JS第四周第二天

1.函数的三种身份

```
1.普通函数  注意this问题
2.new 类    this实例
3.函数类Function的一个实例
<script>
    //函数名,匿名函数的名字是""
    console.log((function () {}).name);//""
    //name的特殊情况
    //1.通过构造函数方式创建的函数 new Function()  name是"anonymous"
    //2.同过bind方法得到的一个函数  name是"bound 原来的函数名"

    //function f(x) {x++;console.log(x);}
    var f= new Function("x","x++;console.log(x)");
    var f1= new Function("x","x++;console.log(x)");
    console.log(f.name);//"anonymous"
    console.log(f1.name);//"anonymous"

    function f2() {};
    var f3=f2.bind();
    console.log(f3.name);//"bound f2"
</script>
```

2.caller

```
caller 调用者 :当前函数在哪个函数中执行的
如果函数直接在全局中执行caller就是null
<script>
    function f() {
      //console.log(f.caller);//f1
      //f.caller();
    }
    function f1() {
      f();
    }
    f1();
    //f();
</script>
```

3.arguments中的callee

```
arguments.callee:当前函数本身
函数.prototype.constructor
注意箭头函数没有arguments
```

4.call

```
call 是函数类Function原型上的一个公有的方法(函数)
那么所有的函数都有call方法
call作用
1.可以让前面的函数执行
2.改变前面函数的this

参数
call() call(null) call(undefined) this都变成window
第一个参数是用来改变前面函数中this的
call(obj) this->obj
从第二个参数开始就是给前面函数传参数用的

 call的（）中写的是基本数据类型，默认将其变成一个对象
```

5.实现call原理

```
<body>
<!--
Function.prototype.call=function(){
```

```
    call方法中的this一定是当前使用call的那个函数
    1.让前面的函数 this执行 this()
    2.call中从第二个参数开始是给this传参数用的
}

f.call()
-->
</body>

<script>

    Function.prototype.call=function (obj,...args) {
      //...args:将除了第一个参数以外的所有参数获取 变成一个数组
      //this->f
      if(obj==undefined){ //obj:不传 undefined null
        //让f执行 将f中的this变成window
        //给this传参数的时候是将args展开传的
        this(...args);//就相当于 f(), this->window
      }else {
        //将this(这里是f)中的this变成obj
        //给obj所属的原型上增加一个 fn属性值是this代表的函数(这里是f)
        //为了防止在控制台打印能看见fn 我们将其加在原型上
        obj.__proto__.fn=this;//他俩共用一个地址
        obj.fn(...args);//都是只同一个地址的函数执行但是这次this就变成了ob
j
        //用完了将其删除
        delete obj.__proto__.fn;
      }
    };
    function f(n,m) {
      console.log(n + m);
      console.log(this);
    }
    f.call(null,1,2);// args=[1,2]  f(1,2) ->f(...args)
    f.call(undefined,1);
    f.call({n:1},1,2)




    //let [x,...x1]=[1,2,3,4,5]

    Function.prototype.call=function () {
      var ary=[];
      for (var i=1;i<arguments.length;i++){
        ary[i-1]=arguments[i];
      }
      if(arguments[0]==undefined){
        eval("this("+ary+")");
```

```
        }else {
          obj.__proto__.fn=this;
          eval("obj.fn("+ary+")");
          delete obj.__proto__.fn;
        }
      };

      Function.prototype.call=function () {
          var ary=[];
          var obj=arguments[0];
          for (var i=1;i<arguments.length;i++){
              ary[i-1]=arguments[i];
          }
          if(obj==undefined){
              //注意:构造函数得到的函数执行里面的this是window 所以使用变量_th
is代表call中的this
              (new Function("_this","_this("+ary+")"))(this);
          }else {
              obj.__proto__.fn=this;
              (new Function("obj","obj.fn("+ary+")"))(obj);
              delete obj.__proto__.fn;
          }
      }

      // f.call(null,1,2);// args=[1,2]  f(1,2) ->f(...args)
      // f.call(undefined,1);
      // f.call({n:1},1,2)

</script>
```

6.call.call

```
call也是函数也有call方法
f.call.call(x1)

call(x1)执行的时候让前面的函数 f.call执行 将f.call中的this变成x1  而f.cal
l()中this是f 将其变成 x1,一旦变成x1 在call的内部就是让 x1执行,此时x1必须是一
个函数

两以及两个以上的call执行最终都是第一个参数通过call执行将剩下参数传给call
所以读第一个参数必须是函数
f.call.call(x1,x2,x3);  x1.call(x2,x3)
f.call.call.call......(x1,x2,3) ->x1.call(x2,x3)
```

7.数组遍历方法

数组遍历方法 第二个参数是用来改变第一个参数函数的this的

```
<script>
    Array.prototype.forEach=function (fn,obj) {
      for (var i=0;i<this.length;i++){
        fn.call(obj,this[i],i,ary);
      }
    };

    let ary=[1,2,3];
    ary.forEach(function (item,index,input) {
      console.log(this);
    })
</script>
```

