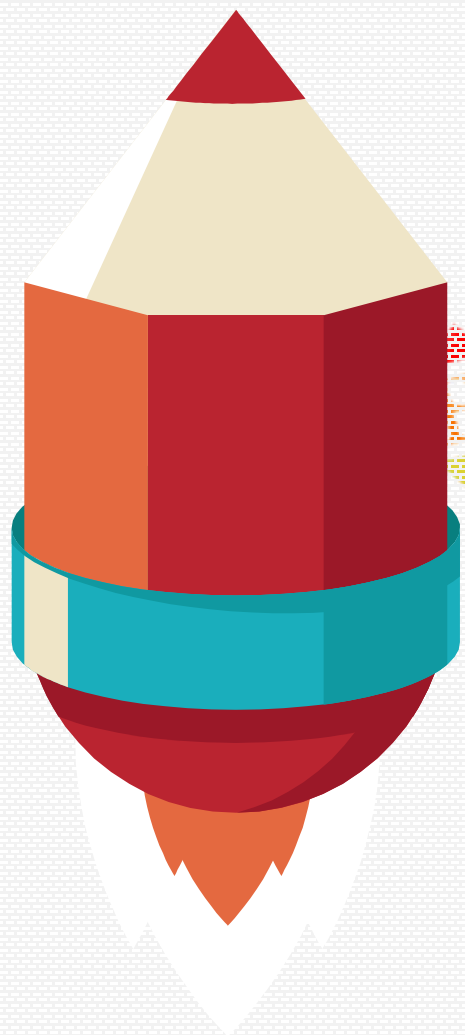




第25课：函数高级

■ 主讲老师：万章



四知

递归函数

闭包与变量

高阶函数

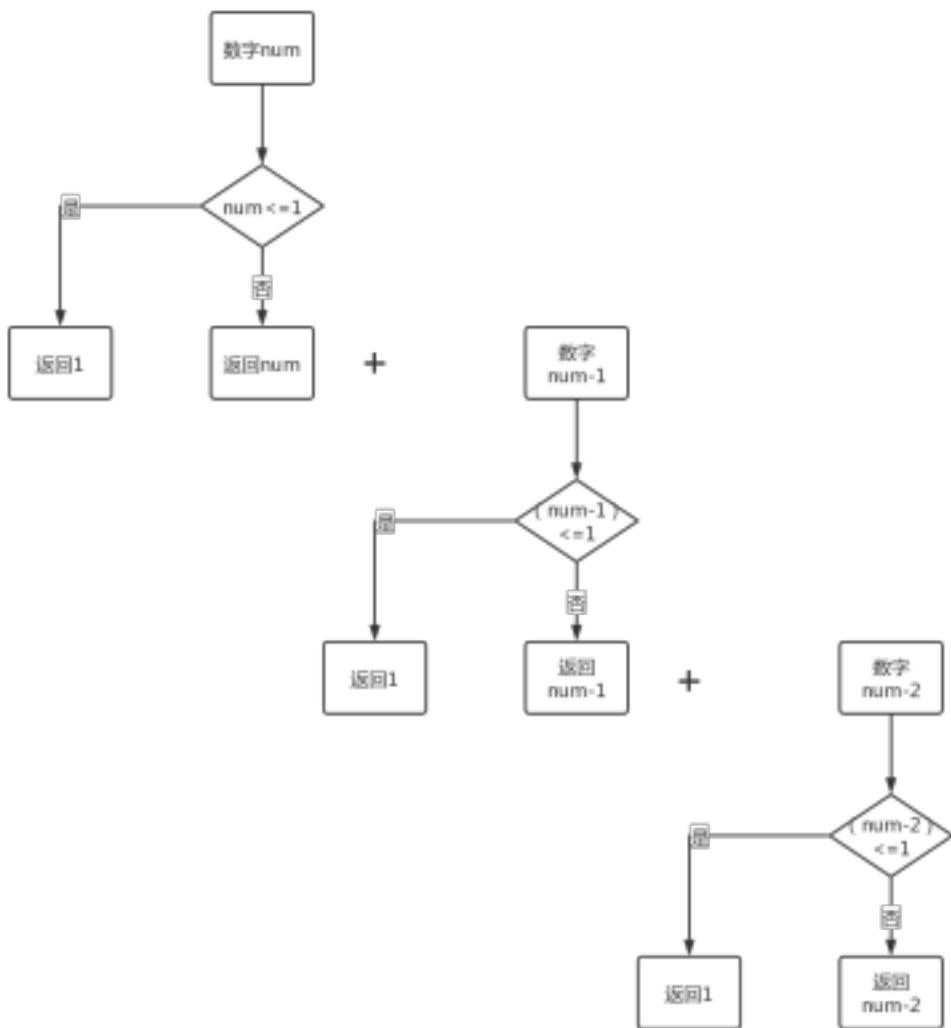
函数柯理化



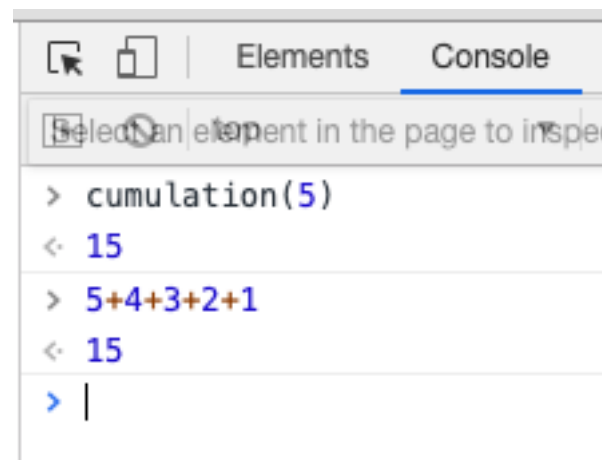
递归函数

递归函数

递归函数是在一个函数通过名字调用自身的情况下构成的



```
function cumulation(num) {//summation
    if (num <= 1) {
        return 1;
    } else {
        return num + cumulation(num - 1);
