link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=202 sentences=598, words=4820

1. 规范

- ES5英文版 (http://es5.github.io)
- ES5中文版 (http://yanhaijing.com/es5)
- ES6英文版 (https://262.ecma-international.org/6.0)

2. 进入全局代码

- 执行全局代码 (https://www.processon.com/diagraming/61bcc05c7d9c087834ef1a64)

2.1 1.global.js

src\1.global.js

```
var a = 1;
function one(c) {
    var b = 2;
    console.log(a, b, c);
}
one(3);
```

2.2 1.global.run.js

src\1.global.run.js

```

const LexicalEnvironment = require('./LexicalEnvironment');
const ExecutionContext = require('./ExecutionContext');
const ECStack = require('./ECStack');

const globalLexicalEnvironment = LexicalEnvironment.NewObjectEnvironment(global, null);
const globalEC = new ExecutionContext(globalLexicalEnvironment, global);
ECStack.push(globalEC);

let env = ECStack.current.lexicalEnvironment.environmentRecord;

let configurableBindings = false;

let strict = false;

let dn = 'a';

let varAlreadyDeclared = env.HasBinding(dn);

if (!varAlreadyDeclared) {

    env.CreateMutableBinding(dn, configurableBindings);

    env.SetMutableBinding(dn, undefined, strict);
}
console.log(env.GetBindingValue('a'));
```

2.3 LexicalEnvironment.js

src\LexicalEnvironment.js

```
const DeclarativeEnvironmentRecords = require("./DeclarativeEnvironmentRecords");
const ObjectEnvironmentRecords = require("./ObjectEnvironmentRecords");
class LexicalEnvironment {

   static GetIdentifierReference(lex, name, strict) {

        if (!lex) throw new Error(`ReferenceError: ${name} is not defined`);
        let envRec = lex.environmentRecord;
        let exists = envRec.HasBinding(name);
        if (exists) {
            const value = envRec.GetBindingValue(name);
            return { name: value };
        } else {
            lex = lex.outer;
            return LexicalEnvironment.GetIdentifierReference(lex, name, strict);
        }
    }

    static NewDeclarativeEnvironment(lexicalEnvironment) {
        let env = new LexicalEnvironment();
        let envRec = new DeclarativeEnvironmentRecords();
        env.environmentRecord = envRec;
        env.outer = lexicalEnvironment;
        return env;
    }

    static NewObjectEnvironment(object, lexicalEnvironment) {
        let env = new LexicalEnvironment();
        let envRec = new ObjectEnvironmentRecords(object);
        env.environmentRecord = envRec;
        env.outer = lexicalEnvironment;
        return env;;
    }
}
module.exports = LexicalEnvironment;
```

2.4 DeclarativeEnvironmentRecords.js

src\DeclarativeEnvironmentRecords.js

```
const EnvironmentRecord = require('./EnvironmentRecord');
class DeclarativeEnvironmentRecords extends EnvironmentRecord {

    HasBinding(N) {
        let envRec = this;
        return N in envRec;
    }

    CreateMutableBinding(N, D) {

        let envRec = this;

        console.assert(!this.HasBinding(N));

        Object.defineProperty(envRec, N, {
            value: undefined,
            writable: true,
            configurable: D
        });

    }

    SetMutableBinding(N, V, S) {

        let envRec = this;

        console.assert(this.HasBinding(N));

        const propertyDescriptor = Object.getOwnPropertyDescriptor(envRec, N);
        if (S && !propertyDescriptor.writable) {
            throw new Error("TypeError: Assignment to constant variable.");
        }

        envRec[N] = V;
    }

    GetBindingValue(N, S) {

        let envRec = this;

        console.assert(this.HasBinding(N));

        const V = envRec[N];
        const propertyDescriptor = Object.getOwnPropertyDescriptor(envRec, N);
        if (!propertyDescriptor.uninitialized) {
            if (!S) return V;
            throw new Error(`ReferenceError: Cannot access '${N}' before initialization`);
        }

        return V;
    }

    DeleteBinding(N) {

        let envRec = this;
        if (!this.HasBinding(N)) {
            return true;
        }
        const propertyDescriptor = Object.getOwnPropertyDescriptor(envRec, N);
        if (!propertyDescriptor.configurable) {
            return false;
        }
        delete envRec[N];
        return true;
    }

    ImplicitThisValue() {
        return undefined;
    }

    CreateImmutableBinding(N) {

        let envRec = this;

        console.assert(!this.HasBinding(N));

        Object.defineProperty(envRec, N, {
            value: undefined,
            uninitialized: true
        });
    }

    InitializeImmutableBinding(N, V) {

        let envRec = this;

        const propertyDescriptor = Object.getOwnPropertyDescriptor(envRec, N);
        console.assert(propertyDescriptor && propertyDescriptor.uninitialized);
        envRec[N] = V;
        propertyDescriptor.uninitialized = true;
    }

}
module.exports = DeclarativeEnvironmentRecords;
```

2.5 EnvironmentRecord.js

src\EnvironmentRecord.js

```
class EnvironmentRecord {

}
module.exports = EnvironmentRecord;
```

2.6 ObjectEnvironmentRecords.js

src\ObjectEnvironmentRecords.js

```
const EnvironmentRecord = require('./EnvironmentRecord');
class ObjectEnvironmentRecords extends EnvironmentRecord {
    constructor(bindings) {
        super();
        this.bindings = bindings;
        this.provideThis = false;
    }

    HasBinding(N) {
        const envRec = this;
        const bindings = envRec.bindings;
        return bindings.hasOwnProperty(N);
    }

    CreateMutableBinding(N, D) {

        const envRec = this;
        const bindings = envRec.bindings;

        console.assert(!bindings.hasOwnProperty(N));
        const configValue = D;

        Object.defineProperty(bindings, N, {
            value: undefined,
            enumerable: true,
            configurable: configValue,
            writable: true
        });
    }

    SetMutableBinding(N, V, S) {

        const envRec = this;
        const bindings = envRec.bindings;

        console.assert(this.HasBinding(N));

        const propertyDescriptor = Object.getOwnPropertyDescriptor(bindings, N);
        if (S && !propertyDescriptor.writable) {
            throw new Error("TypeError: Assignment to constant variable.");
        }

        bindings[N] = V;
    }

    GetBindingValue(N, S) {

        const envRec = this;
        const bindings = envRec.bindings;

        console.assert(bindings.hasOwnProperty(N), `变量${N}尚未定义`);

        const V = bindings[N];
        const propertyDescriptor = Object.getOwnPropertyDescriptor(bindings, N);
        if (!propertyDescriptor.uninitialized) {
            if (!S) return V;
            throw new Error(`ReferenceError: Cannot access '${N}' before initialization`);
        }

        return V;
    }

    DeleteBinding(N) {

        const envRec = this;
        const bindings = envRec.bindings;
        if (!bindings.hasOwnProperty(N)) {
            return true;
        }
        const propertyDescriptor = Object.getOwnPropertyDescriptor(bindings, N);
        if (!propertyDescriptor.configurable) {
            return false;
        }
        delete bindings[N];
        return true;
    }

    ImplicitThisValue() {
        const envRec = this;
        if (provideThis) {
            return envRec.bindings;
        }
        return undefined;
    }
}
module.exports = ObjectEnvironmentRecords;
```

