JSCORE03

复习

- 闭包
  - 作用：在全局作用域下，开辟一个局部作用域 -- 规避变量的全局污染
  - 本质：JS函数的特殊设计 -- 函数在生成时，能够保存自身的作用域--词法环境
    - 可以打印window对象，找到函数 查看 [[scopes]] 属性，看到所有的作用域!
    - scopes中的内容是数组类型，有序.
  - 最基础的写法：

```
// 为什么用变量保存匿名函数自调的 返回值?
// 答案：把返回的函数保存在window中，后期才能调用

// 为什么写匿名函数自调用，是固定的吗？可以不用匿名函数么？
// 答：不是必须匿名，但是用命名函数，会让window中额外出现一个变量 浪费内存，而且还需要手动调用一次，来触发命名函数，浪费代码 --- low
var 变量 = (function xxx(){
    var name = '亮亮是坏人'
    return function(){
        console.log(name)
    }
})()

变量();
```

  - 闭包重要么?

    闭包是底层封装框架使用，可以规避全局污染的风险

    我们后期正常工作开发， 不会用到闭包

    为什么要学： 面试必考

- 听明白原理，但是不知道有什么使用场景??

  JSCORE中讲解的知识点，基本上都是 封装底层框架使用的!

  我们学习的目的就是 面试

  面试造火箭，工作拧螺丝!

- 原型：每个对象类型都会自带一个原型对象，名字叫 __proto__

  需要通过 360浏览器 后台查看才是准确的

  先进的浏览器Google,火狐... 都遵循了最新的规则，把原型属性做了一些处理!

  - 原型中包含什么：构造函数本身 + 1些通用的方法或属性
  - 对象的原型是哪里来的???
    - 对象是通过构造函数构造的. {} 字面量 本质是new Object()
    - 对象中的__proto__变量哪里来的??? 就是 构造函数注入的
    - 构造函数中有一个变量叫 : prototype，当 new 的时候，就不把这个属性赋值给生成的对象... 生成对象.__proto__ = 构造函数.prototype
    - __proto__ 和 prototype 是同一个对象，只是环境不同 称呼不同!
- 原型的用途 有两种!
  - 节省内存：如果构造函数中，有属性保存了不变的内容，大多数是方法

```
function 构造函数(){
    this.aa = function(){ .... } //字面量
    this.bb = new Function(...) //函数的构造写法
}
// 每次new都会触发 构造函数中的代码，会导致生成多个函数对象！浪费内存
new 构造函数()
new 构造函数()
```

应该写：

```
function 构造函数(){
}
构造函数.prototype.aa = funciton (){}

new 构造函数()
new 构造函数()
```

- 扩展系统构造函数 或 第三方构造函数 -- 你不能直接修改的构造！

```
Array.prototype.sum = function(){}

//之后对所有的数组对象，都可以调用
[123,123,123] //字面量
var a = new Array(123,123,123);
a.sum -> a.__proto__.sum  ->Array.prototype.sum
```