    }
}
```

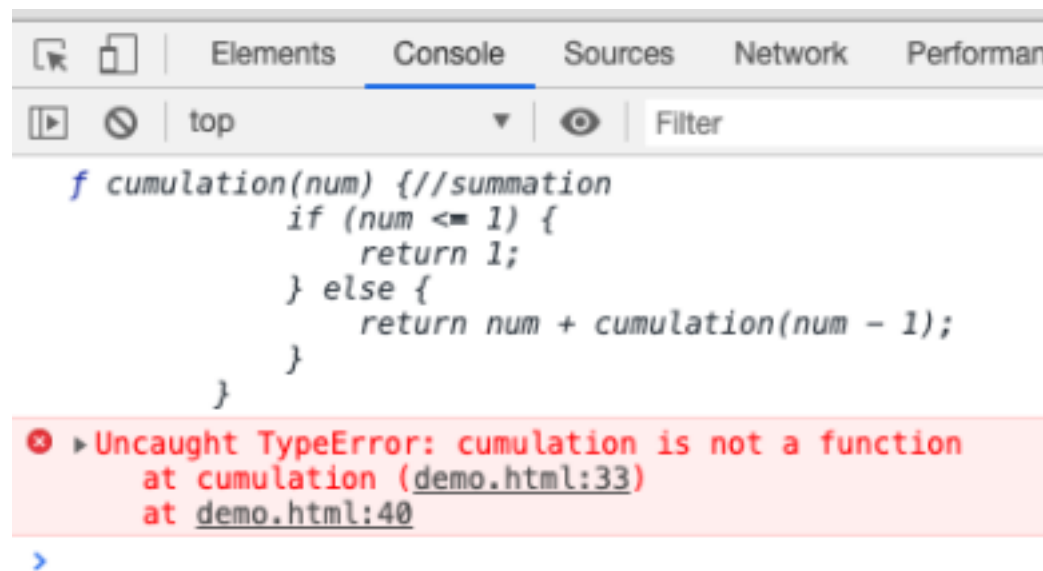


递归函数的问题

这是一个经典的递归累加函数。虽然这个函数表面看来没什么问题，但下面的代码却可能导致它出错。

```
function cumulation(num) {//summation
  if (num <= 1) {
    return 1;
  } else {
    return num + cumulation(num - 1);
  }
}
```

```
let anotherFn=cumulation;
cumulation=null;
console.log(anotherFn);// 输出了函数
anotherFn(5)// 报错
```



