# 第12章 函数进阶

在第5章中我们已经学习过了函数，不过那些只是表面的一些东西。在JavaScript中，函数是一等公民，可以像其他值一样作为参数来进行传递。这也给JavaScript这门语言带来了许多其他语言所没有的特性。

本章将学习如下内容：

- 立即执行的函数表达式
- 递归函数
- 闭包
- 高阶函数
- 函数柯里化
- 函数式编程介绍
- 函数的节流与防抖
- 变量的初始化

## 12-1 立即执行函数表达式

立即执行的函数表达式的英文全称为Immediately Invoked Function Expression，简称就为IIFE。这是一个如它名字所示的那样，在定义后就会被立即调用的函数。
我们在调用函数的时候需要加上一对括号，IIFE同样如此。除此之外，我们还需要将函数变为一个表达式，只需要将整个函数的声明放进括号里面就可以实现。具体的语法如下：

```
(function(){
    //函数体
})()
```

接下来我们来看一个具体的示例：

```
(function(){
    console.log("Hello");
})()
// Hello
```

IIFE可以在执行一个任务的同时，将所有的变量都封装到函数的作用域里面，从而保证了全局的命名空间不会被很多变量名污染。

这里我可以举一个简单的例子，在以前我们要交换两个数的时候，往往需要声明第三个临时变量temp

> 注：从ES6开始已经不需要这么做了，直接使用结构就可以交换两个数了

```
let a = 3,b = 5;
let temp = a;
a = b;
b = temp;
console.log(a);//5
console.log(b);//3
console.log(temp);//3
```

这样虽然我们的两个变量被交换了，但是存在一个问题，那就是我们在全局环境下也存在了一个temp变量，这就可以被称之为污染了全局环境。所以我们可以使用IIFE来解决该问题，如下：

```
let a = 3,b = 5;
(function(a,b){
    let temp = a;
    a = b;
    b = temp;
})(a,b)
console.log(a);//3
console.log(b);//5
console.log(temp);//报错
```

这是一个非常方便的功能，特别是有些时候我们在初始化一些信息时需要一些变量的帮助，但是这些变量除了初始化之后就再也不会用了，那么这个时候我们就可以考虑使用IIFE来进行初始化，这样不会污染到全局环境。

```
(function(){
    let days = ["星期天","星期一","星期二","星期三","星期四","星期五","星期六"];
    let date = new Date();
    let today = [date.toLocaleDateString(),days[date.getDay()]];
    console.log(`今天是${today[0]}, ${today[1]}, 欢迎你回来！`);
})()
//今天是2017-12-20, 星期三, 欢迎你回来!
```

这里我们只是想要输出一条欢迎信息，附上当天的日期和星期几，但是有一个很尴尬的地方在于上面定义的这些变量我们都只使用一次，后面就不会再用了，所以这个时候我们也是可以考虑使用IIFE来避免这些无用的变量声明。

通过IIFE，我们可以对我们的代码进行分块。并且块与块之间不会互相影响，哪怕有同名的变量也没问题，因为IIFE也是函数，在函数内部声明的变量是一个局部变量，示例如下：

```
(function(){
    //block A
    let name = "xiejie";
    console.log(`my name is ${name}`);
})();
(function(){
    //block B
    let name = "song";
    console.log(`my name is ${name}`);
})();
// my name is xiejie
// my name is song
```

在`var`流行的时代，JS是没有块作用域的。什么叫做块作用域呢？目前我们所知的作用域大概有两种：全局作用域和函数作用域。其中，全局作用域是指声明的变量可在当前环境的任何地方使用。函数作用域则只能在当前函数所创造的环境中使用。块级作用域是指每个代码块也可以有自己的作用域，比如在`if`块中声明一个变量，就只能在当前代码块中使用，外面无法使用。而用`var`声明的变量是不存在块级作用域的，所以即使在`if`块中用`var`声明变量，它也能在外部的函数或者全局作用域中使用。

```
function show(valid){
    if(valid){
        var a = 100;
    }
    console.log('a:',a);
}
show(true); // 输出a的值为100
```

这个例子中，a变量是在`if`块中声明，但是它的外部仍然能输出它的结果。

解决这个问题有两种方法，第一：使用ES6中的`let`关键字声明变量，这样它就有块级作用域。第二：使用IIFE，示例如下：

```
function show(valid){
    if(valid){
        (function(){
            var a = 100;
        })();
    }
    console.log('a:',a);
```

```
}
show(true); // 报错: a is not defined
```

当然，只要浏览器支持，建立尽量使用 `let` 的方式来声明变量。

# 12-2 变量初始化

## 12-2-1 执行上下文

在ECMAScript中代码的运行环境分为以下三种：

- 全局级别的代码：这是默认的代码运行环境，一旦代码被载入，JS引擎最先进入的就是这个环境
- 函数级别的代码：当执行一个函数时，运行函数体中的代码。
- Eval级别的代码：在Eval函数内运行的代码。

为了便于理解，我们可以将"执行上下文"粗略的看做是当前代码的运行环境或者说是作用域。下面我们来看一个例子，其中包括了全局以及函数级别的执行上下文，如下：

```
let one = "Hello";
let test = function(){
    let two = "Lucy",three = "Bill";
    let test2 = function(){
        console.log(one,two);
    }
    let test3 = function(){
        console.log(one,three);
    }
    test2();
    test3();
}
test();
```

上面这段代码，本身是没有什么意义的，我们主要是要使用这段代码来分析一下里面存在多少个上下文。在上面的代码中，一共存在4个上下文。一个全局上下文，一个test函数上下文，一个test2函数上下文和一个test3函数上下文。

通过上面的例子，我们就可以得出下面的结论：

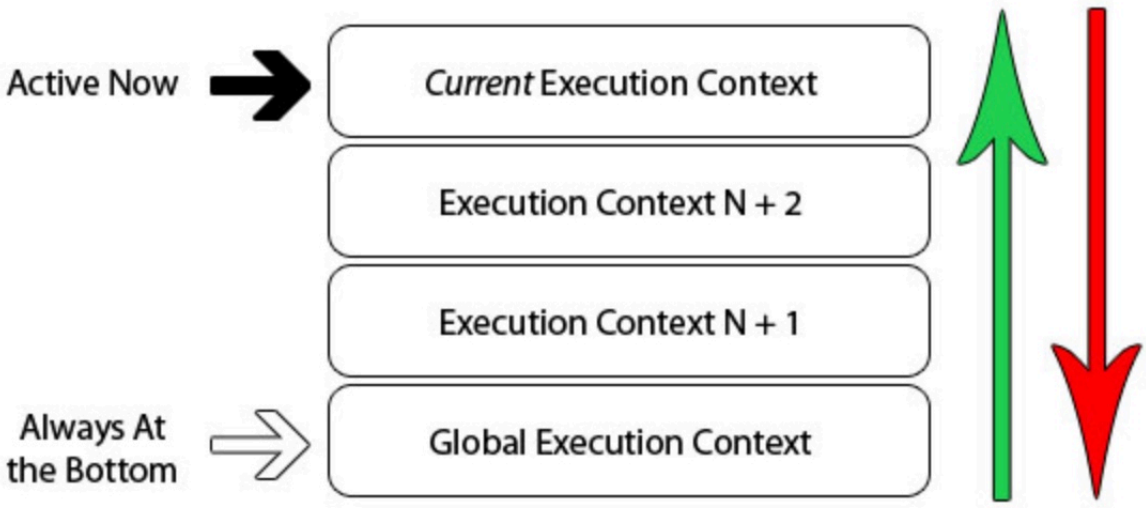
- 不管什么情况下，只存在一个全局的上下文，该上下文能被任何其它的上下文所访问到。也就是说，我们可以在test的上下文中访问到全局上下文中的one变量，当然在函数test2或者test3中同样可以访问到该变量。

- 至于函数上下文的个数是没有任何限制的，每到调用执行一个函数时，引擎就会自动新建出一个函数上下文，换句话说，就是新建一个局部作用域，可以在该局部作用域中声明私有变量等，在外部的上下文中是无法直接访问到该局部作用域内的元素的。

在上述例子中，内部的函数可以访问到外部上下文中的声明的变量，反之则行不通。那么，这到底是什么原因呢？引擎内部是如何处理的呢？这需要了解执行上下文堆栈。

## 执行上下文堆栈

JS引擎的工作方式是单线程的。也就是说，某一个时候只有唯一的一个事件是处于被激活的，其他的事件都是被放入队列中，等待被处理的。下面的示例图就描述了这样一个堆栈，如下：




我们已经知道，当JS代码文件被JS引擎载入后，默认最先进入的是一个全局的执行上下文。当在全局上下文中调用执行一个函数时，程序流就进入该被调用函数内，此时引擎就会为该函数创建一个新的执行上下文，并且将其压入到执行上下文堆栈的顶部。

JS引擎总是执行当前在堆栈顶部的上下文，一旦执行完毕，该上下文就会从堆栈顶部被弹出，然后，进入其下的上下文执行代码。这样，堆栈中的上下文就会被依次执行并且弹出堆栈，直到回到全局的上下文。

来看下面这段代码，分析其一共有多少个上下文：

```
(function foo(i){
    if(i==3)
    {
        return;
    }
    else{
        console.log(i);
        foo(++i);
    }
})(0);
// 全局上下文
// 函数上下文 0
// 函数上下文 1
```

```
// 函数上下文 2
// 函数上下文
```

上述foo被声明后，通过()运算符强制直接运行了。函数代码就是调用了其自身4次，每次是局部变量i增加1。每次foo函数被自身调用时，就会有一个新的执行上下文被创建。每当一个上下文执行完毕，该上下文就被弹出堆栈，回到上一个上下文，直到再次回到全局上下文。所以在本段代码中一共存在了5个不同的执行上下文。