2.7 ExecutionContext.js

src\ExecutionContext.js

```
class ExecutionContext {
    constructor(lexicalEnvironment, thisBinding) {
        this.variableEnvironment = this.lexicalEnvironment = lexicalEnvironment;
        this.thisBinding = thisBinding;
    }
}
module.exports = ExecutionContext;
```

2.8 ECStack.js

src\ECStack.js

```
class ECStack {
    constructor() {
        this.contexts = [];
    }
    push(EC) {
        this.contexts.push(EC);
    }
    get current() {
        return this.contexts[this.contexts.length - 1];
    }
    pop() {
        this.contexts.pop();
    }
}
module.exports = new ECStack();
```

3. 注册函数

3.1 FunctionDeclaration.js

src/FunctionDeclaration.js

```
class FunctionDeclaration {
    static createInstance(fn, FormalParameterList, FunctionBody, Scope) {

        let Strict = false;

        let F = { name: fn };

        F[` [[Class]]`] = Function;

        F[`[[Prototype]]`] = Function.prototype;

        F[`[[Scope]]`] = Scope;

        let names = FormalParameterList;

        F[`[[FormalParameters]]`] = names;

        F[`[[Code]]`] = FunctionBody;

        F[`[[Extensible]]`] = true;

        let len = FormalParameterList.length;

        Object.defineProperty(F, 'length', { value: len, writable: false, enumerable: false, configurable: false });

        let proto = new Object();

        Object.defineProperty(proto, 'constructor', { value: F, writable: true, enumerable: false, configurable: false });

        Object.defineProperty(F, 'prototype', { value: proto, writable: true, enumerable: false, configurable: false });
        return F;
    }
}
module.exports = FunctionDeclaration;
```

3.2 global.run.js

src\1.global.run.js

```
/**
var a = 1;
function one(c) {
    var b = 2;
    console.log(a, b,c);
}
one();
*/
const LexicalEnvironment = require('./LexicalEnvironment');
const ExecutionContext = require('./ExecutionContext');
const ECStack = require('./ECStack');
+const FunctionDeclaration = require('./FunctionDeclaration');

//1.将变量环境设置为全局环境 2.将词法环境设置为 全局环境
const globalLexicalEnvironment = LexicalEnvironment.NewObjectEnvironment(global, null);
const globalEC = new ExecutionContext(globalLexicalEnvironment, global);
ECStack.push(globalEC);

//令 env 为当前运行的执行环境的环境变量的 环境记录
let env = ECStack.current.lexicalEnvironment.environmentRecord;
//如果 code 是 eval 代码 ，则令 configurableBindings 为 true，否则令 configurableBindings 为 false。
let configurableBindings = false;
//如果代码是 严格模式下的代码 ，则令 strict 为 true，否则令 strict 为 false。
let strict = false;
//按源码顺序遍历 code，对于每一个 VariableDeclaration 和 VariableDeclarationNoIn 表达式：
//令 dn 为 d 中的标识符。
let dn = 'a';
//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
let varAlreadyDeclared = env.HasBinding(dn);
//如果 varAlreadyDeclared 为 false，则：
if (!varAlreadyDeclared) {
    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
    env.CreateMutableBinding(dn, configurableBindings);
    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
    env.SetMutableBinding(dn, undefined, strict);
}
console.log(env.GetBindingValue('a'));

+//按源码顺序遍历 code，对于每一个 FunctionDeclaration 表达式 f：
+//令 fn 为 FunctionDeclaration 表达式 f 中的 标识符
+let fn = 'one';
+//创建函数对象
+//指定 FormalParameterList 为可选参数列表
+let FormalParameterList = ['c'];
+//指定 FunctionBody 为函数体
+let FunctionBody = `
+    var b = 2;
+    console.log(a, b);
+`;
+//指定 Scope 为 词法环境
+let Scope = ECStack.current.lexicalEnvironment;
+//按 第 13 章 中所述的步骤初始化 FunctionDeclaration 表达式 ，并令 fo 为初始化的结果。
+let fo = FunctionDeclaration.createInstance(fn, FormalParameterList, FunctionBody, Scope);
+//以 fn 为参数，调用 env 的 HasBinding 具体方法，并令 argAlreadyDeclared 为调用的结果。
+let argAlreadyDeclared = env.HasBinding(fn);
+if (!argAlreadyDeclared) {
+    env.CreateMutableBinding(fn, configurableBindings);
+} else {
+    //否则如果 env 是全局环境的 环境记录 对象，则
+    //令 go 为全局对象。
+    let go = global;
+    //以 fn 为参数，调用 go 和 [[GetProperty]] 内部方法，并令 existingProp 为调用的结果。
+    let existingProp = Object.getOwnPropertyDescriptor(go, fn);
+    if (existingProp.configurable) {
+        Object.defineProperty(go, fn, { value: undefined, writable: true, enumerable: true, configurable: +configurableBindings });
+        if (!existingProp.writable) {
+            throw new Error(`TypeError`);
+        }
+    }
+}
+env.SetMutableBinding(fn, fo, strict);
+console.log(env.GetBindingValue('one'));
```

4. 执行全局代码和函数

4.1 CreateArgumentsObject.js

src\CreateArgumentsObject.js

```
function CreateArgumentsObject(func, names, args, env, strict) {

    let len = args.length;

    let obj = {};

    obj[`[[Class]]`] = 'Arguments';

    obj.constructor = Object;

    obj[`[[Prototype]]`] = Object.prototype;

    Object.defineProperty(obj, 'length', {
        value: len, writable: true, enumerable: false, configurable: true
    });
    let indx = len - 1;
    while (indx >= 0) {
        let val = args[indx];
        Object.defineProperty(obj, indx.toString(), { value: val, writable: true, enumerable: true, configurable: true });
        indx = indx - 1;
    }
    return obj;
}
module.exports = CreateArgumentsObject;
```