- 原型链：JS引擎在 调用一个对象的属性时，如果对象自身没有，就会找他的 __proto__
  原型如果没有，原型有 __proto__ 就会继续向内查找

> 在JAVA中，这种机制叫 继承

系统提供的 Object 对象是所有对象的 最终原型，Object构造函数的原型 的原型是null

`Object.prototype.__proto__ == null`

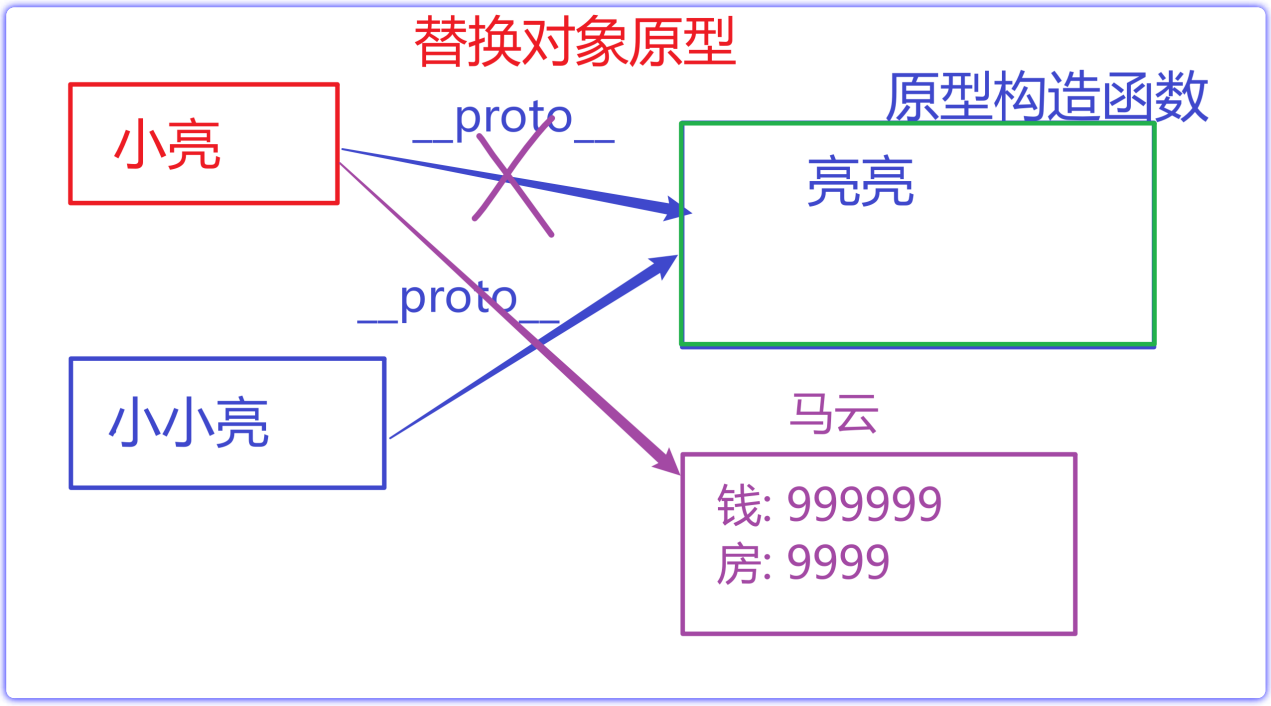
原型的练习

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      //期望：为数组扩展一个 max 方法，能够得到数组中最大值
      //使用方法如下：

      // 向数组的构造方法的原型中增加max
      Array.prototype.max = function () {
        // 对象中的函数,具有关键词 this
        // this永远代表其所在的对象
        // 粗暴的理解,从格式上:   对象.方法()   方法中的this就是对象
        var num_max = this[0];
        for (let i = 0; i < this.length; i++) {
          // 语法糖：利用语法来简化代码 让程序员幸福...
          // 官方会提供很多简化语法：
```

```
          // if(xxx){1行代码}   就可以省略 {}:  if (xxx) xxx
          if (num_max < this[i]) num_max = this[i];
        }
        return num_max;
      };

      var arr = new Array(12, 12, 345, 435, 123);
      console.log(arr);
      console.log(arr.max()); //435
    </script>
  </body>
</html>
```

替换对象的原型

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      function Liang(name, age) {
        this.name = name;
        this.age = age;
      }
      var xiaoLiang = new Liang("小亮", 1);
      console.log(xiaoLiang);

      var maYun = {
        money: 999999999999,
        houses: 1000000,
      };
      // 替换 小亮的 父亲 为 马云..
      // 为对象替换原型
      // 直接修改 __proto__ 的指向，效率比较低，推荐另一种做法:
      // xiaoLiang.__proto__ = maYun;

      // 系统提供了专门替换对象原型的函数: setPrototypeOf()
      // 把参数1 的 proto 换成 参数2
      Object.setPrototypeOf(xiaoLiang, maYun);

      console.log(xiaoLiang);
      // 代码提示生成后的代码有一种特殊的状态高亮，此处书写代码没有代码提示，推荐按 ESC 可以退出此状态，再写代码就有提示了!
      console.log(xiaoLiang.money);

      // 如果通过构造函数再次创建对象，此对象的原型会变成马云吗?
      var xxLiang = new Liang("小小亮", 3);
      console.log(xxLiang);

      // setPrototypeOf(对象，原型):  只会影响修改的对象，不会影响构造函数，之后创建的对象没有变更
    </script>
  </body>
</html>
```



