   闭包与函数对象

1.匿名函数

\* 匿名函数（anonymous）的概念：

     没有名字的函数就是匿名函数。当某个函数，function 关键字后面没有标识符的时候，那么它就是一个匿名函数（lamda，拉姆达函数）。

如：

  function(){

  alert(“亲，我来自无名函数”);

}

以前曾经写过 以下的代码：

   window.onload = function(){

  ……

  }

这就是把匿名函数赋给了事件属性。

\* 看看以下代码：

\* 写法一：  定义一个函数，把函数名赋给 onload属性

function test(){

    alert("亲1");

}

window.onload=test;

\* 写法二 : 定义一个匿名函数，直接赋给 onload属性

window.onload = function(){

    alert("亲2");

}

\* 写法三 ：定义一个匿名函数赋给一个变量；然后把变量赋给onload事件属性； 比写法二多了个中间变量而已

var test = function(){

    alert("亲3");

}

window.onload = test;

\* 写法四： 定义一个匿名函数赋给一个变量，用变量（当作函数名）来调用函数；

var test = function(){

    alert("亲4");

}

test();

\*  结论：函数可以赋给一个变量（其实函数是个对象）

   函数对象

   所以，函数名作为另外一个函数的参数就好理解了。

自运行：

\* 什么是自运行（函数）

\* 自运行（函数）示例：

  能够自己运行的一个函数。即，不用别人调用就能运行的函数。这是学习闭包的基础。

  (function(){alert(“亲4”);})()

推导一下：

1、已知：

var test = function(){

  alert("亲4");

}

test();

2、把test()中的test等量代换就是

     (function(){alert(“亲4”);})()

    不要急着问它的好处，不要急着觉

    得它有点多此一举，因为，还没有

    学习闭包，所以，不能体现它的好处

\* 自运行（函数）的其它写法：

\* 写法一：最外面加括号

//     (function(){

//         console.log("123");

//     }());

 //这是jslint推荐的写法，好处是，能提醒阅读代码的人，这段代码是一个整体。

\* 写法二：最后的括号前面加括号；

//   l (function(){

//         console.log("heihei");

//     })();

\* 写法三： function前面加运算符，常见的是!与void 。

//     !function fun(){

//         console.log("111");

//     }();

//     void function fun(){

//         console.log("111");

//     }();

\* 带参匿名函数

//带参匿名函数

var test = function(str1,str2){

  return str1+str2;

}

window.onload=function(){

  alert(test(12,23));

  alert(test("hello"," 亲"));

}

\* 匿名函数的常见用法：

    \* 函数表达式（var test = function(){}）

    \* 事件处理函数 (window.onload = funciont(){})

    \* 自运行（立即执行）

    \* 闭包里的返回值 （闭包中会讲）

其实，当把匿名函数赋给一个变量时，就知道还可以用在何处（如：作为另外一个函数的参数），但不能进行运算

2.闭包

\* 官方解释：闭包是一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。（函数就是一个表达式）

  JavaScript中所有的function都是一个闭包。不过一般来说，嵌套的function所产生的闭包更为强大，也是大部分时候我们所谓的“闭包”。

\* 什么是闭包：闭包是指在函数外部访问函数作用域中变量（局部变量）的函数;或者说闭包就是能够读取其他函数内部变量的函数；或者说闭包是指有权访问另一个函数作用域中的变量的函数；

  由于在Javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成“定义在一个函数内部的函数”。

  通俗解释：

  闭包是指有权访问另外一个函数作用域中的变量的函数.可以理解为(能够读取其他函数内部变量的函数)

//    记数器:

    //全局变量  全局变量降低函数的独立性

//     var count = 0;

//     function add(){

//         return count++;

//     }

//     console.log(add());

//     console.log(add());

//     console.log(add());

    //局部变量  函数执行外  局部变量销毁

//     function add(){

//         var count = 0;

//         return count++;

//     }

//     console.log(add());

//     console.log(add());

//     console.log(add());