4.2 global.run.js

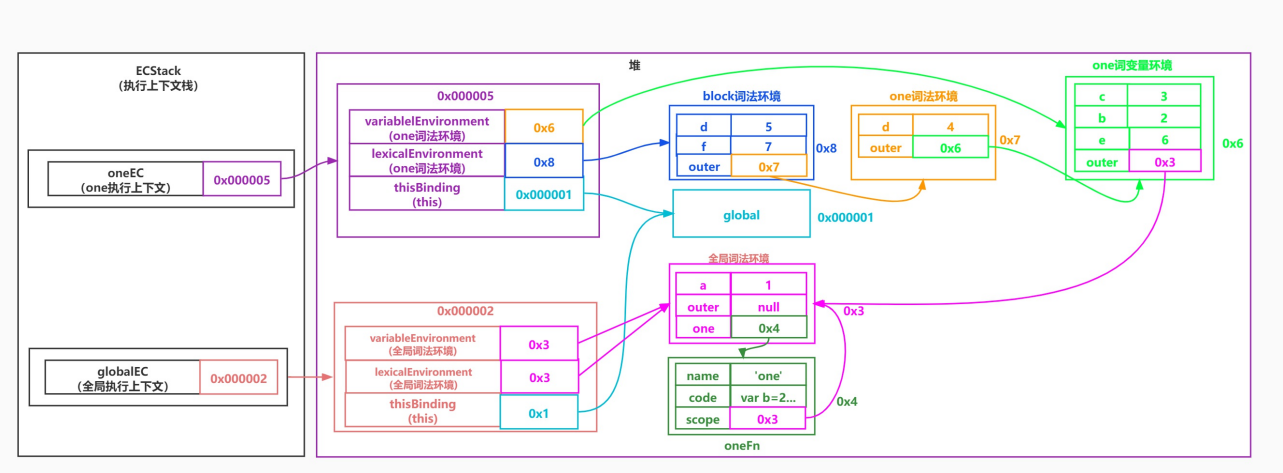
src\1.global.run.js

```
/**
var a = 1;
function one(c) {
    var b = 2;
    console.log(a, b,c);
}
one();
*/
const LexicalEnvironment = require('./LexicalEnvironment');
const ExecutionContext = require('./ExecutionContext');
const ECStack = require('./ECStack');
const FunctionDeclaration = require('./FunctionDeclaration');
+const CreateArgumentsObject = require('./CreateArgumentsObject');

//1.将变量环境设置为全局环境 2.将词法环境设置为 全局环境
const globalLexicalEnvironment = LexicalEnvironment.NewObjectEnvironment(global, null);
const globalEC = new ExecutionContext(globalLexicalEnvironment, global);
ECStack.push(globalEC);

//令 env 为当前运行的执行环境的环境变量的 环境记录
let env = ECStack.current.lexicalEnvironment.environmentRecord;
//如果 code 是 eval 代码 ，则令 configurableBindings 为 true，否则令 configurableBindings 为 false。
let configurableBindings = false;
//如果代码是 严格模式下的代码 ，则令 strict 为 true，否则令 strict 为 false。
let strict = false;
//按源码顺序遍历 code，对于每一个 VariableDeclaration 和 VariableDeclarationNoIn 表达式：
//令 dn 为 d 中的标识符。
let dn = 'a';
//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
let varAlreadyDeclared = env.HasBinding(dn);
//如果 varAlreadyDeclared 为 false，则：
if (!varAlreadyDeclared) {
    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
    env.CreateMutableBinding(dn, configurableBindings);
    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
    env.SetMutableBinding(dn, undefined, strict);
}
console.log(env.GetBindingValue('a'));//undefined

//按源码顺序遍历 code，对于每一个 FunctionDeclaration 表达式 f：
//令 fn 为 FunctionDeclaration 表达式 f 中的 标识符
let fn = 'one';
//创建函数对象
//指定 FormalParameterList 为可选参数列表
let FormalParameterList = ['c'];
//指定 FunctionBody 为函数体
let FunctionBody = `
    var b = 2;
    console.log(a, b);
`;
//指定 Scope 为 词法环境
let Scope = ECStack.current.lexicalEnvironment;
//按 第 13 章 中所述的步骤初始化 FunctionDeclaration 表达式 ，并令 fo 为初始化的结果。
let fo = FunctionDeclaration.createInstance(fn, FormalParameterList, FunctionBody, Scope);
//以 fn 为参数，调用 env 的 HasBinding 具体方法，并令 argAlreadyDeclared 为调用的结果。
let argAlreadyDeclared = env.HasBinding(fn);
if (!argAlreadyDeclared) {
    env.CreateMutableBinding(fn, configurableBindings);
} else {
    //否则如果 env 是全局环境的 环境记录 对象，则
    //令 go 为全局对象。
    let go = global;
    //以 fn 为参数，调用 go 和 [[GetProperty]] 内部方法，并令 existingProp 为调用的结果。
    let existingProp = Object.getOwnPropertyDescriptor(go, fn);
    if (existingProp.configurable) {
        Object.defineProperty(go, fn, { value: undefined, writable: true, enumerable: true, configurable: configurableBindings });
        if (!existingProp.writable) {
            throw new Error(`TypeError`);
        }
    }
}
env.SetMutableBinding(fn, fo, strict);
//console.log(env.GetBindingValue('one'));//1
```

```
+// 给a赋值为1
+env.SetMutableBinding('a', 1);
+console.log(env.GetBindingValue('a'));//1
+//进入函数代码
+//当控制流根据一个函数对象 F、调用者提供的 thisArg 以及调用者提供的 argumentList，进入 函数代码 的执行环境时，执行以下步+骤：
+let thisArg = global;
+let argumentList = [3];
+//以 F 的 [[Scope]] 内部属性为参数调用 NewDeclarativeEnvironment，并令 localEnv 为调用的结果
+let localEnv = LexicalEnvironment.NewDeclarativeEnvironment(fo[`[[Scope]]`]);
+//设词法环境为 localEnv 设变量环境为 localEnv
+let oneExecutionContext = new ExecutionContext(localEnv, thisArg);
+ECStack.push(oneExecutionContext);
+//按 [10.5](#10.5) 描述的方案，使用 函数代码 code 和 argumentList 执行定义绑定初始化步骤
+env = ECStack.current.lexicalEnvironment.environmentRecord;
+configurableBindings = false;
+strict = false;
+//令 func 为通过 [[Call]] 内部属性初始化 code 的执行的函数对象
+let func = fo;
+//令 code 为 F 的 [[Code]] 内部属性的值。
+let code = func[`[[Code]]`];
+//令 names 为 func 的 [[FormalParameters]] 内部属性。
+names = func[`[[FormalParameters]]`];
+let args = [3];
+//令 argCount 为 args 中元素的数量
+let argCount = args.length;
+let n = 0;
+names.forEach(argName => {
+    n += 1;
+    let v = n > argCount ? undefined : args[n - 1];
+    //以 argName 为参数，调用 env 的 HasBinding 具体方法，并令 argAlreadyDeclared 为调用的结果。
+    argAlreadyDeclared = env.HasBinding(argName);
+    if (!argAlreadyDeclared) {
+        env.CreateMutableBinding(argName);
+    }
+    env.SetMutableBinding(argName, v, strict);
+});
+let argumentsAlreadyDeclared = env.HasBinding('arguments');
+if (!argumentsAlreadyDeclared) {
+    let argsObj = CreateArgumentsObject(func, names, args, env, strict);
+    env.CreateMutableBinding('arguments');
+    env.SetMutableBinding('arguments', argsObj);
+}
+
+dn = 'b';
+//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
+varAlreadyDeclared = env.HasBinding(dn);
+//如果 varAlreadyDeclared 为 false，则：
+if (!varAlreadyDeclared) {
+    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
+    env.CreateMutableBinding(dn, configurableBindings);
+    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
+    env.SetMutableBinding(dn, undefined, strict);
+}
+env.SetMutableBinding(dn, 2);
+
+console.log(env.GetBindingValue('arguments'));//['3']
+
+console.log(
+    LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'a'),
+    LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'b'),
+    LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'c')
+);
```

