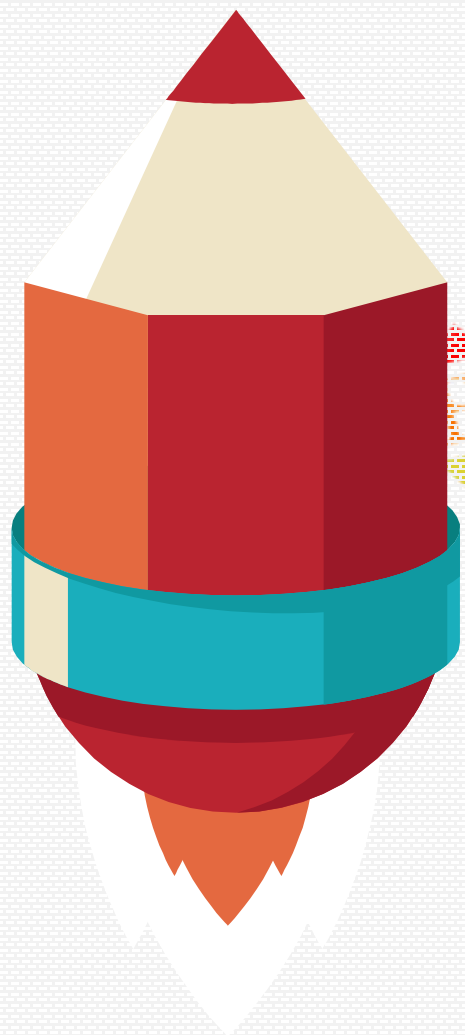




第11课：多种变量的声明 方式let,const

■ 主讲老师：万章



目录

var的变量提升

函数的变量提升

let,const指令详解

JavaScript垃圾回收

函数的闭包原理



var的变量提升

var的变量提升

var 指令的作用就是在当前作用域下声明了一个变量

作用域:就是按照名称查找变量的一套规则/一个空间

```
var a=10;  
console.log(a); //输出数字 10
```

```
console.log(a);  
//报错，浏览器指示变量a未声明
```

```
> console.log(a)  
Uncaught ReferenceError: a is not defined VM100:1  
    at <anonymous>:1:13  
>
```

```
var a;  
console.log(a); //输出undefined  
a=10;  
console.log(a); //输出数字10
```

var的变量提升

无论在哪个作用域中, var指令都会被提前到该作用域代码的最前

```
console.log(a); // 输出undefined  
var a=10;  
console.log(a); // 输出数字10
```

指令1: 输出变量a

指令2: 创造变量a并把数字10
赋值给变量a

指令3: 输出变量a

指令
拆分

```
console.log(a); // 输出undefined  
var a;  
a=10;  
console.log(a); // 输出数字10
```

指令1: 输出变量a

指令2.1: 创造变量a
指令2.2: 把数字10赋值给变量a

指令3: 输出变量a

指令
提升

```
var a;  
console.log(a); // 输出undefined  
a=10;  
console.log(a); // 输出数字10
```

指令2.1: 创造变量a

指令1: 输出变量a

指令2.2: 把数字10赋值给变量a

指令3: 输出变量a



函数的变量提升

函数声明的变量提升

无论在哪个作用域中, 函数声明指令也都会被提前到该作用域代码的最前, 函数声明指令会被提前到var指令之前

```
console.log(a); //输出undefined
console.log(add); //输出函数add的内容
add(); //在控制台输出10
function add(){
    console.log(9527);
}
var a=10;
```

指令1: 输出变量a

指令2: 输出变量add

指令3: 执行变量add内存存储的函数

指令4: 声明变量add, 并给变量add存储一个函数

指令5: 创造变量a并把数字10赋值给变量a

指令
拆分

```
console.log(a); //输出undefined
console.log(add); //输出函数add的内容
add(); //在控制台输出10
function add(){
    console.log(9527);
}
var a;
a=10;
```

指令1: 输出变量a

指令2: 输出变量add

指令3: 执行变量add内存存储的函数

指令4: 声明变量add, 并给变量add存储一个函数

指令5.1: 创造变量a
指令5.2: 把数字10赋值给变量a

指令
提升

```
function add(){
    console.log(9527);
}
var a;
console.log(a); //输出undefined
console.log(add); //输出函数add的内容
add(); //在控制台输出10
a=10;
```

指令4: 声明变量add, 并给变量add存储一个函数

指令5.1: 创造变量a

指令1: 输出变量a

指令2: 输出变量add

指令3: 执行变量add内存存储的函数

指令5.2: 把数字10赋值给变量a

函数声明的变量提升

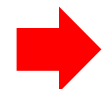
函数声明指令会被提前到var指令之前

```
add( );//输出9527
function add(){
    console.log(9527);
}
var add=10;
console.log(add);//输出10
```