    //plus定义在add的内部，可以访问add局部变量count

    //f为一个全局变量,通过赋值后，成为add的返回值,也就是plus方法

    //访问到了add中的局部变量count

    //所以count虽然是局部变量,但不允许被销毁,plus就是闭包

//     function add(){

//         var count = 0;

//         function plus(){

//             return count++;

//         }

//         return plus;

//     }

//

//     var f = add();

//

//     console.log(f());

//     console.log(f());

//     console.log(f());

    //变身

//     function add(){

//         var count = 0;

//         return function(){

//             return count++;

//         }

//     }

//

//     var f = add();

//

//     console.log(f());

//     console.log(f());

//     console.log(f());

    //继续变身

//     var f = (function (){

//         var count = 0;

//         return function(){

//             return count++;

//         }

//     }());

//

//     console.log(f());

//     console.log(f());

//     console.log(f());

  闭包的作用:

  正常函数执行完毕后,里面声明的变量被垃圾回收处理掉,但是闭包可以让作用域里的 变量,在函数执行完之后依旧保持没有被垃圾回收处理掉

\* 可以读取函数内部的变量

\* 让这些变量的值始终保持在内存中。

\* 增加块级作用域

//java,C,C++都有块作用域,即一对花括号里定义的变量，只在该花括号里起作用。

//JS中，就算在花括号里定义的变量（会做声明的提前），属于整个函数内部

//JS中，没有块作用域，但是在闭包的写法里，可以体现出来。

function outerFunc(){

  var outVar = 10;

  var innerF = function (){

      var innerVar = 20;//该变量虽然隶属于outerFunc内部，但是它的作用域范围只在innerF对应的函数体内，属于块级作用域

  }

  alert(innerVar);

  return innerF;

}

\* 使用闭包的注意事项：

    \* 由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。

    \* 闭包会在父函数外部改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。(了解)

3.闭包案例

\* 示例：局部变量的累加

  看程序分析：

<script type="text/javascript">

function outerF(){

  var t=0;

  function innerF(){

  t++;

  console.log(t);

  }

  return innerF;

}

var f = outerF();

</script>

</head>

<body>

  <input type="button" value="累加一次" onclick="f()" />

</body>

\* 示例：事件绑定

<!DOCTYPE html>

<html>

    <head>

        <meta charset="utf-8">

        <title></title>

        <ul>

            <li>1111</li>

            <li>2222</li>

            <li>3333</li>

            <li>4444</li>

            <li>5555</li>

        </ul>

    </head>

    <body>

    </body>

</html>

<script>

    let list = document.getElementsByTagName("li");

    for(var i=0; i<list.length; i++){

        //list[i].index = i;

        (function fun(i){//自运行

            list[i].onclick = function(){

                //alert(this.index+1);

                alert(i);

            }

        }(i));

    }

</script>

\* 用闭包的方式实现mult(5)(6)(7)，表示三个数的乘法（5\*6\*7）

    function fun(m){

        return function(n){

            return function(k){

                return m\*n\*k;

            };

        }

    }

 console.log(fun(5)(6)(7));

4.函数对象

\* 函数的三种定义方式 (执行顺序稍有不同)

\* 函数声明  声明的方式有函数的提升

function test(ord){

  alert(“亲”+ord+“！”);

}

\* 函数表达式（匿名函数)

var test = function(ord){

  alert(“亲”+ord+“！”);

}

\* 对象的方式：

let test = new Function(参数1,参数2...,"函数体");

示例:

//     test();

//     function test(){ // 有变量提升

//         console.log("heihei");

//     }

//     test();

//     let test = function(){

//         console.log("heihei");

//     }

//    let test = new Function("param","console.log(param)");

//    test(666);

\*  函数是功能完整的类

    \*   ECMAScript 中函数实际上是功能完整的对象。即函数就是一个对象

    \*   格式：

        var functionName = new Function(arg1,..., argN, function\_body)

