day19 原型和继承

面向对象的回顾

对象的声明方式

使用new关键词

使用class

```
class Person{
    constructor(name){
        this.name = name
    }
}
let person = new Person('tom')
```

使用构造函数

```
function Person(name){
    this.name = name
}
let person = new Person('jack')
```

使用工厂函数

```
function factory(name){
    let obj = new Object()
    obj.name = name
    return obj
}
let obj = factory('rose')
```

构造函数的缺陷

```
function Person(){
    this.name = 'jack'
    this.sayHello = function(){
        console.log('hello')
    }
}
let person1 = new Person()
let person2 = new Person()
console.log(person1.name == person2.name)
person1.sayHello() //hello
person2.sayHello() //hello
console.log(person1.sayHello ==  person2.sayHello)//false person1的调用的方法和
person2调用的不一致
```

- person1的调用的方法和person2调用的不一致
- 不一致也就是说person1里面有个sayHello的内存空间 person2里面也有一个sayHello内存空间

- 那么就是new多少次就会创建多少次的sayHello的内存空间 (内存浪费)
- 如果要解决这个问题 那么在调用对应的构造函数的时候就不能创建对应的一个sayHello的空间（**对应的对象的函数定义不要放在构造函数中**）
- 也就是说我们要将所有的sayHello只有一个空间 (**公共空间**)
- 从上面可以看出来对应的这个公共空间必须跟对应的构造函数产生联系 （必须所有实例都能调用）

原型

函数的原型（prototype）

概述

prototype是函数内的一个对象空间，每个函数都有一个，他被称为显式原型。

应用

- 对应的构造函数也属于函数 那么他同样也拥有一个prototype，且对应的prototype只有一个也就是说他会在预编译时候声明一次，也就是prototype是对应的一个构造函数的公共空间，且他只声明一次 那么也就说他就可以解决对应的构造函数的缺陷了。

```
function Person(){
    this.name = 'jack'
}
//构造函数的prototype空间
console.log(Person.prototype)
```

- 从上可得对应的函数的prototype这个空间他其实是一个对象，那么我们是不是可以在这个对象里面存入对应的函数

```
function Person(){
    this.name = 'jack'
}
//在构造函数的prototype中存入对应的函数
Person.prototype.sayHello = function(){
    console.log('hello')
}
let person1 = new Person()
person1.sayHello()
let person2 = new Person()
person2.sayHello()
console.log( person1.sayHello ===  person2.sayHello) //true
```

- 以上代码发现 对应的函数的prototype中存入的属性 可以直接通过对应的实例.属性来访问

总结

- 每个函数内都存在一个**prototype的对象空间** 构造函数也是函数所以他也存在
- prototype这个空间会在预编译的时候进行开辟 （只开辟一次）
- 利用prototype可以解决构造函数内存储的函数开辟多个内存空间的问题
- prototype里面的方法 可以直接通过对应的**实例对象.方法名**来访问
- 建议将对应的属性**存入对应的构造函数** 将对应的**方法存入prototype**

class的机制

- 将contructor外部声明的函数自动加入到原型中

```
class Animal{
    constructor(){

    }
    //声明一个函数 在prototype里面
    say(){
        console.log('hello')
    }
}
new Animal().say()
new Animal().say()
console.log(new Animal().say == new Animal().say)//true
```

对象的原型（__proto__）

__proto__是对象的一个对象空间（实例对象也属于对象），他指向对应的构造函数的prototype，他被称为隐式原型。

```
console.log({}.__proto__)
//对象的__proto__ 指向对应构造函数prototype
console.log(new Object().__proto__ == Object.prototype)
console.log({}.__proto__ == Object.prototype)
```

__proto__ 指向对应的构造函数的prototype 那么也就意味着如果往对象的__proto__里面添加内容，其实就往构造函数的prototype中添加。

```
//往对象的隐式原型中添加对应的内容 其实就是往构造函数的prototype中添加内容
new Person().__proto__.sayHello = ()=>console.log('hello')
new Person().sayHello()
//同一构造函数产生的实例对象的 __proto__的对象是一个
console.log( new Person().__proto__ ==  new Person().__proto__)//true
console.log( new Person().__proto__ == Person.prototype)
```

