

js 一问就跪了

闭包和匿名函数有什么区别？

什么是作用域 作用域链 有什么用？

js 是怎么实现继承的

闭包

javascript的function 就是一个天然的闭包

函数的对象可以通过作用域链互相关联起来，函数体内的var 都可以保存在函数作用域内
这种特性就叫做闭包

bind()方法

将 函数绑定至某个对象

```
var sum = function(x,y){  
    return x+y  
}
```

var lake=sum.bind(null,1) 绑定的时候 x 预留 , y =1
lake(12) 调用

函数式编程

事件编程

事件是js的活力之源

类和模块

一个构造函数 就是一个类 首字母大写

当两个对象继承自 同一个原型对象

他们属于同一个类的实例

`instanceof` 检测对象是否属于某个类

`constructor`属性的值是一个函数对象

```
var F = function(){};
var p = F.prototype;
var c = p.constructor;
```

```
console.info(c===F)
```

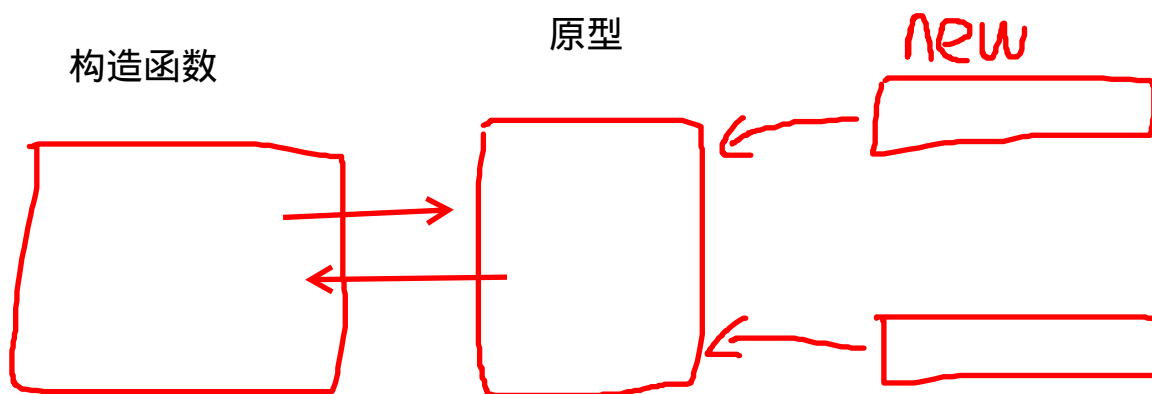
任何javascript 函数都可以做构造函数

并且调用 构造函数需要用到的一个 `prototype` 属性

每个javascript `function` 都自动有一个`prototype`属性

`Function.bind()` 返回的函数除外

`constructor`属性值 不可枚举



```
function lake() {}  
lake.prototype = {  
  age: function() {},  
  aaa: function() {}  
}
```

```
lake.prototype.age = function() {}
```

这是重写 prototype

没重写

2种写法是不一样的

父类 superclass

子类 subclass

子类调用父类中的 fn 叫做 方法链

就是用 call apply

事件驱动

想要程序响应一个事件 写一个fn 就是 事件处理fn

事件监听

回调

注册这个fn

事件发生的时候 就调用

addEventListener() attachEvent()

有浏览器兼容问题

创建可维护的代码对应用程序的成功至关重要

web页面包含不是该页面开发者所写的代码也是比较常见的，例如：

- 第三方的JavaScript库
- 广告方的脚本代码
- 第三方用户跟踪和分析脚本代码
- 不同类型的小组件，标志和按钮

如果大范围使用了全局变量 问题很严重 自己的js代码会和第三方js代码冲突
\$ 老被占用就是例子

不声明全局变量就没问题啦

还是有问题 可能会在写代码中无意的隐身的声明了

```
function sum(a,b){  
    result = a+b;  
    return result  
}
```

这个很容易看的出来 result 就是全局变量

```
function lake(){  
    var a=b=12;  
}
```

b 就是全局的

var的副作用

通过var创建的全局变量（任何函数之外的程序中创建）是不能被删除的。


无var创建的隐式全局变量（无视是否在函数中创建）是能被删除的。

```
var a=12;  
delete a 删不掉
```

关于预解析

```
myname = "global"; // 全局变量  
function func() {  
    alert(myname); // "undefined"  
    var myname = "local";  
    alert(myname); // "local"  
}  
func();
```