道理其实很简单，因为递归函数里面的return语句里面引用的还是cumulation函数，而此时的cumulation函数已经是null了，所以引用自然就报错了。这个问题叫做函数的耦合度高（递归函数与cumulation的名字耦合到一起了）的问

递归函数的问题

自带缓存功能的递归函数，可用来计算阶乘

```
// 计算阶乘，并将结果缓存至函数的属性中
// 这个函数factorial()使用了自身的属性（将自身当做数组来对待）
function factorial(n) {
    if (isFinite(n) && n > 0 && n == Math.round(n)) { //有限的正整数
        if (!(n in factorial)) { //如果没有缓存结果
            factorial[n] = n * factorial(n - 1); //计算结果并缓存之
        }
        return factorial[n]; //返回缓存结果
    } else {
        return NaN;
    } //如
}
factorial[1] = 1; //
```

递归函数的问题

`arguments.callee` 是一个指向正在执行的函数的指针，因此可以用它来实现对函数的递归调用

```
function cumulation(num) {//summation
  if (num <= 1) {
    return 1;
  } else {
    return num + arguments.callee(num - 1);
  }
}
```

```
let anotherFn=cumulation;
cumulation=null;
console.log(anotherFn);// 输出了函数
console.log(anotherFn(5))// 输出15
```

通过使用 `arguments.callee` 代替函数名，可以确保无论怎样调用函数都不会出问题。因此，在编写递归函数时，使用 `arguments.callee` 总比使用函数名更保险。

递归函数的问题

严格模式通过在脚本或函数的头部添加 "use strict"; 表达式来声明
在严格模式下, 不能通过脚本访问 arguments.callee, 访问这个属性会导致错误

```
function cumulation(num) {//summation  
    'use strict';  
  
    if (num <= 1) {  
        return 1;  
    } else {  
        return num + arguments.callee(num - 1);  
    }  
}  
  
let anotherFn=cumulation;  
cumulation=null;  
console.log(anotherFn);// 输出了函数  
console.log(anotherFn(5));// 报错, 函数执行的时候读取到了arguments.callee
```

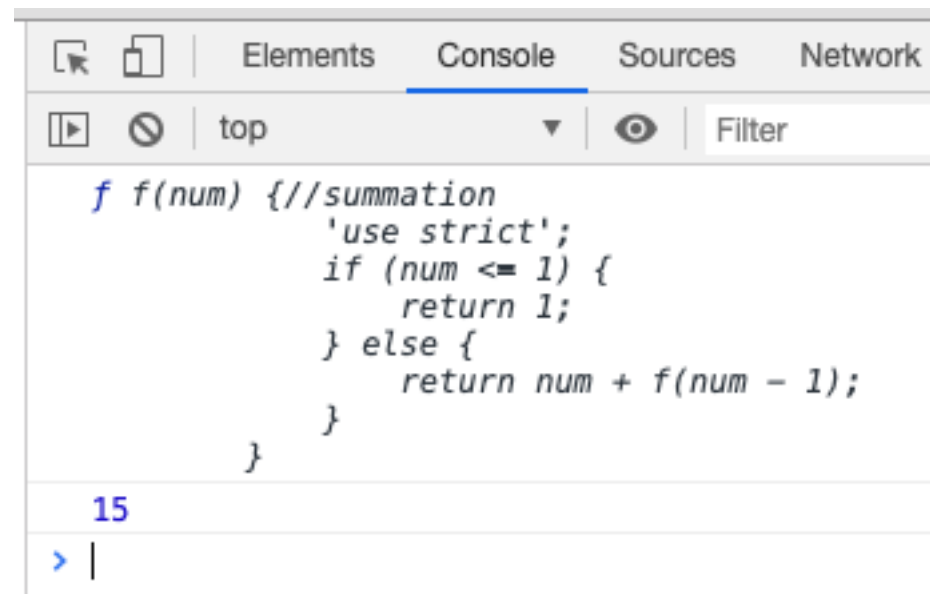
✖ ▶ Uncaught TypeError: 'caller', 'callee', and 'arguments' properties may not be accessed on strict mode functions or the arguments objects for calls to them [demo.html:35](#)
 at cumulation ([demo.html:35](#))
 at [demo.html:42](#)