5. 块级作用域

- 块级作用域 (https://www.processon.com/diagraming/61beb9795653bb5a3e2b12fa)



5.1 block.js

```
var a = 1;
function one(c) {
    var b = 2;
    console.log(a, b, c, e );
    let d = 4;
    {
        let d = 5;
        var e = 6;
        let f = 7;
        console.log(a, b, c, d, e, f);
    }
}
one(3);
```

5.2 block.run.js

src\block.run.js

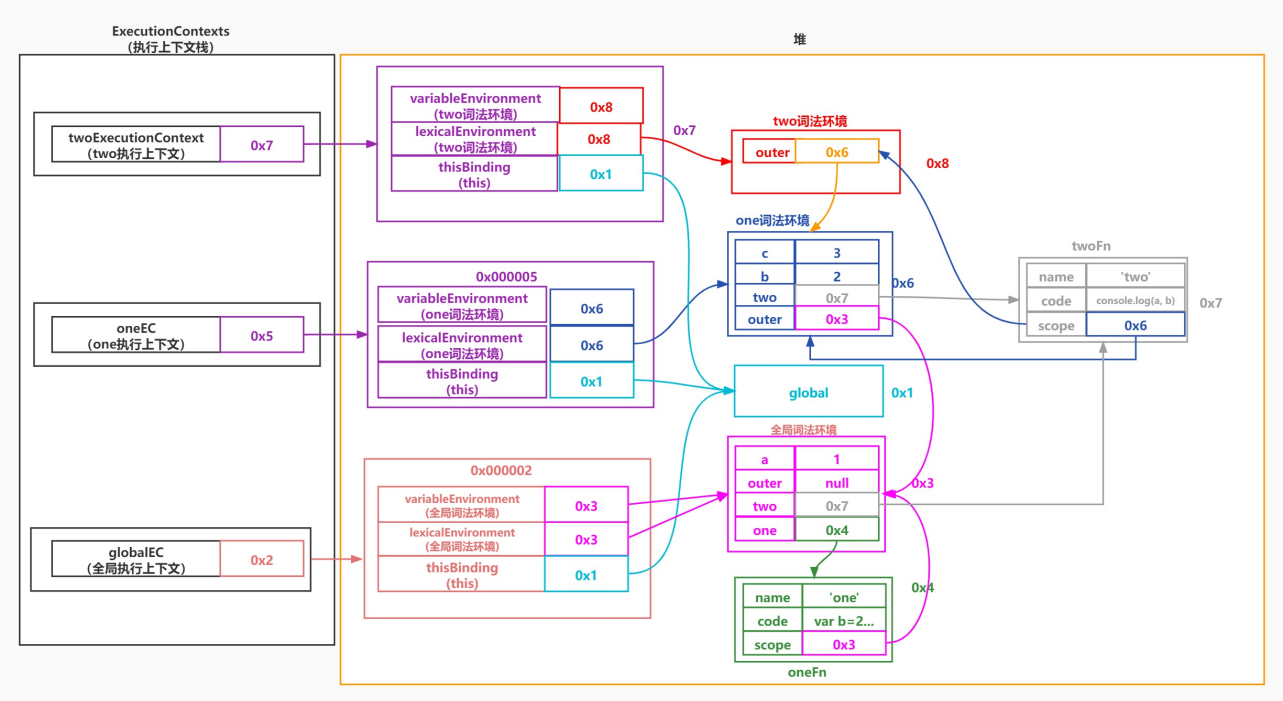
```
/**
var a = 1;
function one(c) {
    var b = 2;
    console.log(a, b, c, e);
    let d = 4;
    {
        let d = 5;
        var e = 6;
        console.log(a, b, c, d, e);
    }
}
one(3);
*/
const LexicalEnvironment = require('./LexicalEnvironment');
const ExecutionContext = require('./ExecutionContext');
const ECStack = require('./ECStack');
const FunctionDeclaration = require('./FunctionDeclaration');
const CreateArgumentsObject = require('./CreateArgumentsObject');

//1.将变量环境设置为全局环境 2.将词法环境设置为 全局环境
const globalLexicalEnvironment = LexicalEnvironment.NewObjectEnvironment(global, null);
const globalEC = new ExecutionContext(globalLexicalEnvironment, global);
ECStack.push(globalEC);

//令 env 为当前运行的执行环境的环境变量的 环境记录
let env = ECStack.current.lexicalEnvironment.environmentRecord;
//如果 code 是 eval 代码 ，则令 configurableBindings 为 true，否则令 configurableBindings 为 false。
let configurableBindings = false;
//如果代码是 严格模式下的代码 ，则令 strict 为 true，否则令 strict 为 false。
let strict = false;
//按源码顺序遍历 code，对于每一个 VariableDeclaration 和 VariableDeclarationNoIn 表达式：
//令 dn 为 d 中的标识符。
let dn = 'a';
//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
let varAlreadyDeclared = env.HasBinding(dn);
//如果 varAlreadyDeclared 为 false，则：
if (!varAlreadyDeclared) {
    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
    env.CreateMutableBinding(dn, configurableBindings);
    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
    env.SetMutableBinding(dn, undefined, strict);
}
console.log(env.GetBindingValue('a'));//undefined

//按源码顺序遍历 code，对于每一个 FunctionDeclaration 表达式 f：
//令 fn 为 FunctionDeclaration 表达式 f 中的 标识符
let fn = 'one';
//创建函数对象
//指定 FormalParameterList 为可选参数列表
let FormalParameterList = ['c'];
//指定 FunctionBody 为函数体
let FunctionBody = `
    var b = 2;
    console.log(a, b);
`;
//指定 Scope 为 词法环境
let Scope = ECStack.current.lexicalEnvironment;
//按 第 13 章 中所述的步骤初始化 FunctionDeclaration 表达式 ，并令 fo 为初始化的结果。
let fo = FunctionDeclaration.createInstance(fn, FormalParameterList, FunctionBody, Scope);
//以 fn 为参数，调用 env 的 HasBinding 具体方法，并令 argAlreadyDeclared 为调用的结果。
let argAlreadyDeclared = env.HasBinding(fn);
if (!argAlreadyDeclared) {
    env.CreateMutableBinding(fn, configurableBindings);
} else {
    //否则如果 env 是全局环境的 环境记录 对象，则
    //令 go 为全局对象。
    let go = global;
    //以 fn 为参数，调用 go 和 [[GetProperty]] 内部方法，并令 existingProp 为调用的结果。
    let existingProp = Object.getOwnPropertyDescriptor(go, fn);
    if (existingProp.configurable) {
        Object.defineProperty(go, fn, { value: undefined, writable: true, enumerable: true, configurable: configurableBindings });
        if (!existingProp.writable) {
            throw new Error(`TypeError`);
        }
    }
}
env.SetMutableBinding(fn, fo, strict);
//console.log(env.GetBindingValue('one'));//1

// 给a赋值为1
env.SetMutableBinding('a', 1);
console.log(env.GetBindingValue('a'));//1
//进入函数代码
//当控制流根据一个函数对象 F、调用者提供的 thisArg 以及调用者提供的 argumentList，进入 函数代码 的执行环境时，执行以下步骤：
let thisArg = global;
```

```
let argumentList = [3];
//以 F 的 [[Scope]] 内部属性为参数调用 NewDeclarativeEnvironment，并令 localEnv 为调用的结果
let localEnv = LexicalEnvironment.NewDeclarativeEnvironment(fo[`[[Scope]]`]);
//设词法环境为 localEnv 设变量环境为 localEnv
let oneExecutionContext = new ExecutionContext(localEnv, thisArg);
ECStack.push(oneExecutionContext);
//按 [10.5](#10.5) 描述的方案，使用 函数代码 code 和 argumentList 执行定义绑定初始化步骤
env = ECStack.current.lexicalEnvironment.environmentRecord;
configurableBindings = false;
strict = false;
//令 func 为通过 [[Call]] 内部属性初始化 code 的执行的函数对象
let func = fo;
//令 code 为 F 的 [[Code]] 内部属性的值。
let code = func[`[[Code]]`];
//令 names 为 func 的 [[FormalParameters]] 内部属性。
names = func[`[[FormalParameters]]`];
let args = [3];
//令 argCount 为 args 中元素的数量
let argCount = args.length;
let n = 0;
names.forEach(argName => {
    n += 1;
