闭包和继承

闭包

如果让我们去开发一个银行账户的系统，账户的钱应该怎么用程序实现呢？

我们首先想到，应该存在变量中。然后我们可以访问到这个变量的值（查询余额）、进行加法运行（存钱）、减法运算（取钱）。。。

是很方便，但问题也随之而来：放在全局中的变量，是任何人都能访问到的，那就是任何人都能进行修改，这样对于账户来说是很不安全的。所以我们将这个变量放在局部，局部变量在全局中访问不到，是很安全，要想访问这个局部变量，我们只能将这个局部变量作为函数的返回值，如下：

```
function account(){
    var balance = 10;
    return balance;
}
var money = account(); // 10
```

这样能访问到，但是没办法进行存钱取钱的操作，因为在外面不管怎么修改，都是在操作money，当我们下次再访问余额的时候，看到的仍然是函数中的num。所以我们想到在account中创建一个更小的局部，那么就可以访问到这个变量，也可以对这个变量进行修改了。代码如下：

```
function account(){
    var balance = 10;
    function handle(){
        return balance;
    }
}
```

此时，没办法在外面访问到了，但是可以将handle函数作为account函数的返回值，这时在外面访问account函数的时候，就能得到handle函数：

```
function account(){
    var balance = 10;
    function handle(){
        return balance;
    }
    return handle;
}
var deal = account();
console.log(deal); // 返回的是handle函数本身，如果要访问到num就需要调用这个函数
var money = deal();
console.log(money) // 10
```

这样能访问到余额，那如果要存钱取钱如何操作呢？我们可以将存入的钱和要取出的钱作为参数传入handle，代码如下：

```
function account(){
    var balance = 10;
    function handle(num,access){
        if(access == 'cun'){ // 表示要存
           balance += num;
        }else if(access == 'qu'){
            balance -= num;
        }
        return balance;
    }
    return handle;
}
var deal = account();
// 查余额
var b = deal();
console.log(b); // 10
// 存钱
var b1 = deal(50,"cun");
console.log(b1); // 60
// 取钱
var b2 = deal(20,"qu");
console.log(b2); // 40
// 查余额
var b3 = deal();
console.log(b3); // 40
```

此时就实现了账户的程序。

观察现在的代码特点：

1. 函数中返回了一个函数，这样在全局中调用大函数可以得到小函数
2. 小函数中可以改变大函数中的数据，并且这个数据可以持久而不是每次调用就销毁

这种形式的代码叫做闭包。

一般函数，每次调用的时候，里面的变量都是重新开始，但闭包不会，因为闭包这个空间不会被销毁。

例：

一般函数：

```
function fn(){
    var num = 1;
    return ++num;
}
var n1 = fn();
console.log(n1); // 2
var n2 = fn();
console.log(n2); // 2
```

函数在执行的时候，会创建一个执行空间，用来执行函数中的代码，当函数中代码执行完毕后，这个空间就会销毁。当下一次调用的时候，会重新创建一个执行空间，重新执行。

闭包：

```
function fun(){
    var num=1;
    return function(){
        return ++num;
    }
}
var fn = fun();
fn(); // 2
fn(); // 3
```

调用fun的时候，在执行空间中，创建了一个空间，里面存储小函数的地址，这个地址在全局中被fn变量接收。只要在全局中能使用fn，那么执行空间就不能销毁，否则fn就不能用了。所以每次调用fn后，执行空间都没有销毁。

闭包：

函数嵌套，小函数内部使用大函数的变量，大函数就形成了闭包。

优点：

1. 作用域空间不销毁，所以变量也不会被销毁，增加了变量的声明周期
2. 在函数外部可以访问函数内部的变量
3. 保护私有变量，将变量定义在函数内，不会污染全局

缺点：

因为函数外部可以访问函数内部的变量，导致变量和内部的函数引用关系一直存在，内存不能销毁，会一直占用，使用量较大时会导致内存溢出

案例：利用闭包完成星星评分

```
var imgs = document.querySelectorAll("img");
for(var i=0;i<imgs.length;i++){
    (function(i){
        imgs[i].onclick=function(){
            // 小函数内部使用大函数的变量，形成闭包
            for(var j=0;j<=i;j++){
                imgs[j].src = 'rank3.jpg';
            }
            for(var j=imgs.length;j>i;j--){
                imgs[j].src = 'rank4.jpg';
            }
        }
    })(i)
}
```

