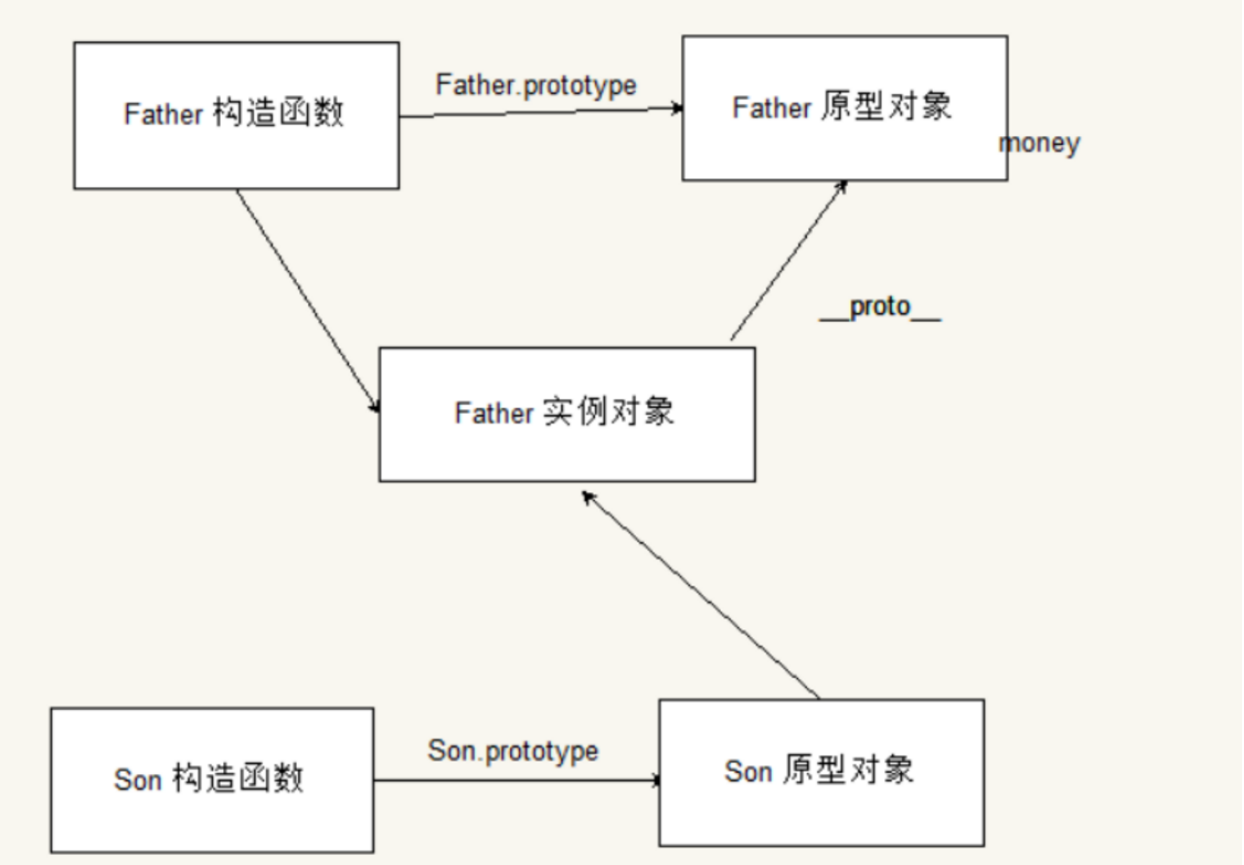
目录

作业讲评

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 借用原型来继承构造函数中的属性和方法
        // 一个构造函数Father中有money cars house company 属性,有管理的方法,另一个构造函数Son要继承构造函数Father的属性和方法 构造函数Son中有自己独特的score分数属性以及study学习方法
        function Father(money,cars,house,company){
            // 构造函数中一般就是对属性赋值
            this.money   = money;
            this.cars    = cars;
            this.house   = house;
            this.company = company;
        }
        // 方法一般会放在原型对象上,这样才可以实现多个实例共享方法
        Father.prototype.manage = function(){
            console.log("管理公司");
        }


        function Son(money,cars,house,company,score){
            // 如何实现子构造器继承父构造器的属性  通过call方法调用父构造器函数,并且改变this指向
            // console.log( this );
            Father.call(this,money,cars,house,company);
            // 子构造器也可以有自己的属性
            this.score = score;
        }


        // 如何实现子构造器继承父构造器的方法


        // 让子原型对象等于父原型对象,弊端:这样会有一个问题,因为原型对象的取值是对象,他们之间的赋值是引用传值,也就是传地址,会同生共死(相互影响)
        // Son.prototype = Father.prototype;


        // 我们可以使用Son.prototype = 父构造器实例对象; 如果原型对象直接赋值一个对象的话,会丢失constructor属性  需要手动设置回来
        Son.prototype = new Father();
        Son.prototype.constructor = Son;


        Son.prototype.study = function(){
            console.log( "好好学习,天天向上" );
        }