递归函数的问题

为了在严格模式下达到相同的效果，我们可以使用命名函数表达式来达成相同的结果

```
let cumulation=(function f(num) {//summation
  'use strict';
  if (num <= 1) {
    return 1;
  } else {
    return num + f(num - 1);
  }
})

let anotherFn=cumulation;
cumulation=null;
console.log(anotherFn);// 输出了函数
console.log(anotherFn(5));// 输出
```



The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the function definition for `f` and the result of the function call `f(5)`. The function definition is as follows:

```
f f(num) {//summation
  'use strict';
  if (num <= 1) {
    return 1;
  } else {
    return num + f(num - 1);
  }
}
```

Below the function definition, the console shows the result of the function call `f(5)`, which is `15`. The prompt `> |` is visible at the bottom of the console.



闭包和变量

闭包的详解

有不少开发人员总是搞不清匿名函数和闭包这两个概念，因此经常混用。**闭包是指有权访问另一个函数作用域中的变量的函数。**创建闭包的常见方式，就是在一个函数内部创建另一个函数

```
function createComparisonFunction(propertyName) {  
  
    return function (object1, object2) {  
        var value1 = object1[propertyName];  
        var value2 = object2[propertyName];  
        if (value1 < value2) {  
            return -1;  
        } else if (value1 > value2) {  
            return 1;  
        } else {  
            return 0;  
        }  
    };  
}
```

假设我们现在有一个数组，数组的每一个项目都是对象，那么要比较对象的先后顺序就要选择对象里面的某个属性的值的参数，但是具体比较哪个属性，我们可以让用户自行选择，这个函数就是根据用的选择创建一个比较函数的函数

突出的那两行代码是内部函数（一个匿名函数）中的代码，这两行代码访问了外部函数中的变量 `propertyName`。即使这个内部函数被返回了，而且是在其他地方被调用了，但它仍然可以访问变量 `propertyName`。之所以还能够访问这个变量，是因为内部函数的作用域链中包含 `createComparisonFunction()` 的作用域