指令
拆分

```
add( );//输出9527
function add(){
    console.log(9527);
}
var add;
add=10;
console.log(add);//输出10
```



指令
提升

```
function add(){
    console.log(9527);
}
var add;
add( );//输出9527
add=10;
console.log(add);//输出10
```

重复的var声明会被忽略

因为var指令的作用就是新建一个变量/地址, 那么我们既然已经有了这个地址, 那么就直接拿来用就行

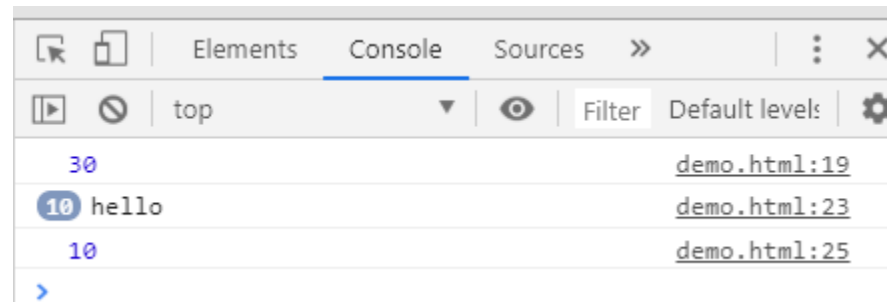


let,const指令详解

let指令的块级作用域

```
var a=10;
var b=20;
if(a<b){
    var c=30;
}
console.log(c);

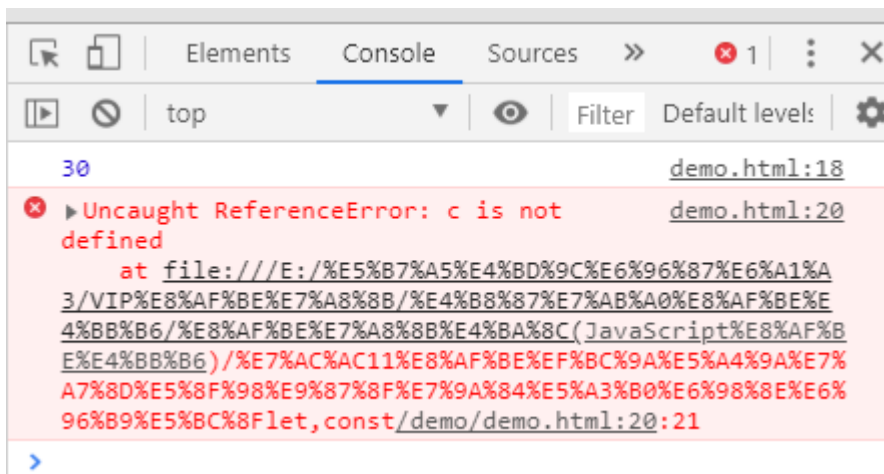
for(var i=0;i<10;i++){
    console.log("hello");
}
console.log(i)
```



var指令创造的变量只有函数作用域, 在控制流程内不会产生作用域, 外面也可以访问到控制流程内部的变量

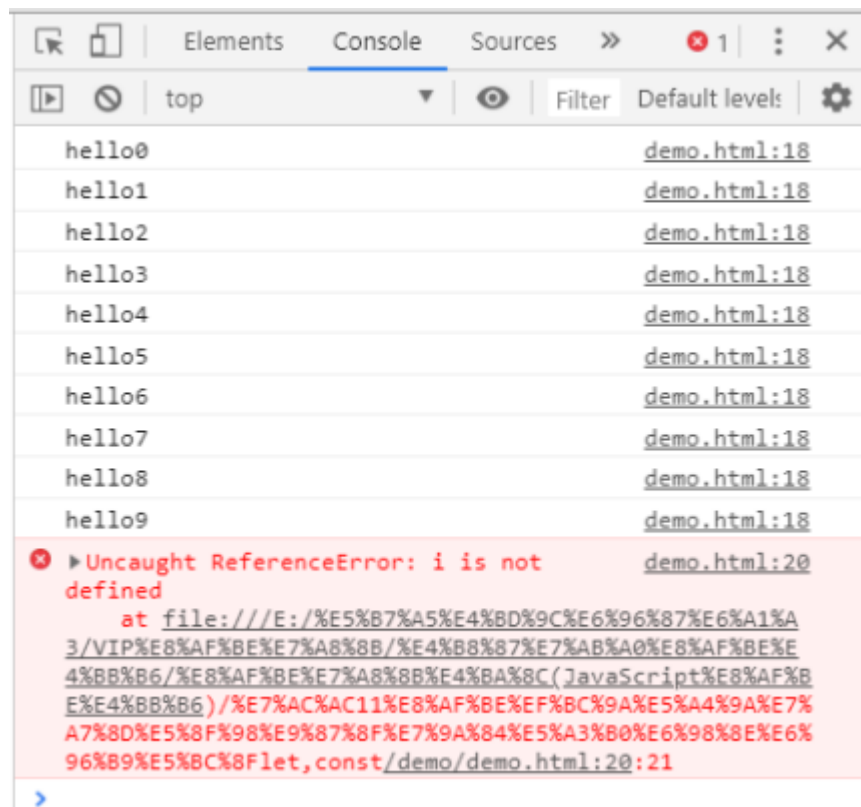
let指令的块级作用域

```
var a=10;
var b=20;
if(a<b){
  let c=30;
  console.log(c)
}
console.log(c);
```