    let v = n > argCount ? undefined : args[n - 1];
    //以 argName 为参数，调用 env 的 HasBinding 具体方法，并令 argAlreadyDeclared 为调用的结果。
    argAlreadyDeclared = env.HasBinding(argName);
    if (!argAlreadyDeclared) {
        env.CreateMutableBinding(argName);
    }
    env.SetMutableBinding(argName, v, strict);
});
let argumentsAlreadyDeclared = env.HasBinding('arguments');
if (!argumentsAlreadyDeclared) {
    let argsObj = CreateArgumentsObject(func, names, args, env, strict);
    env.CreateMutableBinding('arguments');
    env.SetMutableBinding('arguments', argsObj);
}

dn = 'b';
//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
varAlreadyDeclared = env.HasBinding(dn);
//如果 varAlreadyDeclared 为 false，则：
if (!varAlreadyDeclared) {
    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
    env.CreateMutableBinding(dn, configurableBindings);
    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
    env.SetMutableBinding(dn, undefined, strict);
}

+dn = 'e';
+//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
+varAlreadyDeclared = env.HasBinding(dn);
+//如果 varAlreadyDeclared 为 false，则：
+if (!varAlreadyDeclared) {
+    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
+    env.CreateMutableBinding(dn, configurableBindings);
+    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
+    env.SetMutableBinding(dn, undefined, strict);
+}
+
+localEnv = LexicalEnvironment.NewDeclarativeEnvironment(ECStack.current.variableEnvironment);
+env = localEnv.environmentRecord;
+dn = 'd';
+//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
+varAlreadyDeclared = env.HasBinding(dn);
+//如果 varAlreadyDeclared 为 false，则：
+if (!varAlreadyDeclared) {
+    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
+    env.CreateMutableBinding(dn, configurableBindings);
+    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
+    env.SetMutableBinding(dn, undefined, strict);
+}
+ECStack.current.lexicalEnvironment = localEnv;
+//开始执行one
+ECStack.current.variableEnvironment.environmentRecord.SetMutableBinding('b', 2);
+console.log(
+    'a=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'a').name,
+    'b=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'b').name,
+    'c=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'c').name,
+    'e=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'e').name,
+    //LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'f')
+);
+ECStack.current.variableEnvironment.environmentRecord.SetMutableBinding('d', 4);
+
+localEnv = LexicalEnvironment.NewDeclarativeEnvironment(ECStack.current.variableEnvironment);
+ECStack.current.lexicalEnvironment = localEnv;
+env = localEnv.environmentRecord;
+dn = 'd';
+//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
+varAlreadyDeclared = env.HasBinding(dn);
+//如果 varAlreadyDeclared 为 false，则：
+if (!varAlreadyDeclared) {
+    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
+    env.CreateMutableBinding(dn, configurableBindings);
+    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
+    env.SetMutableBinding(dn, undefined, strict);
+}
+dn = 'f';
+//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
+varAlreadyDeclared = env.HasBinding(dn);
+//如果 varAlreadyDeclared 为 false，则：
+if (!varAlreadyDeclared) {
+    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
+    env.CreateMutableBinding(dn, configurableBindings);
+    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
```

```
+    env.SetMutableBinding(dn, undefined, strict);
+}
+ECStack.current.lexicalEnvironment.environmentRecord.SetMutableBinding('d', 5);
+ECStack.current.variableEnvironment.environmentRecord.SetMutableBinding('e', 6);
+ECStack.current.lexicalEnvironment.environmentRecord.SetMutableBinding('f', 7);
+console.log(
+    'a=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'a').name,
+    'b=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'b').name,
+    'c=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'c').name,
+    'd=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'd').name,
+    'e=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'e').name,
+    'f=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'f').name
+);
```