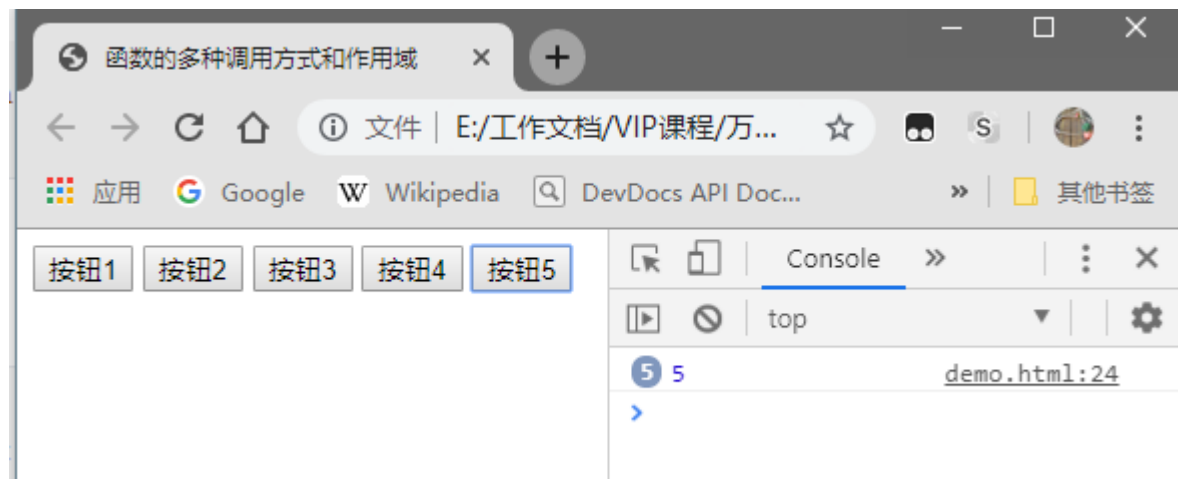
闭包的详解

```
<body>

  <button>按钮1</button>
  <button>按钮2</button>
  <button>按钮3</button>
  <button>按钮4</button>
  <button>按钮5</button>

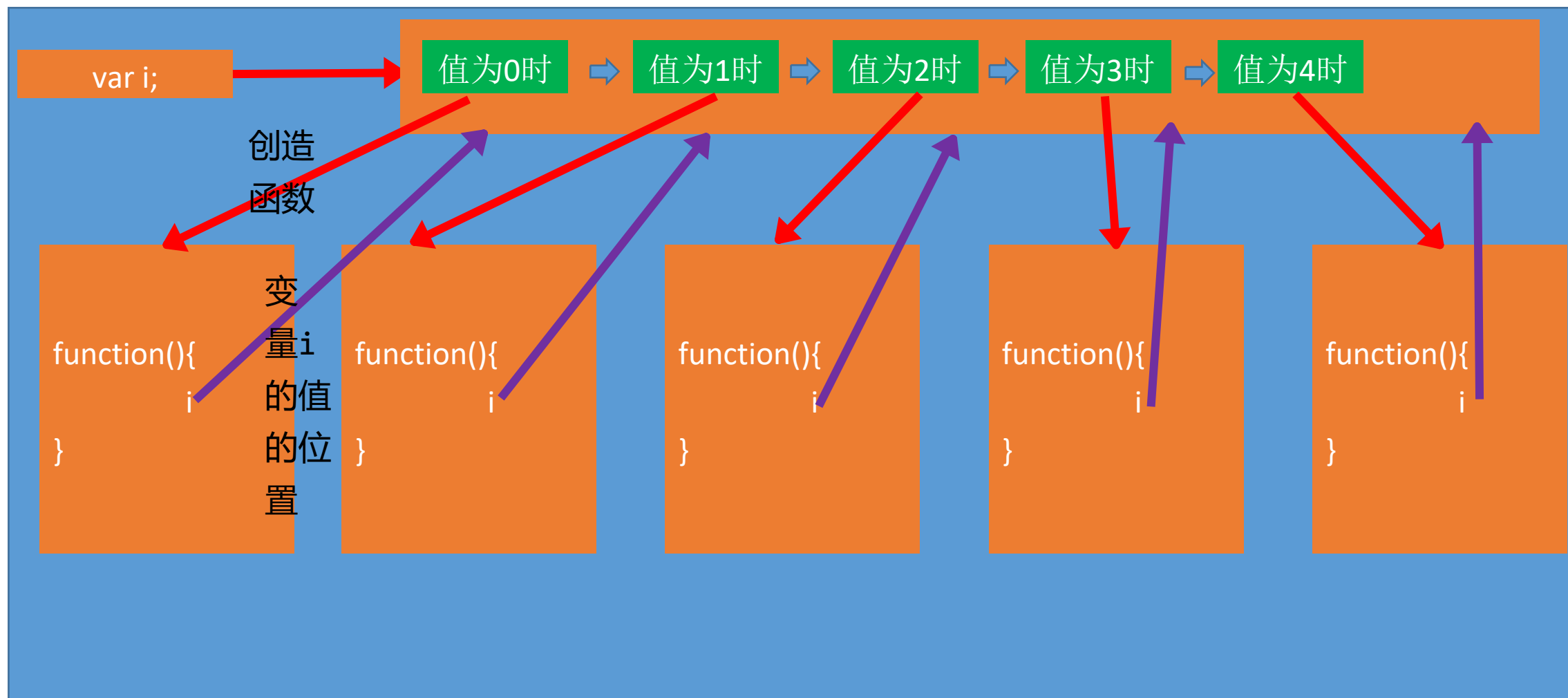
  <script>
    var aBtn=document.querySelectorAll("button");

    for(var i=0;i<5;i++){
      aBtn[i].onclick=function(){
        console.log(i);
      }
    }
  </script>
</body>
```