        Son.prototype.exam = function(){
            console.log( "考试" );
        }


        Father.prototype.smoke = function(){
            console.log( "抽烟" );
        }


        var son = new Son("100百万","上海大众","海景房","上市公司",100);
        console.log( son );
        // 如果父构造器的方法是写在原型对象上的话,用call方法只能继承到父构造器的属性,继承不了父构造器的方法;  但是如果方法父构造器上的,子构造器通过call方法是可以继承到的
        // 根据原型链,son在自己对象身上找不到的东西,就会去原型对象上面找
        console.log("子构造器的原型对象");
        console.log( Son.prototype );
        son.manage();
        son.exam();
        son.study();

        var father = new Father("更多钱","更多车","更多房","更多公司");
        // father.exam();
        // father.study();
        // console.log("");

        console.log("父构造器的原型对象");
        console.log( Father.prototype );
    </script>
</body>
</html>
```



学习目标

- 能够说出函数的多种定义和调用方式
- 能够说出和改变函数内部this的指向
- 能够说出严格模式的特点
- 能够把函数作为参数和返回值传递
- 能够说出闭包的作用
- 能够使用递归
- 能够说出深拷贝和浅拷贝的区别

案例-别踩白块-面向对象实现

```
游戏对象,需要什么属性和方法?
```

```
属性:
  1.游戏区域对象
  2.游戏是否结束标志
  3.向下移动的速度
  4.定时器标志,因为div游戏区需要不断向下运动
```

5.游戏分数

方法:

1.创建一行4个块,并且其中有一个黑块的方法

2.游戏区域向下移动的方法,向下移动到底部判断是否全部黑块被点击

3.判断游戏是否结束的方法 绑定点击事件 在事件中进行判断 点击的是黑块还是白块 如果是黑块就加分变白,白 块就游戏结束

4.游戏初始化方法

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        *{
            margin: 0;
            padding: 0;
        }
        .score{
            margin-left: 50px;
            margin-top: 44px;
        }
        .box{
            width: 400px;
            height: 400px;
            margin-left: 50px;
            margin-top: 50px;
            border:1px solid blue;
            /* 父相 */
            position: relative;
            /* 溢出隐藏 */
            /* overflow: hidden; */
        }
        .box .game {
            /* 因为后期我们需要让游戏区下落运动起来,所以需要给游戏区设置绝对定位 */
            /* 子绝 */
            position: absolute;
            top: -50px;
            left: 0;
        }
        .box .game .row div{
            width: 98px;
            height: 98px;
            border:1px solid gray;
            float: left;
        }
        .box .game .row .black{
            background: black;
        }
    </style>
</head>
<body>
    <!-- 1. HTML,CSS完成可视区以及游戏区的布局 -->
    <!-- 统计分数的文本框  disabled代表禁用属性,不能可以编辑-->
    <input type="text" class="score" value="0" disabled="disabled">

    <!-- 外层大盒子  可视区 -->
    <div class="box">
        <!-- 内层盒子 游戏区域 也就是需要一直向下运动并且可以点击的区域  -->
        <div class="game">
            <div class="row">
                <div class="black"></div>
                <div></div>
                <div></div>
                <div></div>
```

```
            </div>
            <div class="row">
                <div></div>
                <div class="black"></div>
                <div></div>
                <div></div>
            </div>
            <div class="row">
                <div></div>
                <div></div>
                <div class="black"></div>
                <div></div>
            </div>
            <div class="row">
                <div></div>
                <div></div>
                <div></div>
                <div class="black"></div>
            </div>
        </div>
    </div>
    <script>
        // 面向过程的内容怎么转换成面向对象的内容
        // 面向过程是把功能封装到各个函数中,按步骤调用函数即可
        // 面向对象是把封装到对象里面,让对象去调用功能
        // 面相对象有两个比较重要的东西,属性和方法;属性就是特征,可以简单理解为就是变量;方法是功能行为,可以简单理解函数
        // 思考:别踩白块游戏需要封装什么对象?这个对象具有什么属性和方法

        // 封装一个别踩白块游戏对象
        /*
            有什么属性?属性就是需要用到什么变量
            游戏区对象
            文本框分数对象
            游戏是否已经结束的标识符
            当前分数
            游戏区下落速度
         */


        /* 有什么方法?方法就是这个对象有什么功能,可以理解为有什么函数
        生成行的方法(游戏区到达底部的时候)
        游戏区动画下落的方法(不断往下移动)
        判断游戏是否结束的方法(游戏区可以点击,点击黑块怎样,点击白块怎样)
        初始化游戏区的方法(指挥其他方法如何工作) */


        // 缓存this
        var that;


        // 定义一个游戏对象
        function Game(){
            that = this;


            // 在构造函数中,定义相关属性
            // 游戏区对象
            this.game = document.querySelector(".game");
            // 文本框对象(放置分数的对象)
            this.score = document.querySelector(".score");
            // 游戏是否已经结束的标识符  false代表游戏没有结束  true代表游戏已经结束
            this.gameover = false;
            // 当前分数
            this.fenshu = 0;
            // 游戏区下落速度
            this.speed = 1;
            // 因为多个方法需要使用timer这个变量,可以把timer这个变量变成游戏对象中的一个属性
            this.timer = null;


            // 调用初始化方法
            this.init();
        }


        // 定义方法,我们就放在原型对象上
        // 生成行的方法(游戏区到达底部的时候,生成行,并且随机生成黑块的方法(一行里面有一个是黑块))
        Game.prototype.createRow = function(){
```

```
    console.log("创建了一行,并且生成黑块");
}


// 游戏区动画下落的方法(不断往下移动)
Game.prototype.moveDown = function(){
    // 让div游戏区 动起来 向下运动(不断增加游戏区域的top值,让游戏动起来)
    this.timer = window.setInterval(function(){
        // 在定时器里面的this指向window对象
        that.game.style.top = that.game.offsetTop + that.speed + "px";


        // 得到1 2 3 4随机一个数字
        var random = Math.floor(Math.random()*4 + 1);


        // 当游戏区的top值大于等于0的时候,动态添加一行
        if(  that.game.offsetTop >=0 ){
            // 到达底部的时候,我们就去判断当前游戏区中game的行数是否大于等于5,如果大于等于5,我们就删除最后一行
            if( that.game.children.length >= 5){
                that.game.lastElementChild.remove();
            }


            // 创建行
            var row = document.createElement("div");
            row.className = "row";
            for(var i = 1 ; i <= 4; i++ ){
                // 创建列
                var div = document.createElement("div");

                // 设置随机黑块
                if(i == random){
                    div.className = "black";
                }
                row.appendChild(div);
            }
            // 再把创建出来的行添加到游戏区的子元素列表最前面
            that.game.insertBefore(row, that.game.firstElementChild );
            // 设置游戏的top值为-100px
            that.game.style.top = "-100px";
        }
    },30)


}

// 判断游戏是否结束的方法(游戏区可以点击,点击黑块怎样,点击白块怎样)
Game.prototype.isOver = function(){
    // 绑定点击事件 在事件中进行判断 点击的是黑块还是白块;如果是黑块就加分变白,白块就游戏结束;
    this.game.onclick = function(e){
        e = e || window.event;
        if( e.target.className == "black" ){//如果点击的是黑块就加1分,黑块变白
            that.fenshu++;
            that.score.value = that.fenshu;


            // 黑块变白
            e.target.removeAttribute("class");
        }else{
            // 如果点击的白块就弹窗提示游戏结束 清除定时器
            alert("游戏结束");
            window.clearInterval( that.timer );
        }
    }
}


// 初始化游戏区的方法(指挥其他方法如何工作的)
Game.prototype.init = function(){
    console.log("游戏开始了");
    // 在原型对象中this默认是指向实例对象的
    this.moveDown();
    this.createRow();
    this.isOver();
}


// 构造函数中的代码,只要实例化对象就会执行
new Game();
```

```
    </script>
</body>
</html>
```

