link: null
title: 珠峰架构师成长计划
description: src\Reference.js
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=115 sentences=303, words=2886

1.ECMAScript 的类型

- ECMAScript 的类型分为语言类型和规范类型
- ECMAScript 语言类型是开发者直接使用 ECMAScript 可以操作的。其实就是我们常说的Undefined, Null, Boolean, String, Number, 和 Object
- 而规范类型相当于 `meta-values`，是用来用算法描述 ECMAScript 语言结构和 ECMAScript 语言类型的,包括 `Reference`、 `Lexical Environment` 和 `Environment Record`

2.函数调用

src\Reference.js

```
const EnvironmentRecord = require('./EnvironmentRecord');
class Reference {
    constructor(base, name, strict) {
        this.base = base;
        this.name = name;
        this.strict = strict;
    }
    static GetBase(V) {
        return V.base;
    }
    static GetReferencedName(V) {
        return V.name;
    }
    static IsStrictReference(V) {
        return V.strict;
    }
    static HasPrimitiveBase(V) {
        return V.base.toString() === `[Object Boolean]`
            || V.base.toString() === `[Object String]`
            || V.base.toString() === `[Object Number]`
    }
    static IsPropertyReference(V) {
        return (typeof V.base === 'object' && (!(V.base instanceof EnvironmentRecord))) || Reference.HasPrimitiveBase(V)
    }
    static IsUnresolvableReference(V) {
        return typeof V.base === 'undefined';
    }
    static GetValue(V) {

        if (!Reference.Type(V)) return V;

        let base = Reference.GetBase(V);

        if (Reference.IsUnresolvableReference(V)) {
            throw new Error(`ReferenceError ${V} is not defined`);
        }

        if (Reference.IsPropertyReference(V)) {

            if (!Reference.HasPrimitiveBase(V)) {
                let name = Reference.GetReferencedName(V);
                return base[name];
            } else {
                let O = Reference.ToObject(base);
                let name = Reference.GetReferencedName(V);
                let desc = Object.getOwnPropertyDescriptor(O, name);
                if (typeof desc === 'undefined') return undefined;
                if (Reference.IsDataDescriptor(desc)) {
                    return desc.value;
                } else if (IsAccessorDescriptor(desc)) {
                    let getter = desc.get;
                    if (typeof getter === 'undefined') return undefined;
                    return getter.call(base);
                }
            }
        } else {

            let name = Reference.GetReferencedName(V);
            let strict = Reference.IsStrictReference(V);
            base.GetBindingValue(name, strict);
        }
        return base(Reference.GetReferencedName(V));
    }
    static IsDataDescriptor(desc) {
        return desc.value || desc.writable;
    }
    static IsAccessorDescriptor(desc) {
        return desc.get || desc.set;
    }
    static ToObject(PrimitiveValue) {
        let type = Object.prototype.toString.call(PrimitiveValue).slice(8, -1);
        switch (type) {
            case 'Boolean':
                return new Boolean(PrimitiveValue);;
            case 'Number':
                return new Number(PrimitiveValue);
            case 'String':
                return new String(PrimitiveValue);
            default:
                break;
        }
    }
    static Type(V) {
        return V instanceof Reference;
    }
}
module.exports = Reference;
```

src\3.this.js

```
var name = 1;
var obj = {
    name: 2,
    getName: function () {
        return this.name;
    }
}
console.log(obj.getName());

const Reference = require('./Reference');
let ref = new Reference(obj, 'getName', false);

let func = Reference.GetValue(ref);

let argList = [];

let thisValue;
if (Reference.Type(ref)) {
    if (Reference.IsPropertyReference(ref)) {
        thisValue = Reference.GetBase(ref);
    } else {
        thisValue = Reference.GetBase(ref).ImplicitThisValue();
    }
} else {

    thisValue = undefined;
}

let result = func.call(thisValue, ...argList);
console.log(result);
```

3.[[Call]]

