# 放置函数并提升

在不确定在哪里放置函数，以确保需要的时候可以访问到它？

如果要使用一个声明式的函数，可以将其放置在代码中的任何位置。然后，如果要使用的一个函数表达式，必须将其放置在使用函数的位置之前。

## 讨论

在考虑创建一个函数的时候，需要注意的是提升所有的变量声明都会移动或提升到当前作用域的顶部。这仅仅是指声明，而不是实例化。如果访问一个没有声明的变量，将会得到一个引用错误。在变量声明之前访问，将会得到一个undefined值。

如果声明一个变量的时候没有为其初始值，当访问的时候，将会得到一个undefined

console.log(a); //undefined

var a;

再如：

console.log(a);//undefined

var a = 1;

这是因为一个变量的声明提前了。而赋值并没有，仍然在当前位置。因此也将会得到一个undefined。

如果创建一个声明式函数，在访问该函数之前，提升将确保把函数声明移动到当前作用于的顶部。

console.log(fn());//fn函数

function fn(){

return 'fn函数';

}

当使用一个函数表达式时

console.log(fn1());//TypeError: fn1 is not a function

var fn1 = function(){

return 'fn1函数';

}

出现了引用错误，这个是因为fn1变量可能声明了，但是没有被实例化。而代码试图将这个没有实例化的变量当作一个函数来对待。

# 把一个函数当作参数传递给另一个函数

如何将一个函数作为参数传递给另一个函数？

对如下的函数

function fn(x,y,z){

x(y,z)

}

使用一个函数表达式（函数字面值）作为参数：

var fn1 = function(arg1,arg2){

console.log(arg1,arg2);//hello world

}

fn(fn1,'hello','world')

或者：

fn(function(arg1,arg2){console.log(arg1,arg2);},'hello','world')

## 讨论

function关键字是一个运算符，也是一条语句，并且可以用来创建作为表达式的一个函数。以这种方式创建的函数称为函数表达式、函数字面值和匿名函数。

命名一个函数表达式的好处：当调试JavaScript的时候，命名的函数会通过其名称显示在栈追踪中，而不是一个只显示的匿名函数。

可以像传递一个变量一样，将一个函数作为参数传递给另一个函数，或者直接方法参数列表中。函数字面值在页面加载的时候解析，而不是在每次访问的时候解析

## 函数式编程和JavaScript

一个函数接受另一个函数作为参数，或者返回一个函数，或者二者具备，这就是一个高阶函数。这个概念来自一个称为函数式编程的编程范型。函数式编程是对应用程序复杂性进行抽象的一种方式，它使用整齐、干净的函数调用代替了复杂的循环和条件语句。

使用条件、循环和相关联的过程进行JavaScript开发的传统方式，称为命令式编程。

### 作为函数式编程的一个示例

将数组中的所有数字相加。

var arr = [1,2,3,4,5,6,7,8,9];

var total = 0;

for(var i = 0; i < arr.length; i++){

total += arr[i];

}

console.log(total);//45

下面来看看Array.reduce()函数：

var total = arr.reduce(function(n1,n2) {

return n1+n2;

});

console.log(total)

上面的结果是相同的，但不同的是没有那么冗余的代码，可以清楚的理解发生了什么事情从而得到结果：便利数组元素，执行了一些操作，知道最终的结果归结为一个单个的值再次即：所有元素的值相加。

然而函数式编程真正的好处是：代码更具有可读性。当你将应用程序分解为函数块，并且每个函数执行一个单个的任务，唯一的数据通过参数形式来提供给函数使用，操作唯一的输出是由函数返回的。

给定函数一个相同的参数值的时候，总是会得到相同的结果，并且不会修改任何外部数据或触发任何副作用，这叫做纯函数。

换句话说：不管在应用程序中的任何地方发生了什么（在任何库中、对于任何全局变量），应用诸如Array.reduce()的一个函数，其效果总是一致的。

# 实现递归算法

如何实现一个函数，它递归遍历一个数组并返回一个反向的数组字符串？

var reverseArr = function(x,index,str){

return index == 0 ? str : reverseArr(x,--index,(str+=' '+x[index]));

}

var arr = [1,2,3,4,5,6,7,8,9];

var str = reverseArr(arr,arr.length,'');

console.log(str);// 9 8 7 6 5 4 3 2 1

## 讨论

递归是数学领域中众所周知的概念，也是计算机科学领域的概念。

在数学中，递归的一个例子就是裴波那契数列。

f(n) = f(n-1)+f(n-2);

f(0) = 0 f(1) = 1;