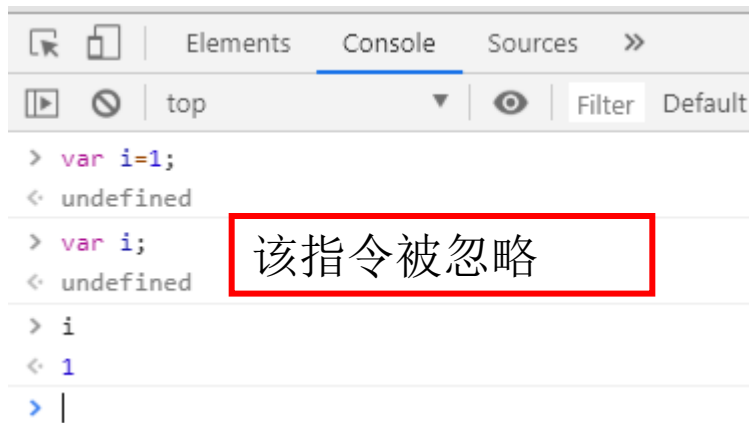


```
for(let i=0;i<10;i++){
  console.log("hello"+i);
}
console.log(i)
```

使用let指令创建的变量, 存在块级作用域, 控制流程内部用let指令定义的变量, 在控制流程外部**无法获取**



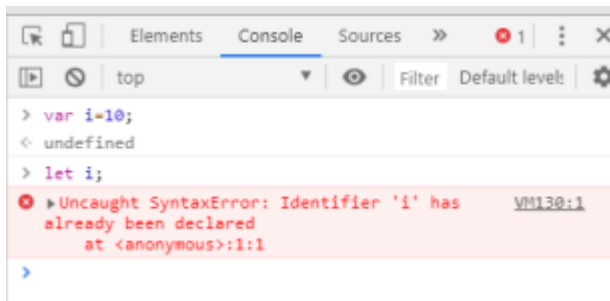
let指令的重复声明问题



```
> var i=1;
< undefined
> var i;
< undefined
> i
< 1
> |
```

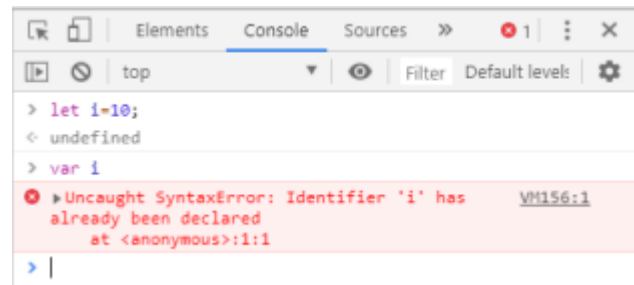
该指令被忽略

如果我们已经使用var指令声明了一个变量, 那么下方如果出现了一个var指令声明了同一个名称的变量, 那么下方的var指令会被忽略

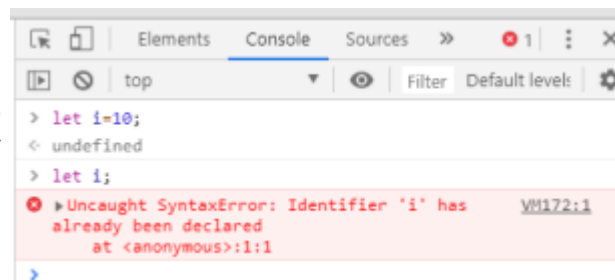


```
> var i=10;
< undefined
> let i;
Uncaught SyntaxError: Identifier 'i' has already been declared
    at <anonymous>:1:1
```

报错信息的意思为:
变量i已经被声明过



```
> let i=10;
< undefined
> var i;
Uncaught SyntaxError: Identifier 'i' has already been declared
    at <anonymous>:1:1
```



```
> let i=10;
< undefined
> let i;
Uncaught SyntaxError: Identifier 'i' has already been declared
    at <anonymous>:1:1
```

如果某个变量已经用let指令声明, 那么就不能再次使用let或是var声明;
如果某个变量已经用var指令声明, 那么就不能再次使用let声明;

let指令的临时死区

var
的
变
量
提
升

```
console.log(a); //输出undefined
var a=10;
console.log(a); //输出数字10
```