由此可见，对于执行上下文这个抽象的概念，可以归纳为以下几点：

- 单线程
- 同步执行
- 唯一的一个全局上下文
- 函数的执行上下文的个数没有限制
- 每次某个函数被调用，就会有个新的执行上下文为其创建，即使是调用的自身函数，也是如此。

## 12-2-2 函数上下文的建立与激活

我们现在已经知道，每当我们调用一个函数时，一个新的执行上下文就会被创建出来。然而，在JS引擎的内部，这个上下文的创建过程具体分为两个阶段，分别是建立阶段和代码执行阶段。这两个阶段要做的事儿也不一样。

建立阶段：发生在当调用一个函数，但是在执行函数体内的具体代码之前

- 建立变量对象(arguments对象，形式参数，函数和局部变量)
- 初始化作用域链
- 确定上下文中this的指向对象

代码执行阶段：发生在具体开始执行函数体内的代码的时候

- 执行函数体内的每一句代码

我们将建立阶段称之为函数上下文的建立，将代码执行阶段称之为函数上下文的激活。

### 变量对象

在上面介绍函数两个阶段中的建立阶段时，提到了一个词，叫做变量对象。这其实是将整个上下文看做是一个对象以后得到的一个词语。具体来讲，我们可以将整个函数上下文看做是一个对象，那么既然是对象，对象就应该有相应的属性。对于我们的执行上下文来说，有如下的三个属性：

```
```

```
executionContextObj = {
    variableObject : {},//变量对象，里面包含arguments对象，形式参数，函数和局部变量
    scopeChain : {},//作用域链，包含内部上下文所有变量对象的列表
    this : {}//上下文中this的指向对象
}
```

可以看到，这里我们的执行上下文对象有3个属性，分别是变量对象，作用域链以及this，这里我们重点来看一下变量对象里面所拥有的东西。

在函数的建立阶段，首先会建立arguments对象。然后确定形式参数，检查当然上下文中的函数声明，每找到一个函数声明，就在variableObject下面用函数名建立一个属性，属性值就指向该函数在内存中的地址的一个引用。如果上述函数名已经存在于variableObject(简称VO)下面，那么对应的属性值会被新的引用给覆盖。最后，是确定当前上下文中的局部变量，如果遇到和函数名同名的变量，则会忽略该变量。

好，接下来我们来通过一个实际的例子来演示函数的这两个阶段以及变量对象是如何变化的。

```
let foo = function(i){
    var a = "Hello";
    var b = function privateB(){};
    function c(){}
}
foo(10);
```

首先在建立阶段的变量对象如下：

```
fooExecutionContext = {
    variavleObject : {
        arguments : {0 : 10,length : 1},//确定arguments对象
        i : 10,//确定形式参数
        c : pointer to function c(),//确定函数引用
        a : undefined,//局部变量 初始值为undefined
        b : undefined //局部变量 初始值为undefined
    },
    scopeChain : {},
    this : {}
}
```

由此可见，在建立阶段，除了arguments，函数的声明，以及形式参数被赋予了具体的属性值外，其它的变量属性默认的都是undefined。并且普通形式声明的函数的提升是在变量的上面的。

一旦上述建立阶段结束，引擎就会进入代码执行阶段，这个阶段完成后，上述执行上下文对象如

下，变量会被赋上具体的值。

```
fooExecutionContext = {
    variavleObject : {
        arguments : {0 : 10,length : 1},
        i : 10,
        c : pointer to function c(),
        a : "Hello",//a变量被赋值为Hello
        b : pointer to function privateB()//b变量被赋值为privateB()函数
    },
    scopeChain : {},
    this : {}
}
```

我们看到，只有在代码执行阶段，局部变量才会被赋予具体的值。在建立阶段局部变量的值都是undefined。这其实也就解释了变量提升的原理。

接下来我们再通过一段代码来加深对函数这两个阶段的过程的理解，代码如下：

```
(function(){
    console.log(typeof foo);
    console.log(typeof bar);
    var foo = "Hello";
    var bar = function(){
        return "World";
    }
    function foo()
    {
        return "good";
    }
    console.log(foo,typeof foo);
})()
```

这里，我们定义了一个IIFE，该函数在建立阶段的变量对象如下：

```
fooExecutionContext = {
    variavleObject : {
        arguments : {length : 0},
        foo : pointer to function foo(),
        bar : undefined
    },
    scopeChain : {},
    this : {}
}
```

首先确定arguments对象，接下来是形式参数，由于本例中不存在形式参数，所以接下来开始确定函数的引用，找到foo函数后，创建foo标识符来指向这个foo函数，之后同名的foo变量不会再被创建，会直接被忽略。然后创建bar变量，不过初始值为undefined。

建立阶段完成之后，接下来进入代码执行阶段，开始一句一句的执行代码，结果如下：

```
(function(){
    console.log(typeof foo);//function
    console.log(typeof bar);//undefined
    var foo = "Hello";//foo被重新赋值 变成了一个字符串
    var bar = function(){
        return "World";
    }
    function foo()
    {
        return "good";
    }
    console.log(foo,typeof foo);//Hello string
})()
```

## 12-2-3 作用域链

前面在讲解函数上下文时，我们将上下文看做了是一个对象，这个对象有3个属性，分别是变量对象，作用域链以及this指向。关于this指向我们之前已经介绍过了，变量对象也在上面做了相关的介绍，最后我们就一起来看一下这个作用域链。

所谓作用域链，就是内部上下文所有变量对象(包括父变量对象)的列表。此链主要是用于变量查询。

关于作用域链，有一个公式`作用域链(ScopeChain) = AO + [[scope]]`
其中AO，简单来说就是VO，AO全称为active object(活动对象)，对于当前的上下文来讲，一般将其称之为AO，对于不是当前的上下文，一般被称为VO
[[scope]]：所有父级变量对象的层级列表(也被称之为层级链)
举个例子：

```
var x = 10;
function foo()
{
    var y = 20;
    console.log(x + y);
}
foo();//30
```

这里，我们来分析一下VO和AO

```
//全局
VO : x : 10
     foo : pointer to foo()
//foo函数上下文
AO : y : 20
ScopeChain : AO(y) + [[scope]](VO:x)
```

这里，在全局上下文下的VO就是一个x变量，而在foo函数上下文下面，AO有一个变量y，接下来是作用域链。作用域链等于当前上下文的AO加上父级的VO，所以在函数内部虽然没有变量x，但是通过作用域链我们找到了父级上下文下面有一个变量x，然后拿来使用。

关于[[scope]]，有一个一个非常重要的特性，那就是[[scope]]是在函数创建的时候，就已经被存储了，是静态的。所谓静态，就是说永远不会变，**函数可以永远不被调用，但是[[scope]]在创建的时候就已经被写入了，**并且存储在函数作用域链对象里面。我们来举一个例子说明，如下：

```
let food = "rice";
let eat = function(){
    console.log(`eat ${food}`);
};
(function(){
    let food = "noodle";
    eat();//eat rice
})();
```

这里的结果为eat rice，原因非常简单。因为对于`eat()`函数来讲，创建的时候它的父级是全局上下文，所以[[scope]]里面就存储了全局上下文的VO，所以food的值为rice。如果我们将代码稍作修改，改成如下：

```
let food = "rice";
(function(){
    let food = "noodle";
    let eat = function(){
        console.log(`eat ${food}`);
    };
    eat();//eat noodle
})();
```

那么这个时候打印出来的值就为eat noodle。因为对于`eat()`函数来讲，这个时候它的父级为IIFE，所以[[scope]]里面存储的是IIFE这个函数上下文的VO，food的值为noodle。

最后，我们用一个稍微复杂一些的例子来贯穿上面所介绍的作用域链。

```
var x = 10;
function foo()
{
    var y = 20;
    function bar()
    {
        var z = 30;
        console.log(x + y + z);
    }
    bar();
}
foo();// 60
```

在这里，我们来分析一下变量对象，函数的[[scope]]属性以及上下文作用域链的变化。
首先，刚开始的时候是全局上下文的变量对象：

```
globalContext.VO = {
    x : 10,
    foo : pointer to foo()
}
```

在`foo()`函数被创建时，此时`foo()`函数的[[scope]]属性为：

```
foo.[[scope]] = [
    globalContext.VO
];
```

之后，`foo()`函数会被激活，确定`foo()`函数里面的活动对象：

```
fooContext.AO = {
    y : 20,
    bar : pointer to bar()
}
```

此时`foo()`函数上下文里面的作用域链的结构为：

```
fooContext.Scope = fooContext.AO + foo.[[scope]]
fooContext.Scope = [
    fooContext.AO,
```

```
    globalContext.VO
]
```

当内部函数`bar()`函数被创建时，其[[scope]]为：

```
bar.[[scope]] = [
    fooContext.AO,
    globalContext.VO
];
```

当`bar()`函数被激活，拥有活动对象时，`bar()`函数的活动对象如下：

```
barContext.AO = {
    z : 30
}
```

此时`bar()`函数上下文的作用域链的结构为：

```
barContext.Scope = barContext.AO + bar.[[scope]]
barContext.Scope = [
    barContext.AO,
    fooContext.AO,
    globalContext.VO
]
```

对`x`，`y`，`z`的标识符解析如下：

```
- x
- barContext.AO // not found
- fooContext.AO // not found
- globalContext.VO // found  10

- y
- barContext.AO // not found
- fooContext.AO // found  20

- z
- barContext.AO // found  30
```

最后总结一下：函数的[[scope]]属性是在函数创建时就确定了，而变量对象则是在函数激活时，也就是说调用函数时才会确定。

# 12-3 闭包

对于JavaScript程序员来说，闭包(closure)是一个难懂又必须征服的概念。接下来我将从四个方面来描述闭包的概念：

- 为什么要使用闭包
- 什么是闭包
- 闭包的原理
- 闭包的作用和使用

## 12-3-1 闭包基本介绍

首先我们来看看为什么要使用闭包，先看下面这个例子：

```
var eat = function(){
    var food = "鸡翅";
    console.log(food);
}
eat();
```