继承

学习构造函数的时候写过一个构造函数Person，通过这个构造函数，可以实例化出很多对象，有张三对象、李四对象。。。这些对象有共同的特点，都有name属性，都有age属性，都有eat方法。。。

仔细想来，实例化出来的对象，是每个人，都从Person这个构造函数出来。我们就可以将Person这个构造函数称之为人类了。

其实每个构造函数都是一个类，可以实例化出很多具体的对象。

写一个兔子类，写一个狗类

```
function Rabbit(name){
    this.name = name;
}
Rabbit.prototype.pao=function(){
    console.log("能跑");
}
function Dog(name){
    this.name = name;
}
Dog.prototype.pao=function(){
    console.log("能跑");
}
```

从上面代码中能看出来，兔子类和狗类，也会有一些共同的特征：都有名字，都能跑。其实，这是每个动物都能有的特征。为了节省代码，也为了描述他们的关系，我们可以写一个动物类，让兔子类和狗类都具备动物的一些特征。这样就是实现了 --- "继承"。

原型继承

将父类的对象赋值给子类的原型。

```
function Animal(){
    this.name = "动物";
    this.attribute = "能动";
}
var animal = new Animal(); // 实例化动物类，得到动物对象
function Dog(){
    this.jiao = "汪汪";
}
Dog.prototype = animal; // 将狗类的原型赋值为动物对象
function Pig(){
    this.jiao = "哼哼";
}
Pig.prototype=animal; // 将猪类的原型赋值为动物对象
var ergou = new Dog();
console.log(ergou.name); // 访问狗对象的name属性 - 动物
var xiaohuang = new Pig();
console.log(xiaohuang.name); // 访问猪对象的name属性 - 动物
```

借用函数继承

把父类构造函数体借用过来使用一下。

```
function Animal(){
    this.name = "动物";
}
Animal.prototype.dong=function(){ // 将父类方法绑定到子类上
    console.log("能动");
}
function Dog(){
    Animal.call(this); // 将父类借用一下，并将函数的this原来是父类改变成子类
    this.jiao = '汪汪';
}
var ergou = new Dog();
console.log(ergou.name); // 访问子类的name属性- 动物
ergou.dong(); // 调用dong方法，报错不存在这个方法
```

借用函数继承有一个缺点：不能继承父类原型上的方法，所以在借用函数继承的基础上再进行原型的方式继承

混合继承

就是把 `原型继承` 和 `借用构造函数继承` 两个方式组合在一起

```
function Animal(){
    this.name = "动物";
}
Animal.prototype.dong=function(){ // 将父类方法绑定到子类上
    console.log("能动");
}
function Dog(){
    Animal.call(this); // 将父类借用一下，并将函数的this原来是父类改变成子类
    this.jiao = '汪汪';
}
Dog.prototype=new Animal();
var ergou = new Dog();
console.log(ergou.name); // 访问子类的name属性- 动物、
ergou.dong(); // 调用dong方法，也能访问了
```

es6的继承

在es6之前，可以说没有类这个概念，es6中新增了定义类（构造函数的方式）

语法：

```
class Animal{

}
var animal = new Animal();
console.log(animal); // Animal {}
```

这种方式定义的类，也必须和new配合使用。

如果在实例化需要传递参数的话，就在这个类中，写一个函数，名字必须是constructor

```
class Animal{
    constructor(name){
        this.name = name;
    }
}
var animal = new Animal("动物");
console.log(animal); // Animal {name: "动物"}
```

要给类添加方法，就像写constructor一样写一个函数，这种写法和我们以前给构造函数的原型上添加方法是一样的

```
class Animal{
    constructor(name){
        this.name = name;
    }
    dong(){
        console.log("能动");
    }
}
var animal = new Animal("动物");
console.log(animal); // Animal {name: "动物"}
animal.dong(); // 能动
```

这种方式的类继承很容易，而且是固定语法：

```
class 子类 extends 父类{
    constructor(){
                super();
        }
}
```

例：

```
class Animal{
    constructor(name){
        this.name = name;
    }
    dong(){
        console.log("能动");
    }
}
class Dog extends Animal{
    constructor(name){
        super(name);
        this.jiao = "汪汪";
    }
}
var ergou = new Dog("狗");
console.log(ergou); // Dog {name: "狗", jiao: "汪汪"}
ergou.dong(); // 能动
```

总结：继承就是让一个类拥有另一个类的属性和方法。