



9. 面向对象 (OOP)

[普通函数执行](#)

[构造函数设计模式](#)

[instanceOf](#)

[装箱与拆箱机制](#)

[obj.hasOwnProperty\(\)](#)

[\[attr\] in obj](#)

[hasOwnProperty:](#)

[prototype\(原型\)](#)

[函数数据类型](#)

[原型重定向](#)

[__proto__\(原型链\)](#)

[对象数据类型值:](#)

[new 执行原理](#)

[类class](#)

[THIS](#)

[继承](#)

[1. 原型继承:](#)

[2. call\(寄生组合继承\)](#)

[class继承](#)



JavaScript是基于面向对象的编程语言, 他本身是基于面向对象的思想构建出来的

1. **对象:** 即每一个研究对象, 树,牛,羊
2. **类:** 对对象的归类与抽象: 人类, 灵长类
 1. 类是函数类型
3. **实例:** 具体到某一个体, 比如个人,某只具体的狗
 1. 实例对象则是对象类型

JavaScript中的类:

内置类:

数据类型: Number, String, Boolean, Symbol,BigInt, Object(Array,RegExp), Function

具体值, 就是每个类对应的实例

标签也是一个实例对象, 如DIV则属于HTMLDivElement类, getComputedStyle则是他们所属的公共方法

Div ⇒ HTMLDivElement ⇒ HTMLElement ⇒ Node ⇒ EventTarget ⇒ Object基类

自定义类:

普通函数执行

```
function Fn(x, y) {
  let total = x + y,
      flag = false;
  this.total = total;
  this.say = function say() {
    console.log(this.total);
  };
}
```

堆内存 `[[scope]]`: EC(G)

Fn(10,20) 普通函数执行
EC(FN1)

AO(FN1)

x	_____	10
y	_____	20
total	_____	30
flag	_____	false

作用域链: <EC(FN1),EC(G)>

初始THIS: window

形参赋值: x=10 y=20

代码执行:

```
let total = x + y,
    flag = false;
this.total = total; window.total = 30
this.say = function say() {
  console.log(this.total);
}; window.say = 0x001 [[scope]]:EC(FN1)
```

return undefined; 不设返回值, 默认返回undefined

```
let result = Fn(10, 20);
console.log(result);
undefined
```

构造函数设计模式



很多后台也称其为单例设计模式

构造函数在创建于执行时与普通函数的差异:



构造函数执行时产生的执行上下文, 与创建的新实例对象没有直接关系

1. 在初始化this指向时, 构造函数的this会指向一个**空的实例对象**.
2. 执行时所有的this.xx访问的变量都为实例对象内的变量
3. 构造函数默认return实例对象
 1. 返回普通数值, 依旧默认返回实例对象.
 2. 返回一个对象, 则覆盖原本的实例对象.

instanceOf



检测某个实例是否属于哪个类

```
obj instanceof Object // true
```

装箱与拆箱机制

```
let n1 = 10 // 字面量
let n2 = new Number(10) // new 一个实例对象

n2.toFixed(2) // ==> 10.00
n1.toFixed(2) // ==> 10.00
// 字面量可以调用数字类型方法的处理机制: 装箱
// 内部处理机制:
// Object(10) => Number{10}
n2 + 10 // 20
// 实例对象与原始值相加不会变成字符串拼接: 拆箱
// n2[Symbol.toPrimitive]/ ValueOf
```

obj.hasOwnProperty()



检测一个属性是不是为对象的私有属性

```
// 那个构造函数的实例来看
function Fn(x, y) {
  let total = x + y,
      flag = false;
  this.total = total;
  this.say = function say() {
    console.log(this.total);
  };
}
let result1 = Fn(10, 20);
let result2 = Fn(10, 20);
```

```
result1.hasOwnProperty('say') => true
result1.hasOwnProperty('hasOwnProperty') => false // 说明hasOwnProperty是result1的公共属性
```

[attr] in obj

[attr]是否为对象的属性, 无论私有还是公有

```
'say' in result1 => true
'hasOwnProperty' in result1 => true
```

hasPubProperty:

```
function hasPubProperty (obj, attr) {
  return (attr in obj) && !obj.hasOwnProperty(attr)
}
```

弊端: 如果属性即是公有属性又是私有属性的话, 则返回的是false

解决办法:

Object.keys(obj)

Object.getOwnPropertyNames(obj)



