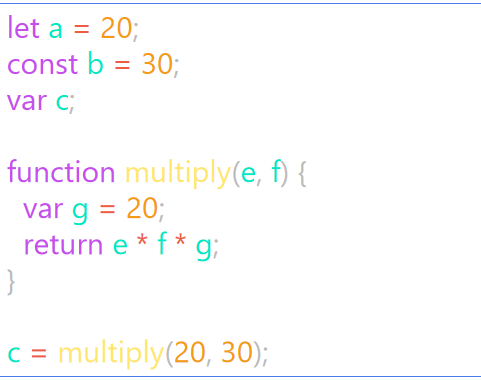
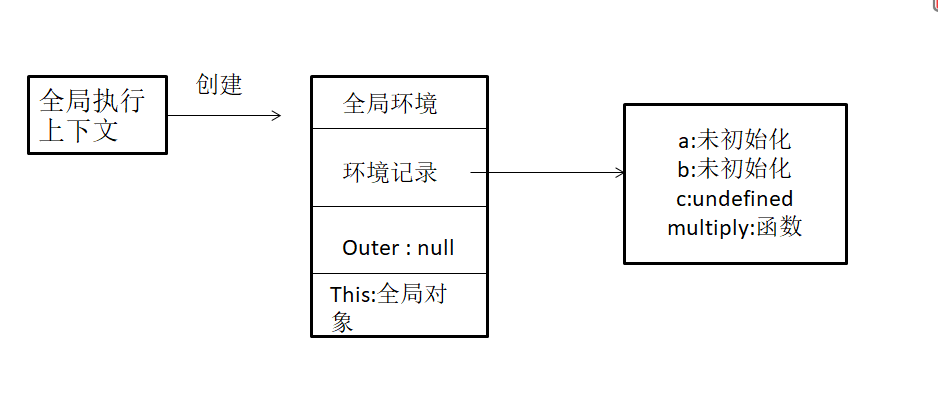
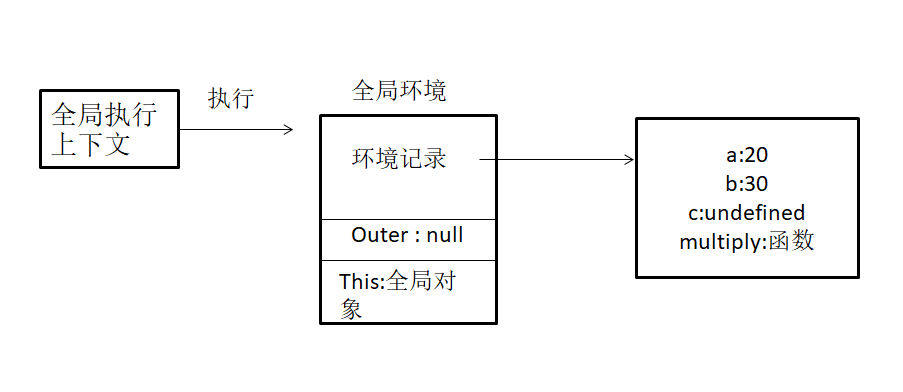
代码