- 当用一个 this 值，一个参数列表调用函数对象 F 的 [[Call]] 内部方法，采用以下步骤：
    - 用 F 的 [[FormalParameters]] 内部属性值，参数列表 args，10.4.3 描述的 this 值来建立 函数代码 的一个新执行环境，令 funcCtx 为其结果。
    - 令 result 为 FunctionBody（也就是 F 的 [[Code]] 内部属性）解释执行的结果。如果 F 没有 [[Code]] 内部属性或其值是空的 FunctionBody，则 result 是 (normal, undefined, empty)。
    - 退出 funcCtx 执行环境，恢复到之前的执行环境。
    - 如果 result.type 是 throw 则抛出 result.value。
    - 如果 result.type 是 return 则返回 result.value。
    - 否则 result.type 必定是 normal。返回 undefined。

```
const ExecutionContext = require('./ExecutionContext');
const LexicalEnvironment = require('./LexicalEnvironment');
const FunctionDeclaration = require('./FunctionDeclaration');
const Reference = require('./Reference');
const ECStack = require('./ECStack');
function sum(a, b) {
    console.log(this, a, b);
}

let fn = 'sum';
let FormalParameterList = "a,b";
let FunctionBody = `
   console.log(this, a, b);
`;
const globalLexicalEnvironment = LexicalEnvironment.NewObjectEnvironment(global, null);
const globalEC = new ExecutionContext(globalLexicalEnvironment, global);
ECStack.push(globalEC);
let env = ECStack.current.lexicalEnvironment.environmentRecord;
let Scope = ECStack.current.lexicalEnvironment;
let fo = FunctionDeclaration.createInstance(fn, FormalParameterList, FunctionBody, Scope);
let ref = new Reference(env, 'sum', false);
let thisValue;
if (Reference.Type(ref)) {
    if (Reference.IsPropertyReference(ref)) {
        thisValue = Reference.GetBase(ref);
    } else {
        thisValue = Reference.GetBase(ref).ImplicitThisValue();
    }
} else {

    thisValue = undefined;
}
let FormalParameters = FormalParameterList.split(',');
let localEnv = LexicalEnvironment.NewDeclarativeEnvironment(fo[`[[Scope]]`]);
let funcCtx = new ExecutionContext(localEnv, thisValue);
ECStack.push(funcCtx);
env = ECStack.current.lexicalEnvironment.environmentRecord;
let strict = false;
let args = [1, 2];

let argCount = args.length;
let n = 0;
FormalParameters.forEach(argName => {
    n += 1;
    let v = n > argCount ? undefined : args[n - 1];
    argAlreadyDeclared = env.HasBinding(argName);
    if (!argAlreadyDeclared) {
        env.CreateMutableBinding(argName);
    }
    env.SetMutableBinding(argName, v, strict);
});
console.log(
    thisValue,
    Reference.GetValue(
        LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'a')),
    Reference.GetValue(
        LexicalEnvironment.GetIdentifierReference(ECStack.current.lexicalEnvironment, 'b'))
);
ECStack.pop();
```

4. call

```
function sum(a, b) {
    console.log(this, a, b);
}
let thisArg = { name: 'obj' };
sum.call(thisArg, 1, 2);

let func = sum;

let argList = [];

argList.push(1);
argList.push(2);

console.log(
    thisArg
);
```

5. apply

```
const { Type } = require('./utils');
function func(a, b) {
    console.log(this, a, b);
}
let thisArg = { name: 'obj' };
func.apply(thisArg, [1, 2]);

let argArray = [1, 2];
if (!argArray) {
    console.log(thisArg, undefined, undefined);
}

if (Type(argArray) !== 'object') {
    throw new Error(`TypeError`);
}

let len = argArray.length;

let n = len;

let argList = [];
let index = 0;
while (index < n) {
    let indexName = index.toString();
    let nextArg = argArray[indexName];
    argList.push(nextArg);
    index = index + 1;
}
```

