`

JavaScript 语言精粹三

所有过失在未犯以前，都已定下应处的惩罚 - 威廉·莎士比亚《一报还一报》

起步：

- JavaScript 中极具特色的设计就是其函数的实现了
- 函数：即一组语句，它们是 JS 的基础模块单元，用于代码复用，信息隐藏和组合调用。
- 编程：什么为编程？即把一组需求分解成一组函数和数据结构的技能。

函数对象

- javascript中的函数就是对象，对象是'键值对'的集合，并且隐形的连接到原型对象
  - 对象字面量产生的对象连接到 Object.prototype
  - 函数对象连接到 Function.prototype（此原型对象本身连接到 Object.prototype）
- 函数对象在创建时，都配有一个**prototype** 属性[Foo.prototype]，它的值是一个拥有**constructor**属性，且属性值为该函数的对象[Foo.prototype.constructor==Foo]
  - 书中的这句话有些拗口且不容易理解，所以我觉得用代码会直观一些

```
/**
 *  Foo() 构造函数
 * Foo.prototype 是构造函数Foo的原型/ 是poo的原型对象
 * poo.__prototype__ 也是指向原型对象
 */
function Foo() {} //构造函数
var poo = new Foo(); //实例化一个对象
console.log(poo.__proto__ === Foo.prototype); //true ,说明是指向同一个原型对象
console.log(poo.__proto__.constructor); //Foo()
console.log(Foo.prototype.constructor); //Foo()
```

函数字面量

- 函数对象可通过函数字面量来创建，比如：

```
var add = function (par1, par2) {
  return par1 + par2;
};
```

- 函数字面量分为四个部分
  - 第 1 部分是保留字function
  - 第 2 部分是函数名（可省略--匿名函数），函数可通过函数名递归调用自身
  - 第 3 部分是在调用（圆括号）中的一组参数，多个以逗号分隔。这些参数称之为函数中的变量
  - 第 4 部分是在执行（大括号）中的一组语句，它们是函数的主题，在函数调用时依次执行

注：函数字面量可出现在任何允许表达式执行的地方，函数可以定义在另一个函数中，并且**函数除了可以访问自身的参数和变量外，还可以自由访问把它嵌套在自身函数体的父函数中的参数和变量**这称之为**闭包。**

- 书中未举例，为便于理解我编写一个简单闭包函数：

```
/**
 * JS的特点就是函数内容可以读取全局/父级里变量
 * 函数外部无法读取函数内部定义的变量（内部声明要加let或var，否则默认声明的是全局变量）
 * 下方parent中的所有变量都是对getname可见的，反之则不成立
 */
function parent() {
  var name = "poo";
  function getname() {
    return name;
  }
  return getname;
}
var Aha = parent();
Aha(); //'poo'
```

函数的调用

- 调用一个函数会暂停当前函数的执行，传递执行权和参数给新函数。
- 除了声明时定义的形参，每个函数还另外接受两个特殊参数`this`和`arguments`
- 在`JavaScript`中，一共有四种调用模式
  - 方法调用
  - 函数调用
  - 构造器调用
  - apply 调用
  - 这些模式在初始化关键参数`this`上存在很大的差异
- 调用运算符是跟在函数表达式后的一对圆括号，括号内可包含零或多个表达式，每个表达式产生一个参数值
- 实参与形参个数不匹配时不会出错，会忽略超出的参数值，会替换缺少的值为`undefined`。

方法调用模式

- 当一个函数被保存为对象的某一属性时，称它为一个方法，当该方法被调用时，this 就绑定到了该对象，若调用表达式包含 `.` 或者 `[ ]` 时，它被当做一个方法调用。
- 例如：

```
var myObj = {
  val: 233,
  getVal: function (par) {
    this.val += typeof par === "number" ? par : 1;
  },
};
myObj.getVal();
console.log(myObj.val); //234
```

```
/*--------------------*/
myObj.getVal(433);
console.log(myObj.val); //666
```