另一个就是阶乘。

4! = 4\*3\*2\*1;

使用JavaScript来实现如下：

var fibonacci = function(n){

return n<2 ? n : fibonacci(n-1)+fibonacci(n-2);

}

console.log(fibonacci(3));//2

和

function factorial(n){

return n === 1 ? 1 : n\*factorial(n-1);

}

console.log(factorial(3));//6

注：两个函数都不能操作负值。

递归函数很消耗内存的。所以应该慎用

## 尾调用优化

ES6提出了一项称为尾调用优化的新的JavaScript功能，或者称之为适当的尾调用。

如下

function factorial(num){

if(num === 0){

return 1;

}else{

return num\*factorial(num-1);

}

}

console.log(factorial(3))

在函数的末尾调用函数，称为尾调用，当前，每次调用递归函数的时候，都会想调用栈添加另一个帧。基本上，所发生的事情是：JavaScript引擎记录了函数调用及传递给函数的数据。当调用足够多的时候，内存将会耗尽，得到一个RangeError。

适当的尾调用优化所做的事情，就是复用相同的帧而不是添加一个新的帧。一旦加入了这一功能，讲不会再发生RangeError错误。

# 使用一个定时器和回调防止代码阻塞

有一段代码可能比较耗时间，并且在等待其完成的时候，不想阻塞其他的代码的处理，但是当消耗时间的函数完成的时候，确实需要执行一些功能？

将一个回调函数和setTimeout结合起来使用，定时器设置为0。

function factorial(n){

console.log(n);

return n === 1 ? 1 : n\*factorial(n-1);

}

function noBlock(n,callback){

setTimeout(function(){

var val = factorial(n);

if(callback && typeof callback == 'function'){

callback(val);

}

},0)

}

console.log('Top');

noBlock(3,function(n){

console.log('first:'+n);

noBlock(n,function(m){

console.log('final:'+m);

})

})

var tst = 0;

for(var i = 0; i < 10; i++){

tst += i;

}

console.log('value:'+tst);

noBlock(4,function(n){

console.log('end:'+n);

})

console.log('not')

输出结果：

Top

value:45

not

3

2

1

first:6

4

3

2

1

end:24

6

5

4

3

2

1

final:720

## 讨论

不考虑底层系统或应用的话，JavaScript不是多线程的：所有的进程都在单个的线程上执行的。通常这不是一个问题，除非当你要运行一段长的代码段，并且不想要阻塞应用程序的其他部分完成其任务。此外，可能想要延迟其他的代码的运行，直到那段长代码完成之后。

这两种编程挑战的一种解决方案：就是讲一个JavaScript定时器和一个回调函数结合起来使用。回调函数的意思就是一个函数作为参数传递给另一个函数，在某种条件下载函数内部调用传入的函数，或者在一个进程的末尾调用它。

当JavaScript定时器事件发生的时候，就像JavaScript中任何其他的异步事件一样，它添加到了事件队列的末尾，而不是立即挤入到队列中。具体如何进入队列以及进入到队列中的何处，这因浏览器和应用程序环境而不容，但是通常，和定时器事件相关的任何功能，都是在同一个队列中的任何其他功能之后处理的。

如果想要一个进程确切的在很多秒后执行，这可能会比较麻烦的，但是，该功能可能是在等待一个比较耗时的事件的同时而不阻塞应用程序的一种方便的方法。通过将setTimeout定时器设置为0，可以创建一个是案件并将其放到执行队列的末尾。通过将耗时的事件放入到定时器的进程中，在等待进程完成的同时不再阻塞。

当一个比较耗时的进程完成之后，通常想要执行一个最终的操作，因此，把一个回调函数传递给定时器进程。该函数只在进程完成之后调用，。

回调及等待进程进或时间的同时不阻塞的概念，对于鼠标单击和Ajax调用的事件处理，以及针对Node的底层处理架构来说，都是必须的。

# 创建能记住其状态的函数

想要创建一个函数，它能够记住数据，但是不必使用全局变量，并且不必对每个函数调用重新发送同样的数据？

创建一个外围函数，它接受一个或多个参数，然后创建一个内部函数，它也能接受一个或多个参数，但是为了执行其他功能，内部使用自己的参数及其复函数的参数。从外围哈拿书返回内部返回，并且将其赋值给一个变量，将该变量当做函数使用。

function greetingmaker(greeting){

function addName(name){

return greeting+ " " + name;

}

return addName;

}

var daytimeGreeting = greetingmaker('good day to you');

var nightGreeting = greetingmaker('good Evening');

console.log(daytimeGreeting('za'));

console.log(nightGreeting('li'))