6. bind

```
function func(a, b) {
    console.log(this, a, b);
}
let thisArg = { name: 'obj' };
let newFunc = func.bind(thisArg, 1);
newFunc(2);

const { IsCallable } = require('./utils');

let Target = thisArg;

func[`[[call]]`] = true;
if (!IsCallable(func)) {
    throw new Error(`TypeError`);
}

let A = [1];

let F = {};

F[`[[TargetFunction]]`] = Target;

F[`[[BoundThis]]`] = thisArg;

F[`[[BoundArgs]]`] = A;

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

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

if (Target[`[[Class]]`] === 'Function') {
    let L = Target.length - A.length;
    F.length = Math.max(0, L);
} else {

    F.length = 0;
}

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

let thrower = (key, value) => { };

Object.defineProperty(F, 'caller', {
    get: thrower,
    set: thrower,
    enumerable: false,
    configurable: false
});

Object.defineProperty(F, 'arguments', {
    get: thrower,
    set: thrower,
    enumerable: false,
    configurable: false
});

let boundArgs = F[`[[BoundArgs]]`];
let boundThis = F[`[[BoundThis]]`];
let target = F[`[[TargetFunction]]`];

let ExtraArgs = [2];
let args = [...boundArgs, ...ExtraArgs];

console.log(
    boundThis,
    ...args
);
```

7. call/apply/bind

- call/apply/bind可以改变函数中this的指向
- 第一个参数是改变this指向(非严格模式下，传递null/undefined指向也是window)
- call参数是依次传递，apply是以数组的方式传递

```
!function (proto) {
    function getContext(context) {
        context = context || window;
        var type = typeof context;
        if (['number', 'string', 'boolean', 'null'].includes(type)) {
            context = new context.constructor(context);
        }
        return context;
    }
    function call(context, ...args) {
        context = getContext(context);
        context._fn = this;
        let result = context._fn(...args);
        delete context._fn;
        return result;
    }
    function apply(context, args) {
        context = getContext(context);
        context._fn = this;
        let result = context._fn(...args);
        delete context._fn;
        return result;
    }

    function bind(context, ...bindArgs) {
        return (...args) => this.call(context, ...bindArgs, ...args);
    }
    proto.call = call;
    proto.apply = apply;
    proto.bind = bind;
}(Function.prototype)
```

- 引用类型用来说明 delete，typeof，赋值运算符这些运算符的行为
- 例如，在赋值运算中左边的操作数期望产生一个引用
- 一个 引用 (Reference) 是个已解决的命名绑定
- 一个引用由三部分组成， 基 (base) 值， 引用名称（referenced name） 和布尔值 严格引用 (strict reference) 标志
- 基值是 undefined, 一个 Object, 一个 Boolean, 一个 String, 一个 Number, 一个 environment record 中的任意一个
- 基值是 undefined 表示此引用可以不解决一个绑定
- 引用名称是一个字符串

```
var a = 1;
var aReference = {
    base: environmentRecords,
    name: 'a',
    strict: false
}

var obj = {
    name: 'zhufeng',
    getName() {
        console.log(this.name);
    }
}
var getNameReference = {
    base: obj,
    name: 'getName',
    strict: false
};

function two() {
    console.log(this);
}
two();
var twoReference = {
    base: environmentRecords,
    name: 'two',
    strict: false
};
```

- 本规范中使用以下抽象操作访问引用的组件

方法 含义 GetBase(V) 返回引用值 V 的基值组件 GetReferencedName(V) 返回引用值 V 的引用名称组件 IsStrictReference(V) 返回引用值 V 的严格引用组件 HasPrimitiveBase(V) 如果基值是 Boolean, String, Number，那么返回 true IsPropertyReference(V) 如果基值是个对象或 HasPrimitiveBase(V) 是 true，那么返回 true；否则返回 false IsUnresolvableReference(V) 如果基值是 undefined 那么返回 true，否则返回 false