8.apply

```
apply 跟call效果一样 唯一不同的是传参数的方式不同
第一个参数:修改this用的
第二参数:数组/类数组 函数执行时候将数组的每一项传给函数

<script>
    function f(x,y) {
      console.log(x + y);
    }
    f(1,2);
    f.call(null,1,2);
    f.apply(null,[1,2]);


    //求数组的最大值
    let ary=[1,23,4,17,8,19,40];

    //1.先排序(从大到小) 获取数组ary[0]
    ary.sort((a,b)=>b-a);
    console.log(ary[0]);

    //2.Math.max
    //Math.max(1,23,4,17,8,19,40);
    //Math.max([1,23,4,17,8,19,40])
    Math.max(...ary);

    let m1=eval("Math.max("+ary+")");
    //"Math.max(1,23,4,17,8,19,40)"
    console.log(m1);
```

```
    //利用apply传参数的特点 传的是一个数组 但是给前面函数max()的时候就是将数组展开
    console.log(Math.max.apply(null, ary));

    //假设法
    var max=0;
    for (var i=0;i<ary.length;i++){
      if(ary[i]>max)max=ary[i];
    }
    console.log(max);
</script>
```

9.bind

```
bind :得到一个新函数跟原来的函数长得一样 但是this变了
只改变this 不能让函数执行

注意bind是有返回值,返回值就是得到的新函数


call 修改this并执行
bind 只修改this不执行
apply 利用传参数的特殊性

<script>
    function f1() {
      console.log(this);
    }
    let f2=f1.bind(1);
    console.log(f2==f1);
    f2();


    // Function.prototype.bind=function (obj) {
    //   //this执行bind的原函数 我们需要返回一个跟this长得一样的函数
    //   var fn=(new Function("return "+this))();
    //   //字符串拼接默认调用toString()
    //   //var fn=(new Function("return "+this.toString()))();
    //   return fn;
    // }
    // var f=f1.bind({});
    // f();

    function change(){
```

```
        this.innerHTML++;
    }
    box.change=change;
    //setInterval(change.bind(box),1000)
    box.onclick=function () {
        change.call(this)
    };
</script>
```

10.toString

```
<body>
<!--
任何数据类型对应的类的原型上都有一个toString 但是Object原型上的toString比较特殊 "[object Object]"



var obj={};
obj.toString() ->"[object Object]"

ary.toString()
ary.__proto__->Array.prototype ->toString

利用Object原型上的toString的结果特点将里面的this变成想要检测的数据,得到对应的数据类型
-->
</body>

<script>

    // delete Array.prototype.toString;
    // delete Number.prototype.toString;
    // delete Boolean.prototype.toString;
    // delete String.prototype.toString;
    console.log(Array.prototype);
    let ary=[1,2];
    console.log(ary.toString());//"[object Array]"
    console.log(1..toString());//"[object Number]"
    console.log(true.toString());//"[object Boolean]"
    console.log("11".toString());//"[object String]"
    //上m面我们将Number Array  Boolean String 原型上的toString属性删除之后 都找的是Object原型上的toString 我们看出结果是 "[object this的类]"

    // Object.prototype.toString=function () {
    //   //typeClass this的类型
```

```
    //    var typeClass=this.constructor.name;
    //    return "[object "+typeClass+"]";
    // };

    console.log(Object.prototype.toString.call([1, 2]));
    console.log(Object.prototype.toString.call(1));
    console.log(Object.prototype.toString.call(true));
    console.log(Object.prototype.toString.call(/\d/));
</script>
```

11.检测数据类型的方法

```
1.typeof 详细检测基本数据类型(null->object) 对于引用数据类型得到"objec
t"和"function"
2.instanceOf 检测实例是否属于某一个类 注意基本数据类型中通过字面量创建方式检测
不到 使用构造函数创建方式可以检测 引用数据类型没有问题
3.通过constructor得到所属类本身可以 获取name属性直接拿到类名(函数名)
4.利用Object原型下面的toString方法将其this通过call方法变成想要检测的数据
```

12.JSON

```
我们在自己创建的.json文件中写的内容是JSON字符串 字符必须使用双引号

<script>
    //JSON.parse() 将JSON字符串变成JSON对象
    //JSON.stringify()将JSON对象变成JSON字符串

    //JSON是一种数据格式
    //JSON对象 严格的对象 使用双引号包起来
    var obj1={"name":"珠峰","age":10};
    var arr1=[{"name":"珠峰1","age":10},{"name":"珠峰2","age":20}];
    console.log(1,JSON.stringify(arr1));
    //JSON字符串  将JSON对使用单引号包起来
    var str='[{"name":"珠峰1","age":10},{"name":"珠峰2","age":20}]';
    console.log(JSON.parse(str));

</script>
```