指令1: 输出变量a

指令2: 创造变量a并把数字10赋值给变量a

指令3: 输出变量a

指令
拆分

```
console.log(a); //输出undefined
var a;
a=10;
console.log(a); //输出数字10
```

指令1: 输出变量a

指令2.1: 创造变量a
指令2.2: 把数字10赋值给变量a

指令3: 输出变量a

指令
提升

```
var a;
console.log(a); //输出undefined
a=10;
console.log(a); //输出数字10
```

指令2.1: 创造变量a

指令1: 输出变量a

指令2.2: 把数字10赋值给变量a

指令3: 输出变量a

let
的
临
时
死
区

```
console.log(b);
let b=10;
// Uncaught ReferenceError: Cannot access 'b' before initialization
// at <anonymous>:1:13
```

let指令不会提升!!!

指令1: 输出变量b

指令2: 创造变量b并把数字10赋值给变量a

指令
执行

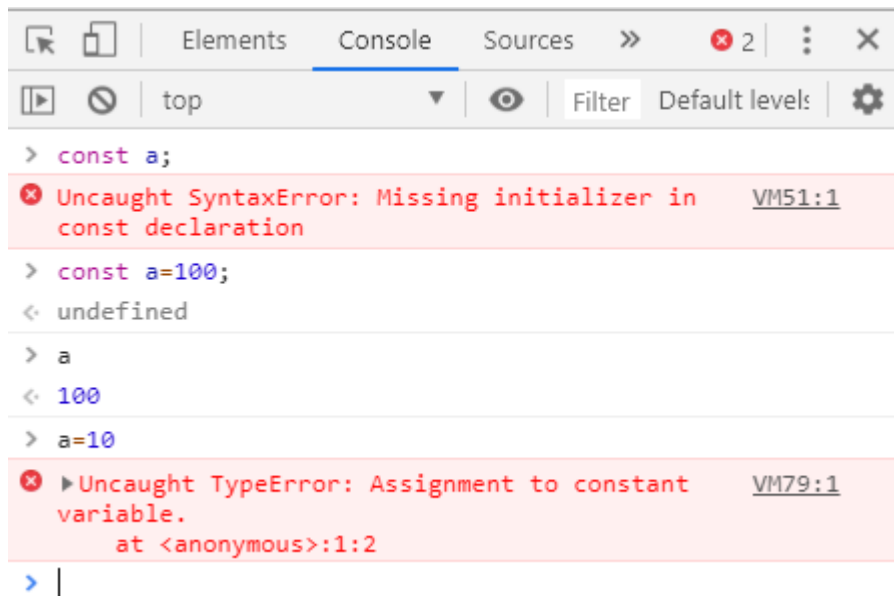
没有在作用域里面找到变量b
报错
(在没有初始化(声明)变量b之前,
不可以访问变量b的数据)

let指令的临时死区

因为let指令没有变量提升
所以在浏览器在执行到let指令
之前, 任何语句中都无法使用
该let指令所定义的变量

```
<script>  
  
    console.log(a); //Uncaught ReferenceError: a is not defined  
    let a;  
    a = 3;  
  
</script>
```

const指令(定义常量)



The screenshot shows a browser's developer console with the 'Console' tab selected. It displays two error messages related to the `const` keyword. The first error is a 'SyntaxError: Missing initializer in const declaration' at VM51:1, which occurs after the command `> const a;`. The second error is a 'TypeError: Assignment to constant variable.' at VM79:1, which occurs after the command `> a=10`. The console also shows the output of `> const a=100;` as `< undefined` and `> a` as `< 100`.

```
> const a;  
✖ Uncaught SyntaxError: Missing initializer in  
  const declaration VM51:1  
  
> const a=100;  
< undefined  
  
> a  
< 100  
  
> a=10  
✖ ▶ Uncaught TypeError: Assignment to constant  
  variable. VM79:1  
    at <anonymous>:1:2  
  
> |
```

`const`声明和`let`非常类似, 区别就是

1:`const`定义好一个值之后, 不允许修改变量值;

2:初始化的时候就要赋值;

3:常量变量名请大写(只是规范而已, 小写也不会报错, 但是大写了可以让程序员快速识别该变量是啥);