获取对象的非Symbol私有属性

获取所有私有属性:

```
function each(obj, callback) {
  let keys = Object.keys(obj)
  let i = 0
  let len = 0
  let key

  if (typeof Symbol !== 'undefined') {
    keys = keys.concat(Object.getOwnPropertySymbols(obj))
  }
  len = keys.length
  if (typeof callback !== 'function') callback = function () {}
  for (; i < len; i++) {
    key = keys[i]
    value = obj[key]
    callback(key, value)
  }
}
each(obj, (value, key) => {
  console.log(value, key)
})
```

检测一个属性是否为当前对象的公有属性:

```
Object.prototype.hasPubProperty = function hasPubProperty(attr) {
  let self = this,
      prototype = Object.getPrototypeOf(attr)
  while (prototype) {
    // 原型对象是否存在私有属性(则实例对象上不存在.)
    if (prototype.hasOwnProperty(attr)) {
      return true
    }
    prototype = Object.getPrototypeOf(prototype)
  }
  return false
}
```

prototype(原型)



大部分'函数数据类型'都具备prototype原型属性, 属性本身是一个对象, 浏览器默认为其开启一个堆内存, 用来存储当前类所属实例, 可以调用的公共属性和方法...

函数数据类型

1. 普通函数
2. 箭头函数
3. 构造函数/类()
4. 生成器函数 generator

不具备prototype的函数

1. 箭头函数
2. 基于ES6给对象某个成员赋值函数数值的快捷操作

```
let obj = {
  fn1: function() {},
},
fn2() {
  // 快捷写法, 不具备prototype属性
}
```

```

    }
  }
  class Fn {
    fn() {} // 不具备
  }

```

原型重定向



手写的内置类允许重定向, 但是内置类只允许添加, 不允许重定向, 但是内置类的方法可以进行单一的重写

原型中添加新方法的方式:

1. Object.assign([obj1], [obj2])

1. 方法返回合并后的obj1.
2. obj2不发生改变.
3. Object.assign({}, obj1, obj2) 返回一个全新的对象

2. 闭包的形式批量填充

```

// 1.
Object.assign(Number.prototype, {
  plus: function plus(num) {
    return num++
  }
})
// 2.
;(function (proto) {
  const plus = function plus(num) {
    //...
    return num++
  }
  proto.plus = plus
})(Number.prototype)

```

__proto__(原型链)



每一个'对象数据类型'的值都具备一个属性'__proto__'(原型链/隐式原则). 属性值指向自己所属类的原型prototype

对象数据类型值:

1. 普通对象
2. 特殊对象: 数组, 正则, 日期, Math, Error
3. 函数对象
4. 实例对象
5. 构造函数.prototype
6.

new 执行原理



new执行构造函数, 并返回一个实例对象, 实例对象的原型链指向构造函数的原型对象prototype

```

// 实现一个new
/*
1. 构造函数Ctor的原型对象存在
2. 不能是Symbol或者BigInt类型, 他们不能被new
3. Ctor必须是一个函数
4. __proto__去改变实例对象的原型链不是一个好的做法, 兼容性不好.
*/

```

```

    应该使用Object.create().
    5. 对于new执行的函数来说, 默认情况下返回实例对象,
       但是如果我们手动改写了return的值, 且是object或者function,
       则return会被改写
*/
function _new(Ctor, ...params) {
    let prototype = Ctor.prototype
    obj,
    result,
    ct = typeof Ctor
    if(ct !== 'function' || Ctor === Symbol || Ctor === BigInt || !prototype) {
        throw new TypeError(`${Ctor} is not a constructor!`)
    }

    obj = Object.create(prototype)
    result = Ctor.apply(obj, params)

    if(result !== null && /^(function|object)/.test(typeof result)) return result
    return obj
}


```

```

// 手写一个Object.create(obj)
Object.create = function create(prototype){
    if (prototype !== null && typeof prototype !== 'object') {
        throw new TypeError('')
    }
    function Proxy(){}
    Proxy.prototype = prototype
    return new Proxy()
}

```

类class

 相当于一个构造函数, 但无法直接执行, 只能通过new的形式执行

```

class Fn {
    // 构造函数体
    constructor(name) {
        //-----
        // 直接挂载实例对象上的内容, this即使实例对象
        this.name = name
        this.getY = function getY() {}
    }
    //-----
    x = 100 // 等价于构造函数体中的“this.x=100”
    // 原型上的内容 「无法直接设置原型上属性值非函数的公有属性」
    //-----
    // + 但是这样这样设置的函数是没有prototype的, 类似于: obj={fn() {}}
    // 直接挂在原型上
    getX() {
        console.log(1)
    }
    getName() {}

    //-----
    // 看做普通对象, 设置私有属性「静态私有属性和方法」
    static x = 1000
    static getX() {
        console.log(2)
    }
}
//-----
Fn.prototype.y = 200

// Fn(); //Uncaught TypeError: Class constructor Fn cannot be invoked without 'new' 基于class声明的构造函数必须基于new执行, 不允许当做普通函数
let f = new Fn('zhufeng')

```

THIS

1. 给当前元素绑定事件时, this指向的就是元素本身(排除IE6~8)
2. 方法执行"点字诀"
3. 箭头函数, 块级上下文, 没有this, 如果上下文中出现了this, 则默认继承上个上下文的this.