置顶解析



扩展内置原型

扩增构造函数的`prototype`属性是个很强大的增加功能的方法

增加内置的构造函数原型（如`Object()`，`Array()`，或`Function()`）

可以把需要的 `fn` 扩展到原型上 就可以直接调用了

把属性添加到原型上 首先要检测下 `name` 是否被占用

一般不直接在原型上扩展 至少我没用过

```
if (typeof Object.prototype.myMethod !== "function") {  
    Object.prototype.myMethod = function () {  
        // 实现...  
    };  
}
```

`eval()`是魔鬼

此方法接受任意的字符串，并当作JavaScript代码来处理

给`setInterval()`，`setTimeout()`和`Function()`构造函数传递字符串，大部分情况下，与使用`eval()`是类似的

```
// 反面示例  
setTimeout("myFunc()", 1000);  
setTimeout("myFunc(1, 2, 3)", 1000);
```

```
// 更好的  
setTimeout(myFunc, 1000);  
setTimeout(function () {  
    myFunc(1, 2, 3);  
}, 1000);
```

`parseInt()`你可以从字符串中获取数值

当字符串以"0" 开头的时候 会被当做8进制 `ecmascript5` 中有改进

```
var month = "06",  
    year = "09";  
month = parseInt(month, 10);  
year = parseInt(year, 10);
```

// 警告： 意外的返回值

```
function func() {
```

```
    return
```

```
    // 下面代码不执行
```

```
    {
```

```
        name : "Batman"
```

```
    }
```

```
}
```

return 会直接加 ;

这里 我有博客有记录 ; 什么时候加 什么时候省

js 里面有没有php 的定义常量 个人定义时候可以 var LAKE=12 大写下

私有方法 _name: function(){}

_ 作区分

构建大型的javascript应用 "不要"构建大型javascript 应用

解耦成一系列 相互平等 独立的部分

不要互相依赖

模型 不必知道 视图 和 控制器的细节

只需要 包含数据 及直接喝这些数据相关的逻辑

与模型无关的逻辑都应当隔离在模型之外

不要这样子写

```
var user = users['foo'];  
destroyUser(user);
```

在全局中 再来个 destroyUser 就跪了 容易冲突

应该写在属性里面

而且 类似 destroy() fn 不用在每个fn 定义一边
很容易继承

```
var user = user.find('foo');  
user.destroy();
```

这样子写

总而言之： 避免对全局造成污染

fn 放在属性里面

闭包里面

跑 都行

闭包 不要多用， 闭包有弊端

常驻内存，会增大内存使用量
this 指向问题

控制器

控制器 从 view 获得 事件 + 输入

对他们进行处理

函数表达式和函数声明

```
function lake() {}
```

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

var fn 解析不一样

IE的ECMAScript实现JavaScript严重混淆了命名函数表达式

例1：函数表达式的标示符泄露到外部作用域

```
var f = function g() {};  
typeof g; // "function"
```

命名函数表达式的标示符在外部作用域是无效的，

IE9貌似已经修复了这个问题

但JavaScript明显是违反了这一规范，

上面例子中的标示符g被解析成函数对象，这就乱了套了

很多难以发现的bug都是因为这个原因导致的

命名函数表达式会创建两个截然不同的函数对象！

```
var f = function g() {};  
f === g; // false
```

```
f.expando = 'foo';  
g.expando; // undefined
```

修改任何一个对象，另外一个没有什么改变，这太恶了

Module模式的基本特征：

- 1 模块化，可重用
- 2 封装了变量和function，和全局的namespace不接触，松耦合
- 3 只暴露可用public的方法，其它私有方法全部隐藏

简单的一个实现

```
var Calculator = function (eq) {  
    //这里可以声明私有成员  
  
    var eqCtl = document.getElementById(eq);  
  
    return {  
        // 暴露公开的成员  
        add: function (x, y) {  
            var val = x + y;  
            eqCtl.innerHTML = val;  
        }  
    };  
};  
  
var calculator = new Calculator('eq');  
calculator.add(2, 2);
```

每次用的时候都要new一下，也就是说每个实例在内存里都是一份copy

不new 应该也可以

更好地进行面向对象编程，五大原则分别是

The Single Responsibility Principle (单一职责SRP)

The Open/Closed Principle (开闭原则OCP)

The Liskov Substitution Principle (里氏替换原则LSP)

The Interface Segregation Principle (接口分离原则ISP)

The Dependency Inversion Principle (依赖反转原则DIP)

1. 类发生更改的原因应该只有一个

遵守单一职责的好处是可以让我们很容易地来维护这个对象，当一个对象封装了很多职责的话，一旦一个职责需要修改，势必会影响该对象想的其它职责代码。通过解耦可以让每个职责工更加有弹性地变化

我们如何知道一个对象的多个行为构造多个职责还是单个职责