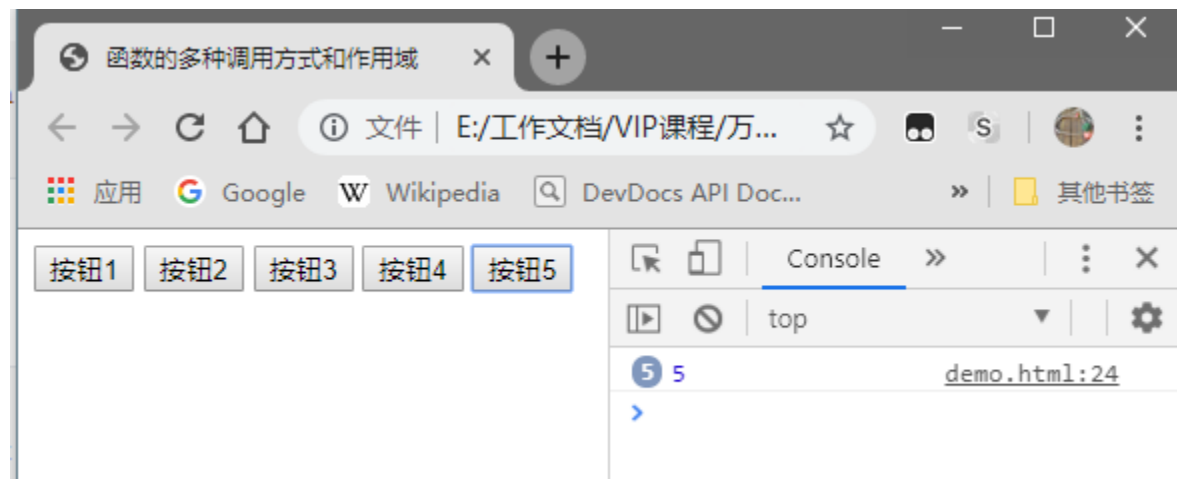
经典案例

```
<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

经典案例

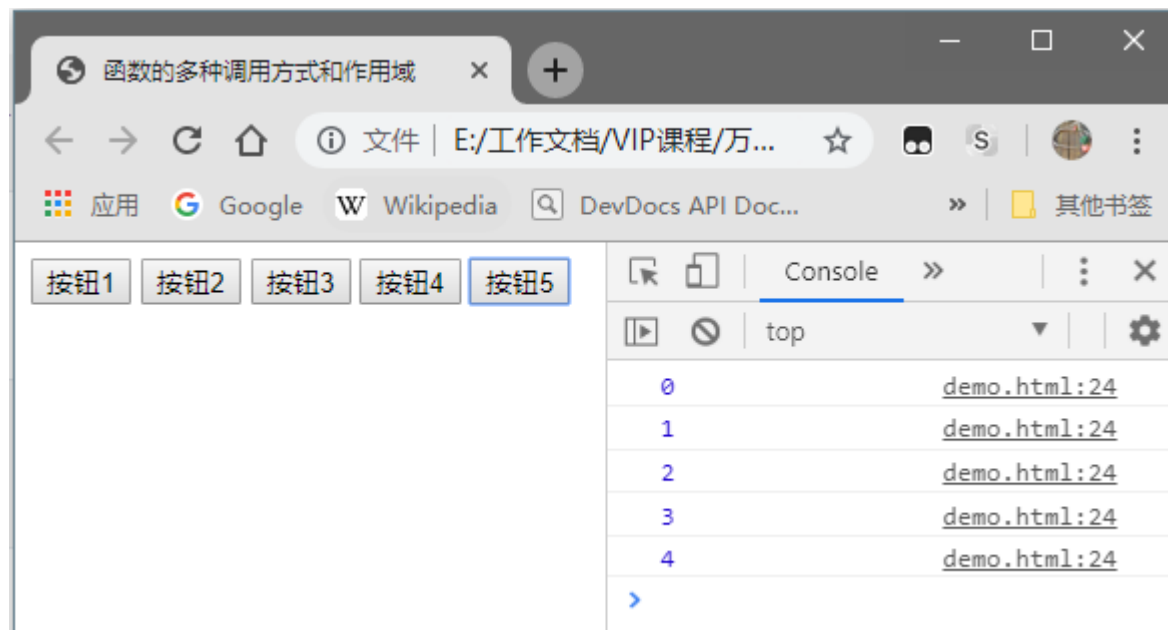
```
<body>

<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");