无论点击哪个按钮，输出的都是5

闭包的详解



所有的函数内部的变量*i*指向的是**全局作用域内**的变量*i*的值

所以当函数执行的时候，就回去寻找此时此刻变量*i*的值，而变量*i*的值在经

历完循环之后数字已经变成了5，所以每一个函数输出的值都是5

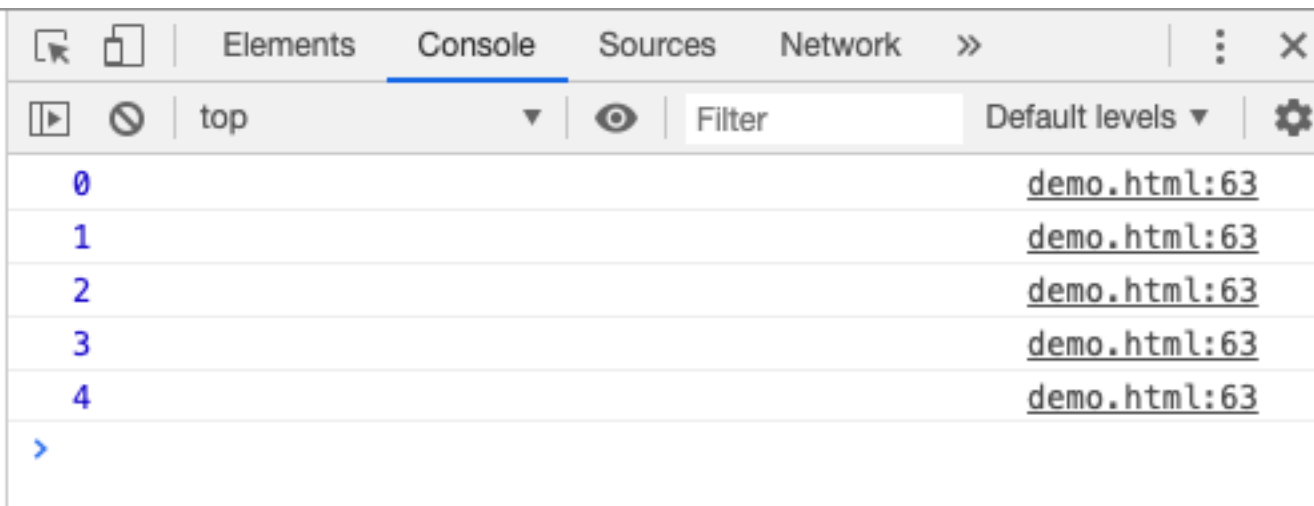
闭包的详解

```
var aBtn=document.querySelectorAll("button");

for(var i=0;i<aBtn.length;i++){
    aBtn[i].onclick=(function(num){
        return function(){
            console.log(num);
        }
    })(i)
}
```

道理相同，利用闭包的原理，将每一次的*i*作为参数值传入到一个立执行的函数中，而这个函数的功能就是创建一个新的函数，新函数的num就是闭包保存的*i*参数的值

按钮1 按钮2 按钮3 按钮4 按钮5





闭包的详解

由于闭包会携带包含它的函数的作用域，因此会比其他函数占用更多的内存。过度使用闭包可能会导致内存占用过多，我们建议读者只在绝对必要时再考虑使用闭包。虽然像 V8 等优化后的 JavaScript 引擎会尝试回收被闭包占用的内存，但请大家还是要慎重使用闭包。

闭包的this变量指向

this 对象是在运行时基于函数的执行环境绑定的：在全局函数中，this 等于 window，而当函数被作为某个对象的方法调用时（call和apply），this 等于那个对象。不过，匿名函数的执行环境具有全局性，因此其 this 对象通常指向 window。但有时候由于编写闭包的方式不同，这一点可能不会那么明显

```
var name = "The Window";
var object = {
  name: "My Object",
  getNameFunc: function () {
    return function () {
      return this.name;
    };
  }
};
console.log(object.getNameFunc()); // 输出"The Window"
```

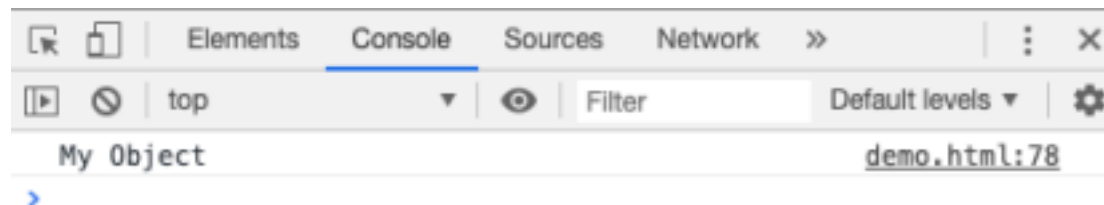
```
> object.getNameFunc()
< f () {
    return this;
}
> |
```

其实简单的说也不难，object的getNameFunc方法的执行结果返回的是一个新函数，这个新函数的执行并不依赖于对象object，而且也没有在闭包里面指定新函数的this的值，所以this的指向那就和普通的全局作用域下的函数一样，就是window对象