替换构造的原型

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      // 构造函数原型变化，会影响所有生成的对象
      function Liang(name, age) {
        this.name = name;
        this.age = age;
      }
      Liang.prototype.firstname = "成";
      Liang.prototype.intro = function () {
        // 反引号是模板字符串，其中 ${} 代表JS代码范围
        console.log(`大家好,我是${this.firstname + this.name}`);
      };

      //替换构造函数的prototype
      Liang.prototype = {
        firstname: "王",
        intro: function () {
          console.log(`${this.name}: 我爸姓${this.firstname}`);
        },
      };

      var xiaoLiang = new Liang("小亮", 1);
      xiaoLiang.intro();

      var xxLiang = new Liang("小小亮", 1);
```

```
      xxLiang.intro();
    </script>
  </body>
</html>
```

for..in的特点

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      // 构造函数原型变化，会影响所有生成的对象
      function Liang(name, age) {
        this.name = name;
        this.age = age;
      }
      Liang.prototype.firstname = "成";
      Liang.prototype.intro = function () {
        // 反引号是模板字符串，其中 ${} 代表JS代码范围
        console.log(`大家好,我是${this.firstname + this.name}`);
      };

      var xiaoliang = new Liang("小亮", 33);
      console.log(xiaoliang);
      // for (属性名 in 对象){ }
      // 用于遍历对象中所有的属性名 :   {属性名: 值, name:value}
      for (var name in xiaoliang) {
        // for .. in 的特点: 会遍历对象及其自定义原型链中所有的属性
        console.log(name);
      }
      // Object.keys(): 可以只查询对象中有什么，不看原型链
      console.log("keys:", Object.keys(xiaoliang));

      // 判断空对象?  {}
      var a = {};
      // 判断元素个数为0  就说明是空的
      console.log(Object.keys(a).length == 0);
    </script>
  </body>
</html>
```

严格模式

拒绝意外创建全局变量

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <!— 严格模式? —>
    <!— ES: ECMAScript  —>
    <!— ECMA: 欧洲计算机组织--负责制定规则,所有计算机厂商都需要遵循这个规则，才能互相合作 —>
    <!— 标准会逐年更新，弃用旧特性，规避风险，保持语言的先进性 —>
    <!— ES3.1 → ES5 : 没有 ES4修改过于极端,被抛弃了! —>

    <!— 严格模式: 从ES5(ES 2009)开始推出的新特性，用来解决旧版本代码中存在的各种风险! —>

    <script>
      "use strict"; //此代码以下的，都会进入严格模式

      //风险1: 意外的全局变量创建
      var servername;
      // 写错变量名的错误 过于常见
      // 作者想: 不写var 声明，也能自动创建变量，让程序员少写几个字母
      servrname = "localhost";
      // servrname is not defined:
      // 在严格模式下，就会禁用这种写法，直接报错

      console.log(servername);
    </script>
  </body>
</html>
```

window中,函数的this为undefined

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      // 开启严格模式. 系统帮你找错误!
      // 严格模式下，全局区的函数中的this 是 undefined,
      // 目的就是防止直接调用函数 向window中注入属性，导致的全局污染
      "use strict";

      // 严格模式:就是系统会帮助你把 有可能错误的地方 直接指出来，进行报错!
      // 规避开始中 常遇到的问题，规避风险
      // 所有框架默认都开 严格模式!
      function Demo(name) {
        // 函数中的this 代表其当前所在的对象
        // 直接调用 Demo() 则，this是window
        console.log("this:", this);

        this.aname = name;

        // 这里就相当于 window.aname = name
        // 此时就会意外创建 aname，造成了全局变量污染
      }
      //构造函数使用时，应该 new 触发，返回对象
      // var obj = new Demo("亮亮");
```

```
      // 常见问题: 构造函数忘记写new
      var xx = Demo("铭铭");
    </script>
  </body>
</html>
```