例子中声明了一个名为eat的函数，并对它进行调用。js引擎会创建一个eat的执行上下文，其中声明food变量并赋值。当该方法执行完后，上下文被销毁，food变量也会跟着消失。这是因为food变量属于eat函数的局部变量，它作用于eat函数中，会随着eat的执行上下文创建而创建，销毁而销毁。
再看下面这个例子：

```
var eat = function(){
    var food = "鸡翅";
    return function(){
        console.log(food);
    }
}
var look = eat();
look();
look();
```

在这个例子中，eat函数返回一个函数，并在这个内部函数中访问food这个局部变量。调用eat函数并将结果赋给look变量，这个look指向了eat函数中的内部函数，然后调用它，最终输出food的值。按照之前的说法，这个food变量应该当eat函数调用完后就销毁，后续为什么还能通过调用

look方法访问到这个变量呢？这是因为闭包起了作用。返回的内部函数和它外部的变量food实际上就是一个闭包。我们不禁想问，为什么它们称为闭包？闭包又能做什么呢？

我们先来看看闭包的概念：
闭包是指引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使离开了创造它的环境也不例外。这里提到了自由变量，它又是什么呢？

**自由变量可以理解成跨作用域的变量，比如子作用域访问父作用域的变量。**

这些概念说起来太过于专业，下面我用一个生活中的例子来解释它。假如小王、小强是某某学校计算机专业的学生。某一天他们的导师在外面接了一个项目，然后为此成立一个项目组，拉了他两人加入。他们的这个项目组既在学校中建立，但又可以独立于学校存在。比如，小王和小强从学校毕业了，他们的项目组仍然可以继续存在。所以，我们可以认为他们组建的这个项目组就是闭包。下面我把这个例子写成了代码：

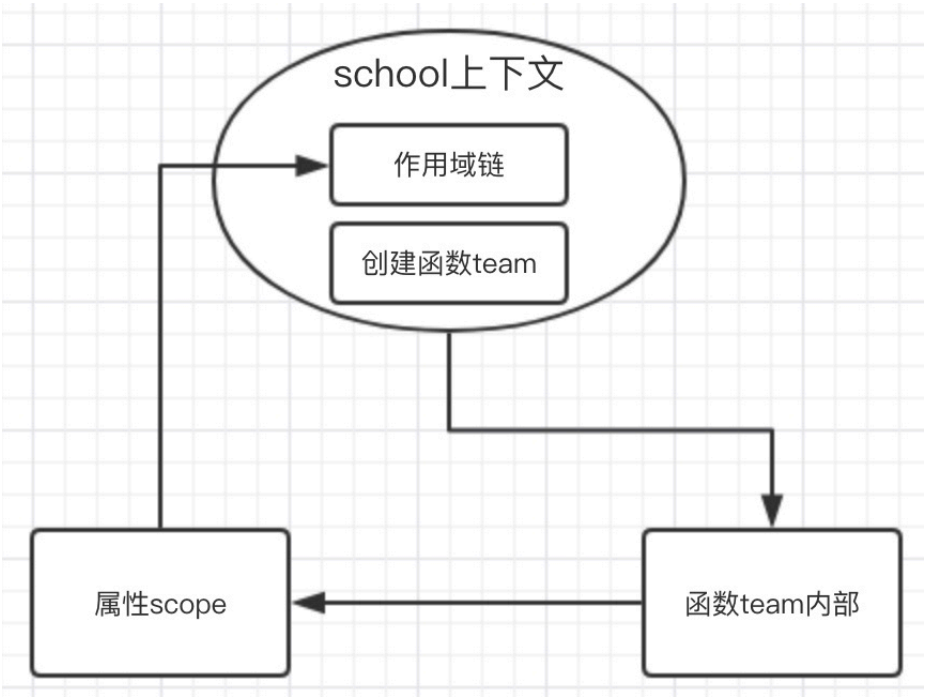
```
var school = function(){
    var s1 = "小强";
    var s2 = "小王";
    var team = function(project){
        console.log(s1 + s2 + project);
    }
    return team;
}
var team = school();
team("做电商项目");//小强、小王做电商项目
team("做微信项目");//小强、小王做微信项目
```

变量s1和s2属于school函数的局部变量，并被内部函数team使用，同时该函数也被返回出去。在外部，通过调用school得到了内部函数的引用，后面多次调用这个内部函数，仍然能够访问到s1和s2变量。这样s1和s2作为自由变量被team函数引用，即使创造它们的函数school执行完了，这些变量依然存在，因此，这就是闭包。

看到这里，我们需要问自己两个问题：

首先，为什么函数内部可以访问外部函数的变量？

这个问题其实很好解释，在变量初始化那节中提到过，当一个函数在创建时其内部会产生一个scope属性，该属性指向创建该函数的执行上下文中的作用域链对象。这句话说起来比较绕口，看看下面这张图：




其中作用域链对象包含了该上下文中的VO/AO对象，还有scope对象，比如school上下文中的作用域链对象就像这样：

```
school.scopeChain = {
    VO:{
        s1:"小强",
        s2:"小王"
    },
    scope:[[scopeChain]]
}
```

接下来，咱们就来回答这个问题。当内部函数中找不到对应的变量，它就会到scope指向的对象中找。该对象保存着外部上下文中的作用域链对象，从该作用域链中就能找到对应的变量。这就是为什么函数内部可以访问到外部函数变量的原因。下面咱们咱来看看第二个问题：

为什么当外部函数的上下文执行完以后，其中的局部变量还是能通过闭包访问到呢?

其实用上一个问题的答案再延伸一下，这个问题的答案就出来了。你想想，外部函数的上下文即使结束了，但内部的函数只要不销毁（被外部引用了，就不会销毁），它当中的scope就会一直引用着刚才上下文的作用域链对象，那么包含在作用域链中的变量也就可以一直被访问到。

把这个理解了，闭包的原理也就明白了。

按照这种说法，在JS中每个函数都有scope对象，并且都会保存外部上下文的作用域链对象。也就是说，任何时候外部上下文销毁了，只要内部函数还在都能访问到外部的变量。那岂不是任何函数都可以称为闭包了吗？事实上确实是这样的，从广义上讲，JS的函数都可以称为闭包(因为它们能访问外部变量)。但我们这里要讲的是狭义上的闭包，这样闭包对于实际应用来讲才会有意

义。

狭义的闭包必须满足两个条件：

- 形成闭包环境的函数能够被外部变量引用，这样就算它外部上下文销毁，它依然存在。
- 在内部函数中要访问外部函数的局部变量。

后面我提到的闭包都是指要满足这两个条件。下面我们来看看闭包有哪些优缺点，先来看看优点：

- 通过闭包可以让外部环境访问到函数内部的局部变量。
- 通过闭包可以让局部变量持续保存下来，不随着它的上下文环境一起销毁。看下面这个例子：

```
let count = 0; //全局变量
let compute = function(){ //将计数器加1
    count++;
    console.log(count);
}
for(let i = 0 ;i < 100;i++){
    compute(); //循环100次
}
```

这个例子是对一个全局变量进行加1的操作，一共加100次，得到值为100的结果。下面用闭包的方式重构它：

```
var compute = function(){
    var count = 0; //局部变量
    return function(){
        count++; //内部函数访问外部变量
        console.log(count);
    }
}
var func = compute(); //引用了内部函数，形成闭包
for(var i = 0 ;i < 100;i++){
    func();
}
```

这个例子就不再使用全局变量，其中count这个局部变量依然可以被保存下来。

下面来看看闭包的缺点：

其实闭包本身并没有什么明显的缺点。但往往人们对闭包有种误解：说闭包会将局部变量保存下

来，如果大量使用闭包，而其中的变量又未得到清理，可能会造成内存泄漏。所以要尽量减少闭包的使用。

局部变量本来应该在函数退出时被解除引用，但如果局部变量被封闭在闭包形成的环境中，那么这个局部变量就能一直生存下去。从这个角度来看，闭包的确会使一些数据无法被及时销毁。使用闭包的一部分原因是我们选择主动把一些变量封闭在闭包中，因为可能在以后还需要使用这些变量。把这些变量放在闭包中和放在全局作用域中，对内存方面的影响是一样的，所以这里并不能说成是内存泄漏。如果在将来需要回收这些变量，我们可以手动把这些变量设置为null。

如果非要说闭包和内存泄漏有关系的地方，那就是使用闭包的同时比较容易形成循环引用，如果闭包的作用域中保存着一些DOM节点，这个时候就有可能造成内存泄漏。但这本身并非闭包的问题，也并非JavaScript的问题。在IE浏览器中，由于BOM和DOM中的对象是使用C++以COM对象的方式实现的，而COM对象的垃圾收集机制采用的是引用计数策略。在基于引用计数策略的垃圾回收机制中，如果两个对象之间形成了循环引用，那么这两个对象都无法被回收，但循环引用造成的内存泄漏在本质上也不是闭包造成的。

同样，如果要解决循环引用带来的内存泄漏问题，我们只需要把循环引用中的变量设为null即可。将变量设置为null意味着切断变量与它此前引用的值之间的连接。当垃圾收集器下次运行时，就会删除这些值并回收它们占用的内存。

接下来我们看看到底什么时候会用到闭包。比如我们经常会使用时间函数对某一个变量进行操作，看这个例子：

```
let a = 100;
setTimeout(function(){
    console.log(++a);
},1000);
```