闭包的this变量指向

在定义匿名函数之前，我们把 `this` 对象赋值给了一个名叫 `that` 的变量。而在定义了闭包之后，闭包也可以访问这个变量，因为它是我们在包含函数中特意声名的一个变量。即使在函数返回之后，`that` 也仍然引用着 `object`，所以调用 `object.getNameFunc()()` 就返回了 "My Object"。

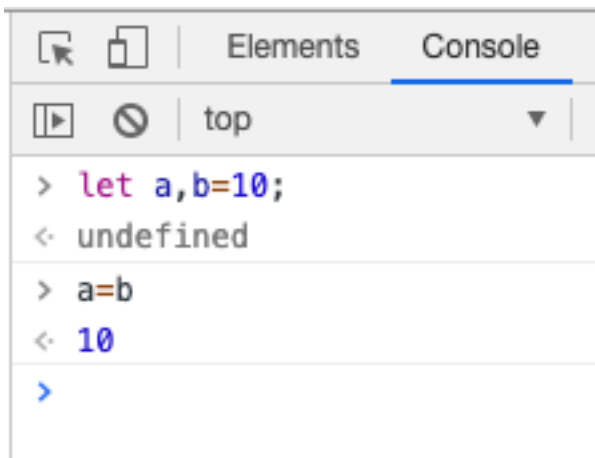
```
var name = "The Window";
var object = {
  name: "My Object",
  getNameFunc: function () {
    let that=this;
    return function () {
      return that.name;
    };
  }
};
console.log(object.getNameFunc()()); // 输出"The Window"
```



闲得蛋疼情况下的this变量指向

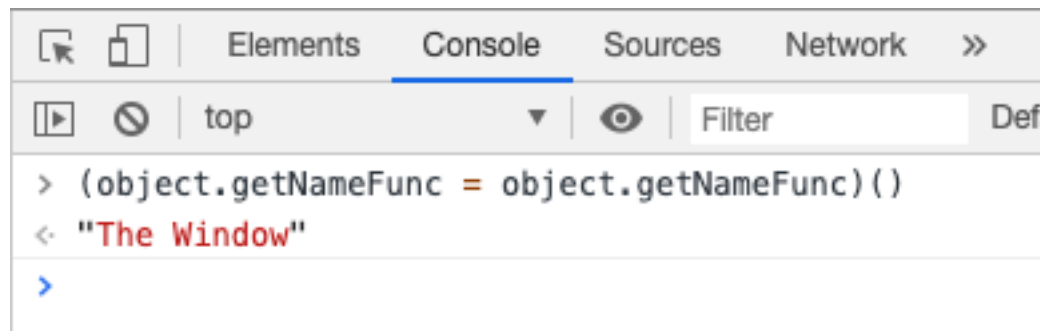
有一些特殊的操作符在操作的时候会默认返回一个值的，比如赋值符号。。。

赋值符号在赋值的过程中会把值本身作为结果输出来



```
> let a,b=10;  
< undefined  
  
> a=b  
< 10  
  
>
```

```
var name = "The Window";  
var object = {  
  name: "My Object",  
  getNameFunc: function () {  
    return this.name;  
  }  
};
```



```
> (object.getNameFunc = object.getNameFunc)()  
< "The Window"  
  
>
```



高级函数

高阶函数

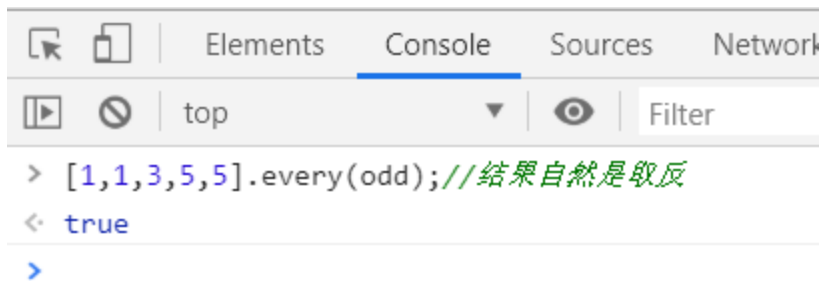
所谓高阶函数 (higher-order function) 就是操作函数的函数, 该函数接受一个或是多个函数作为参数, 并返回一个新的函数

```
function not(f){//传入一个函数
  return function(){//返回一个新函数
    let result=f.apply(this,arguments);
    //在调用返回函数的对象上应用函数f, 参数为调用函数默认参数组, 前三个参数依次是数组项目, 索引, 数组本身
    return !result;//将结果取反
  }
}

let even=function(x){
  return x%2===0;//判断x是否是偶数
}

let odd=not(even);//odd函数的效果正好与even相反

[1,1,3,5,5].every(odd);//结果自然是取反
```