静默失败

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      "use strict";
      // 静默失败:
      var obj = {
        name: "亮亮",
        age: 19,
      };
      // freeze: 冻结，把对象冻结之后，就不能修改了!
      Object.freeze(obj);
      // 由于对象被冻结了，所有age 无法修改
      // 但是默认为 静默失败，不会有任何报错
      obj.age = 88;
      //严格模式报错: Cannot assign to read only property 'age' of object
      console.log(obj);
    </script>
  </body>
</html>
```

callee的弃用

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      // 严格模式: 淘汰了 callee
      "use strict";

      // callee: 是 arguments 中的一个属性，保存了当前执行的函数
      function show() {
        console.log(arguments);
      }
      show(123, 23, true, "哈哈");

      // 回调函数: 函数的内部调用自身，形成一个循环的效果
```

```
      function factorial(n) {
        if (n > 1) {
          return n * factorial(n - 1);
        }
        return 1;
      }
      //计算 5! = 1 * 2 * 3 * 4 * 5
      console.log(factorial(5)); //120

      // 匿名函数自调用
      // 在早期的JS中，不支持命名函数的自调用，只能是匿名函数自调用
      var result = (function (n) {
        if (n > 1) {
          // callee: 代表当前正在执行的函数，适合匿名函数的取用
          // 递归函数多次调用，会多次创建 arguments 对象，浪费内存!
          return n * arguments.callee(n - 1);
        }
        return 1;
      })(5);
      console.log(result);

      // 命名函数自调用，古老的JS版本中不支持，所以只能用 消耗极大的 callee 实现
      // 自从有了 命名函数自调用写法，callee 就被淘汰了!
      var result = (function a(n) {
        if (n > 1) {
          // callee: 代表当前正在执行的函数，适合匿名函数的取用
          return n * a(n - 1);
        }
        return 1;
      })(5);
    </script>
  </body>
</html>
```

精确设置对象属性

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      "use strict";

      // 精确配置对象属性:
      // 此知识点 是封装框架时使用，平常开发很难用到!
      var obj = {
        id: 1001,
        name: "亮亮",
        age: 44,
      };
      // 对象的每个属性，额外有几个选项: 可写，可配置，可遍历
      // 参数1: 要修改的对象
      // 参数2: 要修改的属性名，必须是字符串格式
```

```
      // 参数3: 要修改的具体配置
      Object.defineProperty(obj, "id", {
        // 快捷键: ctrl+i  快速弹出提示
        writable: false, // 是否可写
        configurable: false, //是否可重新配置
        enumerable: false, //是否可遍历
      });

      //遍历对象
      for (var name in obj) {
        console.log(name);
      }

      // 需求: id这个属性 只能读取查看, 不能修改!
      // obj.id = 9999; //不允许
      console.log(obj.id);

      // 不可修改配置: 如果用户使用时发现不可以写, 那么主动把 可写改为真?
      Object.defineProperty(obj, "id", {
        writable: true,
      });
      obj.id = 9999;
      console.log(obj.id);
    </script>
  </body>
</html>
```

精确添加对象属性

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      // 精确添加对象属性
      var obj = {
        name: "亮亮",
        age: 33,
      };
      // 新增id属性, 默认值 1001, 不可写 不可重新配置 不可遍历
      Object.defineProperty(obj, "id", {
        value: 1001, //默认值
        writable: false,
        configurable: false,
        enumerable: false,
      });
      console.log(obj);
    </script>
  </body>
</html>
```