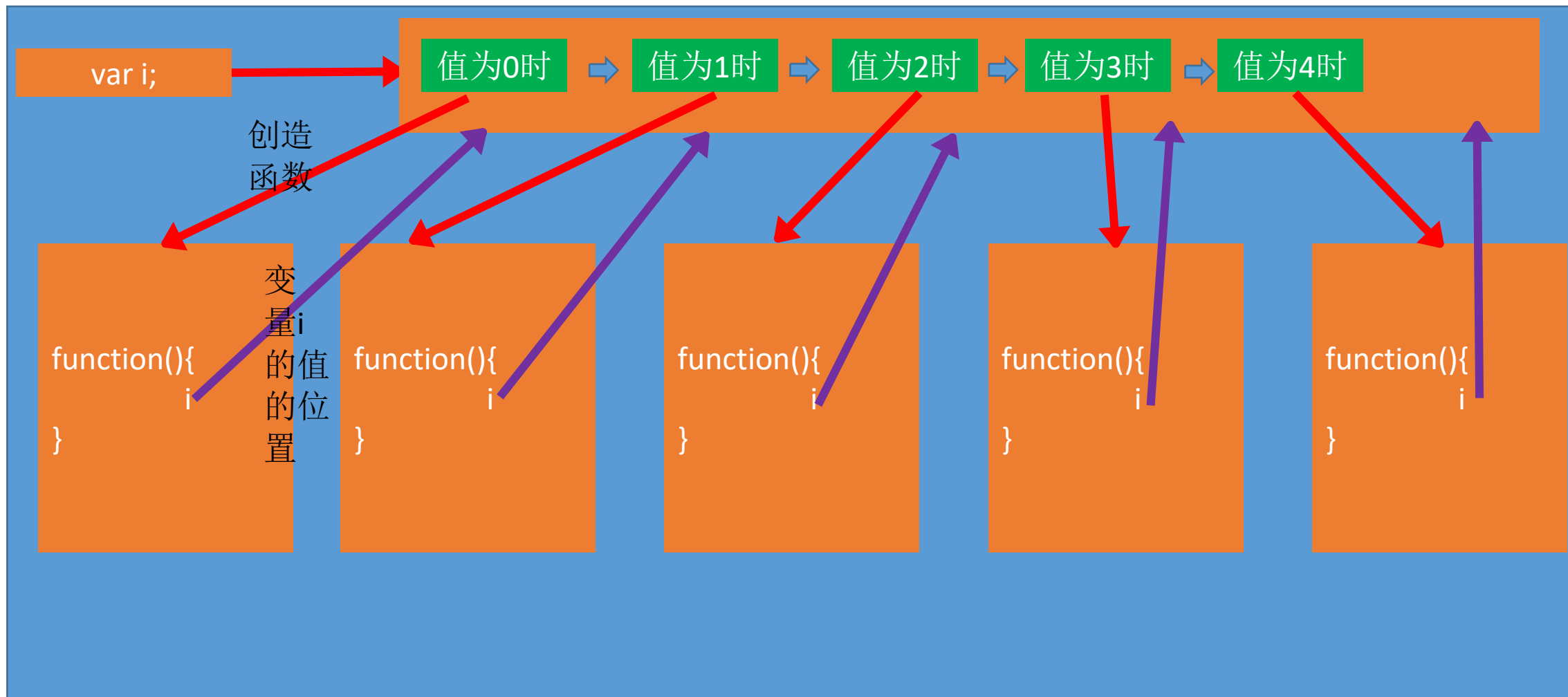
  for(let i=0;i<5;i++){
    aBtn[i].onclick=function(){
      console.log(i);
    }
  }
</script>
</body>
```

把for循环中的var改成let



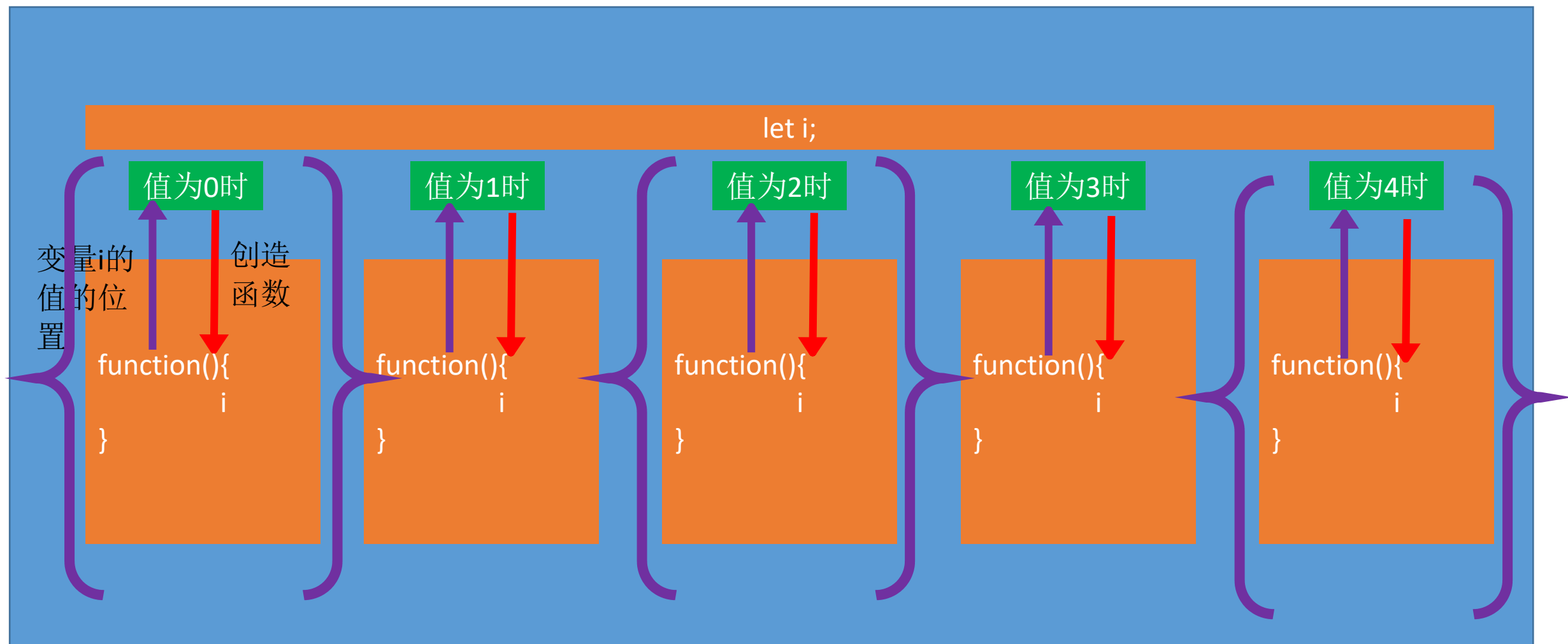
点击按钮1到按钮5, 分别输出0~4

经典案例原理解析 var



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

经典案例原理解析 let



所有的函数内部的变量*i*指向的是**局部块级作用域**内的变量*i*的值
所以for循环执行一次就相当于新建了一个块级作用域, 每个块级作用域内都有一个*i*的值, 而且此值是完全独立的, 所以函数在执行时所用到的*i*的值都是函数在定义时的*i*的值

经典案例原理解析 同理