总结

__proto__是所有对象都拥有的一个对象空间，他指向对应的构造函数的prototype

问题

- 对于引用数据类型来说所有的引用数据类型都是对象类型，那么对应的function是不是也是一个对象，构造函数他也是对象同样他是不是也拥有__proto__，那么它的__proto__又指向谁呢?
- 对于构造函数来说它有一个prototype属性它也一个对象空间，那么这个对象空间的__proto__又指向谁呢?

原型链

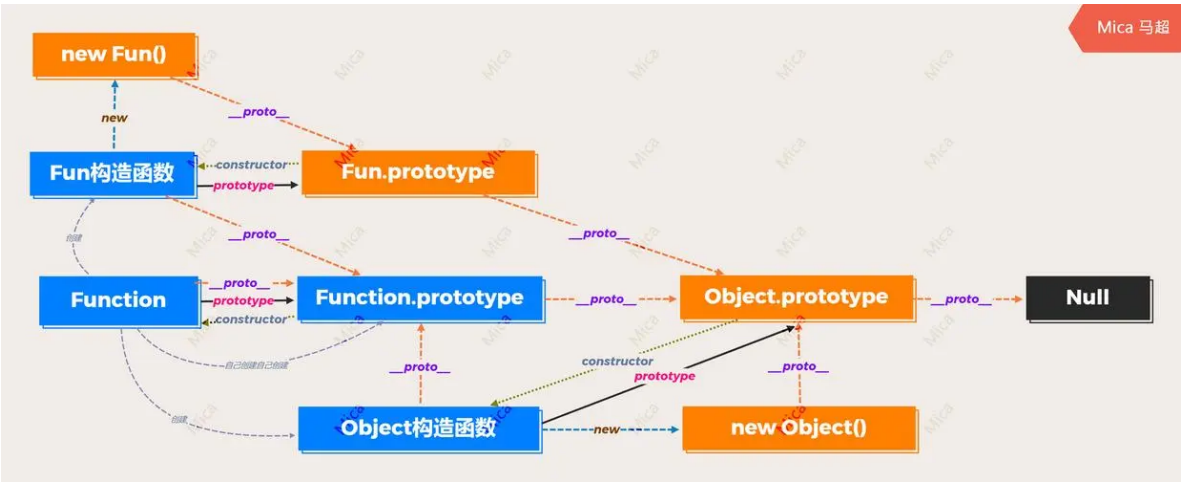
概述

在__proto__中寻找属性的过程形成的链子 被称为原型链

示例

```
function Person() {
console.log(111)
}
let person = new Person()
console.log(Person.__proto__.__proto__.__proto__) //
//函数的 __proto__  ƒ () { [native code] } ==> Object ==> null
console.log(person.__proto__) //Person的prototype 对象
console.log(person.__proto__.__proto__) //对象的__proto__ (Object.prototype)
console.log(person.__proto__.__proto__.__proto__) //Object.prototype的__proto__
==> null
// 找到对应null结束  Person -- Object -- null
```

原型链查找过程



- **对应的函数的 `__proto__` 指向对应的Funtion的构造函数的prototype**
- **对应的指向关系 （原型链查找）**

  先指向自身的构造函数的prototype

  再指向对应的父类的构造函数的prototype

  再指向上级父类 直到找到 Object的构造函数的prototype

  Object的构造函数的prototype的 `__proto__` 指向null （查找属性找到null 还没有返回undefined）
- **对象赋值不遵从原型链 （如果存在属性那么对应的就修改 如果没有就添加）**