6. 闭包

- 闭包 (https://www.processon.com/diagraming/61bebb770e3e74525cd0137b)



6.1 closure.js

```
var a = 1;
function one(c) {
    var b = 2;
   function two() {
       console.log(a,b,c);
   }
   return two;
}
var two = one(3);
two();
```

6.2 closure.run.js

closure.run.js

```
/**
var a = 1;
function one(c) {
    var b = 2;
    return function two() {
        console.log(a, b, c);
    }
}
var two = one(3);
two();
*/
const LexicalEnvironment = require('./LexicalEnvironment');
const ExecutionContext = require('./ExecutionContext');
const ECStack = require('./ECStack');
const FunctionDeclaration = require('./FunctionDeclaration');
const CreateArgumentsObject = require('./CreateArgumentsObject');

//1.将变量环境设置为全局环境 2.将词法环境设置为 全局环境
const globalLexicalEnvironment = LexicalEnvironment.NewObjectEnvironment(global, null);
const globalEC = new ExecutionContext(globalLexicalEnvironment, global);
ECStack.push(globalEC);

//令 env 为当前运行的执行环境的环境变量的 环境记录
let env = ECStack.current.lexicalEnvironment.environmentRecord;
//如果 code 是 eval 代码 ，则令 configurableBindings 为 true，否则令 configurableBindings 为 false。
let configurableBindings = false;
//如果代码是 严格模式下的代码 ，则令 strict 为 true，否则令 strict 为 false。
let strict = false;
//按源码顺序遍历 code，对于每一个 VariableDeclaration 和 VariableDeclarationNoIn 表达式：
//令 dn 为 d 中的标识符。
```

```
let dn = 'a';
//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
let varAlreadyDeclared = env.HasBinding(dn);
//如果 varAlreadyDeclared 为 false，则：
if (!varAlreadyDeclared) {
    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
    env.CreateMutableBinding(dn, configurableBindings);
    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
    env.SetMutableBinding(dn, undefined, strict);
}
//console.log(env.GetBindingValue('a'));//undefined
+dn = 'two';
+//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
+varAlreadyDeclared = env.HasBinding(dn);
+//如果 varAlreadyDeclared 为 false，则：
+if (!varAlreadyDeclared) {
+    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
+    env.CreateMutableBinding(dn, configurableBindings);
+    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
+    env.SetMutableBinding(dn, undefined, strict);
+}
//console.log(env.GetBindingValue(dn));//undefined

//按源码顺序遍历 code，对于每一个 FunctionDeclaration 表达式 f：
//令 fn 为 FunctionDeclaration 表达式 f 中的 标识符
let fn = 'one';
//创建函数对象
//指定 FormalParameterList 为可选参数列表
let FormalParameterList = ['c'];
//指定 FunctionBody 为函数体
let FunctionBody = `
    var b = 2;
    console.log(a, b);
`;
//指定 Scope 为 词法环境
let Scope = ECStack.current.lexicalEnvironment;
//按 第 13 章 中所述的步骤初始化 FunctionDeclaration 表达式 ，并令 fo 为初始化的结果。
let fo = FunctionDeclaration.createInstance(fn, FormalParameterList, FunctionBody, Scope);
//以 fn 为参数，调用 env 的 HasBinding 具体方法，并令 argAlreadyDeclared 为调用的结果。
let argAlreadyDeclared = env.HasBinding(fn);
if (!argAlreadyDeclared) {
    env.CreateMutableBinding(fn, configurableBindings);
} else {
    //否则如果 env 是全局环境的 环境记录 对象，则
    //令 go 为全局对象。
    let go = global;
    //以 fn 为参数，调用 go 和 [[GetProperty]] 内部方法，并令 existingProp 为调用的结果。
    let existingProp = Object.getOwnPropertyDescriptor(go, fn);
    if (existingProp.configurable) {
        Object.defineProperty(go, fn, { value: undefined, writable: true, enumerable: true, configurable: configurableBindings });
        if (!existingProp.writable) {
            throw new Error(`TypeError`);
        }
    }
}
env.SetMutableBinding(fn, fo, strict);
//console.log(env.GetBindingValue('one'));//1

// 给a赋值为1
env.SetMutableBinding('a', 1);
//进入函数one代码
//当控制流根据一个函数对象 F、调用者提供的 thisArg 以及调用者提供的 argumentList，进入 函数代码 的执行环境时，执行以下步骤：
let thisArg = global;
//以 F 的 [[Scope]] 内部属性为参数调用 NewDeclarativeEnvironment，并令 localEnv 为调用的结果
let localEnv = LexicalEnvironment.NewDeclarativeEnvironment(fo[`[[Scope]]`]);
//设词法环境为 localEnv 设变量环境为 localEnv
let oneExecutionContext = new ExecutionContext(localEnv, thisArg);
ECStack.push(oneExecutionContext);
//按 [10.5](#10.5) 描述的方案，使用 函数代码 code 和 argumentList 执行定义绑定初始化步骤
env = ECStack.current.lexicalEnvironment.environmentRecord;
configurableBindings = false;
strict = false;
//令 func 为通过 [[Call]] 内部属性初始化 code 的执行的函数对象
let func = fo;
//令 code 为 F 的 [[Code]] 内部属性的值。
let code = func[`[[Code]]`];
//令 names 为 func 的 [[FormalParameters]] 内部属性。
names = func[`[[FormalParameters]]`];
let args = [3];
//令 argCount 为 args 中元素的数量
let argCount = args.length;
let n = 0;
names.forEach(argName => {
    n += 1;
    let v = n > argCount ? undefined : args[n - 1];
    //以 argName 为参数，调用 env 的 HasBinding 具体方法，并令 argAlreadyDeclared 为调用的结果。
    argAlreadyDeclared = env.HasBinding(argName);
    if (!argAlreadyDeclared) {
        env.CreateMutableBinding(argName);
    }
    env.SetMutableBinding(argName, v, strict);
});
let argumentsAlreadyDeclared = env.HasBinding('arguments');
if (!argumentsAlreadyDeclared) {
    let argsObj = CreateArgumentsObject(func, names, args, env, strict);
    env.CreateMutableBinding('arguments');
    env.SetMutableBinding('arguments', argsObj);
}

dn = 'b';
//以 dn 为参数，调用 env 的 HasBinding 具体方法，并令 varAlreadyDeclared 为调用的结果
varAlreadyDeclared = env.HasBinding(dn);
//如果 varAlreadyDeclared 为 false，则：
if (!varAlreadyDeclared) {
```

```
    //以 dn 和 configurableBindings 为参数，调用 env 的 CreateMutableBinding 具体方法。
    env.CreateMutableBinding(dn, configurableBindings);
    //以 dn、undefined 和 strict 为参数，调用 env 的 SetMutableBinding 具体方法。
    env.SetMutableBinding(dn, undefined, strict);
}

//开始执行
env.SetMutableBinding(dn, 2);

+//按源码顺序遍历 code，对于每一个 FunctionDeclaration 表达式 f：
+//令 fn 为 FunctionDeclaration 表达式 f 中的 标识符
+fn = 'two';
+//创建函数对象
+//指定 FormalParameterList 为可选参数列表
+FormalParameterList = [];
+//指定 FunctionBody 为函数体
+FunctionBody = `
+    console.log(a, b, c);
+`;
+//指定 Scope 为 词法环境
+Scope = ECStack.current.lexicalEnvironment;
+//按 第 13 章 中所述的步骤初始化 FunctionDeclaration 表达式 ，并令 fo 为初始化的结果。
+fo = FunctionDeclaration.createInstance(fn, FormalParameterList, FunctionBody, Scope);
+//退出one
+ECStack.pop();
+env = ECStack.current.lexicalEnvironment.environmentRecord;
+//以 fn 为参数，调用 env 的 HasBinding 具体方法，并令 argAlreadyDeclared 为调用的结果。
+argAlreadyDeclared = env.HasBinding(fn);
+if (!argAlreadyDeclared) {
+    env.CreateMutableBinding(fn, configurableBindings);
+} else {
+    //否则如果 env 是全局环境的 环境记录 对象，则
+    //令 go 为全局对象。
+    let go = global;
+    //以 fn 为参数，调用 go 和 [[GetProperty]] 内部方法，并令 existingProp 为调用的结果。
+    let existingProp = Object.getOwnPropertyDescriptor(go, fn);
+    if (existingProp.configurable) {
+        Object.defineProperty(go, fn, { value: undefined, writable: true, enumerable: true, configurable: +configurableBindings });
+        if (!existingProp.writable) {
+            throw new Error(`TypeError`);
+        }
+    }
+}
+env.SetMutableBinding(fn, fo, strict);
+//执行two
+//当控制流根据一个函数对象 F、调用者提供的 thisArg 以及调用者提供的 argumentList，进入 函数代码 的执行环境时，执行以下步+骤：
+thisArg = global;
+//以 F 的 [[Scope]] 内部属性为参数调用 NewDeclarativeEnvironment，并令 localEnv 为调用的结果
+localEnv = LexicalEnvironment.NewDeclarativeEnvironment(fo[`[[Scope]]`]);
+//设词法环境为 localEnv 设变量环境为 localEnv
+oneExecutionContext = new ExecutionContext(localEnv, thisArg);
+ECStack.push(oneExecutionContext);
+//按 [10.5](#10.5) 描述的方案，使用 函数代码 code 和 argumentList 执行定义绑定初始化步骤
+env = ECStack.current.lexicalEnvironment.environmentRecord;
+configurableBindings = false;
+strict = false;
+//令 func 为通过 [[Call]] 内部属性初始化 code 的执行的函数对象
+func = fo;
+//令 code 为 F 的 [[Code]] 内部属性的值。
+code = func[`[[Code]]`];
+//令 names 为 func 的 [[FormalParameters]] 内部属性。
+names = func[`[[FormalParameters]]`];
+args = [3];
+//令 argCount 为 args 中元素的数量
+argCount = args.length;
+n = 0;
+names.forEach(argName => {
+    n += 1;
+    let v = n > argCount ? undefined : args[n - 1];
+    //以 argName 为参数，调用 env 的 HasBinding 具体方法，并令 argAlreadyDeclared 为调用的结果。
+    argAlreadyDeclared = env.HasBinding(argName);
+    if (!argAlreadyDeclared) {
+        env.CreateMutableBinding(argName);
+    }
+    env.SetMutableBinding(argName, v, strict);
+});
+argumentsAlreadyDeclared = env.HasBinding('arguments');
+if (!argumentsAlreadyDeclared) {
+    let argsObj = CreateArgumentsObject(func, names, args, env, strict);
+    env.CreateMutableBinding('arguments');
+    env.SetMutableBinding('arguments', argsObj);
+}
+console.log(
+    'a=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'a').name,
+    'b=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'b').name,
+    'c=' + LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'c').name
+);
```