- 对象身上的方法可以通过 this 访问该对象身上的属性，因此也能进行取值和修改。
- 通过 this 取得它所属对象的上下文的方法，称之为公共方法。

构造器调用模式

- JavaScript是基于原型继承的语言，对象可直接从其他对象继承属性
- 如果在一个函数前面加上 new 关键字调用，那么底层是会创建一个连接到该函数 prototype 成员的新对象，同时绑定 this 至新对象
- 例如：

```
var Poo=function(city){
  this.citys=city;
}
Poo.prototype.getCity=function(){
  return this.citys;
}
var goWhere=new Poo('ZhengZhou');
console.log(goWhere.getCity());// ZhengZhou
```

- 函数若通常使用 new 来结合调用，那么它就被称之为构造器函数
- 约定俗成，这类函数要首字母大写用以区分

Apply 调用模式

- apply 方法可以构建一个参数数组传递给调用函数，并且可自由选择 this 的指向
- apply 中通常第一个参数为 this 绑定的值，第二个为参数数组

```
var add = function (par1, par2) {
  return par1 + par2;
};
var arrs=[1,2];
var count=add.apply(null,arrs) //3
//-构造一个包含type成员的对象,types并没有getRtx方法，但通过apply，仍可使用
var Poo=function(city){
  this.type=type;
}
Poo.prototype.getRtx=function(){
  return this.type;
}
var types={
  type:'RTX3080'
```

```
}
var getRtx=Poo.prototype.getRtx.apply(types);//RTX3080
```

参数

- 当函数被调用时，都会拥有一个 arguments数组，它里面包含了所有被调用时，传递给它的参数

```
var count = function () {
  var i = 0;
  sum = 0;
  for (i; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
};
console.log(count(1, 3, 4, 6, 7, 12)); //33
```

- 因为设计上的失误，导致 arguments 并不是一个真正的数组，而是一个类数组对象 (array-like)
- arguments是一个拥有 length 属性的特殊对象，但它不具备任何数组方法
    - 数组对象的类型是 Array， arguments的类型是 Object；
    - arguments 不能直接调用数组 API（不具备任何数组方法）；
    - 数组遍历可以用 for in 和 for 循环，arguments 只能用 for 循环遍历；

```
//把函数中的arguments转为数组
var newArr = Array.prototype.slice.call(arguments);
console.log(Array.isArray(arguments)); //false
console.log(Array.isArray(newArr)); //true
```

返回

- 当函数被调用时，从起始语句执行到遇到函数体 } 时结束，然后移交控制权至调用该函数的程序。
- retrun 语句可使函数结束执行提前返回，若无指定返回值，则返回 undefined。
- js 中大部分函数在未设置 return 值时都会默认返回一个 underfined。
- getReturn 不加执行体 () 时，只是单纯的是一个指向函数的指针而已。
- getReturn() 则是表示执行函数，并且获取函数执行的结果。

```
var getReturn = function () {
  return "hi~ Poo";
};
getReturn; //ƒ (){ return 'hi~ Poo'}
getReturn(); //hi~ Poo
```

扩充类型的功能

- JavaScript 可以扩充语言的基本类型，比如避免每次添加方法时都写入 prototype 名称

```
Function.prototype.method = function (name, fun) {
  this.prototype[name] = fun;
  return this;
};
```

- 可以通过给 Number.prototype 绑定自定义方法来简化我们的工作
- 当参数为负时向上取整，为正时向下取整

```
Number.methods('getInteger',function(){
  return Math[this<0?'ceil':'floor'>](this);
})
console.log((5/3).getInteger())//1
```

- 这种方法虽然方便，但是有很大的隐患，比如可能会覆盖类库中原有的方法
- 所以上方的代码优化后如下：

```
Function.prototype.method = function (name, fun) {
  if (!this.prototype[name]) {
    this.prototype[name] = fun;
  }
  return this;
};
```

> 注意：通常避免混淆，可以利用 hasOwnProperty 来判断是否是自身属性（非继承）来的

- 例如：

```
var Poo=function(){
  this.name='poo'
  this.getName=function(){
    console.log(`hi~ `${this.name})
  }
}
let aHa=new Poo()
console.log(aHa.hasOwnProperty('name')) //true
console.log(aHa.hasOwnProperty('getName')) //true
console.log(aHa.hasOwnProperty('toNumber'))//false
```