高阶函数

下面的代码展示了一个高阶函数，`memorize()`接收一个函数作为实参，并返回带有记忆能力的函数

```
// 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) {
    var cache = {}; // 将值保存在闭包内
    return function () { // 将实参转换为字符串形式，并将其用做缓存的键
        var key = arguments.length + Array.prototype.join.call(arguments, ",");
        if (key in cache) {
            return cache[key]; // 存在值的时候直接调用值
        } else {
            return cache[key] = f.apply(this, arguments)
            // 存在值的时候直接计算，并把计算最终结果的值存在cache里
        }
    };
};

function fn(n){//经典的阶乘函数
    if(n<=1){
        return 1
    }else{
        return n*factorial(n-1);
    }
}

let factorial=memorize(fn);

factorial(5);
```



函数柯理化

函数柯里化

在 JavaScript 中，函数柯里化是函数式编程的重要思想，也是高阶函数中一个重要的应用，其含义是给函数分步传递参数，每次传递部分参数，并返回一个更具体的函数接收剩下的参数，这中间可嵌套多层这样的接收部分参数的函数，直至返回最后结果

```
// 原函数
function add(a, b, c) {
    return a + b + c;
}
```

```
// 调用原函数
add(1, 2, 3); // 6
```

最基本的柯里化拆分

```
// 柯里化函数
function addCurrying(a) {
    return function (b) {
        return function (c) {
            return a + b + c;
        }
    }
}
```

```
// 调用柯里化函数
addCurrying(1)(2)(3) // 6
```

被柯里化的函数 `addCurrying` 每次的返回值都为函数，并使用下一个参数作为形参，直到三个参数都被传入后，返回的最后一个函数内部执行求和操作，其实是充分的利用了闭包的特性来实现的。

函数柯里化

上面的柯里化函数没涉及到高阶函数，也不具备通用性，无法转换形参个数任意或未知的函数，我们接下来封装一个通用的柯里化转换函数，可以将任意函数转换成柯里化。

```
function currying(func, args) {  
    var arity = func.length; // 形参个数，即func需要正常运行所需要的参数个数  
    var args = args || [];  
    // 在柯里化函数的时候是否传入了其他参数  
    // 如果传入了就保存改参数, 没传就把参数初始化一个空数据  
  
    return function () {  
        // 将参数转化为数组  
        var _args = [].slice.call(arguments);  
  
        // 将上次的参数与当前参数进行组合并修正传参顺序  
        Array.prototype.unshift.apply(_args, args);  
        // 即把args的数组项目加入到_args数组的前面  
  
        // 如果参数不够，返回新的闭包函数继续收集参数  
        if (_args.length < arity) {  
            return currying.call(null, func, _args);  
        }  
  
        // 参数够了则直接执行被转化的函数  
        return func.apply(null, _args);  
        // 之所以要用apply是因为现在的参数是一个数组，而函数的执行需要把参数一个个的传入到函数  
        // 为了简便，我们用apply的方式直接来实现  
    }  
}
```


函数柯里化

柯里化的一个很大的好处是可以帮助我们基于一个被转换函数，通过对参数的拆分实现不同功能的函数

```
// 柯里化通用式应用 — 普通函数
// 被转换函数，用于检测传入的字符串是否符合正则表达式
function checkFun(reg, str) {
  return reg.test(str);
}

// 转换柯里化
const check = currying(checkFun);

// 产生新的功能函数
const checkPhone = check(/^1[34578]\d{9}$/);
const checkEmail = check(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/);
```

```
// 一个普通函数
function loc(a, b, c) {
  console.log(a + b + c);
}

let hasA = currying(loc, [10]);

hasA(10)(10); //30

let hasAB = currying(loc, [10, 10]);
hasAB(10); //30
```