        每个 arg 都是一个参数，最后一个参数是函数主体（要执行的代码）。这些参数必须是字符串。

    \*   示例一（无参）：

        var sayHello = new Function("alert(\"hello 哥们1111！\");");

    \*   示例二（有参）：

        var sayHi = new Function("sN", "sM", "alert(sN + sM);");

        等价于以前的：

        function sayHi(sN, sM) {

          alert(sN + sM);

        }

    \*  既然定义一个函数就是创建一个函数对象。

    \*  那么只要执行函数定义的代码，就会创建新的函数对象

    回忆闭包中，一旦父函数被调用一次，闭包就会新创建一个函数。

    function f1(){

      var n = 250;

      function f2(){

            n++;

            alert(n);

      }

      return f2;

    }

    var f21 = f1();//f21就是f2

    f21();//调用f2;

    f21();//调用f2;

    var f22= f1();//f22就是f2

    f22();//调用f2;

    f22();//调用f2;

    虽然f21和f22的代码一样，但是，它们是两个不同的对象，即，每个的n值不一样。

    所以函数等价于类？

    \* 函数名作为参数和返回值：

    \* 函数名能作为返回值，那么匿名函数也就可以作为返回值。

 当我们知道函数是对象，函数名是对象名（变量名）时，函数名作为另外一个函数的参数和返回值就不难理解了。还记得当时讲数组的sort函数是，参数就是另外一个函数名。

\* 函数内置对象（ arguments和this），自动产生，不需要用函数名来调用

this的回顾:

1.事件绑定的方法

2.构造函数

3.类中出现的方法

4.普通函数

\* arguments对象保存函数的所有参数，虽然可以像数组一样的访问每个参数，但是，并不是标准的数组（没法使用push等函数）;arguments对象不但保存着所有的参数，而且还有一个名叫callee的属性。Callee属性是个指针，指向了arguments对象所在的函数。Callee属性有个好处，看如下阶

示例：

    function fun(a){

        console.log(a);

        for(let i=0;i<arguments.length; i++){

            console.log(arguments[i]);

        }

    }

    fun(11,23,34,53,63);

乘递归的示例，即可明白

function fun(n){

        let c = 0;

        if(n==1){

            return c = 1;

        }else{

            c = arguments.callee(n-1)\*n;

        }

        return c;

    }

    console.log(fun(10));

如果函数名进行修改，函数体不用修改。

5.函数的属性和方法

prototype属性:

          对于引用类型来说，prototype保存着所有实例方法的真正所在，即所有的实例方法都是在prototype中保存着，平时，我们在使用实例方法时，

          虽然用对象直接调用，但是真正的保存是在prototype中。在继承中prototype的作用更加明显

          我们创建的每个函数都有一个属性是prototype（原型），这是属性是个指针，指向一个对象，该对象的用途是包含所有实例共享的属性和方法。

          所有通过同一个构造函数创建的实例对象，都会共享同一个prototype。

示例：

    function Student(newId,newName,newAge){

        this.id = newId;

        this.name = newName;

        this.age = newAge;

        //该行为是属于某个对象还是属于整个类族？

//         this.sleep = function(){

//             console.log("晚上睡不着，上课睡最香");

//         }

    }

    Student.prototype.sleep = function(){

        console.log("晚上睡不着，上课睡最香");

    }

    let s1 = new Student(1,"大黄",18);

    let s2 = new Student(2,"小黄",19);

    s1.sleep();

    s2.sleep();