完整代码如下:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        *{
            margin: 0;
            padding: 0;
        }
        .score{
            margin-left: 50px;
            margin-top: 44px;
        }
        .box{
            width: 400px;
            height: 400px;
            margin-left: 50px;
            margin-top: 50px;
            border:1px solid blue;
            /* 父相 */
            position: relative;
            /* 溢出隐藏 */
            overflow: hidden;
        }
        .box .game {
            /* 因为后期我们需要让游戏区下落运动起来,所以需要给游戏区设置绝对定位 */
            /* 子绝 */
            position: absolute;
            top: -50px;
            left: 0;
        }
        .box .game .row div{
            width: 98px;
            height: 98px;
            border:1px solid gray;
            float: left;
        }
        .box .game .row .black{
            background: black;
        }
    </style>
</head>
<body>
    <!-- 1. HTML,CSS完成可视区以及游戏区的布局 -->
    <!-- 统计分数的文本框  disabled代表禁用属性,不能可以编辑-->
    <input type="text" class="score" value="0" disabled="disabled">

    <!-- 外层大盒子  可视区 -->
    <div class="box">
        <!-- 内层盒子 游戏区域 也就是需要一直向下运动并且可以点击的区域  -->
        <div class="game">
            <div class="row">
                <div class="black"></div>
                <div></div>
                <div></div>
                <div></div>
            </div>
            <div class="row">
                <div></div>
                <div class="black"></div>
                <div></div>
                <div></div>
            </div>
            <div class="row">
                <div></div>
                <div></div>
                <div class="black"></div>
                <div></div>
            </div>
            <div class="row">
                <div></div>
```

```
            <div></div>
            <div></div>
            <div class="black"></div>
        </div>
    </div>
</div>
<script>
    // 面向过程的内容怎么转换成面向对象的内容
    // 面向过程是把功能封装到各个函数中,按步骤调用函数即可
    // 面向对象是把封装到对象里面,让对象去调用功能
    // 面相对象有两个比较重要的东西,属性和方法;属性就是特征,可以简单理解为就是变量;方法是功能行为,可以简单理解函数
    // 思考:别踩白块游戏需要封装什么对象?这个对象具有什么属性和方法

    // 封装一个别踩白块游戏对象
    /*
        有什么属性?属性就是需要用到什么变量
        游戏区对象
        文本框分数对象
        游戏是否已经结束的标识符
        当前分数
        游戏区下落速度
     */


    /* 有什么方法?方法就是这个对象有什么功能,可以理解为有什么函数
    生成行的方法(游戏区到达底部的时候)
    游戏区动画下落的方法(不断往下移动)
    判断游戏是否结束的方法(游戏区可以点击,点击黑块怎样,点击白块怎样)
    初始化游戏区的方法(指挥其他方法如何工作) */


    // 缓存this
    var that;


    // 定义一个游戏对象
    function Game(){
        that = this;


        // 在构造函数中,定义相关属性
        // 游戏区对象
        this.game = document.querySelector(".game");
        // 文本框对象(放置分数的对象)
        this.score = document.querySelector(".score");
        // 游戏是否已经结束的标识符  false代表游戏没有结束  true代表游戏已经结束
        this.gameover = false;
        // 当前分数
        this.fenshu = 0;
        // 游戏区下落速度
        this.speed = 1;
        // 因为多个方法需要使用timer这个变量,可以把timer这个变量变成游戏对象中的一个属性
        this.timer = null;