这个例子用到了时间函数setTimeout，并在等待1秒钟后对变量a进行加1的操作。这是一个闭包，因为setTimeout中的匿名函数对外部变量进行访问，然后该函数又被setTimeout方法引用。满足了形成闭包的两个条件。所以你看，即使外部上下文结束了，1秒后仍然能对变量a进行加1操作。
在DOM的事件操作中，也经常用到闭包，比如下面这个例子：

```
<input id="count" type="button" value="计数">
<script>
    (function(){
        var cnt = 0;
        var count = document.getElementById("count");
        count.onclick = function(){
            console.log(++cnt);
        }
```

```
    })()
</script>
```

onclick指向的函数中访问了外部变量cnt，同时该函数又被onclick事件引用了，满足两个条件，是闭包。所以当外部上下文结束后，你继续点击按钮，在触发的事件处理方法中仍然能访问到变量cnt。
在有些时候闭包还会引起一些奇怪的问题，比如下面这个例子：

```
for(var i = 1;i <= 3;i++){
    setTimeout(function(){
        console.log(i);
    },1000);
}
```

过1秒后分别输出i变量的值为1,2,3。但是，执行的结果是：4,4,4。实际上，问题就出在闭包身上。你看，循环中的setTimeout访问了它的外部变量i，形成闭包。而i变量只有1个，所以循环3次的setTimeout中都访问的是同一个变量。循环到第4次，i变量增加到4，不满足循环条件，循环结束，代码执行完后上下文结束。但是，那3个setTimeout等1秒钟后才执行，由于闭包的原因，所以它们仍然能访问到变量i，不过此时i变量值已经是4了。

既然是闭包引起的问题，那么解决的方法就是去掉闭包。我们知道形成闭包有两个条件，只要不满足其一，那就不再是闭包。条件之一，内部函数被外部引用，这个我们没办法去掉。条件二，内部函数访问外部变量。这个条件我们有办法去掉，比如：

```
for(var i = 1;i <= 3;i++){
    (function(index){
        setTimeout(function(){
            console.log(index);
        },1000);
    })(i)
}
```

这样setTimeout中就可以不用访问for循环声明的变量i了。而是采用调用函数传参的方式把变量i的值传给了setTimeout，这样它们就不再形成闭包。也就是说setTimeout中访问的已经不是外部的变量i，所以即使i的值增长到4，跟它内部也没关系，最后达到了我们想要的效果。

当然，解决这个问题还有个更简单的方法，就是使用ES6中的let关键字。它声明的变量有块作用域，如果将它放在循环中，那么每次循环都会有一个新的变量i，这样即使有闭包也没问题，因为每个闭包保存的都是不同的i变量，那么刚才的问题也就迎刃而解。

```
for(let i = 1;i <= 3;i++){
```

```
    setTimeout(function(){
        console.log(i);
    },1000);
}
```

## 12-3-2 闭包的更多作用

### 1. 封装变量

闭包可以帮助把一些不需要暴露在全局的变量封装成"私有变量"。假设有一个计算乘积的简单函数：

```
// 计算传入的数字的乘积
let mult = function(){
    let a = 1;
    for(let i=0;i<arguments.length;i++)
    {
        a *= arguments[i];
    }
    return a;
}
console.log(mult(1,2,3)); // 6
```

mult函数接受一些number类型的参数，并返回这些参数的乘积。现在我们觉得对于那些相同的参数来说，每次都进行计算是一种浪费，我们可以加入缓存机制来提高这个函数的性能，如下：

```
// 缓存对象，用于对计算结果进行缓存
let cache = {};
let mult = function(){
    // 将传入的数字组成字符串作为缓存对象的键
    let args = Array.prototype.join.call(arguments,',');
    if(cache[args])
    {
        return cache[args];
    }
    let a = 1;
    for(let i=0;i<arguments.length;i++)
    {
        a *= arguments[i];
    }
    return cache[args] = a;
}
console.log(mult(1,2,3)); // 6
console.log(mult(1,2,3)); // 6
```

我们看到，cache这个变量仅仅是在mult函数中被使用，与其让cache变量跟mult函数一起平行地暴露在全局作用域下，不如把它封闭在mult函数内部，这样可以减少页面中的全局变量，以避免这个变量在其他对方被不小心修改而引发错误。代码如下：

```
let mult = (function(){
    // 缓存对象，用于结果进行缓存
    let cache = {};
    return function(){
        let args = Array.prototype.join.call(arguments,',');
        if(cache[args])
        {
            return cache[args];
        }
        let a = 1;
        for(let i=0;i<arguments.length;i++)
        {
            a *= arguments[i];
        }
        return cache[args] = a;
    }
})();
console.log(mult(1,2,3)); // 6
console.log(mult(1,2,3)); // 6
```

提炼函数是代码重构中的一种常见技巧。如果在一个大函数中有一些代码块能够独立出来，那么我们常常把这些代码块封装在独立的小函数里面。独立出来的小函数有助于代码的复用，如果这些小函数有一个良好的命名，它们本身也起到了注释的作用。如果这些小函数不需要在程序的其他地方使用，那么最好是把它们用闭包封闭起来。重构后的代码如下：

```
let mult = (function(){
    // 缓存对象，用于结果进行缓存
    let cache = {};
    // 计算乘积函数 和cache一样，该函数同样被闭包封闭了起来
    let calc = function(){
        let a = 1;
        for(let i=0;i<arguments.length;i++)
        {
            a *= arguments[i];
        }
        return a;
    }
    return function(){
        let args = Array.prototype.join.call(arguments,',');
        if(cache[args])
```

```
        {
            return cache[args];
        }
        return cache[args] = calc.apply(null,arguments);
    }
})();
console.log(mult(1,2,3)); // 6
console.log(mult(1,2,3)); // 6
```

**2. 延续局部变量的寿命**

img对象经常用于进行数据上报，如下所示：

```
let report = function(src){
    let img = new Image();
    img.src = src;
}
report('http://xxx.com/getUserInfo');
```

但是通过查询后台的记录我们得知，因为一些低版本的浏览器的实现存在bug，在这些浏览器下使用report函数进行数据上报时会丢失30%左右的数据，也就是说，report函数并不是每一次都成功发起了HTTP请求。丢失数据的原因是img是report函数中的局部变量，当report函数在调用结束后，img局部变量随即被销毁，而此时或许还没来得及发出HTTP请求，所以此次请求就会丢失掉。

现在我们把img变量用闭包封闭起来，便能解决请求丢失的问题，如下：

```
let report = (function(){
    let imgs = [];
    return function(src){
        let img = new Image();
        imgs.push(img);
        img.src = src;
    }
})();
```

# 12-3-3 闭包和面向对象设计

过程与数据的结合是形容面向对象中的"对象"时经常使用的表达。对象以属性的形式包含了数据，以方法的形式包含了过程。而闭包则是在过程中以环境的形式包含了数据。通常用面向对象思想能实现的功能，用闭包也能够实现，反之亦然。

在JavaScript语言的祖先Scheme语言中，甚至都没有提供面向对象的原生设计，但却可以使用闭包来实现一个完整的面向对象的系统。下面我们来看看这段跟闭包相关的代码：

```
let Test = function(){
    let value = 0;
    return {
        call : function(){
            value++;
            console.log(value);
        }
    }
}
let test = new Test();
test.call(); // 1
test.call(); // 2
test.call(); // 3
```

如果换成面向对象的写法，那就是如下：

```
let test = {
    value : 0,
    call : function(){
        this.value++;
        console.log(this.value);
    }
}
test.call(); // 1
test.call(); // 2
test.call(); // 3
```

或者

```
let Test = function(){
    this.value = 0;
}
Test.prototype.call = function(){
    this.value++;
    console.log(this.value);
}
let test = new Test();
test.call(); // 1
test.call(); // 2
test.call(); // 3
```

# 12-4 递归函数

递归函数是一个一直直接或者间接调用它自己本身，直到满足某个条件才会退出的函数。当需要设计到迭代过程时，这是一个很有用的工具。下面我们以计算阶乘来进行示例：

```
let numCalc = function(i){
    if(i == 1)
    {
        return 1;
    }
    else{
        return i * numCalc(i-1);
    }
}
console.log(numCalc(4));//24
```

这里，我们要计算4的阶乘，那么我们可以看作是4乘以3的阶乘。而3的阶乘又可以看作是3乘以2的阶乘，以此类推。

下面是几个常见的递归函数的例子，通过下面的例子可以帮助我们加深对递归函数的理解

1.使用递归计算从m加到n

```
let numCalc = function(m,n){
    if(m == n)
    {
        return m;
    }
    else{
        return n + numCalc(m,n-1);
    }
}
console.log(numCalc(1,100));//5050
```

2.使用递归计算出某一位的斐波那契数

```
let numCalc = function(i){
    if(i == 1)
    {
        return 0;
    }
    else if(i == 2)
```

```
    {
        return 1;
    }
    else{
        return numCalc(i-1) + numCalc(i-2);
    }
}
console.log(numCalc(8));//13
```

3.使用递归打印出多维数组里面的每一个数字

```
let arr = [1,2,[3,4,[5,6],7,8],9,10];
let test = function(arr){
    for(let i=0;i<arr.length;i++)
    {
        if(typeof arr[i] == 'object')
        {
            test(arr[i]);
        }
        else{
            console.log(arr[i]);
        }
    }
};
test(arr);
```

# 12-5 高阶函数

在本小结中，我们将向大家介绍JavaScript中函数的高阶用法。

## 12-5-1 高阶函数介绍

高阶函数(higher-order-function)指的是操作函数的函数，一般有以下两种情况：

- 函数可以作为参数被传递
- 函数可以作为返回值输出

JavaScript中的函数显然满足高阶函数的条件，在实际开发中，无论是将函数当作参数传递，还是让函数的执行结果返回另外一个函数，这两种情形都有很多应用场景。下面将对这两种情况进行详细介绍

## 12-5-2 参数传递

把函数当作参数传递，代表可以抽离出一部分容易变化的业务逻辑，把这部分业务逻辑放在函数参数中，这样一来可以分离业务代码中变化与不变的部分。其中一个常见的应用场景就是回调函数。

