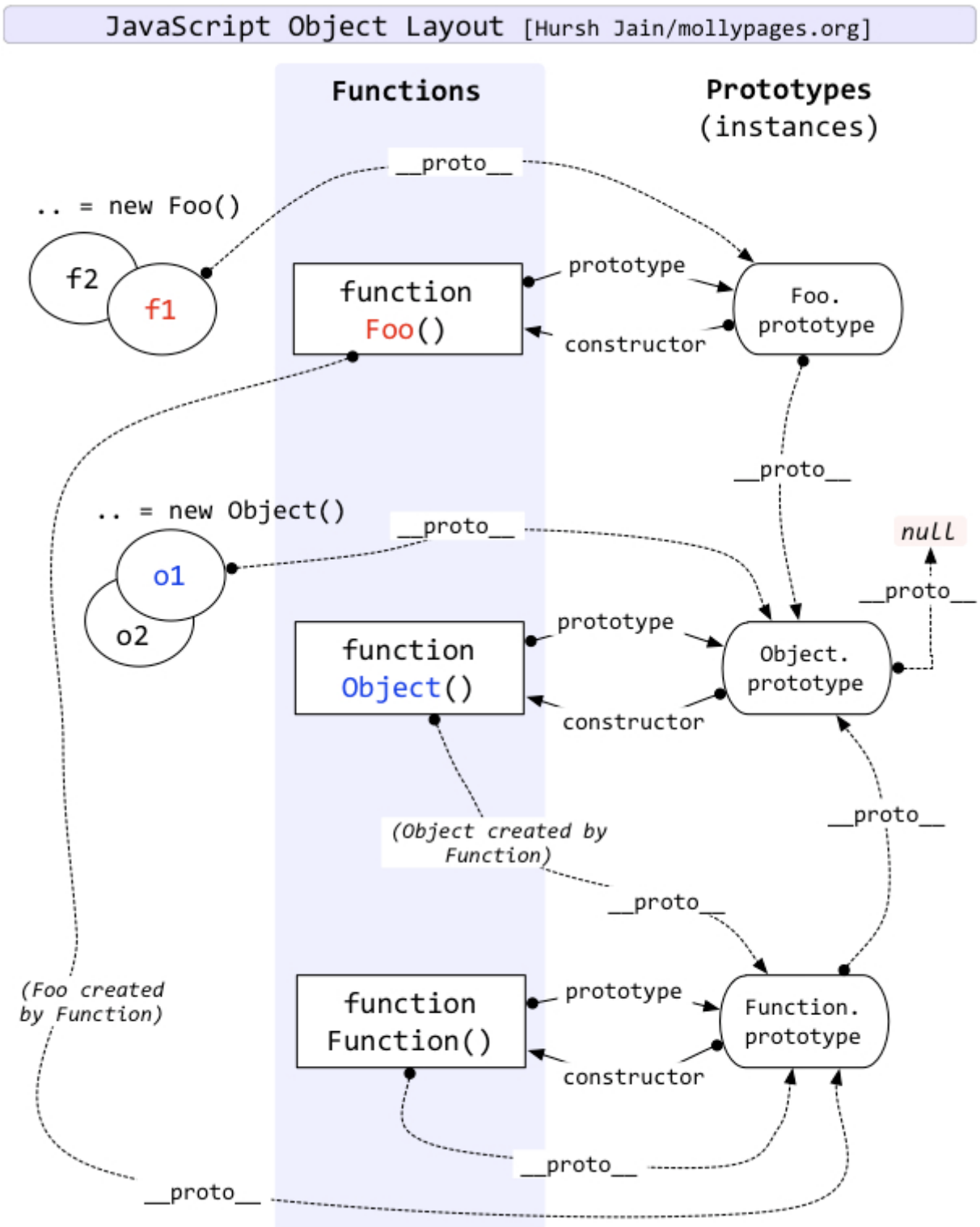
### constructor

一个构造函数可以生成多个实例，但是原型指向构造函数，每个原型都有一个constructor属性指向关联的构造函数。

### 相关性

1. JavaScript中的**继承**是通过原型链实现的
2. 所有构造函数的`__proto__`都指向Function.prototype
3. 每个构造函数constructor都有一个原型对象prototype，原型对象都包含一个指向构造函数的指针，而实例instance都包含一个指向原型对象的内部指针。