        // 调用初始化方法
        this.init();
    }


    // 定义方法,我们就放在原型对象上
    // 生成行的方法(游戏区到达底部的时候,生成行,并且随机生成黑块的方法(一行里面有一个是黑块))
    Game.prototype.createRow = function(){
        // 得到1 2 3 4随机一个数字
        var random = Math.floor(Math.random()*4 + 1);

        // 创建行
        var row = document.createElement("div");
        row.className = "row";
        for(var i = 1 ; i <= 4; i++ ){
            // 创建列
            var div = document.createElement("div");

            // 设置随机黑块
            if(i == random){
                div.className = "black";
            }
            row.appendChild(div);
        }
        // 再把创建出来的行添加到游戏区的子元素列表最前面
```

```
        that.game.insertBefore(row, that.game.firstElementChild );
        // 设置游戏的top值为-100px
        that.game.style.top = "-100px";
    }


    // 游戏区动画下落的方法(不断往下移动)
    Game.prototype.moveDown = function(){
        // 让div游戏区 动起来 向下运动(不断增加游戏区域的top值,让游戏动起来)
        this.timer = window.setInterval(function(){
            // 在定时器里面的this指向window对象
            that.game.style.top = that.game.offsetTop + that.speed + "px";


            // 当游戏区的top值大于等于0的时候,动态添加一行
            if(  that.game.offsetTop >=0 ){
                // 判断游戏区最后一行上的自定义属性的值是否为null
                // 如果是则清除定时器,设置游戏状态为结束,并且弹窗提示用户,最后加上一个return终止函数体后续的代码执行
                if( that.game.lastElementChild.getAttribute("pass") == null ){
                    alert("游戏结束,请重新开始");
                    clearInterval( that.timer );
                    that.gameover = true;
                    return ;
                }


                // 到达底部的时候,我们就去判断当前游戏区中game的行数是否大于等于5,如果大于等于5,我们就删除最后一行
                if( that.game.children.length >= 5){
                    that.game.lastElementChild.remove();
                }


                // 创建一行并且随机出现黑块
                that.createRow();
            }
        },30)


    }

    // 判断游戏是否结束的方法(游戏区可以点击,点击黑块怎样,点击白块怎样)
    Game.prototype.isOver = function(){
        // 绑定点击事件 在事件中进行判断 点击的是黑块还是白块;如果是黑块就加分变白,白块就游戏结束;
        this.game.onclick = function(e){
            e = e || window.event;
            // 在判断点击黑白块之前,需要先判断游戏是否已经结束,如果结束了我们就弹窗提示用户游戏已经结束
            if( that.gameover ){// 判断gameover属性值是否为true
                alert("游戏已经结束,请重新开始");
            }else if( e.target.className == "black" ){//如果点击的是黑块就加1分,黑块变白
                that.fenshu++;
                that.score.value = that.fenshu;


                if(that.fenshu %5 ==0 ){// 如果分数达到5的倍数,比如5分 10分 15分,每达到一个5分,我们就让速度加快
                    that.speed++;
                }


                // 黑块变白
                e.target.removeAttribute("class");


                // 如果当前行的黑块被成功点击,我们就给这行添加一个自定义属性pass,并且赋值为true
                e.target.parentNode.setAttribute("pass",true);


            }else{
                // 如果点击的白块就弹窗提示游戏结束 清除定时器
                alert("游戏结束");
                window.clearInterval( that.timer );
                // 如果点击的白块,除了清除定时器,还需要修改gameover的值为true
                that.gameover = true;
            }
        }
    }


    // 初始化游戏区的方法(指挥其他方法如何工作的)
    Game.prototype.init = function(){
        console.log("游戏开始了");
```

```
            // 在原型对象中this默认是指向实例对象的
            this.moveDown();
            this.isOver();
        }


        // 构造函数中的代码,只要实例化对象就会执行
        new Game();
    </script>
</body>
</html>
```

函数的定义和调用

函数的定义方式

1.方式1 函数声明方式 function 关键字 (命名函数)

```
function fn(){}
```

2.方式2 函数表达式(匿名函数)

```
var fun = function(){}
```

3.方式3 new Function()

```
var fn = new Function('参数1','参数2'..., '函数体')