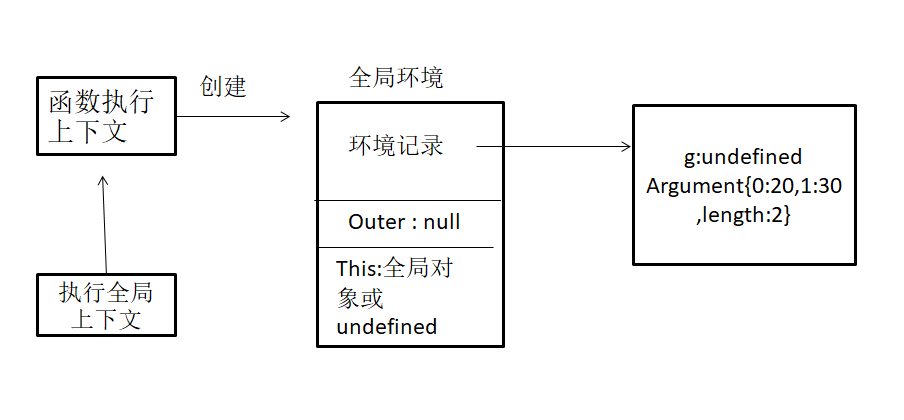
1. 代码开始运行，创建一个全局执行上下文来执行全局代码



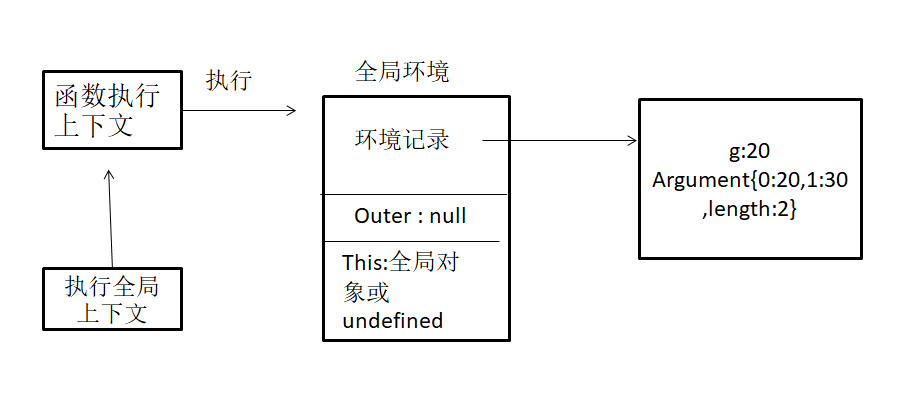
1. 执行阶段，变量赋值完成



1. 创建一个新的函数执行上下文来执行该函数代码



1. 函数变量赋值完成，执行上下文进入执行阶段



let a = 20;

const b = 30;

var c;

function multiply(e, f) {

var g = 20;

return e \* f \* g;

}

c = multiply(20, 30);

复制代码

我们用伪代码来描述上述代码中执行上下文的创建过程：

复制代码

GlobalExectionContext = { // 全局执行上下文

ThisBinding: <Global Object>, // this指向

LexicalEnvironment: { // 词法环境

EnvironmentRecord: { // 环境记录

Type: "Object", // 全局环境

// 标识符绑定在这里

a: < uninitialized >,

b: < uninitialized >,

multiply: < func >

}

outer: <null> // 对外部环境的引用为null

},

VariableEnvironment: { // 变量环境

EnvironmentRecord: { // 环境记录

Type: "Object", // 全局环境

// 标识符绑定在这里

c: undefined,

}

outer: <null> // 对外部环境对引用为null

}

}

FunctionExectionContext = {

ThisBinding: <Global Object>, // 函数执行上下文

LexicalEnvironment: { // 词法环境

EnvironmentRecord: { // 环境记录

Type: "Declarative", // 函数环境

// 标识符绑定在这里

Arguments: {0: 20, 1: 30, length: 2},

},

outer: <GlobalLexicalEnvironment> // 对外部环境对引用

},

VariableEnvironment: { // 变量环境

EnvironmentRecord: { // 环境记录

Type: "Declarative", // 函数环境

// 标识符绑定在这里

g: undefined

},

outer: <GlobalLexicalEnvironment> // 对外部环境对引用

}

}

不知道你有没有发现，在执行上下文创建阶段，函数声明与var声明的变量在创建阶段已经被赋予了一个值，var声明被设置为了undefined，函数被设置为了自身函数，而let const被设置为未初始化。

现在你总知道变量提升与函数声明提前是怎么回事了吧，以及为什么let const为什么有暂时性死域，这是因为作用域创建阶段JS引擎对两者初始化赋值不同。

上下文除了创建阶段外，还有执行阶段，这点大家应该好理解，代码执行时根据之前的环境记录对应赋值，比如早期var在创建阶段为undefined，如果有值就对应赋值，像let const值为未初始化，如果有值就赋值，无值则赋予undefined。