### 1. 回调函数

前面无论是在介绍函数基础的时候，还是在介绍异步编程的时候，我们都有接触过回调函数。回调函数，就是典型的将函数作为参数来进行传递。

在Ajax异步请求的应用中，回调函数的使用非常频繁。想在Ajax请求返回之后做一些事情，但又并不知道请求返回的确切时间时，最常见的方案就是把回调函数当作参数传入发起Ajax请求的方法中，待请求完成之后执行回调函数。(我们将在14章详细介绍Ajax技术)

示例代码如下：

```
let getUserInfo = function (userId, callback) {
    $.ajax('http://xx.com/getUserInfo?' + userId, function (data) {
        if (typeof callback === 'function') {
            callback(data);
        }
    });
}
getUserInfo(123, function (data) {
    alert(data.userName);
```

```
});
```

回调函数的应用不仅只在异步请求中，当一个函数不适合执行一些请求时，也可以把这些请求封装成一个函数，并把它作为参数传递给另外一个函数，"委托"给另外一个函数来执行。比如，想在页面中创建100个div节点，然后把这些div节点都设置为隐藏。下面是一种编写代码的方式：

```
let appendDiv = function () {
    for (var i = 0; i < 100; i++)
    {
        var div = document.createElement('div');
        div.innerHTML = i;
        document.body.appendChild(div);
        div.style.display = 'none';
    }
};
appendDiv();
```

把`div.style.display = 'none'`的逻辑硬编码在`appendDiv`里显然是不合理的，`appendDiv`未免有点个性化，成为了一个难以复用的函数，并不是每个人创建了节点之后就希望它们立刻被隐藏，于是把`div.style.display = 'none'`这行代码抽出来，用回调函数的形式传入`appendDiv()`方法

```
let appendDiv = function (callback) {
    for (let i = 0; i < 100; i++)
    {
        let div = document.createElement('div');
        div.innerHTML = i;
        document.body.appendChild(div);
        if (typeof callback === 'function')
        {
            callback(div);
        }
    }
};
appendDiv(function (node) {
    node.style.display = 'none';
});
```

可以看到，隐藏节点的请求实际上是由客户发起的，但是客户并不知道节点什么时候会创建好，于是把隐藏节点的逻辑放在回调函数中，"委托"给`appendDiv()`方法。`appendDiv()`方法当然知道节点什么时候创建好，所以在节点创建好的时候，`appendDiv()`会执行之前客户传入的回调函数。

### 2. 数组排序

前面在介绍数组排序时有介绍过`sort()`方法，该方法就接收一个函数作为参数。`sort()`方法封装了数组元素的排序方法。从`sort()`方法的使用可以看到，我们的目的是对数组进行排序，这是不变的部分；而使用什么规则去排序，则是可变的部分。把可变的部分封装在函数参数里，动态传入`sort()`方法，使`sort()`方法方法成为了一个非常灵活的方法。我们这里来复习一下：

```
// 从小到大排列，输出：[ 1, 3, 4 ]
[1, 4, 3].sort(function (a, b) {
    return a - b;
});

// 从大到小排列，输出：[ 4, 3, 1 ]
[1, 4, 3].sort(function (a, b) {
    return b - a;
});
```

除了这个`sort()`方法以外，还有诸如`forEach()`，`map()`，`every()`，`some()`等函数，也是常见的回调函数。这些函数在前面都已经介绍过了，这里不再花篇幅进行介绍。

## 12-5-3 返回值输出

相比把函数当作参数传递，函数当作返回值输出的应用场景也许更多。让函数继续返回一个可执行的函数，意味着运算过程是可延续的。

#### 1. 判断数据的类型

我们来看下面的例子，判断一个数据是否为数组。在以往的实现中，可以判断这个数据有没有`length`属性，有没有`sort`方法或者`slice`方法等。但是更好的方法是使用`Object,prototype.toString`来计算。`Object.prototype.toString.call(obj)`返回一个字符串，比如`Object.prototype.toString.call([1,2,3])`总是返回"[object Array]"，而`Object.prototype.toString.call("str")`总是返回"[object String]"。下面是使用`Object,prototype.toString.call()`方法来判断数据类型的一系列的`isType`函数。

```
let isString = function (obj) {
    return Object.prototype.toString.call(obj) === '[object String]';
};
let isArray = function (obj) {
    return Object.prototype.toString.call(obj) === '[object Array]';
};
let isNumber = function (obj) {
```

```
    return Object.prototype.toString.call(obj) === '[object Number]';
};
```

我们发现，实际上这些函数的大部分实现都是相同的，不同的只是 `Object.prototype.toString.call(obj)` 返回的字符串。为了避免多余的代码，我们尝试把这些字符串作为参数提前传入 `isType` 函数。代码如下：

```
let isType = function (type) {
    return function (obj) {
        return Object.prototype.toString.call(obj) === '[object ' + type + '
]';
    }
};
let isString = isType('String');
let isArray = isType('Array');
let isNumber = isType('Number');
console.log(isArray([1, 2, 3]));// 输出：true
```

当然，还可以用循环语句，来批量注册这些 `isType` 函数：

```
let Type = {};
for (let i = 0, type; type = ['String', 'Array', 'Number'][i++];)
{
    (function (type) {
    Type['is' + type] = function (obj) {
        return Object.prototype.toString.call(obj) === '[object ' + type + '
]';
    }
    })(type)
}
console.log(Type.isArray([]));    // 输出：true
console.log(Type.isString("str")); // 输出：true
```

### 2. getSingle

下面的例子是一个单例模式的例子。

> 注：单例模式的定义是：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

```
<body>
    <script>
        let getSingle = function(fn){
            let ret;
```

```
            return function(){
                return ret || (ret = fn.apply(this.arguments));
            }
        }
        let createDiv = getSingle(function(){
            return document.createElement('div');
        });
        let div1 = createDiv();
        let div2 = createDiv();
        console.log(div1 === div2); // true
    </script>
</body>
```

在这个高阶函数的例子中，既把函数当作参数传递，又让函数执行后返回了另一个函数。

## 12-5-4 面向切面编程

AOP(面向切面编程)的主要作用是把一些跟核心业务逻辑模块无关的功能抽离出来，这些跟业务逻辑无关的功能通常包括日志统计、安全控制、异常处理等。把这些功能抽离出来之后，再通过"动态织入"的方式掺入业务逻辑模块中。这样做的好处首先是可以保持业务逻辑模块的纯净和高内聚性，其次是可以很方便地复用日志统计等功能模块

在Java语言中，可以通过反射和动态代理机制来实现AOP技术。而在JavaScript这种动态语言中，AOP的实现更加简单，这是JavaScript与生俱来的能力。通常，在JavaScript中实现AOP，都是指把一个函数"动态织入"到另外一个函数之中。下面通过扩展 `Function.prototype` 来实现，示例如下：

```
Function.prototype.before = function (beforefn) {
    let _self = this;    // 保存原函数的引用
    return function () {    // 返回包含了原函数和新函数的"代理"函数
        beforefn.apply(this, arguments);    // 先执行新函数，修正this
        return _self.apply(this,arguments);    // 再执行原函数
    }
};
Function.prototype.after = function (afterfn) {
    let _self = this;
    return function () {
        let ret = _self.apply(this, arguments); //先执行原函数
        afterfn.apply(this, arguments); //再执行新函数
        return ret;
    }
};
let func = function () {
    console.log(2);
```

```
};
func = func.before(function () {
    console.log(1);
}).after(function () {
    console.log(3);
});
func();
// 1
// 2
// 3
```

把负责打印数字1和打印数字3的两个函数通过AOP的方式动态植入`func()`函数。通过执行上面的代码，控制台顺利地返回了执行结果1、2、3。

本小结介绍了高阶函数的基础使用，主要包括参数传递和返回值输出两种形式。其中，高阶函数的一个重要应用是函数柯里化(currying)，将在下一小节中进行详细介绍。

# 12-6 函数柯里化

函数柯里化(currying)的概念最早由俄国数学家Moses Schönfinkel发明，而后由著名的数理逻辑学家Haskell Curry将其丰富和发展，currying由此得名。本小结将详细介绍函数柯里化(curring)。

## 12-6-1 函数柯里化定义

柯里化(currying)又称部分求值。一个柯里化的函数首先会接受一些参数，接受了这些参数之后，该函数并不会立即求值，而是继续返回另外一个函数，刚才传入的参数在函数形成的闭包中被保存起来。待到函数被真正需要求值的时候，之前传入的所有参数都会被一次性用于求值。

## 12-6-2 每月开销函数

从字面上理解柯里化并不太容易，下面通过编写一个计算每月开销的函数来解释函数柯里化(currying)。在每天结束之前，都要记录今天花掉了多少钱，示例代码如下：

```
let monthlyCost = 0;
let cost = function( money ){
  monthlyCost += money;
};
cost( 100 ); // 第 1 天开销
cost( 200 ); // 第 2 天开销
cost( 300 );   // 第 3 天开销
//...
cost( 700 );   // 第 30 天开销
alert ( monthlyCost );     // 输出1个月的总开销
```

通过这段代码可以看到，每天结束后都会记录并计算到今天为止花掉的钱。但其实我们并不太关心每天结束后，到今天为止花掉了多少钱，而只想知道到月底的时候会花掉多少钱。也就是说，实际上只需要在月底计算一次即可。

如果在每个月的前29天，我们都只是保存好当天的开销，直到第30天才进行求职计算，这样就达到了我们的要求。虽然下面的cost函数还不是一个currying函数的完整实现，但是有助于我们了解其思想：

```
let cost = (function () {
    let args = [];//该数组用于保存传入的参数
    //返回一个函数
    return function () {
        //如果没有参数，则计算args数组中的和
```

```
        if (arguments.length === 0)
        {
            let money = 0;
            for (let i = 0, l = args.length; i < l; i++)
            {
                money += args[i];
            }
            return money;
        }
        //如果有参数，则只能是将数据传到args数组中
        else {
            [].push.apply(args, arguments);
        }
    }
})();
cost(100); // 未真正求值
cost(200); // 未真正求值
cost(300); // 未真正求值
console.log(cost()); // 求值并输出：600
```