模拟实现instanceOf

```
//模拟实现instanceOf关键词
function MyInstanceof(obj,con){
    while( obj.__proto__ ){
        obj  = obj.__proto__
        if(obj.constructor == con){
            return true
        }
    }
    return false
}
function Person(){

}
console.log(MyInstanceof(dog,Dog))
console.log(MyInstanceof(dog,Animal))
```

```
console.log(MyInstanceof(dog,Object))
console.log(MyInstanceof(dog,Person))
```

模拟实现new关键词

```
//模拟实现new
function myNew(fn){
    //自动创建对象
    let obj = {}
    //将对应的obj的原型指向对应的构造函数的原型
    obj.__proto__ = fn.prototype
    //手动属性赋值
    fn.call(obj)
    //自动返回
    return obj
}
console.log(myNew(Person))
```

在于对应的一些内置对象里面的方法大部分是放在原型上

```
function fn(){
    //遍历对应的arguments
    //先将arguments转为数组再调用对应的方法
    // return Array.from(arguments).reduce((prev,current)=>{
    //   return prev + current
    // })
    //reduce其实数组的原型方法
    return Array.prototype.reduce.call(arguments,(prev,current)=>{
        return prev + current
    })
}
console.log(fn(1,2,3,4,5))
```

模拟数组的高阶函数

forEach

```
//模拟forEach方法
Array.prototype.myForEach = function(callback){
    if(typeof callback != 'function'){
        throw new Error()
    }
    //遍历对应的数组 this当前调用的数组
    for(let i=0;i<this.length;i++){
        callback(this[i],i,this)
    }
}
new Array(1,3,5,7).myForEach((v,i,arr)=>{
    console.log(v,i,arr)
})
```

reduce

```
Array.prototype.myReduce = function (callback, previos) {
    if (typeof callback != 'function') {
```

```
        throw new Error()
    }
    let index = 0
    //如果第二个参数没有传递
    if (!previos) {
        previos = this[0]
        index = 1
    }
    for (; index < this.length; index++) {
        previos = callback(previos, this[index], index, this)
    }
    return previos
}
let result = new Array(1, 3, 5, 7).myReduce((prev,v, i, arr) => {
    return prev + v
})
console.log(result)
```

继承

面向对象三大特性

- 封装
- 继承
- 多态

继承的实现

class 继承实现 extends (es6)

```
class Animal {
    constructor() {
        this.name = 'jack'
    }
    say() {
        console.log('hello')
    }
    static run() {
        console.log('running')
    }
}
class Cat extends Animal {
    constructor() {
        super() //父类构造
        this.color = '白色'
    }
}
//获取父类的方法和属性
let cat = new Cat()
console.log(cat.name)
cat.say()
//静态方法 也可以继承
Cat.run()
```

构造函数的继承 （继承不了静态的方法和属性）

原型链继承

将对应的父类对象放入到对应的子类构造的原型上

```
function Person(name,age){
    this.name = name
    this.age = age
}
Person.prototype.sayHello = function(){
    console.log('hello')
}
Person.run = function(){
    console.log('running!!!')
}
function Student(classNumber,name,age) {
   this.class = classNumber
}
//原型链继承
//将父类对象放在子类的原型链上 默认继承的元素的属性在原型上显示为undefined
//缺点  不能进行初始化赋值操作（undefined）
//覆盖子类原型 子类的原型只能在原型继承之后
Student.prototype = new Person()
```

缺点

- 不能进行初始化赋值操作（undefined）
- 覆盖子类原型 （子类的原型方法只能在原型继承之后声明）

对象冒充

将对应的父类构造函数 当作普通函数执行 传入对应的子类构造中的this

```
//对象冒充继承
//将对应的父类构造函数 当作普通函数执行 传入对应的子类构造中的this
//子类构造
function Student(classNumber,name,age) {
    //对象冒充
    Person.call(this,name,age)
    this.class = classNumber
}
```

缺点

- 获取不了原型上的内容

组合继承 （原型链继承 + 对象冒充）

```
//组合继承
//对象冒充继承
//将对应的父类构造函数 当作普通函数执行 传入对应的子类构造中的this
//子类构造
function Student(className, name, age) {
    //对象冒充
    Person.call(this, name, age)
    this.class = className
}
// 原型链继承
Student.prototype = new Person()
```

