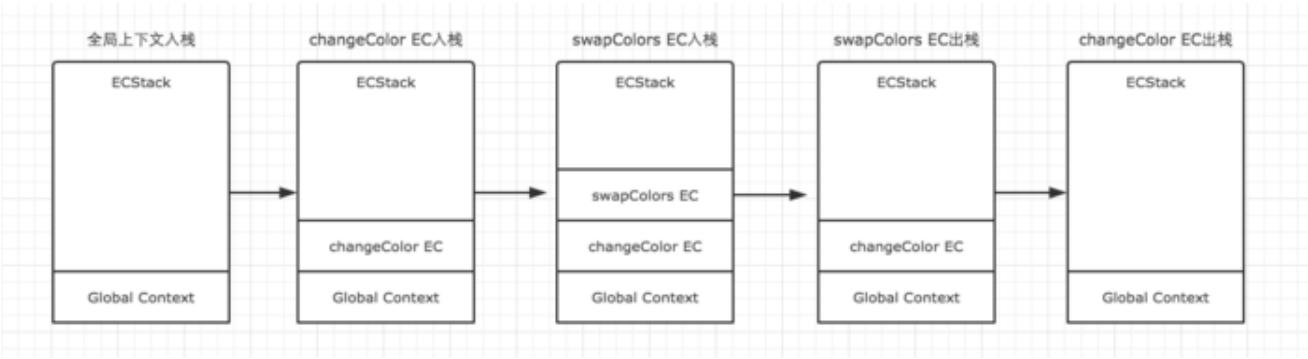
执行上下文




```
console.log(a);   // 这里会打印出什么？
var a = 20;
```

暂时先不管这个例子，我们先引入一个`JavaScript`中最基础，但同时也是最重要的一个概念**执行上下文**（Execution Context）。

> 执行上下文可以理解为函数执行的环境，每一个函数执行时，都会给对应的函数创建这样一个执行环境。

每次当控制器转到可执行代码的时候，就会进入一个执行上下文。执行上下文可以理解为当前代码的执行环境，它会形成一个作用域。`JavaScript`中的运行环境大概包括三种情况。

- 全局环境：`JavaScript`代码运行起来会首先进入该环境
- 函数环境：当函数被调用执行时，会进入当前函数中执行代码
- `eval`（不建议使用，可忽略）

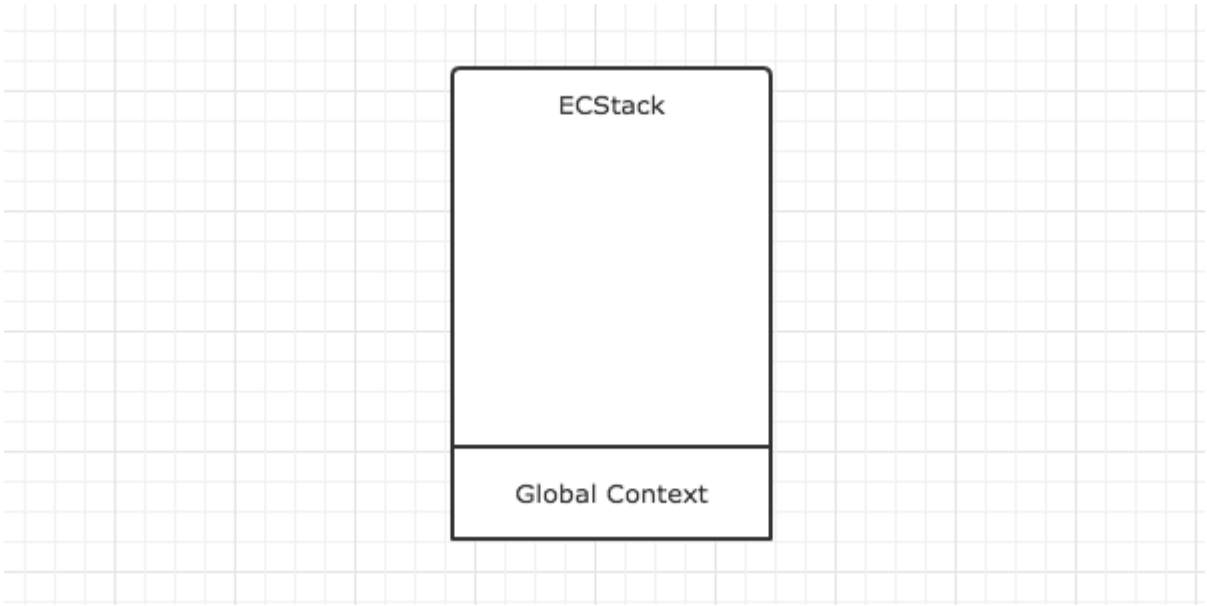
在一个`JavaScript`程序中，必定会产生多个执行上下文，`JavaScript`引擎会以栈的方式来处理它们，这个栈，我们称其为函数调用栈(`call stack`)。栈底永远都是全局上下文，而栈顶就是当前正在执行的上下文。