- 如果 Type(V) 不是引用 , 返回 V。
- 令 base 为调用 GetBase(V) 的返回值。
- 如果 IsUnresolvableReference(V), 抛出一个 ReferenceError 异常。
- 如果 IsPropertyReference(V), 那么
    - 如果 HasPrimitiveBase(V) 是 false, 那么令 get 为 base 的 [[Get]] 内部方法 , 否则令 get 为下面定义的特殊的 [[Get]] 内部方法。
    - 将 base 作为 this 值，传递 GetReferencedName(V) 为参数，调用 get 内部方法，返回结果。
- 否则 , base 必须是一个 environment record。
- 传递 GetReferencedName(V) 和 IsStrictReference(V) 为参数调用 base 的 GetBindingValue(见 10.2.1) 具体方法，返回结果。
- GetValue 中的 V 是原始基值的 属性引用 时使用下面的 [[Get]] 内部方法。它用 base 作为他的 this 值，其中属性 P 是它的参数。采用以下步骤： -令 O 为 ToObject(base)。 -令 desc 为用属性名 P 调用 O 的 [[GetProperty]] 内部方法的返回值。 -如果 desc 是 undefined，返回 undefined。 -如果 IsDataDescriptor(desc) 是 true，返回 desc.[[Value]]。 -否则 IsAccessorDescriptor(desc) 必须是 true，令 getter 为 desc.[[Get]]。 -如果 getter 是 undefined，返回 undefined。 -提供 base 作为 this 值，无参数形式调用 getter 的 [[Call]] 内部方法，返回结果。
- 如果 Type(V) 不是引用，抛出一个 ReferenceError 异常。
- 令 base 为调用 GetBase(V) 的结果。
- 如果 IsUnresolvableReference(V)，那么
    - 如果 IsStrictReference(V) 是 true，那么抛出 ReferenceError 异常。
    - 用 GetReferencedName(V)，W，false 作为参数调用全局对象的 [[Put]] 内部方法。
- 否则如果 IsPropertyReference(V)，那么
    - 如果 HasPrimitiveBase(V) 是 false，那么令 put 为 base 的 [[Put]] 内部方法，否则令 put 为下面定义的特殊的 [[Put]] 内部方法。
    - 用 base 作为 this 值，用 GetReferencedName(V)，W，IsStrictReference(V) 作为参数调用 put 内部方法。
- 否则 base 必定是 environment record 作为 base 的引用。所以， 用 GetReferencedName(V), W, IsStrictReference(V) 作为参数调用 base 的 SetMutableBinding (10.2.1) 具体方法。 返回。

- PutValue 中的 V 是原始基值的属性引用时使用下面的 [[Put]] 内部方法。用 base 作为 this 值，用属性 P，值 W，布尔标志 Throw 作为参数调用它。采用以下步骤：
- 令 O 为 ToObject(base)。
- 如果用 P 作为参数调用 O 的 [[CanPut]] 内部方法的结果是 false，那么
    - 如果 Throw 是 true，那么抛出一个 TypeError 异常。
    - 否则返回。
- 令 ownDesc 为用 P 作为参数调用 O 的 [[GetOwnProperty]] 内部方法的结果。
- 如果 IsDataDescriptor(ownDesc) 是 true，那么
    - 如果 Throw 是 true，那么抛出一个 TypeError 异常。
    - 否则返回。
- 令 desc 为用 P 作为参数调用 O 的 [[GetProperty]] 内部方法的结果。这可能是一个自身或继承的访问器属性描述或是一个继承的数据属性描述。
- 如果 IsAccessorDescriptor(desc) 是 true，那么
    - 令 setter 为 desc.Set，他不能是 undefined。
    - 用 base 作为 this 值，用只由 W 组成的列表作为参数调用 setter 的 [[Call]] 内部方法。
- 否则，这是要在临时对象 O 上创建自身属性的请求。
    - 如果 Throw 是 true，抛出一个 TypeError 异常。
- 返回。
- 当用属性描述 Desc 调用抽象操作 IsAccessorDescriptor，采用以下步骤：
    - 如果 Desc 是 undefined，那么返回 false
    - 如果 Desc.[[Get]] 和 Desc.[[Set]] 都不存在，则返回 false
    - 返回 true
- 当用属性描述 Desc 调用抽象操作 IsDataDescriptor，采用以下步骤：
    - 如果 Desc 是 undefined，那么返回 false。
    - 如果 Desc.[[Value]] 和 Desc.[[Writable]] 都不存在，则返回 false。
    - 返回 true