## 12-6-3 通用函数

下面来编写一个通用的柯里化函数currying，currying接受一个参数，即将要被currying的函数。如果和上面的例子结合，则这个函数的作用是遍历本月每天的开销并求出它们的总和，如下：

```
let currying = function (fn) {
    let args = [];
    return function () {
        if (arguments.length === 0)
        {
            return fn.apply(this, args);
        }
        else {
            [].push.apply(args, arguments);
            return arguments.callee;
        }
    }
};
let cost = (function () {
    let money = 0;
    return function () {
        for (let i = 0, l = arguments.length; i < l; i++)
        {
            money += arguments[i];
        }
        return money;
```

```
    }
})();
cost = currying(cost); // 转化成 currying 函数
cost(100); // 未真正求值
cost(200); // 未真正求值
cost(300);   // 未真正求值
console.log(cost());  // 求值并输出：600
```

至此，完成了一个currying函数的编写。当调用`cost()`时，如果明确地带上了一些参数，表示此时并不进行真正的求值计算，而是把这些参数保存起来，此时让`cost()`函数返回另外一个函数。只有以不带参数的形式执行`cost()`时，才利用前面保存的所有参数，真正开始进行求值计算。

## 12-6-4 可传参函数

实际上，柯里化函数不仅可以接收要柯里化的函数作为参数，也可以接收一些必要参数，下面是函数柯里化(currying)的改进代码。

```
let currying = function (fn) {
    let args = [];
    //储存传到curring函数中的除了fn之外的其他参数，并储存到args函数中
    args = args.concat([].slice.call(arguments, 1));
    return function () {
        if (arguments.length === 0)
        {
            return fn.apply(this, args);
        }
        else {
            //将fn中的参数展开，然后再储存到args数组中
            [].push.apply(args, arguments);
        }
    }
};
let cost = (function () {
    let money = 0;
    return function () {
        for (let i = 0, l = arguments.length; i < l; i++)
        {
            money += arguments[i];
        }
        return money;
    }
})();
let cost = currying(cost, 100, 200); // 转化成 currying 函数
cost(100, 200); // 未真正求值
```

```
cost(300);   // 未真正求值
console.log((cost()));  // 求值并输出：900
```

## 12-6-5 求值柯里化

如果函数柯里化(curring)之后，传参的同时伴随着求值的过程，则代码简化如下：

```
let currying = function (fn) {
    //获取除了fn之外的其他参数
    let args = [].slice.call(arguments, 1);
    return function () {
        //获取fn里的所有参数
        let innerArgs = [].slice.call(arguments);
        //最终的参数列表为args和innerArgs的结合
        let finalArgs = args.concat(innerArgs);
        //将finalArgs里的参数展开，传到fn中执行
        return fn.apply(null, finalArgs);
    };
};
let cost = (function () {
    let money = 0;
    return function () {
        for (let i = 0, l = arguments.length; i < l; i++)
        {
            money += arguments[i];
        }
        return money;
    }
})();
let cost = currying(cost, 100, 200); // 转化成 currying 函数
cost(300);//100+200+300=600
cost(100, 100);//(100+200+300)+(100+200+100+100)=1100
```

## 12-6-6 反柯里化

在JavaScript中，当我们调用对象的某个方法时，其实不用关心该对象原本是否被设计为拥有这个方法，这是动态类型语言的特点，也是常说的鸭子类型思想。

同理，一个对象也未必只能使用它自身的方法，那么有什么办法可以让对象去借用一个原本不属于它的方法呢？答案非常简单，使用call和apply都可以完成这个需求：

```
let obj1 = {name : 'xiejie'};
let obj2 = {
```

```
    getName : function(){
        return this.name;
    }
};
console.log(obj2.getName.call(obj1)); // xiejie
```

我们常常让类数组对象去借用Array.prototype的方法，这是call和apply最常见的应用场景之一：

```
(function(){
    Array.prototype.push.call(arguments,4);
    console.log(arguments);
    // { '0': 1, '1': 2, '2': 3, '3': 4 }
})(1,2,3);
```

在我们的预期中，Array.prototype上的方法原本只能用来操作`array`对象。但用`call`和`apply`可以把任意对象当作`this`传入某个方法，这样一来，方法中用到`this`的地方就不再局限于原来规定的对象，而是加以泛化并得到更广的适用性

那么有没有办法把泛化`this`的过程提取出来呢？反柯里化(uncurrying)就是用来解决这个问题的。反柯里化主要用于扩大适用范围，创建一个应用范围更广的函数。使本来只有特定对象才适用的方法，扩展到更多的对象。

`uncurrying`的话题来自JavaScript之父`Brendan Eich`在2011年发表的一篇文章。以下代码是`uncurrying`的实现方式之一：

```
Function.prototype.uncurrying = function () {
    let self = this;
    return function () {
        let obj = Array.prototype.shift.call(arguments);
        return self.apply(obj, arguments);
    };
};
```

在讲解这段代码的实现原理之前，我们先来瞧瞧它有什么作用。在类数组对象arguments借用Array.prototype的方法之前，先把Array.prototype.push.call这句代码转换为一个通用的push函数：

```
Function.prototype.uncurrying = function () {
    let self = this;
    return function () {
        let obj = Array.prototype.shift.call(arguments);
        return self.apply(obj, arguments);
```

```
    };
};
// 把Array.prototype.push.call这句代码转换为一个通用的push函数
let push = Array.prototype.push.uncurrying();
(function(){
    push(arguments,4);
    console.log(arguments);
    // { '0': 1, '1': 2, '2': 3, '3': 4 }
})(1,2,3);
```

通过uncurrying的方式，Array.prototype.push.call变成了一个通用的push函数。这样一来，push函数的作用就跟Array.prototype.push一样了，同样不仅仅局限于只能操作array对象。而对于使用者而言，调用push函数的方式也显得更加简洁和意图明了。

我们还可以一次性地把Array.prototype上的方法"复制"到array对象上，同样这些方法可操作的对象也不仅仅只是array对象：

```
Function.prototype.uncurrying = function () {
    let self = this;
    return function () {
        let obj = Array.prototype.shift.call(arguments);
        return self.apply(obj, arguments);
    };
};
for(let i=0,fn,ary=['push','shift','forEach'];fn=ary[i++];)
{
    Array[fn] = Array.prototype[fn].uncurrying();
}
let obj = {
    "length" : 3,
    "0" : 1,
    "1" : 2,
    "2" : 3
};
Array.push(obj,4);
console.log(obj.length); // 4
let first = Array.shift(obj);
console.log(first); // 1
console.log(obj); // { '0': 2, '1': 3, '2': 4, length: 3 }
Array.forEach(obj,function(ele,index){
    console.log(index + ":" + ele);
    // 0:2
    // 1:3
    // 2:4
});
```

甚至Function.prototype.call和Function.prototype.apply本身也可以被uncurrying，不过这没有实用价值，只是使得对函数的调用看起来更像JavaScript语言的前身Scheme：

```
Function.prototype.uncurrying = function () {
    let self = this;
    return function () {
        let obj = Array.prototype.shift.call(arguments);
        return self.apply(obj, arguments);
    };
};
let call = Function.prototype.call.uncurrying();
let fn = function(name){
    console.log(name); // xiejie
}
call(fn,global,'xiejie');
let apply = Function.prototype.apply.uncurrying();
let fn2 = function(name){
    console.log(this.name); // song
    console.log(arguments); // { '0': 1, '1': 2, '2': 3 }
}
apply(fn2,{name : 'song'},[1,2,3]);
```

目前我们已经给出了Function.prototype.uncurrying的一种实现。现在来分析调用Array.prototype.push.uncurrying()这句代码时发生了什么事情：

```
Function.prototype.uncurrying = function () {
    let self = this; // self此时是Array.prototype.push
    return function () {
        let obj = Array.prototype.shift.call(arguments);
        // 此时obj是{"length":1,"0":1}
        // arguments对象的第一个元素被截去，剩下[2]
        return self.apply(obj, arguments);
        // 相当于Array.prototype.push.apply(obj,2);
    };
};
let push = Array.prototype.push.uncurrying();
let obj = {
    'length' : 1,
    '0' : 1
};
push(obj,2);
console.log(obj); // { '0': 1, '1': 2, length: 2 }
```

除了上面所提供的代码实现，下面的代码是uncurrying的另一种实现方法：

```
Function.prototype.currying = function () {
    let self = this;
    return function () {
        return Function.prototype.call.apply(self, arguments);
    }
}
```

最终是都把`this.method`转化成`method(this,arg1,arg2....)`以实现方法借用和this的泛化。下面是一个让普通对象具备push方法的例子：

```
let push = Array.prototype.push.uncurrying(),
    obj = {};
push(obj, 'first', 'two');
console.log(obj);
/*obj {
: "first",
: "two"
}*/
```

通过`uncurrying`的方式，`Array.prototype.push.call`变成了一个通用的`push()`函数。这样一来，`push()`函数的作用就跟`Array.prototype.push`一样了，同样不仅仅局限于只能操作array对象。而对于使用者而言，调用`push()`函数的方式也显得更加简洁和意图明了。

最后，再看一个例子：

```
let toUpperCase = String.prototype.toUpperCase.uncurrying();
console.log(toUpperCase('avd')); // AVD
function AryUpper(ary) {
    return ary.map(toUpperCase);
}
console.log(AryUpper(['a', 'b', 'c'])); // ["A", "B", "C"]
```