```
<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");
    let i=0;
    for(;i<5;i++){
      aBtn[i].onclick=function(){
        console.log(i);//点击每个按钮输出的都是5
      }
    }
  </script>
</body>
```

因为此时变量*i*是在全局作用域下定义的所以效果和在for循环内部用var定义变量完全一致, 所有的函数输出的都是5



函数的闭包原理

标记清除(mark-and-sweep)

JavaScript最常用的是标记清除：当变量进入环境(作用域)，则将变量标记为进入环境，当变量离开环境的时候，将其标记为离开环境。

垃圾收集器在运行时会给存储在内存中所有变量标记，然后去掉环境变量与被环境变量引用的变量，剩下的就是环境无法访问的变量，这些变量以及其占用的内容空间将被清理回收。

引用计数(reference counting)

不太常见的垃圾回收策略：引用计数。

跟踪每一个值得引用次数。当声明一个引用并将一个引用类型赋值给这个变量得时候，这个值得引用计数加1，如果又把这个引用给了第三个变量，那么引用计数又加1，变成2。如果有变量得引用被指向了别的值，那么引用计数减1，直到等于0。意味着这个值已经不会被变量引用，垃圾收集器下次运行得时候就会清理引用为0得值所占据得内存。

如果一个值得引用出现闭环得话，这个值得引用不会变为0，循环引用使得值得内存永远得不到回收。意味着永远占用内存。

```
var o1={};  
var o2={};  
o1.x=o2;  
o2.x=o1;
```

引用闭环

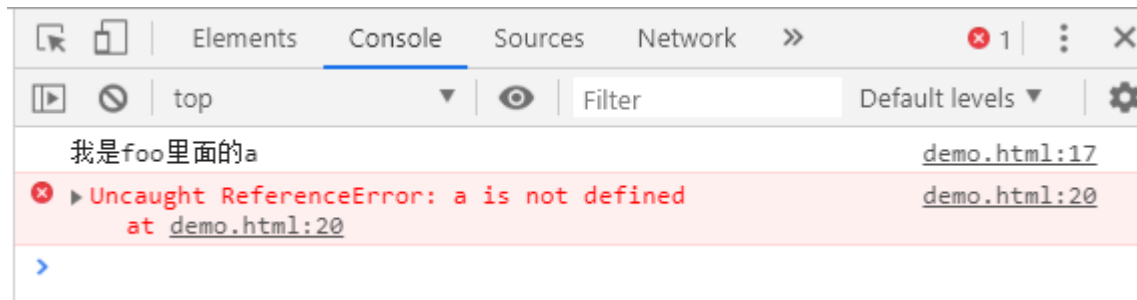
所以：对于不需要得值，或者使用完的值，请手动将变量得指向指向清除。



函数的闭包原理

函数的闭包原理

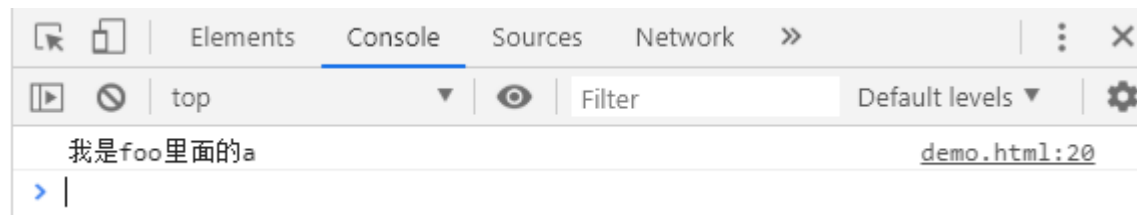
```
<script>
  function foo() {
    var a = "我是foo里面的a";
    console.log(a);
  }
  foo();
  console.log(a);
</script>
```



函数内部的变量在函数外部是无法获取的, 但是如果我们要获得函数内部的某个参数或是变量的值的话, 我们可以用return的方法来实现

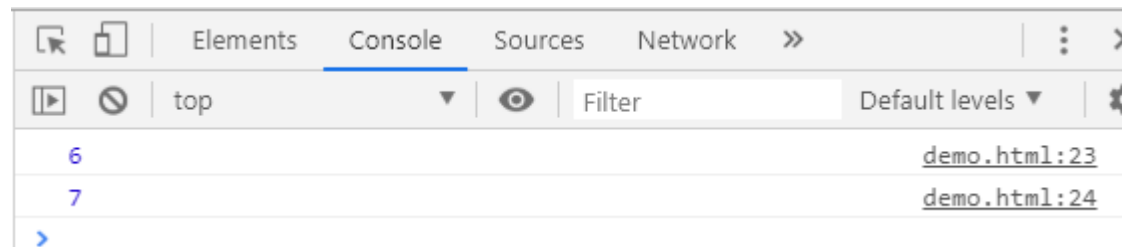
```
<script>