当用属性名 P 调用 O 的 [[Get]] 内部方法，采用以下步骤：

- 令 desc 为用属性名 P 调用 O 的 [[GetProperty]] 内部方法的结果。
- 如果 desc 是 undefined，返回 undefined。
- 如果 IsDataDescriptor(desc) 是 true，返回 desc.[[Value]]。
- 否则，IsAccessorDescriptor(desc) 必定是真，所以，令 getter 为 desc.[[Get]]。
- 如果 getter 是 undefined，返回 undefined。
- 用 O 作为 this，无参数调用 getter 的 [[Call]] 内部方法，返回结果。
- ToObject 运算符根据下表将其参数转换为对象类型的值：

输入类型 结果 Undefined 抛出 TypeError 异常。 Null 抛出 TypeError 异常。 Boolean 创建一个新的Boolean对象，其 [[PrimitiveValue]]属性被设为该布尔值的值。 Number 创建一个新的Number对象，其 [[PrimitiveValue]]属性被设为该数字值。 String 创建一个新的String对象，其 [[PrimitiveValue]] 属性被设为该字符串值。 Object 结果是输入的参数（不转换）。

- 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。在 ECMA 脚本代码执行过程中，PrimaryExpression : Identifier 这一语法产生式将按以下算法进行解释执行： -令 env 为正在运行的执行环境的 词法环境 。 -如果正在解释执行的语法产生式处在 严格模式下的代码 中，则仅 strict 的值为 true，否则令 strict 的值为 false。 -以 env，Identifier 和 strict 为参数，调用 GetIdentifierReference 函数，并返回调用的结果。
    - 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。
- 产生式 PrimaryExpression : (Expression) 按照下面的过程执行 :
- 返回执行Expression的结果，它可能是Reference类型。

> 这一算法并不会作用GetValue于执行Expression的结果。这样做的原则是确保delete和typeof这样的运算符可以作用于括号括起来的表达式。

- 产生式 AssignmentExpression : LeftHandSideExpression = AssignmentExpression 按照下面的过程执行 :
- 令 lref 为解释执行 LeftH 和 SideExpression 的结果 .
- 令 rref 为解释执行 AssignmentExpression 的结果 .
- 令 rval 为 GetValue(rref).
- 抛出一个 SyntaxError 异常，当以下条件都成立 :
    - Type(lref) 为 Reference
    - IsStrictReference(lref) 为 true
    - Type(GetBase(lref)) 为环境记录项
    - GetReferencedName(lref) 为 "eval" 或 "arguments"
- 调用 PutValue(lref, rval).
- 返回 rval.
- 产生式 CallExpression : MemberExpression Arguments 按照下面的过程执行 :
    - 令 ref 为解释执行 MemberExpression 的结果 .
    - 令 func 为 GetValue(ref).
    - 令 argList 为解释执行 Arguments 的结果 , 产生参数值们的内部列表 (see 11.2.4).
    - 如果 Type(func) is not Object ， 抛出一个 TypeError 异常 .
    - 如果 IsCallable(func) is false ， 抛出一个 TypeError 异常 .
    - 如果 Type(ref) 为 Reference，那么 如果 IsPropertyReference(ref) 为 true，那么 令 thisValue 为 GetBase - (ref). 否则 , ref 的基值是一个环境记录项 令 thisValue 为调用 GetBase(ref) 的 ImplicitThisValue 具体方法的结 - 果
    - 否则 , 假如 Type(ref) 不是 Reference. 令 thisValue 为 undefined.
    - 返回调用 func 的 [[Call]] 内置方法的结果 , 传入 thisValue 作为 this 值和列表 argList 作为参数列表
- 产生式 CallExpression : CallExpression Arguments以完全相同的方式执行，除了第1步执行的是其中的CallExpression。
- 返回根据表 20 由 Type(val) 决定的字符串。

val类型 结果 Undefined "undefined" Null "null" Boolean "boolean" Number "number" String "string" Object（原生，且没有实现 [[call]]） "object" Object（原生或者宿主且实现了 [[call]] "function" Object（宿主且没实现 [[call]]） 由实现定义，但不能是 "undefined", "boolean", "number", or "string"