**神图**




1. **实例的隐式原型等于构造函数的显式原型**，`f1.__proto__==Foo.prototype`
2. 函数类型有`prototype`，对象有`__proto__`

3. 所有的构造函数也属于函数，所以`Foo/Object.__proto__==Function.protoype`
4. 函数的构造函数也属于函数，所以`Function.__proto__== Function.protoype`
5. `Object.prototype.__proto__ == null`

## 继承

https://juejin.im/post/58f94c9bb123db411953691b#heading-3
https://juejin.im/post/5b654e88f265da0f4a4e914c#heading-0
https://juejin.im/post/58f94c9bb123db411953691b
在面向对象中，继承分为**接口继承**和**实现继承**。而JavaScript中，ECMAScript中无法实现接口继承，只支持实现继承，而实现继承主要依靠的是**原型链**来实现的。
在JS中实现继承的方式有如下6种：
1. **原型链**
2. **借用构造函数**
3. **组合继承**
4. **原型式继承**
5. **寄生式继承**
6. **寄生组合式继承**

### 原型链

ECMAScript中描述了原型链的概念，并将原型链作为实现继承的主要方法。其基本思想是利用原型链 上让一个引用类型继承另一个引用类型的属性和方法。每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个纸箱原型对象的内部指针。

```
// 《JavaScript高级程序设计》中给出了如下实现方式
function SuperType(){
  this.property = true;
}
SuperType.prototype.getSuperValue = function(){
  return this.property;
}
function SubType(){
  this.subProperty = false;
}
//继承 SuperType
SubType.prototype = new SuperType();//SubType.prototype被重写,SubType.prototype.constructor也一同被重写
SubType.prototype.getSubVaule = function(){
```

```
  return this.subProperty;
}
// 比较严谨的写法要添加下边这段
SuperType.prototype.getSuperValue = function(){
  return false;
}
var instance = new SubType();
alert(instance.getSuperValue());//true
```

但是这种写法存在一个很严重的问题：

1. 当原型链中包含引用类型值的原型时，该引用类型值会被所有实例共享；
2. 在创建子类型（例如创建SubType实例）时，不能向超类型（SuperType）的构造函数中传递参数。

### 构造函数

为了解决上面的两个问题，所以要借用构造函数的技术（也叫经典继承）。主要原理就是**在子类型构造函数的内部调用超类型构造函数。**

```
function SuperType(){
    this.colors = ["red","blue","green"];
}
function SubType(){
    //继承了SuperType,且向父类型传递参数
    SuperType.call(this);
}
var instance1 = new SubType();
instance1.colors.push("black")
console.log(instance1.colors);//"red,blue,green,black"

var instance2 = new SubType(); console.log(instance2.colors);//"red,blue,green" 可见引用类型值是独立的
```

借用构造函数的方式解决了上面原型链方法带来的问题：

1. 保证了原型链中引用类型值的独立，不再被所有实例共享；
2. 子类型在创建时也能向父类型传递参数。

但是如果仅仅借用构造函数，那么将无法避免构造函数模式存在的问题--方法都在构造函数中定义，因此函数复用就无从谈起了。而且，在超类型的原型中定义的方法，对子类型而言也是不可见的。结果所有类型都只能使用构造函数模式。考虑到这些，借用构造函数的技术也是很少单独使用的。

### 组合继承

组合继承，也称作伪经典继承，指的是将原型链和借用构造函数的技术组合到一块。其背后的思路是**使用原型属性和方法的继承，而通过借用构造函数来实现对实力属性的继承**。这样，即通过原型上定义方法实现了函数复用，又能保证每个势力都有自己的属性。

```
function SuperType(name){
    this.name = name;
    this.colors = ["red","blue","green"];
 }
SuperType.prototype.sayName = function(){
    alert(this.name);
};
function SubType(name,age){
    SuperType.call(this,name);//继承实例属性，第一次调用SuperType()          t
    this.age = age;
}
SubType.prototype = new SuperType();//继承父类方法,第二次调用SuperType()
SubType.prototype.constructor = SubType;
SubType.prototype.sayAge = function(){
    alert(this.age);
}
var instance1 = new SubType("louis",5);
instance1.colors.push("black"); console.log(instance1.colors);//"red,blue,green,black" =
instance1.sayName();//louis
instance1.sayAge();//5

var instance1 = new Son("zhai",10); console.log(instance1.colors);//"red,blue,green"
instance1.sayName();//zhai
instance1.sayAge();//10
```

