JS第四周第一天

1.单例模式

模块化开发:
将一个完整的项目分模块,大家合作开发
例如:个人中心,首页,登录注册....

合作开发很容易出现变量冲突　所以说一般每一个模块的属性和方法将其写在一个对象中,只需要对象名字不一样即可,这时候这个对象名就可以叫做命名空间
每一个命名空间下的方法中的this也是当前命名空间(对象)

```
<script>
    //高级单例模式 通过一个自行函数返回一个对象 这个自执行函数执行的时候形成的私有作用域可以保护里面的私有变量不受全局污染

    //单例模式中的链式写法核心就是 return this
    let obj1=(function () {
      let name="obj1";
      function f1() {
        return this;
      }
      function f2() {
        return this;
      }
      return{
        name,
        f1,
        f2
      }
```

```
    })();
    obj1.f1().f2().name;
</script>
```

2.单例模式场景切换

```
利用return
```

3.面向对象

```
面向对象：万物皆对象（除了null和undefined），任何东西都看成一个对象，可以通过原型链（原型指针__proto__）找到基类Object
JS中任何数据类型都有自己所属的类，   null和undefined没有
内置类：数据类型，元素，arguments...
JS提供了一种自己定义类的方式，通过自定义函数来实现，使用new关键字执行函数，这个函数就有个新的身份叫类
构造函数
```

4.构造函数

```
一个函数同new方式执行  就是构造函数
1.里面的this是当前实例
2.通过this给当前实例增加私有属性
3.默认返回的当前实例
4.实例都是对象
new 有执行功能  如果函数不需要传参数可以省略()
<script>
    function fn() {
      this.name="zf";
    }
    let f1=new fn;// f1是fn的一个实例
    //{name:"zf"}
    //fn();


    //构造函数return问题
    //return出的内容是基本数据类型对实例没有影响  如果是引用数据类型对实例有影响
    function Fn() {
      this.a=10;
      //return 100;
```

```
        return [];
    }
    let f2=new Fn;
    console.log(f2);
</script>
```

5.构造函数创建方式

```
<script>
    //字面量创建方式
    let n=10;

    //构造函数创建方式 new创建  得到都是对象

    let n1=new Number(1);//此时n1是Number类的实例 n1也是一个对象
    console.log(typeof n1);
    console.log(n1);

    //对于基本数据类型 使用构造函数方式创建和字面创建方式不一样,引用数据类型没什么区别
    let arr1=[1,2,3];
    let arr2=new Array(1,2,3);
    console.log(arr1, arr2);

    //构造函数方式创建函数
    // function f() {
    //
    // }
    //new Function("函数体")
    //new Function("形参","函数体")
    let f1= new Function("x","x++;return x;");
    // function f1(x){
    //   x++;
    //   return x;
    // }
    console.log(f1(10));

    let s="[1,2,3,4]";
    //console.log(eval(s));
    let ff=new Function("return "+s);
    console.log(ff());
</script>
```

6.原型

类都是函数
任何类都有一个天生自带的属性prototype叫做原型,这个原型(prototype)是一个对象,既然是对象就是引用数据类型 浏览器就会默认给他开辟一个堆内存,这个堆内存中有一个天生自带的属性叫constructor指向当前类本身,原型中存储的是公有的属性和方法

任何一个对象(实例,prototype....)有一个属性__proto__,指向所属类的原型

7.原型链

当一个对象获取属性的时候先看是不是自己的私有属性 没有通过__proto__找到所属类的原型上看公有属性有没有,没有继续通过原型的__proto__往上一级找 一直找到Object的原型没有就是undefined,这个查找过程就是原型链

函数既有原型prototype 也有__proto__

8.Function和Object

1.类是函数,函数就是Function类的实例 类都是Function类的实例
2.Object是基类 任何类都是Object类的实例

Object是一个类就是一个函数 ,函数是Function类的实例,所以Object就是Function类的实例
Object.__proto__指向所属类的原型,所属的类是Function,原型是Function.prototype