var f = new Function('a', 'b', 'console.log(a + b)');
f(1, 2);
```

注意:Function 里面参数都必须是字符串格式

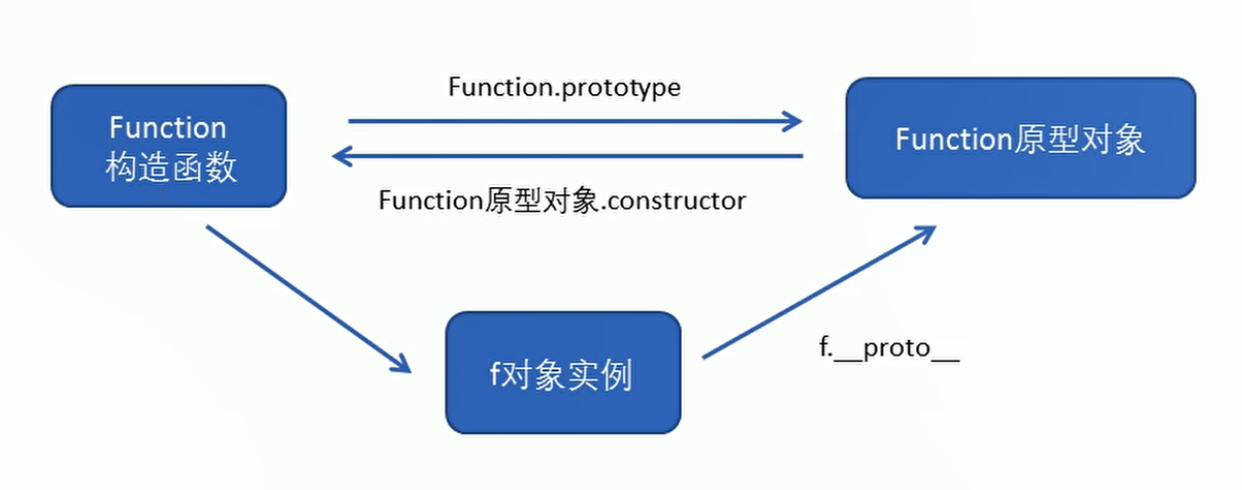
小结:

- 第三种方式执行效率低，也不方便书写，因此较少使用
- 所有函数都是 Function 的实例(对象)
- 函数也属于对象

举例:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        /* 我们之前讲过定义函数的语法是:
        function 函数名([形参1,形参2,形参3...]){
            函数体;
            return 返回值;
        }
        我们之前讲过调用函数的语法是
        函数名([实参1,实参2,实参3...]) */


        // 函数的定义方式
        // 1.方式1 函数声明方式 function 关键字 (命名函数)
        // function fn(){}
        function fn1(){
            console.log("fn1的函数");
        }
        fn1();


        // 2.方式2 函数表达式(匿名函数)
        // var fun = function(){}


        // 匿名函数可以赋值给一个变量或者给事件绑定
        var fn2 = function(){
            console.log("我是匿名函数");
        }
        // 通过变量名就可以调用匿名函数了
        fn2();


        // 3.方式3 通过构造函数new实例化得到   new Function()
        // var fn = new Function('参数1','参数2'..., '函数体')
        // new Function里面的参数必须要加引号
        // new Function因为局限性多,也非常麻烦,实际开发的时候,几乎不用
        var fn3 = new Function("a","b","console.log('传进来的两个数的和为:'+(a+b) );");
        fn3(10,20);
    </script>
</body>
</html>
```

函数的三角关系



函数的调用

目前,我们已经学过了以下6种函数

```
1. 普通函数
2. 对象的方法
3. 构造函数
4. 绑定事件函数
5. 定时器,延时器函数
6. 立即执行函数
```

举例:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <button>按钮</button>


    <script>
        // 1. 普通函数
        function fn(){
            console.log("普通函数");
        }
        fn();


        // 2. 对象的方法
        var obj = {
            sayHello:function(){
                console.log("我是对象里面的sayHello方法");
            }
        }
        // 对象里面的方法,是通过对象去调用的
        obj.sayHello();
```

```
        // 3. 构造函数
        function Star(uname,age){
            this.uname = uname;
            this.age = age;
            console.log( "只要实例化了,就会调用构造函数里面的代码 ");
        }
        new Star();



        // 4. 绑定事件函数
        var btn = document.querySelector("button");
        btn.onclick = function(){
            // 绑定事件的函数不会马上调用,触发的时候才会调用
            alert("点击了按钮");
        }

        // 5. 定时器,延时器函数
        window.setInterval(function(){
            console.log("我是定时器里面的代码,我每隔1秒就会执行");
        },1000);


        window.setTimeout(function(){
            console.log("我是延时器里面的代码,隔3秒以后才会执行,只执行一次");
        },3000);


        // 6. 立即执行函数(自调用函数,自己调用自己,立马执行函数体)
        (function(){
            console.log("我是立即执行函数");
        })()
    </script>
</body>
</html>
```

this

函数内部的this指向

这些 this 的指向，是当我们调用函数的时候确定的。调用方式的不同决定了this的指向不同

this一般指向我们的**调用者**

调用方式	this指向
普通函数调用	window

构造函数调用	实例对象, 另外原型对象里面的方法也指向实例对象
对象方法调用	该方法所属对象
事件绑定方法	绑定事件的对象
定时器和延时器函数	window
立即执行函数	window

举例:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <button>按钮</button>
    <script>
        // 这些 this 的指向，是当我们调用函数的时候确定的。调用方式的不同决定了this 的指向不同
        // this一般指向我们的调用者


        // 普通函数调用 this指向window
        function fn(){
            console.log("普通函数里面的this");
            console.log( this );
            console.log( "" );
        }
        fn();


        // 构造函数调用 this指向实例对象, 另外原型对象里面的方法也指向实例对象
        function Star(uname,age){
            this.uname = uname;
            this.age = age;
            console.log("构造函数里面的this");
            console.log( this );
            console.log( "" );
        }
        Star.prototype.sing = function(){
            console.log("原型对象里面的this");
            console.log( this );
            console.log( "" );
        }
        var ldh = new Star("刘德华",18);
        ldh.sing();


        // 对象方法调用 this指向该方法所属对象
        var obj = {
            uname:"张三",
            age:23,
            sayHello:function(){
                console.log("调用对象中的方法,方法里面的this");
                console.log( this );
                console.log( "" );
            }
```

```
        }
        obj.sayHello();

        // 事件绑定方法 this指向绑定事件的对象
        var btn = document.querySelector("button");
        btn.onclick = function(){
            // 绑定事件的函数不会马上调用,触发的时候才会调用
            console.log("事件绑定方法中的this");
            console.log( this );
            console.log( "" );
        }

        // 定时器和延时器函数 this指向window
        window.setInterval(function(){
            console.log("定时器里面的this");
            console.log( this );
            console.log( "" );
        },1000);

        window.setTimeout(function(){
            console.log("延时器里面的this");
            console.log( this );
            console.log( "" );
        },3000);

        // 立即执行函数 this指向window
        (function(){
            console.log("立即调用函数里面的this");
            console.log( this );
            console.log( "" );
        })()
    </script>
</body>
</html>
```

改变函数内部 this 指向

Javascript为我们专门提供了一些函数方法帮我们更优雅的处理函数内部this的指向问题,常用的有 call() 、bind() 、apply() 三种方法

call方法

call()方法调用一个对象。简单理解为 是一种调用函数的方式，但是它可以改变函数的 this 指向

应用场景: 经常用call实现继承.