4. 构造函数执行, this指向实例对象
5. Function.prototype中提供了 call/apply/bind三个方法去改变this的指向.
6. apply方法要求第二个参数为一个数组, 将需要的实参以数组的形式传入.
7. bind则类似于异步执行, 不会要求函数一旦传入就立即执行(事件单击)



call(this, [value])与apply(this, [arr])

```
// 通过call让类数组借用数组方法.
const sum = function sum() {
  let params = arguments
  if (params.length === 0) return 0
  // 不转换了, 直接借用即可
  return [].reduce.call(params, (total, item) => total + item, 0)
}
```

```
let obj = {
  2: 3,
  3: 4,
  length: 2,
  push: Array.prototype.push
}
obj.push(1);
obj.push(2);
console.log(obj);
/*
输出:
let obj = {
  2: 1,
  3: 2,
  length: 4,
  push: Array.prototype.push
}

*/

Array.prototype.push = function push(val) {
  this[this.length] = val
  // this.length++ length属性自动递增
}
```

call方法实现

```
Function.prototype._call = function _call(context, ...params) {
  // this => obj
  // params传给原函数执行, 在将返回值原路返回
  context == null ? (context = window) : null
  if (!/(function|object)$/i.test(typeof context)) context = Object(context)
  let self = this,
      result,
      key = Symbol('key')

  context[key] = self
  result = context[key](...params)
  delete context[key]
  return result
}
```

bind的实现原理:

```
//=>bind方法在IE6-8中不兼容, 接下来我们自己基于原生JS实现这个方法
function bind(ctx, ...params) {
  let self = this
  return function proxy() {
    params = params.concat(...arguments)
    return self._call(ctx, ...params) // 这里实现与_call是一样的
  }
}
Function.prototype.bind = bind
```

继承

1. 原型继承:



子类指向了父类的实例, 使得子类能够使用父类的私有和公有属性方法, 与其他语言不同, 原型继承不是把父类的属性和方法"拷贝"给子类, 而是让子类的实例基于__proto__原型链找到了父类的实例和原型的方法与属性.

```
function parent () {  
  this.x = 100;  
}  
Parent.prototype.getX = function(){}  
  
function Child () {  
  this.y = 100;  
}  
Child.prototype = new Parent; // 让子类的原型指向父类的实例  
Child.prototype.getY = function(){}  

```

注意:

1. 修改某一个子类的原型, 会导致其他继承的子类有影响.
2. 同样, 可以通过原型链去修改父亲原型, 这样不仅会影响其他子类, 也会影响父类的实例

2. call(寄生组合继承)

```
function parent () {  
  this.x = 100;  
}  
Parent.prototype.getX = function(){}  
  
function Child () {  
  
  Parent.call(this) // 让子类的实例继承了父类私有的属性, 也变为子类私有的属性  
  this.y = 100;  
}  
Child.prototype.__proto__ = Parent.prototype // 将子类原型链指向父类原型  
Child.prototype.getY = function(){} // 这样子类就可以访问父类原型的公有方法  

```

__proto__访问不支持的兼容办法:



Object.create(obj): 创建一个空对象, 其原型链指向obj

```
function parent () {  
  this.x = 100;  
}  
Parent.prototype.getX = function(){}  
  
function Child () {  
  
  Parent.call(this) // 让子类的实例继承了父类私有的属性, 也变为子类私有的属性  
  this.y = 100;  
}  
Child.prototype = Object.create(Parent.prototype)  
Child.prototype.constructor = Child  
Child.prototype.getY = function(){}  

```

class继承

```
class Parent {  
  constructor() {  
    this.x = 100  
  }  
}
```



```
    getX() {  
        return this.x  
    }  
}  
// 类的继承要在子类的构造体中加一个super()  
class Child extends Parent {  
    constructor() {  
        super() // 类似于call继承  
        this.Y = 200  
    }  
    getX() {  
        return this.y  
    }  
}
```