精确配置多个对象属性

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      // 精确配置多个属性
      "use strict";

      var obj = {
        id: 100, //此方式声明的属性，所有配置都是 true!
      };
      // id不可修改，配置不可改
      // 新增 name属性，默认值:亮亮，不可修改,配置不可改
      // 薪资 salary属性 默认值9999，不能改 不能配置 不能遍历
      Object.defineProperties(obj, {
        id: { writable: false, configurable: false },
        // 用 此方式添加新的属性，所有的默认值都是false
        name: {
          value: "亮亮",
          // 默认值是false，所以下方两行可以不写，效果也一样!
          writable: false,
          configurable: false,
          // 如果不主动声明 可以遍历，则默认是不可遍历
          enumerable: true,
        },
        salary: {
          value: 9999,
          writable: false,
          configurable: false,
          enumerable: false,

        },
      });
      console.log(obj);
      for (var name in obj) {
        console.log(name); //salary 不可遍历，不可见
      }
      obj.id = 2222;
    </script>
  </body>
</html>
```

计算属性:getter

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
```

```
    <title>Document</title>
  </head>
  <body>
    <!-- getter: 计算属性 -->
    <script>
      // 矩形:
      var r1 = {
        width: 100,
        height: 40,
        //面积: 特点, 值是计算出来的!
        // area: ???
      };
      //声明计算属性: getter
      Object.defineProperty(r1, "area", {
        // 计算属性get  跟 value/writable 配置项是互斥的!
        // value: 300,
        // get: 获取,  当读取一个属性时, 会自动触发 get属性的函数, 把此函数的返回值 作为属性的值
        get: function () {
          console.log("有人要读取面积!");
          return this.width * this.height;
        },
      });

      console.log(r1.area);

      // 练习: 希望可以得到周长 = (长+宽)*2
      Object.defineProperty(r1, "perimeter", {
        get: function () {
          return (this.width + this.height) * 2;
        },
      });
      console.log(r1.perimeter);
    </script>
  </body>
</html>
```

计算属性的语法糖

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      // 语法糖: 作者提供的一种简化语法, 能够把一些常用代码进行简化
      // 字面量写法 就属于语法糖的一种
      var r1 = {
        width: 100,
        height: 30,
        // 语法糖写法, 固定格式,  计算属性
        // 使用时: 对象.area  计算属性比函数方法 使用时少写()
        get area() {
          return this.width * this.height;
        },
```

```
        // 使用时: 对象.area1();
        area1: function () {
          return this.width * this.height;
        },
      };
      console.log(r1.area);
      console.log(r1.perimeter);
    </script>
  </body>
</html>
```

赋值监听: setter

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      "use strict";
      // 赋值监听
      var yanan = {
        name: "亚楠",
        age: 19,
      };
      //要求: 亚楠的年龄范围是: 1~100 , 超过就要报错

      // 生成一个额外的变量, 来保存age的值, 防止下方的无限循环
      var __age; //__没有特殊作用, 习惯上把这种专门负责保存 set 方法设定的属性值的额外属性, 加__ 前缀, 以示特殊!

      Object.defineProperty(yanan, "age", {
        // 赋值,设置值: set
        set: function (value) {
          console.log("有新的年龄传入:", value);
          // 判断: 1-100 就正常赋值, 否则就报错!
          if (value ≥ 1 && value ≤ 100) {
            // __age 是var声明的, 不用加this, 属于普通作用域

            __age = value; //赋值, 会触发 age 的set方法
          } else {
            throw Error("年龄范围错误: " + value);
          }
        },
        // 读取时 返回 __age 的值,  这个变量的值是 set 设置的
        get: function () {
          return __age;
        },
      });

      yanan.age = 50; //应该报错!
      console.log(yanan);
      console.log(yanan.age); //触发get
```

```
      // 匿名函数自调用，就可以形成一个 闭包作用域
      // set 和 get 方法都存储在 yanan里，yanan 在window里，所以大家都是活的
      (function () {
        var __salary;

        Object.defineProperty(yanan, "salary", {
          get: function () {
            return __salary;
          },
          set: function (value) {
            if (value ≥ 1000 && value ≤ 100000) {
              __salary = value;
            } else {
              throw Error("薪资范围错误" + value);
            }
          },
        });
      })();

      // yanan.salary = 250; //会报错
      yanan.salary = 11250; //正常
      console.log(yanan.salary);
    </script>
  </body>
</html>

<!—
  set: 用来监听属性的赋值
  为什么? 防止一些明显错误的值的添加!
  如何做?
    修改属性的 set，在此方法中接受赋值，然后对值进行检测，如果正确才赋值-- 此处不能给当前属性赋值，会无限循环 -- 此处必须额外声明一个变量，来存储值

    读取时，再通过 get 方法，来返回那个存值的变量
 —>

<!—
   练习: 增加一个薪资属性 salary
   此属性的取值范围 1000~100000

   如果赋值错误，就抛出错误，否则正常赋值
—>
```