## 讨论

尽可能的避免全局变量，因为，它和库有潜在的冲突。然而，有时候需要存储要跨越几个函数调用使用的数据，并且不想每次都必须重复的给太函数发送这些个信息。

维持这些从一个函数到另一个函数的数据的一种方法是：在一个函数内部创建一个函数，让他们都能访问该数据，然后，从外围函数返回内部函数。从另一个函数返回一个函数，当返回的函数使用外围的函数的作用域的时候，这称为函数闭包。

在JavaScript中没有一个最外围的应用程序创建的作用于。所有的全局变量、函数和对象都包含在这个外围的作用域之中。

当创建一个函数的时候，创建了一个新的作用于，只要这个函数存在，其作用于就存在。函数可以访问其作用于内的所有的变量，以及来自外围作用于的所有的变量。但那时外围作用域不能访问函数内的变量。由于这些作用域规则，可以在浏览器应用程序中访问window和document对象，并且解决方案中的内部函数也可以访问传递给它的数据，或者最初位于包围它的外围函数中的那些数据。

然，外围函数不能访问内部函数的参数及局部变量，因为他们存在于另一个作用域内。

当一个函数返回了一个函数，而返回的函数引用了外部函数的局部作用于：

function outer(x){

return function(y){

return x\*y;

}

}

var muliThree = outer(5);

console.log(muliThree(3));//15

返回的函数形成了一个闭包。JavaScript闭包是一个函数，也是创建该函数的时候所存在的环境。此外，上面的示例还展示了局部应用，在函数执行之前，函数的参数部分满足了。

当内部函数从外部函数返回，它此时的应用程序作用域，包含了对外部函数的变量的所有引用，现在，随该函数的持久化。因此，即便外部函数的应用程序作用域不再存在，内部函数的作用域在函数返回的时候还是存在的。其中包含了外部函数的数据的一个快照，内部函数的作用域将一直存在，直到该应用程序完成。

注：产生闭包的另一种方式是：一个内部函数赋值给一个全局变量。

# 把函数参数转换到一个数组中

想要对一个函数的参数使用Array功能，但是arguments对象并不是数组？

使用Array.prototype.slice()，然后是函数call方法。将arguments集合转换到一个数组中。

function fn(){

var args = Array.prototype.slice.call(arguments);

console.log(Object.prototype.toString.call(args));//[object Array]

console.log(Object.prototype.toString.call(args) == '[object Array]');//[object Array]

console.log(args);

}

fn(1,2,3,4,5,6,7);

function fn1(){

var args = [].slice.call(arguments);

console.log(Object.prototype.toString.call(args));//[object Array]

console.log(Object.prototype.toString.call(args) == '[object Array]');//[object Array]

console.log(args);

}

fn1(1,2,3,4,5,6,7);

## 讨论

arguments对象在一个函数中可用（并且只在函数中可用），并且是一个类似数组的对象，其中包含了传递给函数的参数。之所以说是类似数组，是因为具有一个length属性（参数的长度）。

slice方法返回了对数组的一部分的一个浅拷贝，或者如果没有给定起始值或结束值的话，就是整个数组的浅拷贝。slice方法也是一个函数，这就意味着诸如call这样的函数方式方法可以和其他一起使用。在此处，call传入了函数arguments，它对参数雷彪执行必要的转换，把得到的数组传递给slice。

call方法的第一个参数是this值，通常是调用对象自身，后面跟着任意数目的参数。再次：this是slice方法，并且外围的函数的参数作为slice的参数传递。这一技术所做的有效的事情是：迫使外围函数的arguments以一种可接受的格式充当slice的参数。

# 使用一个局部应用减少冗余性

有一个函数带有3个参数，但是前两个参数通常给予特定的用途而重复。那么如何避免参数的重复性？

创建一个函数，它接受三个值并返回一个结果，

然后创建一个可以接受两个参数的函数，并返回之前创建的函数。

function makeString(ldelim,str,rdelim){

return ldelim + str + rdelim;

}

function quoteString(str){

return makeString("'",str,"'");

}

function barString(str){

return makeString('-',str,'-');

}

function namedEntity(str){

return makeString('&#',str,';');

}

console.log(quoteString('apple'));//'apple'

console.log(barString('apple'));//-apple-

console.log(namedEntity('apple'));//&#apple;

## 讨论

一个函数在给定数目的参数上执行一个过程并返回一个结果集，而另一个函数就充当函数工厂：产生出返回抵押给函数的函数。

## 一个局部工厂函数