  function foo() {
    var a = "我是foo里面的a";
    return a; // 将变量a给return出来
  }
  let b = foo(); // 此时b就获得foo里面的a了。
  console.log(b); "我是foo里面的a";
</script>
```



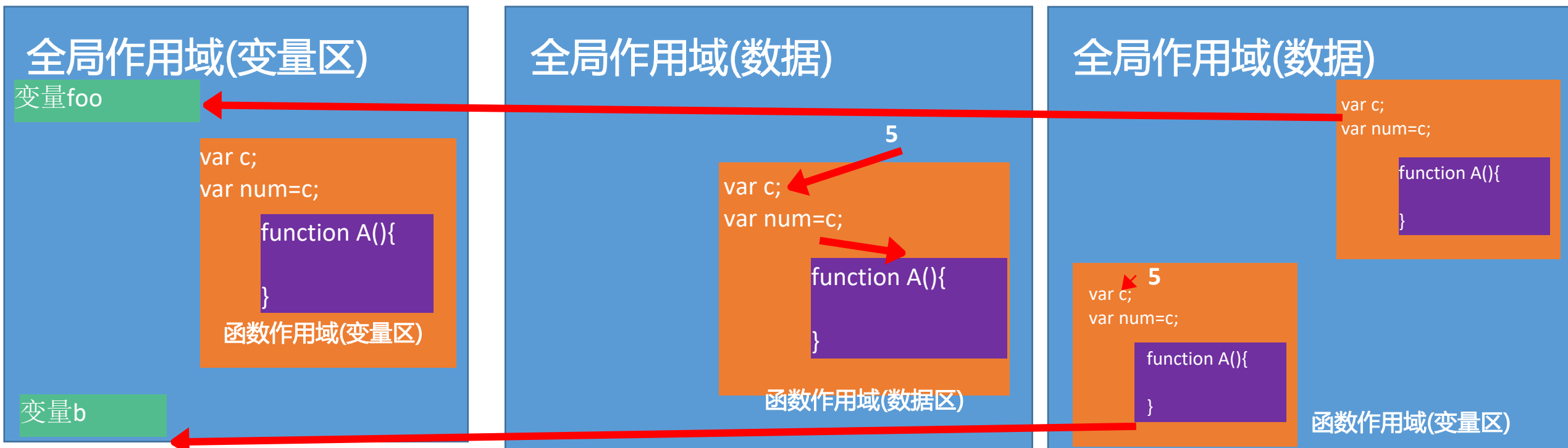
函数的闭包原理

```
<script>
  function foo(c) {
    var num = c;
    return function A() {
      num++;
      return num;
    }
  }
  var b = foo(5); //b是什么? 他保留了什么?
  console.log(b()); //6
  console.log(b()); //7
</script>
```



函数可以通过作用域链互相关联起来, 函数体内部的变量都可以保存在函数作用域内, 这种特性被称为闭包

函数的闭包原理(保证参数被引用, 避免垃圾回收)



函数执行时建立作用域

函数执行时, 传输数据

函数执行后, 垃圾回收
因为参数num被函数A引用了, 所以num的数据不会被回收, 下一次, b
函数执行时还能访问到变量num