组合继承避免了原型链和借用构造函数的缺陷，融合了他们的优点，成为JavaScript中最常见的继承模式，而且，instanceof和isPrototypeOf()也能用于识别基本组合继承的对象。但是调用了两次父类构造函数，造成了不必要的消耗。

### 原型继承

基本思路是借助原型可以基于已有的对象创建新对象，同时还不必因此创建自定义类型。**在object()函数内部，先创建一个历史性的构造函数，然后将传入对象作为这个构造函数的原型，最后返回了这个临时类型的**

**一个新实例。从本质上讲，object()对传入其中的对象执行了一次浅拷贝。**
在ES5中，新增了一个`Object.create()`方法规范了原型式继承。这个方法接受两个参数：一个是用做新对象原型的对象和一个为新对象定义额外属性的对象。在传入一个参数的情况下，和`Object()`方法行为相同。

```
var person = {
    name: 'Jokul',
    friends:['JS','CSS','HTML']
}
var anotherPerson = Object.create(person, { name: { value: 'HEIHEI' } })
alert(anotherPerson.name) //"HEIHEI"
```

**注意：在原型继承中，包含引用类型值的属性，始终都会共享相应的值，就像使用原型模式一样。**

### 寄生式继承

寄生式继承是与原型式继承紧密相关的一种思路。它的思路与寄生构造函数和工厂模式类似，即**创建一个仅用于封装继承过程的函数，该函数在内部已某种方式来增强对象**，最后再像真地是它做了所有工作一样返回对象。

```
function createAnother(original){
    var clone = object(original);//通过调用object函数创建一个新对象
    clone.sayHi = function(){//以某种方式来增强这个对象
        alert("hi");
    };
    return clone;//返回这个对象
}
```

**注意：使用寄生式继承来为对象添加函数，会由于不能做到函数复用而降低效率。**

### 寄生组合式继承

组合继承是JavaScript最常见的继承模式，不过它最大的缺点就是无论在什么时候，都会调用两次超类型构造函数：一次是在创建子类型原型的时候；一次是在子类型构造函数内部。**寄生组合式继承就是为了降低调用父类构造函数的开销而出现的。**

```
function inheritPrototype(subClass,superClass){
    var prototype = object(superClass.prototype);//创建对象
    prototype.constructor = subClass;//增强对象
    subClass.prototype = prototype;//指定对象
}
```

### 终极版继承实现方式

```
function Rectangle(length,width){
    this.l = length
    this.w = width
}
Rectangle.prototype.getArea = function(){
    return this.l*this.w
}
function Square(length){
    // 继承Rectangle的变量及私有变量
    Rectangle.call(this,length,length)
}
// Square对Rectangle的原型的继承    将Square写入Rectangle的constructor
Square.prototype = Object.create(Rectangle.prototype,{
    constructor:{
    value:Square
    }
})
// 上面这段等于下边的,
// Rectangle.prototype.constructor = Square
var square = new Square(3)
console.log(square.getArea())
console.log(square instanceof Square)
console.log(square instanceof Rectangle)
```

### New操作符

new操作符具体进行下面这三步操作：

1. 创建一个空对象；
2. 将这个对象的`__proto__`成员指向父函数对象的`prototype`成员对象；
3. 将父函数对象的this指针替换成子对象，然后在调用父函数

## 深拷贝浅拷贝

### 浅拷贝

创建一个新对象，这个对象有这原始对象属性值的一份精确拷贝。如果属性是基本数据类型，拷贝的就是基本类型的值，如果是引用类型，拷贝的就是内存地址。所以如果其中一个对象改变了这个地址，就会影响到另一个对象。

```
// 浅克隆的方法如下
Object.assign(target,...source)
let target = {};
let source = {a:{b:2}}
Object.assign(target,source);
target.a === source.a //true
```