当代码在执行过程中，遇到以上三种情况，都会生成一个执行上下文，放入栈中，而处于栈顶的上下文执行完毕之后，就会自动出栈。

```
var color = 'blue';

function changeColor() {
    var anotherColor = 'red';

    function swapColors() {
        var tempColor = anotherColor;
        anotherColor = color;
        color = tempColor;
    }

    swapColors();
}
```

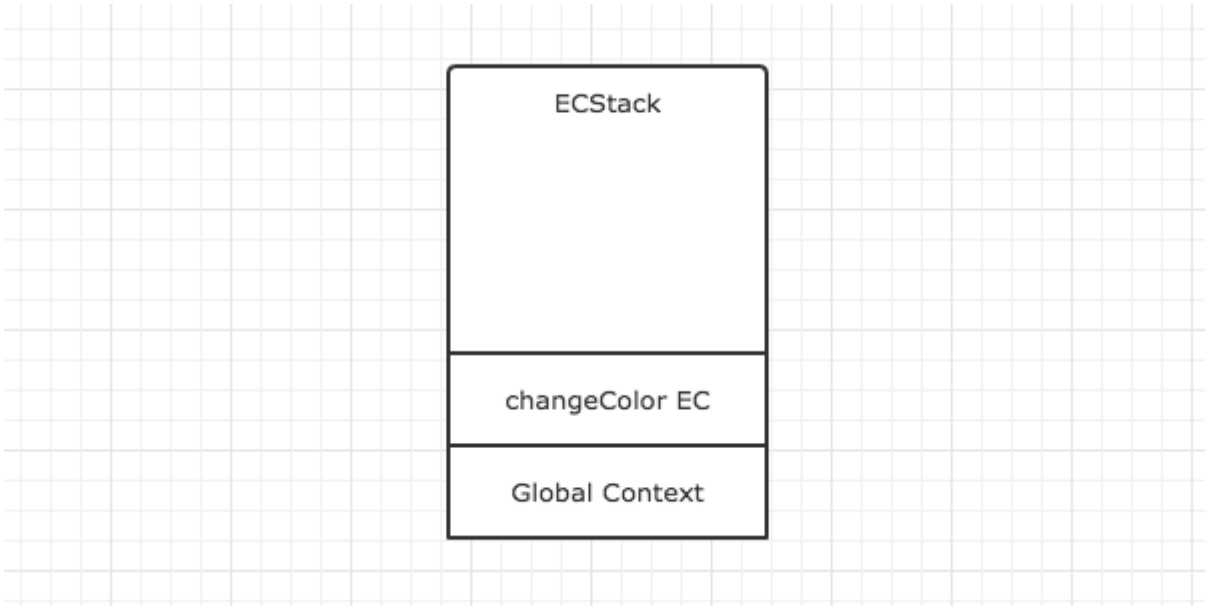
```
changeColor();
```