# 12-7 高阶函数的其他应用

JavaScript中的函数大多数情况下都是由用户主动调用触发的，除非是函数本身的实现不合理，否则一般不会遇到跟性能相关的问题。但在一些少数情况下，函数的触发不是由用户直接控制的。在这些场景下，函数有可能被非常频繁地调用，而造成大的性能问题。解决性能问题的处理办法就是函数节流和函数防抖。

下面是函数被频繁调用的常见的几个场景：

- `mousemove`事件。如果要实现一个拖拽功能，需要一路监听`mousemove`事件，在回调中获取元素当前位置，然后重置DOM的位置来进行样式改变。如果不加以控制，每移动一定像素而触发的回调数量非常惊人，回调中又伴随着DOM操作，继而引发浏览器的重排与重绘，性能差的浏览器可能就会直接假死。
- `window.onresize`事件。为window对象绑定了`resize`事件，当浏览器窗口大小被拖动而改变的时候，这个事件触发的频率非常之高。如果在`window.onresize`事件函数里有一些跟DOM节点相关的操作，而跟DOM节点相关的操作往往是非常消耗性能的，这时候浏览器可能就会吃不消而造成卡顿现象。
- 射击游戏的`mousedown/keydown`事件(单位时间只能发射一颗子弹)
- 搜索联想(keyup事件)
- 监听滚动事件判断是否到页面底部自动加载更多(scroll事件)

对于这些情况的解决方案就是函数节流(throttle)或函数去抖(debounce)，核心其实就是限制某一个方法的频繁触发

## 12-7-1 函数防抖

函数防抖的原理是将即将被执行的函数用setTimeout延迟一段时间执行。对于正在执行的函数和新触发的函数冲突问题有两种处理，也分别对应了定时器管理的两种机制。

第一种是只要当前函数没有执行完成，任何新触发的函数都会被忽略，简易代码如下：

```
function debounce(method, context) {
    //忽略新函数
    if (method.tId)
    {
        return false;
    }
    method.tId = setTimeout(function () {
        method.call(context);
```

```
    }, 1000);
}
```

第二种是只要有新触发的函数，就立即停止执行当前函数，转而执行新函数，简易代码如下：

```
function debounce(method, context) {
    //停止当前函数
    clearTimeout(method.tId);
    method.tId = setTimeout(function () {
        method.call(context);
    }, 1000);
}
```

当然，不论是哪种处理，函数去抖的目的是让要执行的函数停止一段时间之后才执行。下面是一个比较完整的防抖函数(debounce)，该函数接受2个参数，第一个参数为需要被延迟执行的函数，第二个参数为延迟执行的时间，示例如下：

```
let debounce = function (fn, interval) {
    let _self = fn,    // 保存需要被延迟执行的函数引用
        timer,    // 定时器
        firstTime = true;    // 是否是第一次调用
    return function () {
        let args = arguments,
            _me = this;
        if (firstTime) // 如果是第一次调用，不需延迟执行
        {
            _self.apply(me, args);
            return firstTime = false;
        }
        if (timer) // 如果定时器还在，说明前一次延迟执行还没有完成
        {
            return false;
        }
        timer = setTimeout(function () { // 延迟一段时间执行
            clearTimeout(timer);
            timer = null;
            _self.apply(_me, args);
        }, interval || 500);
    };
};
window.onresize = debounce(function () {
    console.log(1);
}, 500);
```

# 12-7-2 函数节流

函数节流使得连续的函数执行，变为固定时间段间断地执行。关于节流的实现，有两种主流的实现方式，一种是使用时间戳，一种是设置定时器。

## 1. 使用时间戳

触发事件时，取出当前的时间戳，然后减去之前的时间戳(最一开始值设为`0`)，如果大于设置的时间周期，就执行函数，然后更新时间戳为当前的时间戳，如果小于，就不执行。

```
function throttle(func, wait) {
    let context, args;
    let previous = 0;
    return function () {
        let now = +new Date();
        context = this;
        args = arguments;
        if (now - previous > wait) {
            func.apply(context, args);
            previous = now;
        }
    }
}
```

## 2. 使用定时器

触发事件时，设置一个定时器，再触发事件的时候，如果定时器存在，就不执行，直到定时器执行，然后执行函数，清空定时器，这样就可以设置下个定时器。

```
function throttle(func, wait) {
    let timeout, args, context;
    let previous = 0;
    return function () {
        context = this;
        args = arguments;
        if (!timeout)
        {
            timeout = setTimeout(function () {
                timeout = null;
                func.apply(context, args)
            }, wait)
        }
    }
}
```

## 12-7-3 数组分块

在前面关于函数节流和函数防抖的讨论中，提供了限制函数被频繁调用的解决方案。下面将遇到另外一个问题，某些函数确实是用户主动调用的，但因为一些客观的原因，这些函数会严重地影响页面性能。

一个例子是创建WebQQ的QQ好友列表。列表中通常会有成百上千个好友，如果一个好友用一个节点来表示，在页面中渲染这个列表的时候，可能要一次性往页面中创建成百上千个节点。

在短时间内往页面中大量添加DOM节点显然也会让浏览器吃不消，看到的结果往往就是浏览器的卡顿甚至假死。代码如下：

```
let ary = [];
for (let i = 1; i <= 1000; i++)
{
    ary.push(i);// 假设 ary 装载了 1000 个好友的数据
};
let renderFriendList = function (data) {
    for (let i = 0, l = data.length; i < l; i++)
    {
        let div = document.createElement('div');
        div.innerHTML = i;
        document.body.appendChild(div);
    }
};
renderFriendList(ary);
```

这个问题的解决方案之一就是使用数组分块技术(注：又被称之为分时函数)，数组分块是一种使用定时器分割循环的技术，为要处理的项目创建一个队列，然后使用定时器取出下一个要处理的项目进行处理，接着再设置另一个定时器。

在数组分块模式中，array变量本质上就是一个"待办事宜"列表，它包含了要处理的项目。使用`shift()`方法可以获取队列中下一个要处理的项目，然后将其传递给某个函数。如果在队列中还有其他项目，则设置另一个定时器，并通过`arguments.callee`调用同一个匿名函数。

数组分块的重要性在于它可以将多个项目的处理在执行队列上分开，在每个项目处理之后，给予其他的浏览器处理机会运行，这样就可能避免长时间运行脚本的错误。一旦某个函数需要花50ms以上的时间完成，那么最好看看能否将任务分割为一系列可以使用定时器的小任务。

下面的`timeChunk()`函数让创建节点的工作分批进行，比如把1秒钟创建1000个节点，改为每隔200毫秒创建8个节点。以下是数组分块模式的简易代码：

```
function timeChunk(array, process, context) {
    setTimeout(function () {
        //取出下一个条目并处理
        let item = array.shift();
        process.call(context, item);
        //若还有条目，再设置另一个定时器
        if (array.length > 0) {
            setTimeout(arguments.callee, 100);
        }
    }, 100);
}
let data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0];
function printValue(item) {
    let div = document.getElementById('myDiv');
    div.innerHTML += item + '<br>';
}
timeChunk(data.concat(), printValue);
```

下面是数组分块的详细代码，timeChunk函数接受3个参数，第1个参数是创建节点时需要用到的数据，第2个参数是封装了创建节点逻辑的函数，第3个参数表示每一批创建的节点数量：

```
let timeChunk = function (ary, fn, count) {
    let obj, t;
    let len = ary.length;
    let start = function () {
        for (let i = 0; i < Math.min(count || 1, ary.length); i++) {
            let obj = ary.shift();
            fn(obj);
        }
    };
    return function () {
        t = setInterval(function () {
            if (ary.length === 0) // 如果全部节点都已经被创建好
            {
                return clearInterval(t);
            }
            start();
        }, 200); // 分批执行的时间间隔，也可以用参数的形式传入
    };
};
// 最后我们进行一些测试，假设我们有1000个好友的数据
// 我们利用timeChunk函数，每一批只往页面中创建8个节点
let ary = [];
for(let i = 1; i <= 1000; i++)
{
    ary.push(i);
```

```
}
let renderFriendList = timeChunk(ary, function (n) {
    let div = document.createElement('div');
    div.innerHTML = n;
    document.body.appendChild(div);
}, 8);
renderFriendList();
```

## 12-7-4 惰性加载函数

在Web开发中，因为浏览器之间实现的差异，一些嗅探工作总是不可避免的。比如我们需要一个在各个浏览器中能够通用的事件绑定函数addEvent，常见的写法如下：

```
let addEvent = function(elem,type,handler){
    if(window.addEventListener)
    {
        return elem.addEventListener(type,handler,false);
    }
    if(window.attachEvent)
    {
        return elem.attachEvent('on' + type,handler);
    }
}
```

这个函数的缺点是，当它每次被调用，也就是说每次给DOM元素绑定事件的时候，都会执行里面的if条件分支。虽然执行这些if分支的开销不算大，但也许有一些方法可以让程序避免这些重复的执行过程。

第二种方案是这样，我们把嗅探浏览器的操作提前到代码加载的时候，在代码加载的时候就立刻进行第一次判断，以便让addEvent返回一个包裹了正确逻辑的函数，代码如下：

```
let addEvent = (function(){
    if(window.addEventListener)
    {
        return function(elem,type,handler){
            elem.addEventListener(type,handler,false);
        }
    }
    if(window.attachEvent)
    {
        return function(elem,type,handler){
            elem.attachEvent('on' + type,handler);
        }
```

```
    }
})();
```

目前addEvent函数依然有个缺点，也许我们从头到尾都没有使用过addEvent函数，也就是说也许编写的页面压根儿就没有绑定过事件。这样看来，前一次的浏览器嗅探工作就是完全多余的操作，而且也会稍稍延长页面ready的时间。