- 产生式 LogicalANDExpression : LogicalANDExpression && BitwiseORExpression 按照下面的过程执行 :
    - 令 lref 为解释执行 LogicalANDExpression 的结果 .
    - 令 lval 为 GetValue(lref).
    - 如果 ToBoolean(lval) 为 false ，返回 lval.
    - 令 rref 为解释执行 BitwiseORExpression 的结果 .
    - 返回 GetValue(rref).
- 产生式 Expression : Expression , AssignmentExpression 按照下面的过程执行 :
    - 令 lref 为解释执行 Expression 的结果
    - Call GetValue(lref).
    - 令 rref 为解释执行 AssignmentExpression 的结果 .
    - 返回 GetValue(rref).
- 当用一个 this 值，一个参数列表调用函数对象 F 的 [[Call]] 内部方法，采用以下步骤：
- 用 F 的 [[FormalParameters]] 内部属性值，参数列表 args，10.4.3 描述的 this 值来建立 函数代码 的一个新执行环- 境，令 funcCtx 为其结果。
- 令 result 为 FunctionBody（也就是 F 的 [[Code]] 内部属性）解释执行的结果。如果 F 没有 [[Code]] 内部属性或其- 值是空的 FunctionBody，则 result 是 (normal, undefined, empty)。
- 退出 funcCtx 执行环境，恢复到之前的执行环境。
- 如果 result.type 是 throw 则抛出 result.value。
- 如果 result.type 是 return 则返回 result.value。
- 否则 result.type 必定是 normal。返回 undefined。
- 当以 thisArg 和 argArray 为参数在一个 func 对象上调用 apply 方法，采用如下步骤：
- 如果 IsCallable(func) 是 false, 则抛出一个 TypeError 异常 .
- 如果 argArray 是 null 或 undefined, 则
    - 返回提供 thisArg 作为 this 值并以空参数列表调用 func 的 [[Call]] 内部方法的结果。
- 如果 Type(argArray) 不是 Object, 则抛出一个 TypeError 异常 .
- 令 len 为以 "length" 作为参数调用 argArray 的 [[Get]] 内部方法的结果。
- 令 n 为 ToUint32(len).
- 令 argList 为一个空列表 .
- 令 index 为 0.
- 只要 index < n 就重复
    - 令 indexName 为 ToString(index).
    - 令 nextArg 为以 indexName 作为参数调用 argArray 的 [[Get]] 内部方法的结果。
    - 将 nextArg 作为最后一个元素插入到 argList 里。
    - 设定 index 为 index + 1.
- 提供 thisArg 作为 this 值并以 argList 作为参数列表，调用 func 的 [[Call]] 内部方法，返回结果。
- apply 方法的 length 属性是 2。

> 在外面传入的 thisArg 值会修改并成为 this 值。thisArg 是 undefined 或 null 时它会被替换成全局对象，所有其他值会被应用 ToObject 并将结果作为 this 值，这是第三版引入的更改。

- 当以 thisArg 和可选的 arg1, arg2 等等作为参数在一个 func 对象上调用 call 方法，采用如下步骤：
- 如果 IsCallable(func) 是 false, 则抛出一个 TypeError 异常。
- 令 argList 为一个空列表。
- 如果调用这个方法的参数多余一个，则从 arg1 开始以从左到右的顺序将每个参数插入为 argList 的最后一个元素。
- 提供 thisArg 作为 this 值并以 argList 作为参数列表，调用 func 的 [[Call]] 内部方法，返回结果。
- call 方法的 length 属性是 1。

> 在外面传入的 thisArg 值会修改并成为 this 值。thisArg 是 undefined 或 null 时它会被替换成全局对象，所有其他值会被应用 ToObject 并将结果作为 this 值，这是第三版引入的更改。