缺点

- 原型上有重复的属性

寄生组合继承

主要寄生原型（将父类原型对象加入到子类的原型上） + 对象冒充

```
//寄生继承
//利用寄生原型 + 对象冒充
function Student(className, name, age) {
    //对象冒充
    Person.call(this, name, age)
    this.class = className
}
// 寄生原型
Student.prototype = Object.create(Person.prototype)
```

面向对象的轮播图

```
class Carousel{
  //轮播图的属性
  constructor(element){
      //切换的ul
      this.toggleUl = element.querySelector('.toggleUl')
      //焦点 容器
      this.focusElement = element.querySelector('.focus')
      //箭头
      this.prevArrow = element.querySelector('.prev')
      this.nextArrow = element.querySelector('.next')
      //定时器
      this.timer = null
      //下标
      this.index = 0
      //原始个数
      this.size = this.toggleUl.children.length
      //记录li的宽度和高度
      this.elementWidth = this.toggleUl.children[0].clientWidth
      this.elementHeight = this.toggleUl.children[0].clientHeight
      //方向 上下 （top）2 3 左右 （left）0 1
      this.direction = 1
      //初始化
      this.init()
  }
```

```
    //移动方法
    move(isAsc){
        let key = this.direction/2 >= 1 ? 'top' : 'left'
        this.distance = this.direction/2 >= 1 ? this.elementHeight :
this.elementWidth
        //判断index的值和方向 正向 ++ () 逆向 --
        //正向
        if(this.index >= this.size && isAsc ){
            this.index = 0
            //变成0的位置
            this.toggleUl.style[key] = this.index * -1 * this.distance + 'px'
        }
        //逆向
        if(this.index <= 0 && !isAsc ){
            this.index = this.size
            //变成最后的位置
            this.toggleUl.style[key] = this.index * -1 * this.distance + 'px'
        }
        //控制对应的index进行变化
        isAsc ? this.index++ : this.index--
        //移动操作
        let targetObj = {
            top:{
                top: this.index * -1 * this.distance
            },
            left:{
                left: this.index * -1 * this.distance
            }
        }
        bufferAnimation(this.toggleUl,targetObj[key])
        //焦点变化
        this.toggleFocus()
    }
    //轮播方法
    autoMove(){
        this.timer = setInterval(()=>{
            this.move(true)
        },2000)
    }
    //焦点切换的方法
    toggleFocus(){
        //主要焦点的选择
        //排他
        Array.prototype.forEach.call(this.focusElement.children,(v)=>{
            v.className = ''
        })
        this.focusElement.children[this.index%this.size].className = 'selected'
    }
    //处理点击事件的方法
    //处理焦点点击
    handlerFocusClick(){
        Array.prototype.forEach.call(this.focusElement.children,(v,i)=>{
            v.onclick = ()=>{
                this.index = i-1
                //移动过去
                this.move(true)
            }
        })
```

```
    }
    //处理箭头点击
    handlerArrowClick(){
        this.prevArrow.onclick = ()=>{
          this.move(false)
        }
        this.nextArrow.onclick = ()=>{
          this.move(true)
        }
    }
    //初始化方法
    init(){
        let _this = this
        //初始化焦点
        //声明index记录对应的图片位置
        //根据图片的个数去生成对应的焦点
        Array.from(this.toggleUl.children).forEach((v, i) => {
            let li = document.createElement('li')
            li.innerText = i + 1
            //给第一个添加对应的selected
            if (i == 0) {
                li.className = 'selected'
            }
            //再将声明的li添加给ul
            _this.focusElement.appendChild(li)
        })
        //在最后一张后面添加一个第一张
        this.toggleUl.appendChild(this.toggleUl.firstElementChild.cloneNode(true))
        //加入点击事件
        this.handlerFocusClick()
        this.handlerArrowClick()
        //调用自动轮播
        this.autoMove()
    }
}
```