\* 区分原型的属性和实例的属性：

  实例：用new调用构造函数创建出来的对象叫做实例。

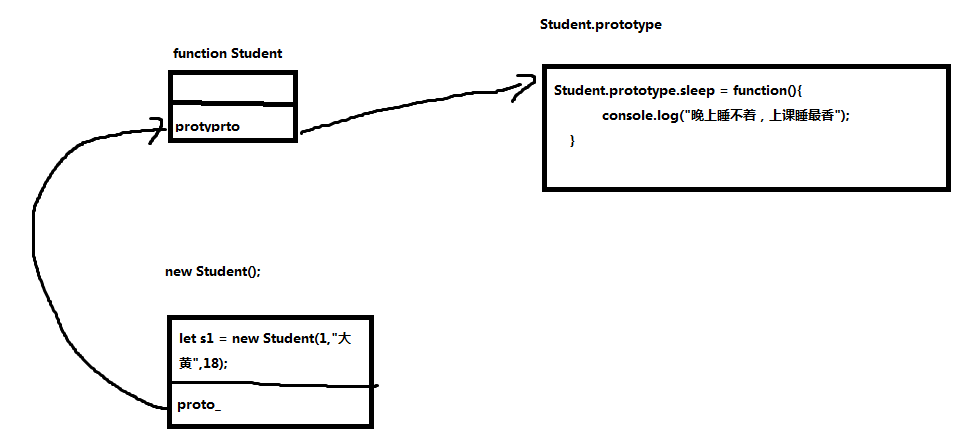
  原型属性：写在prototype后面的叫做原型属性 。

  实例属性：创建出来的对象，重新给原型属性赋值后，就成为了实例属性。

注意原型图：

    函数(类)的属性:prototype

    实例的属性:\_proto\_



    function Student(newId,newName,newAge){

        this.id = newId;

        this.name = newName;

        this.age = newAge;

    }

    Student.prototype.sleep = function(){

        console.log("晚上睡不着，上课睡最香");

    }

    Student.prototype.tall = "180";属于原型的属性，所有的实例（new Student()出来的叫实例）共享该属性

    let s1 = new Student(1,"大黄",18);

    console.log(s1.tall);

    s1.tall = "666";//此处s1修改了tall,则等价于为s1的实例对象添加了tall属性,这是该对象的个人行为，并不影响原型属性tall

    delete s1.tall;

    console.log(s1.tall);

    let s2 = new Student(2,"小黄",19);

    console.log(s2.tall);

给官方对象添加方法：

    也可以通过prototype属性给官方对象增加方法，如：给Array可以增加求数组最小数的函数。

      Array.prototype.max = function(){

        let tmax = this[0];

        for(let i=0; i<this.length; i++){

            if(tmax < this[i]){

                tmax = this[i];

            }

        }

        return tmax;

    }

    let a = [3,4,5,8,17,9];

    console.log(a.max());

6.函数的方法：apply()和call()

\*  apply()和call()方法

        每个函数都有两个非继承而来的方法apply()和call()，这两个方法的用途都是用来调用函数（在特定的作用域中），实际上等于设置函数体内的this对象的值。调用函数，实际上就是调用该函数对象的call内部方法。

  两者的区别：

        call（）方法的第一个参数和apply()的第一个参数一样，其它参数就是调用函数的参数（相当于把，apply第二个参数的每个元素单列出来）

        apply（）方法有两个参数，分别是运行函数的作用域（this），另一个是参数数组（可以是Array也可以是arguments）。

    function Monkey(newId,newName){

        this.id = newId;

        this.name = newName;

    }

    function Snake(newId,newName){

        this.id = newId;

        this.name = newName;

    }

    function eat(str,str1){

        console.log(this.name + "吃" + str + "和" + str1);

    }

    let m = new Monkey(1,"泰山");

    let s = new Snake(2,"小可爱");

    //引申出接口的概念

    //参数1为实力对象,后续参数为eat的参数

    eat.call(m,"香蕉","牛奶");

    eat.call(s,"老鼠","人");

    eat.apply(m,["香蕉","牛奶"]);

    eat.apply(s,["老鼠","人"]);

    apply与call的区别

    function fun(){

        let m = new Monkey(1,"泰山");

        eat.apply(m,arguments);

    }

    fun("香蕉","牛奶");

\* apply()和call() 真正的用途：

    把函数和对象之间进行解耦

    即对象和函数之间可以没有关联

作业

1.整理笔记，课堂案例

2.掌握闭包的特点和原理

3.掌握闭包的应用场景

4.prototype和\_proto\_概念

5.预习继承