- bind 方法需要一个或更多参数，thisArg 和（可选的）arg1, arg2, 等等，执行如下步骤返回一个新函数对象：
- 令 Target 为 this 值 .
- 如果 IsCallable(Target) 是 false, 抛出一个 TypeError 异常 .
- 令 A 为一个（可能为空的）新内部列表，它包含按顺序的 thisArg 后面的所有参数（arg1, arg2 等等）。
- 令 F 为一个新原生 ECMAScript 对象。
- 依照 8.12 指定，设定 F 的除了 [[Get]] 之外的所有内部方法。
- 依照 15.3.5.4 指定，设定 F 的 [[Get]] 内部属性。
- 设定 F 的 [[TargetFunction]] 内部属性为 Target。
- 设定 F 的 [[BoundThis]] 内部属性为 thisArg 的值。
- 设定 F 的 [[BoundArgs]] 内部属性为 A。
- 设定 F 的 [[Class]] 内部属性为 "Function"。
- 设定 F 的 [[Prototype]] 内部属性为 15.3.3.1 指定的标准内置 Function 的 prototype 对象。
- 依照 15.3.4.5.1 描述，设定 F 的 [[Call]] 内置属性。
- 依照 15.3.4.5.2 描述，设定 F 的 [[Construct]] 内置属性。
- 依照 15.3.4.5.3 描述，设定 F 的 [[HasInstance]] 内置属性。
- 如果 Target 的 [[Class]] 内部属性是 "Function", 则
    - 令 L 为 Target 的 length 属性减 A 的长度。
    - 设定 F 的 length 自身属性为 0 和 L 中更大的值。
- 否则设定 F 的 length 自身属性为 0.

- 设定 F 的 length 自身属性的特性为 15.3.5.1 指定的值。
- 设定 F 的 [[Extensible]] 内部属性为 true。
- 令 thrower 为 [[ThrowTypeError]] 函数对象 (13.2.3)。
- 以 "caller", 属性描述符 {[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: - false}, 和 false 作为参数调用 F 的 [[DefineOwnProperty]] 内部方法。
- 以 "arguments", 属性描述符 {[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}, 和 false 作为参数调用 F 的 [[DefineOwnProperty]] 内部方法。
- 返回 F.
- bind 方法的 length 属性是 1。

> Function.prototype.bind 创建的函数对象不包含 prototype 属性或 [[Code]], [[FormalParameters]], [[Scope]] 内部属性

- 当调用一个用 bind 函数创建的函数对象 F 的 [[Call]] 内部方法，传入一个 this 值和一个参数列表 ExtraArgs，采用如下步骤：
- 令 boundArgs 为 F 的 [[BoundArgs]] 内部属性值。
- 令 boundThis 为 F 的 [[BoundThis]] 内部属性值。
- 令 target 为 F 的 [[TargetFunction]] 内部属性值。
- 令 args 为一个新列表，它包含与列表 boundArgs 相同顺序相同值，后面跟着与 ExtraArgs 是相同顺序相同值。
- 提供 boundThis 作为 this 值，提供 args 为参数调用 target 的 [[Call]] 内部方法，返回结果。

3.面试题

```
var name = "window";
var obj = {
    name: "hello",
    getName() {
        console.log(this.name);
    }
};
function getName() {
    var objGetName = obj.getName;
    objGetName();
    obj.getName();
    (obj.getName)();
    (true && obj.getName)();
}
getName();
```

```
var name = 'window';
var obj = {
    name: 'obj',
    getName1() {
        console.log(this.name)
    },
    getName2: () => console.log(this.name),
    getName3() {
        return function () {
            console.log(this.name)
        }
    },
    getName4() {
        return () => {
            console.log(this.name)
        }
    }
}
var obj2 = { name: 'obj2' }
obj.getName1();
obj.getName1.call(obj2);
obj.getName2();
obj.getName2.call(obj2);
obj.getName3()();
obj.getName3.call(obj2)();
obj.getName3().call(obj2);
obj.getName4()();
obj.getName4.call(obj2)();
obj.getName4().call(obj2);
```