语法

```
函数.call(thisArg,arg1,arg2,...)

参数说明:
   thisArg      : 在函数运行时指定的 this 值,如果不需要改变函数内部this指向,可以传递null
   arg1,arg2... : 传递的其他参数,使用逗号隔开


返回值就是函数的返回值，因为它就是调用函数
因此当我们想改变 this 指向，同时想调用这个函数的时候，可以使用 call，比如继承
```

举例:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // call()方法调用一个对象。简单理解为  是一种调用函数的方式 ,  但是它可以改变函数的 this 指向
        // 应用场景:  经常用call实现继承.


        // 语法
        // 函数.call(thisArg,arg1,arg2,...)


        // 参数说明:
        // thisArg     : 在函数运行时指定的 this 值,如果不需要改变函数内部this指向,可以传递null
        // arg1,arg2... : 传递的其他参数,使用逗号隔开


        // 返回值就是函数的返回值，因为它就是调用函数
        // 因此当我们想改变 this 指向，同时想调用这个函数的时候，可以使用 call，比如继承
        function fn(x,y){
            console.log( this );
            console.log( x + y);
        }
        fn(10,20);


        // call的第一个功能,就是调用函数
        fn.call();
        // call的第二个功能,改变函数里面的this指向
        var obj = {
            uname:"zhangsan"
        };
        fn.call(obj);
        // call在改变this的时候,还可以传递参数
        fn.call(obj,50,100);


        // call()方法主要用于实现es5中的构造函数+原型对象组合继承
    </script>
</body>
</html>
```

以上代码运行结果为:

```
▶ Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}
30
▶ Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}
NaN
▶ {uname: "zhangsan"}
NaN
▶ {uname: "zhangsan"}
150
```

apply方法

apply()方法调用一个对象。简单理解为 是一种调用函数的方式，但是它可以改变函数的 this 指向

应用场景：经常跟数组有关系

语法

```
函数.apply(thisArg, [argsArray])

参数说明:
    thisArg        : 在函数运行时指定的 this 值,如果不需要改变函数内部this指向,可以传递null

    argsArray      : 传递的值，必须包含在"数组"里面，数组中的元素会一个一个对应传递给apply()前面函数的形参列表中


返回值就是函数的返回值，因为它就是调用函数
因此 apply 主要应该跟数组有关系，比如使用 Math.max() 求数组的最大值
```

举例:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
    // apply方法
    // apply()方法调用一个对象。简单理解为  是一种调用函数的方式 ,  但是它可以改变函数的 this 指向
    // 应用场景:  经常跟数组有关系
    // 语法
    // 函数.apply(thisArg, [argsArray])


    // 参数说明:
    // thisArg        : 在函数运行时指定的 this 值,如果不需要改变函数内部this指向,可以传递null
    // argsArray      : 传递的值，必须包含在"数组"里面，数组中的元素会一个一个对应传递给apply()前面函数的形参列表中


    // 返回值就是函数的返回值，因为它就是调用函数
    // 因此 apply 主要应该跟数组有关系，比如使用 Math.max() 求数组的最大值


    function fn(x,y){
        console.log( this );
        console.log( x );
        console.log( y );
        console.log( x + y);
        console.log( "" );
    }
    var obj = {
        uname:"zhangsan"
    };
    // 使用apply()调用函数
    fn.apply();
    // 使用apply改变函数里面this的指向
    fn.apply( obj );
    // 如果使用的是apply调用函数,传参的话,必须把每个参数都放在一个数组里面
    fn.apply( obj, [10,20] );
    // 如果只想调用函数并且参数,但是又不想改变原函数中this指向,只需要第一参数传递一个null即可
    fn.apply( null, [50,80] );
    console.log("");
    console.log("");
    console.log("");


    // apply 主要应该跟数组有关系，比如使用 Math.max() 求数组的最大值
    // Math.max(数值1,数值2,数值3,数值4,数值5...)  返回多个数值中最大的那个值
    console.log(  Math.max( 10,5,20,70,3 )  );
    console.log("");


    var arr = [30,1,20,100,7];
    console.log( Math.max( arr ) );// NaN
    console.log("");


    // 使用apply方法,让Math.max可以求出数组中的最大值
    // 函数名.apply(this指向 , 数组参数 )
    console.log( Math.max.apply(null, arr ) );
    // 使用apply方法,让Math.max可以求出数组中的最小值
    console.log( Math.min.apply(null, arr ) );
    </script>
</body>
</html>
```

```
                                                                    06-apply方法.html:24
▶ Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}
undefined                                                           06-apply方法.html:25
undefined                                                           06-apply方法.html:26
NaN                                                                 06-apply方法.html:27
                                                                    06-apply方法.html:28
▶ {uname: "zhangsan"}                                               06-apply方法.html:24
undefined                                                           06-apply方法.html:25
undefined                                                           06-apply方法.html:26
NaN                                                                 06-apply方法.html:27
                                                                    06-apply方法.html:28
▶ {uname: "zhangsan"}                                               06-apply方法.html:24
10                                                                  06-apply方法.html:25
20                                                                  06-apply方法.html:26
30                                                                  06-apply方法.html:27
                                                                    06-apply方法.html:28
                                                                    06-apply方法.html:24
▶ Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}
50                                                                  06-apply方法.html:25
80                                                                  06-apply方法.html:26
130                                                                 06-apply方法.html:27
                                                                    06-apply方法.html:28
                                                                    06-apply方法.html:41
                                                                    06-apply方法.html:42
                                                                    06-apply方法.html:43
70                                                                  06-apply方法.html:47
                                                                    06-apply方法.html:48
NaN                                                                 06-apply方法.html:51
                                                                    06-apply方法.html:52
100                                                                 06-apply方法.html:56
1                                                                   06-apply方法.html:58
```