setter的语法糖

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      // 语法糖写法: 为了简化 set 监听操作
      var yanan = {
        name: "亚楠",
```

```
        set age(value) {
          if (value ≥ 1 && value ≤ 100) {
            // var a = {}; a.__age = 3;
            // 就会在当前对象中自动创建__age属性
            this.__age = value;
          } else {
            throw Error("年龄范围错误!");
          }
        },
        get age() {
          return this.__age;
        },
      };
      // __age: 应该是一个不可遍历的属性，藏起来，是搭配age使用的附庸产物
      // 默认 使用此方式新增的属性，所有配置项都是false，即 不可写，不可遍历，不可重新配置.   此处的
__age 应该是可写的才能存储值
      Object.defineProperty(yanan, "__age", { writable: true });

      // yanan.age = -1;
      yanan.age = 50;
      console.log(yanan);
    </script>
  </body>
</html>
```

闭包与其他代码在一起书写的问题

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      // day01 文档最上方 有哔哩哔哩视频地址
      //

      var a = 5;

      var yanan = {
        // __age: 33, //属性的具体配置都是true:  可写 可改 可遍历
        set age() {
          this.__age = 33;
        },
      };
      yanan.age = 44; //触发 __age 的赋值，也是所有配置都是true
      // yanan.__age = 33; //同上，都是true

      // 新增属性，所有配置都是假!
      // 因为 22 行的执行，导致__age 已经声明，所以此处是修改!
      Object.defineProperty(yanan, '__age', {})


      // 代码结尾一定要加分号，特别是 和闭包代码连在一起写的时候
      // 会把 闭包的() 当成函数的调用
```

```
      Object.defineProperty(
        yanan,
        "age",
        {}
      );  ← 这个分号十分重要!!!，如果不写 就会和下方的闭包联合在一起，会报错
       会认为是 xxx()()() 的结构!

      (function () {
        Object.defineProperty(yanan, "salary", {});
      })();
    </script>
  </body>
</html>
```

什么是函数：函数本质是对象，其中保存了一些代码.

调用函数就可以执行里面的代码 -- 好处是可以多次调用! --- 复用性

作业

制作一个 yanan 对象，新增一个 手机号属性:

当设置手机号 phone 时，必须格式正确，才能赋值，否则要弹出错误

> 手机号正则: ^1[3-9]\d{9}$

手机号可以正常读取

制作一个 立方体 cube 对象， 此对象有长宽高3个属性: length，width，height，随意赋值

制作计算属性，实现 面积area，体积volume，周长perimeter的读取

面积=(长x宽+宽x高+长x高)*2; 体积=长x宽x高; 周长=(长+宽+高)x4

原型:

为Date 对象 通过原型的注入新增 format1 方法

实现: 需要复习亮亮讲的日期对象部分

```
var d = new Date();
console.log(d.format1());  就能打印出 xxxx年xx月xx日 xx:xx:xx 的结构
```