举个例子：

<script>

var scope = "global";

function fn1(){

return scope;

}

function fn2(){

return scope;

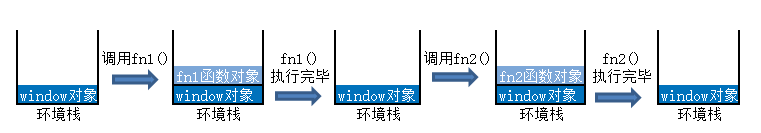
}

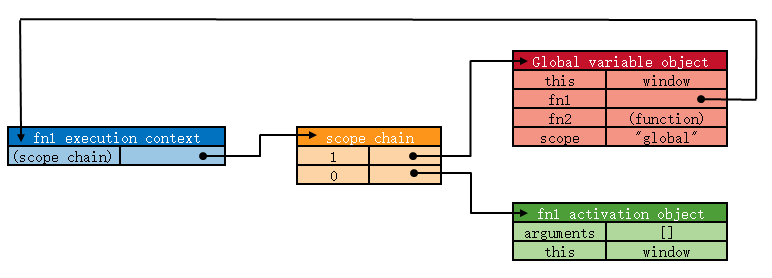
fn1();

fn2();

</script>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

上面代码执行情况演示：  


了解了环境变量，再详细讲讲作用域链。  
当某个函数第一次被调用时，就会创建一个执行环境(execution context)以及相应的作用域链，并把作用域链赋值给一个特殊的内部属性([scope])。然后使用this，arguments(arguments在全局环境中不存在)和其他命名参数的值来初始化函数的活动对象(activation object)。当前执行环境的变量对象始终在作用域链的第0位。  
以上面的代码为例，当第一次调用fn1()时的作用域链如下图所示：  
（因为fn2()还没有被调用，所以没有fn2的执行环境）  
  
可以看到fn1活动对象里并没有scope变量，于是沿着作用域链(scope chain)向后寻找，结果在全局变量对象里找到了scope，所以就返回全局变量对象里的scope值。

标识符解析是沿着作用域链一级一级地搜索标识符地过程。搜索过程始终从作用域链地前端开始，然后逐级向后回溯，直到找到标识符为止（如果找不到标识符，通常会导致错误发生）—-《JavaScript高级程序设计》

那作用域链地作用仅仅只是为了搜索标识符吗？  
再来看一段代码：

<script>

function outer(){

var scope = "outer";

function inner(){

return scope;

}

return inner;

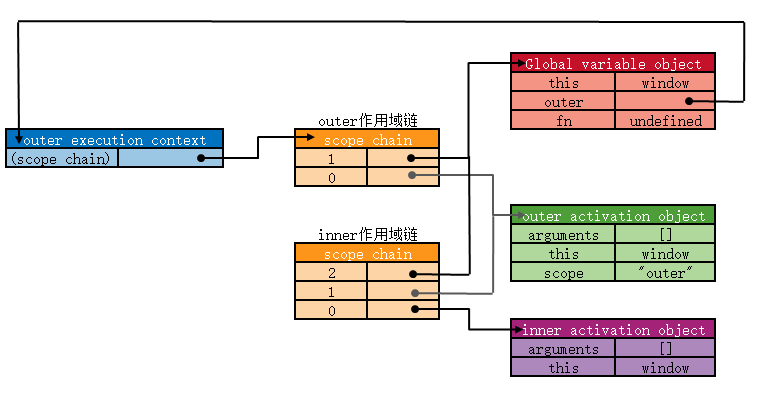
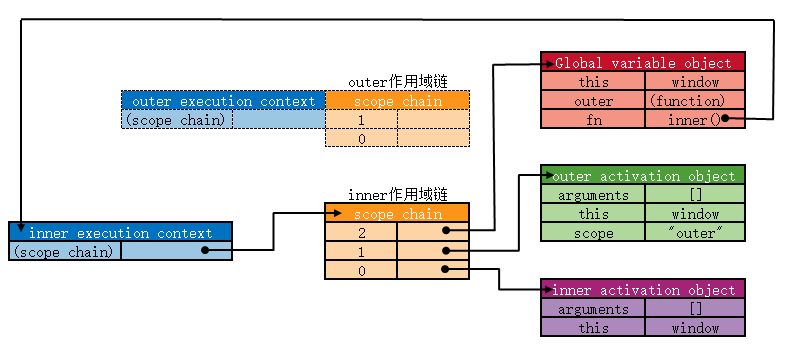
}

var fn = outer();

fn();

</script>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

outer()内部返回了一个inner函数，当调用outer时，inner函数的作用域链就已经被初始化了（复制父函数的作用域链，再在前端插入自己的活动对象），具体如下图：  
  
一般来说，当某个环境中的所有代码执行完毕后，该环境被销毁（弹出环境栈），保存在其中的所有变量和函数也随之销毁（全局执行环境变量直到应用程序退出，如网页关闭才会被销毁）  
但是像上面那种有内部函数的又有所不同，当outer()函数执行结束，执行环境被销毁，但是其关联的活动对象并没有随之销毁，而是一直存在于内存中，因为该活动对象被其内部函数的作用域链所引用。  
具体如下图：  
outer执行结束，内部函数开始被调用  
outer执行环境等待被回收，outer的作用域链对全局变量对象和outer的活动对象引用都断了  


像上面这种内部函数的作用域链仍然保持着对父函数活动对象的引用，就是****闭包(closure)****