递归

- 简单来说：递归就是直接或间接调用自身的一种函数,基本概念分为
  - 递归条件：递归函数调用自己的条件

    - 基线条件：递归函数结束调用自己的条件
- such as:

```
function Recursive(num) {
  if (num > 1) {
    return num + Recursive(num - 1);
  } else return num;
}
Recursive(233); // 求233+222+...+2+1
```

> 总结：不考虑复杂的尾递归时，递归只是让解决方案更清晰，并没有任何性能上的优势，
> 甚至劣势于传统的 for 循环，还会有内存崩溃隐患。
>
> 至于递归底层涉及内存的压栈和弹出，你可以通过点击
> 👉 GitHub 查看详情 或者掘金里的 👉递归

作用域

- 作用域控制着参数与变量的生命周期 ， 例如：

```
var poo = function () {
  var a = 1,
    b = 2;
  var pooTx = function () {
    var b = 3,
      c = 4;
    //--此时这里可读取外部值，但b被就近覆盖，所以a=1,b=3,c=4
    a += b + c;
    //--经过运算影响后，a=8,b=3,c=4
    /* 补充 +优先级高于+=,所以上方相当于a+=(b+c) 或 a=a+(b+c)*/
  };
  //这里因在pooTx外，所以不受影响a=1,b=2[ 无法读取c ]
  pooTX();
  //-执行过了，因此a=8,b=2
};
```

- 但是令人头疼的是 `JavaScript` 有着函数作用域，意味着函数中的参数变量，内部任何地方都可见。
- （ `ES6` 中的 `let` 块级作用域声明符解决了这个痛点）

```
function testPar() {
  var flag = "poo";
  var update = (function () {
    flag = "aHaHa";
  })();
  var getpar = (function () {
    console.log(flag);
  })();
```

```
}
testPar(); //被别的函数篡改了， "aHaHa"
```

闭包

- 作用域的好处是，可以使父级函数内部的子级函数访问父级函数里的参数和变量
- 有趣的是，内部函数往往拥有着比它外部函数更长的生命周期。
- 该函数返回一个包含两个方法的对象，并且这两个方法可以继续访问该函数中的变量 `val`

```
var myObj = function () {
  var val = 1;
  return {
    increment: function (par) {
      val += typeof par === "number" ? par : 2;
    },
    getval: function () {
      return val;
    },
  };
};
myObj();
```

- 下方这个经典示例中：
  - 回调函数都会在循环结束以后才会执行，因此才会每次输出一个 6 出来。
  - 试图在循环中每次运行都捕获一个副本 i,但是根据作用域的工作原理
  - 它们五个函数都会被封印在一个共享的全局作用域中，才会只有同一个 i。
  - 等价于把延迟函数重复定义五次，不使用循环。

```
function star() {
  for (var j = 1; j <= 5; j++) {
    setTimeout(function timer() {
      console.log(j);
    }, j * 1000);
  }
}
/*结果为每隔一秒输出一个数字6总共五个*/
```

- 解决方案
  - 改用 `ES6` 中 `let` 关键字形成块级作用域(推荐)
  - 使用 `IIFE` 形成单独的函数作用域

```
for (var i = 1; i <= 5; i++) {
  (function () {
    var j = i;
    setTimeout(function timer() {
      console.log(j);
```

```
    }, j * 1000);
  })();
}
/*因为在各自的作用域中保留了各自所得到的i值这样结果就与自己预期相符了*/
```

模块

- 模块：是提供接口缺隐藏状态与函数的对象（函数）-使用闭包和函数来进行构造
- 通过使用函数产生模块，可以大幅度减少全局变量的使用
- 比如说给 `String` 添加一个自定义的 `pooMethods`方法，功能：
  - 检测到字符串中的 `HTML` 字符并转换为对应汉字

```
String.method('pooMethods',function(){
var transform={
  '>':'大于',
  '<':'小于',
  '=':'等于'
}
return function(){
  //‘#’开头和‘;’结尾内字符进行替换
  return this.replace(/#([^#;]+);/g,
  function(a,b){
    var r=transform[b];
    return typeof r==='String'?r:a;
  })
}
}());
console.log('123#<;213'.pooMethods())//123小于213
```