10. 可执行代码与执行环境

10.1 可执行代码类型

10.1.1 全局代码

- 全局代码 是指被作为 ECMA 脚本 程序 处理的源代码文本

```
console.log('global');
```

10.1.2 Eval 代码

- Eval 代码 是指提供给 eval 内置函数的源代码文本

```
eval('console.log("hello")');
```

10.1.3 函数代码

- 函数代码 是指作为 函数体 被解析的源代码文本
- 函数代码 同时还特指 以构造器方式调用 Function 内置对象 时所提供的源代码文本

```
function one() {
    console.log('one');
}
one();

const sum = new Function("a,b", "return a+b");
console.log(sum(1, 2));
```

10.2 词法环境

- 词法环境 是一个用于定义特定变量和函数标识符在 ECMAScript 代码的词法嵌套结构上关联关系的规范类型
- 一个词法环境由一个环境记录项和可能为空的外部词法环境引用构成
- 通常词法环境会与特定的 ECMAScript 代码诸如 `FunctionDeclaration`, `WithStatement` 或者 `TryStatement` 的 `Catch` 块这样的语法结构相联系，且类似代码每次执行都会有一个新的语法环境被创建出来
- 环境记录项记录了在它的关联词法环境域内创建的标识符绑定情形
- 外部词法环境引用用于表示词法环境的逻辑嵌套关系模型
- （内部）词法环境的外部引用是逻辑上包含内部词法环境的词法环境
- 外部词法环境自然也可能有多个内部词法环境
- 词法环境和环境记录项是纯粹的规范机制，而不需要 ECMAScript 的实现保持一致。ECMAScript 程序不可能直接访问或者更改这些值

```
let a = 1;
function one() {
    let a = 2;
    console.log(a);
    try {
        let a = 3;
        console.log(a);
        throw new Error('wrong');
    } catch (error) {
        let a = 4;
        console.log(a);
        with ({}) {
            let a = 5;
            console.log(a);
        }
    }
}
one();
```