第三种方案即是我们将要讨论的惰性赞入函数方法。此时addEvent依然被声明为一个普通函数，在函数里依然有一些分支判断。但是在第一次进入条件分支之后，在函数内部会重写这个函数，重写之后的函数就是我们所期望的addEvent函数，在下一次进入addEvent函数的时候，addEvent函数里不再存在条件分支语句，具体的代码如下：

```
<body>
    <div id="div1">点击我绑定事件</div>
    <script>
        let addEvent = function(elem,type,handler){
            if(window.addEventListener)
            {
                addEvent = function(elem,type,handler){
                    elem.addEventListener(type,handler,false);
                }
            }
            if(window.attachEvent)
            {
                addEvent = function(elem,type,handler){
                    elem.attachEvent('on' + type,handler);
                }
            }
            addEvent(elem,type,handler); // 第一次绑定事件的时候需要手动调用一下
        };
        let div = document.getElementById('div1');
        addEvent(div,'click',function(){
            alert(1);
        });
        addEvent(div,'click',function(){
            alert(2);
        });
    </script>
</body>
```

# 12-8 函数式编程

和Lisp、Haskell不同，JavaScript并非标准的函数式编程语言，但在JavaScript中可以像操控对象一样操控函数，也就是说可以在JavaScript中应用函数式编程技术。例如前面有介绍过的`map()`，`reduce()`等方法就非常适合于函数式编程风格，本节将针对函数式编程做一个简单的介绍。

## 12-8-1 函数处理数组

假设有一个数组，数组元素里面都是数字，想要计算这些元素的平均值和标准差。若使用非函数式编程风格的话，如下所示：

```
let data = [1,1,3,5,5];
let total = 0;
for(let i=0;i<data.length;i++)
{
    total += data[i];
}
let mean = total / data.length;
total = 0;
for(let i=0;i<data.length;i++)
{
    let deviation = data[i] - mean;
    total += deviation * deviation;
}
let stddev = Math.sqrt(total / (data.length-1));
console.log(`平均值为${mean}，标准差为${stddev}`);//平均值为3，标准差为2
```

可以使用数组方法`map()`和`reduce()`来实现同样的计算，这种实现极其简洁，示例如下：

```
let sum = function(x,y){
    return x + y;
}
let square = function(x){
    return x * x;
}
let data = [1,1,3,5,5];
let mean = data.reduce(sum)/data.length;
let deviations = data.map(function(x){
    return x - mean;
});
let stddev = Math.sqrt(deviations.map(square).reduce(sum)/(data.length-1));
```

```
console.log(`平均值为${mean}，标准差为${stddev}`);//平均值为3，标准差为2
```

## 12-8-2 不完全函数

不完全函数是一种函数的变换技巧，即把一次完整的函数调用拆成多次函数调用，每次传入的实参都是完整实参的一部分，每个拆分开的函数叫做不完整函数，每次函数的调用叫做不完全调用。这种函数变换的特点是每次调用都返回一个函数，直到得到最终运行结果为止。

接下来我们来看一个具体一点的示例，代码如下：

```
//该函数的作用是将伪数组对象(或对象)转换为真的数组
let array = function(obj,n){
    return Array.prototype.slice.call(obj,n || 0);
}
//这个函数的实参传递到左侧
function partialLeft(f){
    //arguments:{ '0': [Function: f], '1': 2 }
    let args = arguments;
    return function(){
        let a = array(args,1);//[ 2 ]
        a = a.concat(array(arguments));//[ 2, 3, 4 ]
        return f.apply(this,a);
    };
}
let f = function(x,y,z){
    return x * (y - z);
}
console.log(partialLeft(f,2)(3,4));//2*(3-4) = -2
```

上面示例的这个不完整函数，函数的实参传递到了左侧，接下来示例一个函数的实参传递到右侧的示例：

```
//该函数的作用是将伪数组对象(或对象)转换为真的数组
let array = function(obj,n){
    return Array.prototype.slice.call(obj,n || 0);
}
//这个函数的实参传递到右侧
function partialRight(f){
    var args = arguments;//{ '0': [Function: f], '1': 2 }
    return function(){
        let a = array(arguments);//[ 3, 4 ]
        a = a.concat(array(args,1));//[ 3, 4, 2 ]
        return f.apply(this,a);
    };
```

```
}
let f = function(x,y,z){
    return x * (y - z);
}
console.log(partialRight(f,2)(3,4));//3*(4-2) = 6
```

还可以让函数的实参被用作模板，实参列表中的undefined值都被填充，示例如下：

```
//该函数的作用是将伪数组对象(或对象)转换为真的数组
let array = function(obj,n){
    return Array.prototype.slice.call(obj,n || 0);
}
//这个函数的实参被用作模板，实参列表中的undefined值都被填充
function partial(f){
    var args = arguments;
    return function(){
        var a = array(args,1);
        var i = 0, j = 0;
        //遍历args，从内部实参填充undefined值
        for(;i<a.length;i++){
            if(a[i] === undefined){
                a[i] = arguments[j++];
            }
            //现在将剩下的内部实参都追加进去
        };
        a = a.concat(array(arguments,j));
        return f.apply(this,a);
    }
}
let f = function(x,y,z){
    return x * (y - z);
}
console.log(partial(f,undefined,2)(3,4)); //3*(2-4)=-6
```

利用这种不完全函数的编程技巧，可以编写一些有意思的代码，例如可以使用已有的函数来定义新的函数，如下：

```
let increment = partialLeft(sum,1);
let cuberoot = partialRight(Math.pow,1/3);
String.prototype.first = partial(String.prototype.charAt,0);
String.prototype.last = partial(String.prototype.substr,-1,1);
```

当将不完全调用和其他高阶函数整合在一起时，事件就变得格外有趣了。比如，下例定义了not()函数

```
let not = partialLeft(compose,function(x){
    return !x;
});
let even = function(x){
    return x % 2 === 0;
};
let odd = not(even);
let isNumber = not(isNaN);
```

## 12-8-3 记忆

将上次的计算结果缓存起来，在函数式编程中，这种缓存技巧叫做记忆(memorization)。记忆只是一种编程技巧，本质上是牺牲算法的空间复杂度以换取更优的时间复杂度，在客户端JavaScript中代码的执行时间复杂度往往成为瓶颈，因此在大多数场景下，这种牺牲空间换取时间的做法以提升程序执行效率的做法是非常可取的。

```
//返回f()的带有记忆功能的版本
//只有当f()的实参的字符串表示都不相同时它才会工作
function memorize(f) {
    let cache = {};//将值保存到闭包内
    return function () {
        //将实参转换为字符串形式，并将其用做缓存的键
        let key = arguments.length + Array.prototype.join.call(arguments, ",
");
        if (key in cache)
        {
            return cache[key];
        }
        else {
            return cache[key] = f.apply(this, arguments);
        }
    }
}
```

`memorize()`函数创建一个新的对象，这个对象被当作缓存的宿主，并赋值给一个局部变量，因此对于返回的函数来说它是私有的。所返回的函数将它的实参数组转换成字符串，并将字符串用做缓存对象的属性名。如果在缓存中存在这个值，则直接返回它；否则，就调用既定的函数对实参进行计算，将计算结果缓存起来并返回。

```
//返回两个整数的最大公约数
function gcd(a, b)
{
```

```
    let t;
    if (a < b)
    {
        t = b, b = a, a = t;
    }
    while (b != 0)
    {
        t = b, b = a % b, a = t;
    }
    return a;
}
let gcdmemo = memorize(gcd);
gcdmemo(85, 187);//17
```

写一个递归函数时，往往需要实现记忆功能，我们更希望调用实现了记忆功能的递归函数，而不是原递归函数

```
let factorial = memorize(function (n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
});
factorial(5);//120
```

## 12-8-4 连续调用单参函数

下面利用连续调用单参函数来实现一个简易的加法运算

```
add(num1)(num2)(num3)…;
add(10)(10) = 20
add(10)(20)(50) = 80
add(10)(20)(50)(100) = 180
```

如果完全按照上面实现，则无法实现，因为add(1)(2)如果返回3，add(1)(2)(3)必然报错。于是，有以下两种变形方法

第一种变形如下：

```
function add(n) {
    return function f(m) {
        if (m === undefined)
        {
            return n;
        }
        else {
```

```
            n += m;
            return f;
        }
    }
}
console.log(add(10)());//10
console.log(add(10)(10)());//20
console.log(add(10)(10)(10)());//30
```

第二种变形如下：

```
function add(n) {
    function f(m) {
        n += m;
        return f;
    };
    f.toString = f.valueOf = function () { return n }
    return f;
}
console.log(+add(10));//10
console.log(+add(10)(10));//20
console.log(+add(10)(10)(10));//30
```

# 总结

1. 立即执行函数表达式又被称之为IIFE，指的是在定义后就会立即被调用的函数。
2. 函数上下文的创建可以分为两个阶段，分别是建立阶段和代码执行阶段。
3. 如果将一个执行上下文看作是一个对象的话，那么这个对象拥有三个属性，分别是变量对象，作用域链以及this指向
4. 闭包可以分为广义和侠义的理解。从广义上来讲，所有的函数就是闭包。如果从侠义上来讲，必须满足两个条件，一是一个函数中要嵌套一个内部函数，并且内部函数要访问外部函数的变量。第二是内部函数要被外部引用。
5. 递归函数是一个直接或者间接调用自己本身，直到满足某个条件才会退出的函数。
6. 高阶函数是指操作函数的函数，一般分为两种情况，函数可以作为参数被传递，以及函数可以作为返回值输出。
7. 函数柯里化又被称之为部分求值。是指前面几次传入的参数在函数里面形成闭包保存起来，待到函数真正需要求值的时候，之前传入的所有参数都会被一次性用于求值。
8. JavaScript并非标准的函数式编程语言，但在JavaScript中可以像操控对象一样操控函数，也就是说可以在JavaScript中应用函数式编程技术。
9. 在一些时候，函数有可能被非常频繁地调用，而造成大的性能问题。解决性能问题的处理办法就是函数节流和函数防抖。