这里用`ECStack`来表示处理执行上下文组的堆栈。第一步，首先是全局上下文入栈。



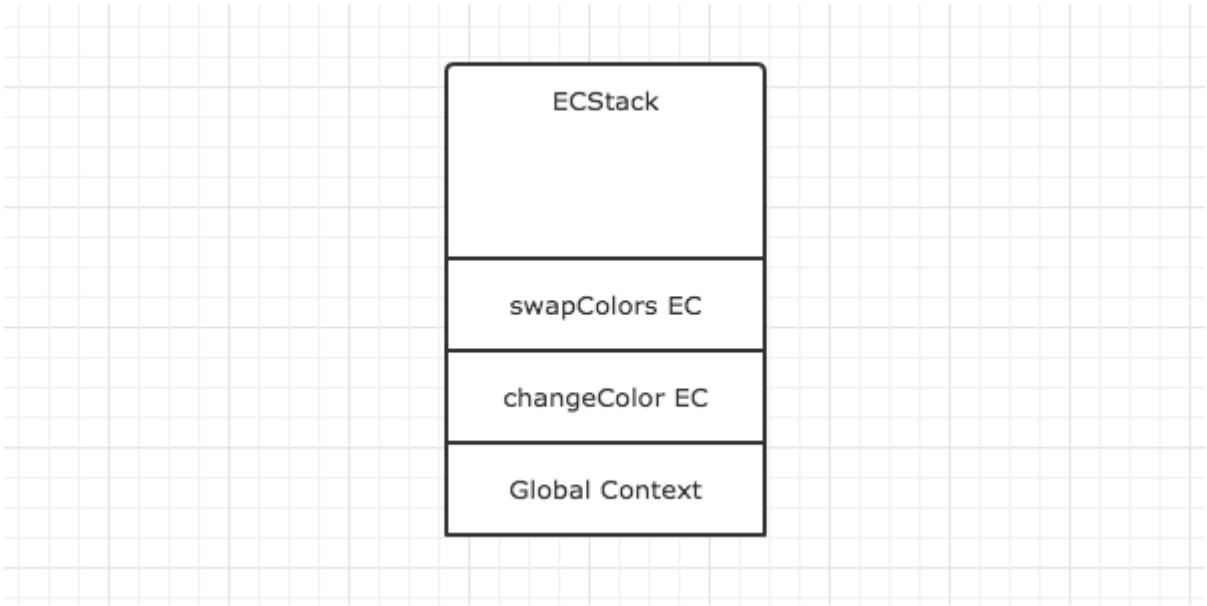
第一步：全局上下文入栈

全局上下文入栈之后，其中的可执行代码开始执行，直到遇到了`changeColor()`，这一句激活函数`changeColor`创建它自己的执行上下文，因此第二步就是`changeColor`的执行上下文入栈。



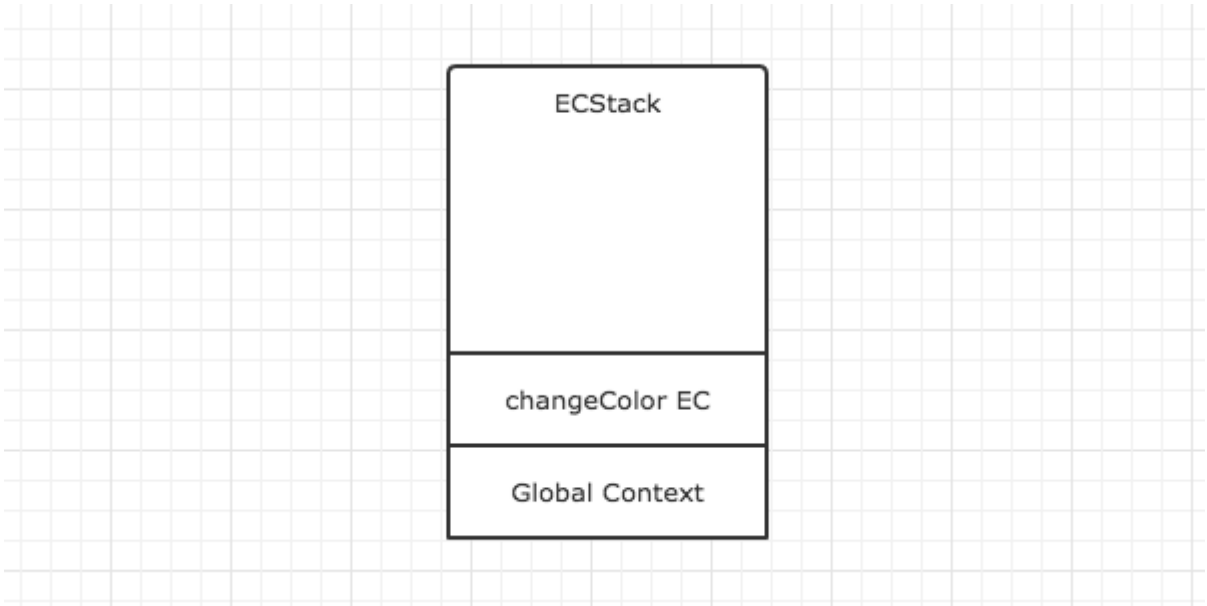
第二步：changeColor的执行上下文入栈

`changeColor`的上下文入栈之后，控制器开始执行其中的可执行代码，遇到`swapColors()`之后又激活了一个执行上下文。因此第三步是`swapColors`的执行上下文入栈。