10.2.1 环境记录项

- 在本标准中，共有 2 类环境记录项： `&#x58F0;&#x660E;&#x5F0F;&#x73AF;&#x5883;&#x8BB0;&#x5F55;&#x9879;` 和 `&#x5BF9;&#x8C61;&#x5F0F;&#x73AF;&#x5883;&#x8BB0;&#x5F55;&#x9879;`
- 声明式环境记录项用于定义那些将 标识符 与语言值直接绑定的 ECMA 脚本语法元素，例如 函数定义， 变量定义 以及 Catch 语句
- 对象式环境记录项用于定义那些将 标识符 与具体对象的属性绑定的 ECMA 脚本元素，例如 程序 以及 With 表达式
- 出于标准规范的目的，可以将环境记录项理解为面向对象中的一个简单继承结构，其中环境记录项是一个抽象类,它有 2 个具体实现类，分别为声明式环境记录项和对象式环境记录项

10.2.1.1 声明式环境记录项

- 每个声明式环境记录项都与一个包含变量和（或）函数声明的 ECMA 脚本的程序作用域相关联
- 声明式环境记录项用于绑定作用域内定义的一系列标识符
- 除了所有环境记录项都支持的可变绑定外，声明式环境记录项还提供不可变绑定
- 在不可变绑定中，一个标识符与它的值之间的关联关系建立之后，就无法改变
- 创建和初始化不可变绑定是两个独立的过程，因此类似的绑定可以处在已初始化阶段或者未初始化阶段

10.2.1.2 对象式环境记录项

- 每一个对象式环境记录项都有一个关联的对象，这个对象被称作 绑定对象
- 对象式环境记录项直接将一系列标识符与其绑定对象的属性名称建立一一对应关系
- 不符合IdentifierName 的属性名不会作为绑定的标识符使用
- 无论是对象自身的，还是继承的属性都会作为绑定,无论该属性的 [[Enumerable]] 特性的值是什么
- 由于对象的属性可以动态的增减，因此对象式环境记录项所绑定的标识符集合也会隐匿地变化，这是增减绑定对象的属性而产生的副作用
- 通过以上描述的副作用而建立的绑定，均被视为可变绑定，即使该绑定对应的属性的 Writable 特性的值为 false
- 对象式环境记录项没有不可变绑定

10.2.2 词法环境的运算

- GetIdentifierReference (lex, name, strict)
- NewDeclarativeEnvironment (E)
- NewObjectEnvironment (O, E)

10.2.3 全局环境

- 全局环境 是一个唯一的词法环境 ，它在任何 ECMA 脚本的代码执行前创建
- 全局环境的 环境数据 是一个 `object-environment-record`对象环境数据
- 该环境数据使用 全局对象作为 绑定对象
- 全局环境的 外部环境引用 为 null
- 在 ECMA 脚本的代码执行过程中，可能会向 全局对象 添加额外的属性，也可能修改其初始属性的值

10.2.3.1 全局对象

- 唯一的全局对象建立在控制进入任何执行环境之前
- 全局对象的标准内置属性拥有特性 {[[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}
- 全局对象没有 [[Construct]] 内部属性 ; 全局对象不可能当做构造器用 new 运算符调用
- 全局对象没有 [[Call]] 内部属性，全局对象不可能当做函数来调用
- 全局对象的 [[Prototype]] 和 [[Class]] 内部属性值是依赖于实现的
- 除了本规范定义的属性之外，全局对象还可以拥有额外的宿主定义的属性 全局对象可包含一个值是全局对象自身的属性；例如，在 HTML 文档对象模型中全局对象的 window 属性是全局对象自身

10.3 执行环境

- 当控制器转入 ECMA 脚本的可执行代码时，控制器会进入一个执行环境
- 当前活动的多个执行环境在逻辑上形成一个栈结构
- 该逻辑栈的最顶层的执行环境称为当前运行的执行环境
- 任何时候，当控制器从当前运行的执行环境相关的可执行代码转入与该执行环境无关的可执行代码时，会创建一个新的执行环境
- 新建的这个执行环境会推入栈中，成为当前运行的执行环境
- 执行环境包含所有用于追踪与其相关的代码的执行进度的状态

组件 作用目的 词法环境 指定一个词法环境对象，用于解析该执行环境内的代码创建的标识符引用 变量环境 指定一个词法环境对象，其环境数据用于保存由该执行环境内的代码通过变量表达式和函数表达式创建的绑定 This绑定 指定该执行环境内的 ECMA 脚本代码中 this 关键字所关联的值

- 执行环境的词法环境和变量环境组件始终为 词法环境 对象
- 当创建一个执行环境时，其词法环境组件和变量环境组件最初是同一个值
- 在该执行环境相关联的代码的执行过程中，变量环境组件永远不变，而词法环境组件有可能改变
- 在本标准中，通常情况下，只有正在运行的执行环境（执行环境栈里的最顶层对象）会被算法直接修改
- 因此当遇到 词法环境， 变量环境和 This 绑定这三个术语时，指的是正在运行的执行环境的对应组件
- 执行环境是一个纯粹的标准机制，并不代表任何 ECMA脚本实现的工件。在 ECMA脚本程序中是不可能访问到执行环境的

10.3.1 标识符解析

- 标识符解析是指使用正在运行的执行环境中的词法环境,通过一个标识符获得其对应的绑定的过程
- 解释执行一个标识符得到的结果必定是 引用 类型的对象，且其引用名属性的值与 Identifier字符串相等

10.4 建立执行环境

- 解释执行全局代码或使用 eval函数输入的代码会创建并进入一个新的执行环境
- 每次调用 ECMA 脚本代码定义的函数也会建立并进入一个新的执行环境，即便函数是自身递归调用的
- 每一次 return 都会退出一个执行环境
- 抛出异常也可退出一个或多个执行环境
- 当控制流进入一个执行环境时，会设置该执行环境的 this绑定，定义变量环境和初始词法环境，并执行定义绑定初始化过程

10.4.1 进入全局代码

- 按 10.4.1.1 描述的方案，使用 全局代码 初始化执行环境
- 按 10.5 描述的方案，使用 全局代码 执行定义绑定初始化步骤。

10.4.1.1 初始化全局执行环境

- 将变量环境设置为 全局环境
- 将词法环境设置为 全局环境
- 将 this 绑定设置为 全局对象

10.5 定义绑定初始化

- 每个执行环境都有一个关联的变量环境
- 当在一个执行环境下执行一段ECMA脚本时，变量和函数定义会以绑定的形式添加到这个变量环境的环境记录中
- 对于函数函数代码，参数也同样会以绑定的形式添加到这个变量环境的环境记录中
- 选择使用哪一个、哪一类型的 环境记录 来绑定定义，是由执行环境下执行的 ECMA 脚本的类型决定的