9.constructor问题

如果给原型重新赋值新的地址 造成constructor丢失问题,可以手动的加上原来的constructor

10.公有属性和私有属性

```
<script>
    let obj1={name:"A",age:10,sex:0};
    console.log(Object.prototype);
    Object.prototype.getX=function () {}

    let arr=[1,3,4,6];
    let arr1=[1,3,4];
```

```
    console.log(arr);
    //数组中的方法都是公有方法 只要是Array的实例都可以使用这些方法

    console.log(arr.hasOwnProperty("length"));
    //in :即可检测私有也可以检测公有的方法和属性

    console.log("toString" in obj1);


    //1.先遍历属性名是数字的 按照从小到大的顺序
    //2.既可以遍历私有属性也可以遍历公有属性,但是一些内置属性遍历不到
    //想这些不可以使用for in 遍历到的属性是不可枚举属性

    for (var key in arr){
      console.log(key);
    }

    //查看对一个私有属性的描述信息
    console.log(Object.getOwnPropertyDescriptor(arr, "length"));
    //configurable:是否可配置,可不可以删除这个属性
    //enumerable:false
    //writable:true 是否可以修改
    console.log(Object.getOwnPropertyDescriptor(obj1, "name"));
    console.dir(Object);
    //查看一个对象下面所有私有属性的描述信息
    console.log(Object.getOwnPropertyDescriptors(obj1));

    //getOwnPropertyNames 获取所有的私有属性的名字 返回一个数组
    console.log(Object.getOwnPropertyNames(obj1));// ["name", "ag
e", "sex"]

    //获取当前对象的所属类的原型
    console.log(Object.getPrototypeOf(obj1));

    let a1=[NaN,NaN,1,true,1,true];

    //Object.is 比较 使用的是===但是解决了NaN的问题
    console.log(Object.is(NaN, NaN));//true
    console.log(Object.is(0, -0));//false

</script>
```

11.数组原型方法的重写

```
<script>
```

```
//1.pop()
//原数组改变,返回值删除的那一项
Array.prototype.pop=function () {
  let item=this[this.length-1];
  this.length--;
  return item;
};
let arr1=[1,2,3];
console.log(arr1.pop());
console.log(arr1);
//2.shift()
Array.prototype.shift=function () {
  let item=this[0];
  for (var i=0;i<this.length;i++){
    this[i]=this[i+1];
  }
  this.length--;
  return item;
};

let arr2=[1,2,3,4,5];
arr2.shift();
console.log(arr2);

//3.push()
//返回值:length 原数组改变
//参数:一个或多个
Array.prototype.push=function () {
  let ary=[...this,...arguments];
  for (var i=0;i<ary.length;i++){
    this[i]=ary[i]
  }
  return this.length;
};
Array.prototype.push=function(){
  for (var i=0;i<arguments.length;i++) {
    this[this.length+i]=arguments[i];
  }
  return this.length;
};

Array.prototype.push=function(){
  let str=this.toString();
  for (var i=0;i<arguments.length;i++){
    str+=","+arguments[i]
  }
  let ary=eval(str);
  for (var i=0;i<ary.length;i++){
    this[i]=ary[i]
```

```
    }
    return this.length;
  };
  arr2.push(10,100);


  //4.unshift()


  //indexOf(item,i)
  Array.prototype.indexOf=function (item,i) {
    for(i;i<this.length;i++){
      if(item===this[i]){
        return i;
      }
    }
    //能走到这里说明循环完了也没有找到
    return -1;
  };

  let a1=[NaN,0,1,NaN];
  console.log(a1.indexOf(NaN));
  console.log(a1.includes(NaN));//Object.is


  //forEach()
  Array.prototype.forEach=function (fn) {
    for (var i=0;i<this.length;i++){
      fn(this[i],i,this);
    }
  };

  //map
  Array.prototype.map=function (fn) {
    let ary=[];
    for (var i=0;i<this.length;i++){
      ary[i]=fn(this[i],i,this);
    }
    return ary;
  };

  console.log([1, 2, 3].map(function (item, index) {
    return item * 10;
  }));
</script>
```