bind方法

bind() 方法不会调用函数,但是能改变函数内部this 指向

返回值是原函数改变this之后产生的新函数

如果**只是想改变 this 指向，并且不想调用这个函数的时候，可以使用bind**

应用场景: **不调用函数,但是还想改变this指向**

语法:

```
函数.bind(thisArg, arg1, arg2, ...)
thisArg：在 fun 函数运行时指定的 this 值
arg1，arg2：传递的其他参数

返回由指定的 this 值和初始化参数改造的"原函数拷贝"
因此当我们只是想改变 this 指向，并且不想调用这个函数的时候，可以使用 bind
```

举例:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <button class="btn1">按钮1</button>


    <div class="box1">
        <button>box1盒子里面的按钮1</button>
        <button>box1盒子里面的按钮2</button>
        <button>box1盒子里面的按钮3</button>
    </div>


    <div class="box2">
        <button>box2盒子里面的按钮1</button>
        <button>box2盒子里面的按钮2</button>
        <button>box2盒子里面的按钮3</button>
    </div>


    <script>
        /* bind方法
        bind() 方法不会调用函数,但是能改变函数内部this 指向
        返回的是原函数改变this之后产生的新函数
        如果只是想改变 this 指向，并且不想调用这个函数的时候，可以使用bind
        应用场景：不调用函数,但是还想改变this指向 */


        /* 语法:
        函数.bind(thisArg, arg1, arg2, ...)
        thisArg：在 fun 函数运行时指定的 this 值
        arg1，arg2：传递的其他参数

        返回由指定的 this 值和初始化参数改造的"原函数拷贝"
        因此当我们只是想改变 this 指向，并且不想调用这个函数的时候，可以使用 bind */


        function fn(x , y ){
            console.log( this );
            console.log( x );
            console.log( y );
            console.log( "" );
        }
        fn(10,20);
```

```
var obj = {
    uname: "李四",
    age: 24
};


// 函数.bind(thisArg, arg1, arg2, ...)
// bind不会调用函数,可以改变函数体中的this指向,返回值是一个改变this以后的新函数
var f = fn.bind( obj, 30, 40 );
console.log( f );
f();


// 比如我们有一个按钮,当我们点击了之后,就禁用这个按钮,3秒钟之后开启这个按钮
var btn1 = document.querySelector(".btn1");
btn1.onclick = function(){
    // 禁用按钮
    this.disabled = true;
    setTimeout(function(){
        // 重新开始按钮
        btn1.disabled = false;
    },3000)
}


// 如果有多个按钮,想实现点击之后,就禁用对应的按钮,3秒以后,再开启这个按钮
// 第一个方式解决,使用缓存this
var box1Btns = document.querySelectorAll(".box1 button");
for(var i=0;i<box1Btns.length;i++){
    box1Btns[i].onclick = function(){
        // 禁用按钮
        this.disabled = true;
        // 缓存this
        var that = this;


        setTimeout(function(){
            // console.log( that );

            // 重新开始按钮
            // box1Btns[i].disabled = false;


            that.disabled = false;
        },3000)
    }
}


// 如果有多个按钮,想实现点击之后,就禁用对应的按钮,3秒以后,再开启这个按钮
// 第二个方式解决,使用bind方法改变setTimeout中匿名函数里面this指向
var box2Btns = document.querySelectorAll(".box2 button");
for(var i=0;i<box2Btns.length;i++){
    box2Btns[i].onclick = function(){
        // 禁用按钮
        this.disabled = true;


        // 使用bind把延时器外边的this传进延时器的匿名函数中,改变了匿名函数中this的指向
        // 为什么要选中bind,而不call或者apply因为bind不会立即调用函数
        setTimeout( function(){
            this.disabled = false;
        }.bind( this ),   3000)
    }
}
```

```
        }
    </script>
</body>
</html>
```

call、apply、bind三者的异同

- 相同点 : 都可以改变函数内部的this指向.
- 不同点:
  - call 和 apply 会调用函数, 并且改变函数内部this指向.
  - call 和 apply传递的参数不一样, call 传递参数 arg1, arg2..形式 apply 必须数组形式[arg1,arg2...]
  - bind 不会调用函数, 可以改变函数内部this指向.
- 应用场景

  a. call 经常做继承.

  b. apply经常跟数组有关系. 比如借助于数学对象实现数组最大值最小值

  c. bind 不调用函数,但是还想改变this指向. 比如改变定时器内部的this指向.

高阶函数

高阶函数是对其他函数进行操作的函数，它接收函数作为参数或将函数作为返回值输出。

简单理解就是 函数可以作为参数传递到另一个中； 函数也可以作为返回值,返回给函数的调用者,因为函数也是数据类型