- 模块一般是：
  - 一个定义私有变量和函数的函数
  - 利用闭包创建可访问私有变量和函数的**特权函数**
  - 最后把这个函数保存到公共可访问的地方
- 好处：
  - 摒弃大量全局变量使用
  - 便于应用程序的封装和构造
  - 模块模式可以更改的来创建安全的对象

```
/**根据书中的示例创建*/
var serial_maker = function () {
  var prefix = "";
  var seq = 0;
  return {
    set_pre: function (p) {
      prefix = String(p);
    },
    set_seq: function (s) {
```

```
      seq = s;
    },
    gensym: function () {
      var result = prefix + seq;
      seq += 1;
      return result;
    },
  };
};
var seqer = serial_maker();
seqer.set_pre("poo");
seqer.set_seq("12345");
var unique = seqer.gensym(); // 'poo12345'
```

- 如上方示例用来产生一个无法影响内部 `prefix` 和 `seq` 安全的序列号

柯里化

> 柯里化(Currying)指的是将原来接受两个参数的函数变成新的接受一个参数的函数的过程。新的函数返回一个以原有第二个参数为参数的函数。

- 柯里化的作用：是函数式编程的一个重要概念。它既能减少代码冗余，也能增加可读性。
- 柯里化的定义：在数学和计算机科学中，柯里化是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术
- 简单应用如下：

```
/**
 * 解析:
 *  若函数以cc(xxx)(yyy)形式可以调用
 *  则sum(xxx)的返回值必然为一个函数，因为后方还要以(yyy)调用
 *  实现一个add方法，使计算结果能够满足如下预期:
 *  add(1)(2)(3) = 6;
 *  add(1, 2, 3)(4) = 10;
 *  add(1)(2)(3)(4)(5) = 15;
 */
function add() {
  // arguments类数组对象转化为数组对象备份
  var _args = Array.prototype.slice.call(arguments);
  // 在内部声明一个函数，利用闭包的特性保存_args并收集所有的参数值
  var _adder = function () {
    _args.push(...arguments);
    return _adder;
  };
  // 利用toString隐式转换的特性，当最后执行时隐式转换，并计算最终的值返回
  _adder.toString = function () {
    return _args.reduce(function (a, b) {
      return a + b;
    });
  };
  return _adder;
}
```

```
multi(1, 3, 5); //9
multi(1, 3)(5); //9
```

记忆

- 函数可将之前操作的结果记录在某对象中，从而避免无谓的重复运算（记忆）
- 这是一种很棒的优化方式，JavaScript 的对象和数组实现很简单

```
/**
 * flag 为想查询数列中第几项
 * curr为一项 next为二项 二者相加为第三项
 * 这是比较简便的方法
 */
function fbnq(flag) {
  function fn(flag, curr = 1, next = 1) {
    if (flag == 1) return curr;
    else return fn(flag - 1, next, curr + next);
  }
  return fn(flag);
}
console.log("斐波那契数列中第六项为", fbnq(6));
// ---这种方法比较更容易理解
function fb(n) {
  return n - 2 > 0 ? fb(n - 2) + fb(n - 1) : 1;
}
console.log("斐波那契数列中第六项为", fb(6));
//---输出一个指定长度的斐波那契数列
function arrs(num) {
  let arr = [];
  if (num - 2 > 0) {
    arr[0] = 1;
    arr[1] = 1;
    for (let i = 1; i <= num - 2; i++) {
      arr.push(arr[i - 1] + arr[i]);
    }
  } else {
    while (num > 0) {
      arr.push(1);
      num--;
    }
  }
  console.log(arr);
}
arrs(14);
```

总结：

> 本章节描述的是在JavaScript语言中的函数的一些核心知识点，但是因为本书定位是精粹系列，所以描述的较为简洁，虽然我补充了一些函数示例，但是先掌握这些知识点还是要自己多多练习。