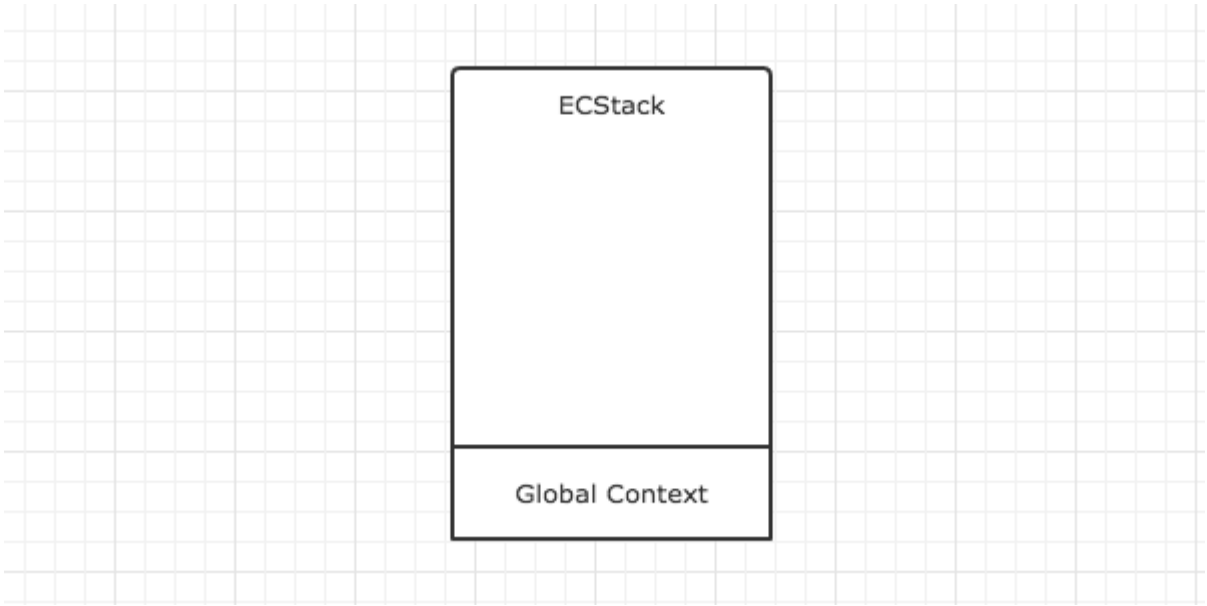
第三步：swapColors的执行上下文入栈

在`swapColors`的可执行代码中，再没有遇到其他能生成执行上下文的情况，因此这段代码顺利执行完毕，`swapColors`的上下文从栈中弹出。

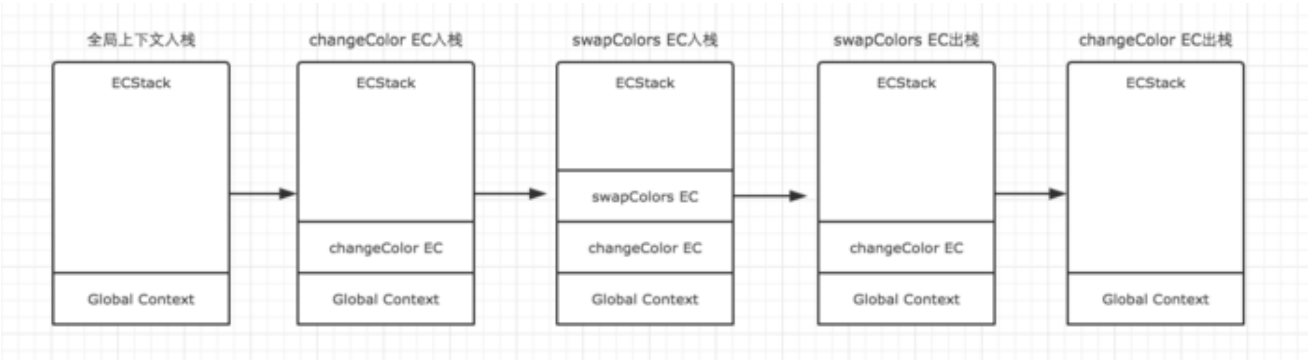


第四步：swapColors的执行上下文出栈

`swapColors`的执行上下文弹出之后，继续执行`changeColor`的可执行代码，也没有再遇到其他执行上下文，顺利执行完毕之后弹出。这样，`ECStack`中就只身下全局上下文了。全局上下文在浏览器窗口关闭后出栈。



第五步：changeColor的执行上下文出栈



> 注意：函数中，遇到return能直接终止可执行代码的执行，因此会直接将当前上下文弹出栈。

详细了解了这个过程之后，我们就可以对执行上下文总结一些结论了。

- 单线程
- 同步执行，只有栈顶的上下文处于执行中，其他上下文需要等待
- 全局上下文只有唯一的一个，它在浏览器关闭时出栈
- 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。

为了巩固一下执行上下文的理解，我们再来绘制一个例子的演变过程，这是一个简单的闭包例子。

```
function f1(){
    var n = 999;
    function f2(){
        console.log(n);
    }
    return f2;
}
var result = f1();
result(); // 999
```

因为 `f1` 中的函数 `f2` 在 `f1` 的可执行代码中，并没有被调用执行，因此执行 `f1` 时， `f2` 不会创建新的上下文，而直到 `result` 执行时，才创建了一个新的。具体演变过程如下。