```
<script>
function fn(callback){
  callback&&callback();
}
fn(function(){alert('hi')}
</script>
```

```
<script>
function fn(){
    return function() {}
}
 fn();
</script>
```

以上两种情况, fn 都是一个高阶函数

函数也是一种数据类型，同样可以作为参数，传递给另外一个参数使用。最典型的就是作为回调函数。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <!-- 引入cdn上面jquery.min.js -->
    <script src="https://cdn.bootcss.com/jquery/1.8.3/jquery.min.js"></script>
    <style>
        div{
            width: 100px;
            height: 100px;
            background: pink;
            position: absolute;
        }
    </style>
</head>
<body>
    <div>div盒子</div>


    <script>
        // 函数也是一种数据类型，同样可以作为参数，传递给另外一个参数使用。最典型的就是作为回调函数。 我们之前学过的回调函数,就是这样的情况,把函数作为参数传递进另一个函数中
        function fn(a,b,callback){
            console.log( a+b);


            console.log( callback );


            // 先判断callback参数是否有值
            /* if( callback ){
                // 调用函数
                callback();
            } */


            // 以上代码可以优化为  &&前面如果可以转成true的值,就会执行&&后面的代码;&&前面如果转成false的值,就不会执行&&后面的代码
            callback && callback();
        }
        // 只传递前两个参数
        fn(10,20);
        // 传递三个参数
        fn(50,80,function(){
            console.log("我是回调函数");
        });



        // 在后续课程中,大家使用回调函数的几率会大点,比如jquery中有一个animate方法里面就可以传递一个回调函数
        $("div").animate({"left":"500px"},function(){
            $("div").css({"background":"skyblue"});
            $("div").animate({"bottom":"200px"},function(){
                $("div").css({"background":"orange"});
            })
        })
    </script>
</body>
</html>
```

同理函数也可以作为返回值传递回来

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 函数也可以作为返回值传递回来
        function fn(a,b){
            // return关键字,会把return后面的内容返回给函数的调用者
            return function(){
                console.log( a+b );
            };
        }
        var f = fn(10,20);
        console.log( f );
        f();

        console.log("");


        // 以下方式也可以正常调用
        fn(30,50)();
    </script>
</body>
</html>
```

闭包 难点

变量的作用域复习

变量根据作用域的不同分为两种：全局变量和局部变量。

1. **函数内部可以使用全局变量。**
2. **函数外部不可以使用局部变量。**
3. **当函数执行完毕，本作用域内的局部变量会销毁。**

什么是闭包

闭包（closure）指有权**访问**另一个函数作用域中**变量**的函数。

其实,**闭包就是一个函数**

简单理解就是， 一个作用域可以访问另外一个函数内部的局部变量。

```
<script>
 function fn1(){     // fn1 就是闭包函数
       var num = 10;
       function fn2(){
           console.log(num); // 10
       }
       fn2()
  }
   fn1();
</script>
```

举例: 可以通过浏览器检查工具 打断点,在scope栏中可以查看 Closure

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        function fn(){
            var num = 10;

            function fn2(){// fn2这个函数就用到另一个函数fn作用域里面num变量,那么这个num所在的函数,就是闭包函数
                console.log( num );// 10
            }
            fn2();
        }


        fn();
    </script>
</body>
</html>
```

闭包的作用

作用：延伸变量的作用范围。

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 闭包的作用 : 延伸变量的作用范围
        // 定义函数
        function fn(){
            // 默认情况下,函数里面使用了var关键字定义的变量就是局部变量,只能在局部作用域里面使用
            var num = 10;


            /* function fn2(){
                console.log( num );
            }
            return fn2; */


            return function(){
                console.log( num );
            }


        }

        var f = fn();
        // 我们现在在fn函数外面输出num的值,就是延伸了num变量的作用范围
        f();
    </script>
</body>
</html>
```

闭包的应用

利用闭包的方式，点击li输出当前li 的索引号

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <ul class="studentNames">
        <li>张三</li>
        <li>李四</li>
        <li>王五</li>
        <li>赵六</li>
        <li>孙七</li>
    </ul>

    <ul class="fruit">
        <li>苹果</li>
        <li>香蕉</li>
        <li>葡萄</li>
        <li>火龙果</li>
        <li>桔子</li>
    </ul>
    <script>
        // 点击li输出当前li 的索引号
        // 方式一:使用自定义属性
        var studentLis = document.querySelectorAll(".studentNames li");
        for(var i=0;i<studentLis.length;i++){
            // 把索引号保存在自定义属性中
            studentLis[i].setAttribute("data-index",i);


            studentLis[i].onclick = function(){
                var index = this.getAttribute("data-index");
                alert( index );
            }
        }


        // 方式二:使用闭包的方式
        // 需要借助立即执行函数,立即执行函数也能成为闭包
        var fruitLis = document.querySelectorAll(".fruit li");
        for(var i=0;i<fruitLis.length;i++){
            // 立即执行函数会马上执行的  会把i传递给index这个形参,接着在单击事件的匿名函数中就可以使用这个index
            (function(index){
                fruitLis[i].onclick = function(){
                    alert( index );
                }
            })(i)
        }
    </script>
</body>
</html>
```

今日总结

xmind今天要做

今日作业

请查看作业文件夹