## 深拷贝

将一个对象从内存中完成的拷贝一份出来，从堆内存中开辟一个新的区域存放新的对象，且修改新对象不会影响原对象。

```
const isComplexDataType = obj => typeof obj === "object";
const deepClone = function(obj, hash = new WeakMap()) {
  if (hash.has(obj)) return hash.get(obj);
  let type = [Date, RegExp, Set, Map, WeakMap, WeakSet];
  if (type.includes(obj.constructor)) return new obj.constructor(obj);
  //  如果成环了，参数obj = obj.loop = 最初obj会在WeakMap中找到第一次放入的obj提前返回，第一次放入WeakMap的cloneObj
  // 遍历传入参数所有键的特性
  let allDesc = Object.getOwnPropertyDescriptors(obj);
  // 继承原型
  let cloneObj = Object.create(Object.getPrototypeOf(obj), allDesc);
  hash.set(obj, cloneObj);
  for (let key of Reflect.ownKeys(obj)) {
    // Reflect.ownKeys(obj)可以拷贝不可枚举属性和符号类型
    // 如果值是引用类型（非函数）则递归调用deepClone
    cloneObj[key] =
      isComplexDataType(obj[key]) && typeof obj[key] !== "function"
        ? deepClone(obj[key], hash)
        : obj[key];
  }
  return cloneObj;
};
```

```
22
```

# JS模块化

## CommonJS

CommonJS的出发点：JS没有完善的模块系统，标准库较少，缺少包管理工具。伴随着NodeJS的兴起，能让JS在任何地方运行，特别是服务端，也达到了具备开发大型项目的能力，所以CommonJS营运而生。

### CommonJS规范

- 一个文件就是一个模块，拥有单独的作用域
- 普通方式定义的变量、函数、对象都属于该模块内
- 通过require来加载模块
- 通过exports和module.exports来暴露块中的内容

### 注意

1. 当exports和module.exports同时存在时，module.exports会覆盖exports
2. 当模块内全是exports时，就等同于module.exports
3. exports就是module.exports的子集
4. 所有代码都运行在模块作用域，不会污染全局作用域
5. 模块可以多次加载，但只会在第一次加载时候运行，然后运行结果就被缓存了，以后再加载，就直接读取缓存结果
6. 模块加载顺讯，按照代码出现的顺序同步加载
7. __dirname代表当前磨具爱文件所在的文件夹路径
8. __filename代表当前模块文件所在的文件夹路径+文件名

## ES6模块化

## AMD

Asynchronous Module Definition，异步加载模块。它是一个在浏览器端模块化开发的规范，不是原生js的规范，使用AMD规范进行页面开发需要用到对应的函数库，RequireJS。

RequireJS主要解决的问题：

1. 多个js文件可能有依赖关系，被依赖的文件需要早于依赖它的文件加载到浏览器
2. js加载的时候浏览器会停止页面渲染，加载文件愈多，页面相应事件就越长
3. 异步前置加载

语法

```
define(id,dependencies,factory)
——id 可选参数，用来定义模块的标识，如果没有提供该参数，脚本文件名（去掉拓展名）
——dependencies 是一个当前模块用来的模块名称数组
——factory 工厂方法，模块初始化要执行的函数或对象，如果为函数，它应该只被执行一次，如果是对象，此对象应该为模块的输出值。
```

## CMD

因为CMD推崇一个文件一个模块，所以经常就用文件名作为模块id； CMD推崇依赖就近，所以一般不在define的参数中写依赖，而是在factory中写。

```
define(id, deps, factory)

factory有三个参数： function(require, exports, module){}
```

1. require require 是 factory 函数的第一个参数，require 是一个方法，接受 模块标识 作为唯一参数，用来获取其他模块提供的接口；
2. exports exports 是一个对象，用来向外提供模块接口；
3. module module是一个对象，上面存储了与当前模块相关联的一些属性和方法。

## 总结

commonjs是同步加载的。主要是在nodejs 也就是服务端应用的模块化机制，通过module.export 导出声明，通过require('')加载。每个文件都是一个模块。他有自己的作用域，文件内的变量，属性函数等不能被外界访问。node会将模块缓存，第二次加载会直接在缓存中获取。
AMD是异步加载的。主要应用在浏览器环境下。requireJS是遵循AMD规范的模块化工具。他是通过define()定义声明，通过require('',function(){})加载。
ES6的模块化加载时通过export default 导出 用import导入 可通过 {} 对导出的内容进行解构。
ES6的模块的运行机制与common不一样，js引擎对脚本静态分析的时候，遇到模块加载指令后会生成一个只读引用，等到脚本真正执行的时候才会通过引用去模块中获取值，在引用到执行的过程中 模块中的值发生了变化，导入的这里也会跟着变，ES6模块是动态引用，并不会缓存值，模块里的比那辆绑定所在的模块。