函数

递归

通常的形式是一个函数通过名称调用自己
可以用arguments.callee 实现解耦（严格模式报错）
严格模式可以使用命名函数表达式

```
const factorial = (function f(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * f(num - 1);
    }
});
```

尾调用优化

ES6新增内存管理优化机制

```
function outerFunction() {
 return innerFunction(); // 尾调用
}
```

优化之前

1. 执行到 outerFunction 函数体，第一个栈帧被推到栈上
2. 执行 outerFunction 函数体，到 return 语句。计算返回值必须先计算 innerFunction
3. 执行到 innerFunction 函数体，第二个栈帧被推到栈上
4. 执行 innerFunction 函数体，计算其返回值
5. 将返回值传回 outerFunction，然后 outerFunction 再返回值
6. 将栈帧弹出栈外

ES6 优化之后

1. 执行到 outerFunction 函数体，第一个栈帧被推到栈上
2. 执行 outerFunction 函数体，到达 return 语句。为求值返回语句，必须先求值 innerFunction
3. 引擎发现把第一个栈帧弹出栈外也没问题，因为 innerFunction 的返回值也是 outerFunction的返回值
4. 弹出 outerFunction 的栈帧
5. 执行到 innerFunction 函数体，栈帧被推到栈上
6. 执行 innerFunction 函数体，计算其返回值
7. 将 innerFunction 的栈帧弹出栈外

优化前每多调用一次嵌套函数，就会多增加一个栈帧
优化后无论调用多少次嵌套函数，都只有一个栈帧

尾调用优化的条件

- 代码在严格模式下执行
- 外部函数的返回值是对尾调用函数的调用
- 尾调用函数返回后不需要执行额外的逻辑（即，直接返回）
- 尾调用函数不是引用外部函数作用域中自由变量的闭包

```
"use strict";
// 无优化：尾调用没有返回
function outerFunction() {
    innerFunction();
}
// 无优化：尾调用没有直接返回
function outerFunction() {
    let innerFunctionResult = innerFunction();
    return innerFunctionResult;
}
// 无优化：尾调用返回后必须转型为字符串
function outerFunction() {
    return innerFunction().toString();
}
// 无优化：尾调用是一个闭包
function outerFunction() {
    let foo = 'bar';
    function innerFunction() { return foo; }
    return innerFunction();
}
```

尾调用优化的代码

```
function fib(n) {
    if (n < 2) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
console.log(fib(0)); // 0
console.log(fib(1)); // 1
console.log(fib(2)); // 1
console.log(fib(3)); // 2
console.log(fib(4)); // 3
console.log(fib(5)); // 5
console.log(fib(6)); // 8

// 优化后

"use strict";
// 基础框架
function fib(n) {
    return fibImpl(0, 1, n);
}
// 执行递归
function fibImpl(a, b, n) {
```

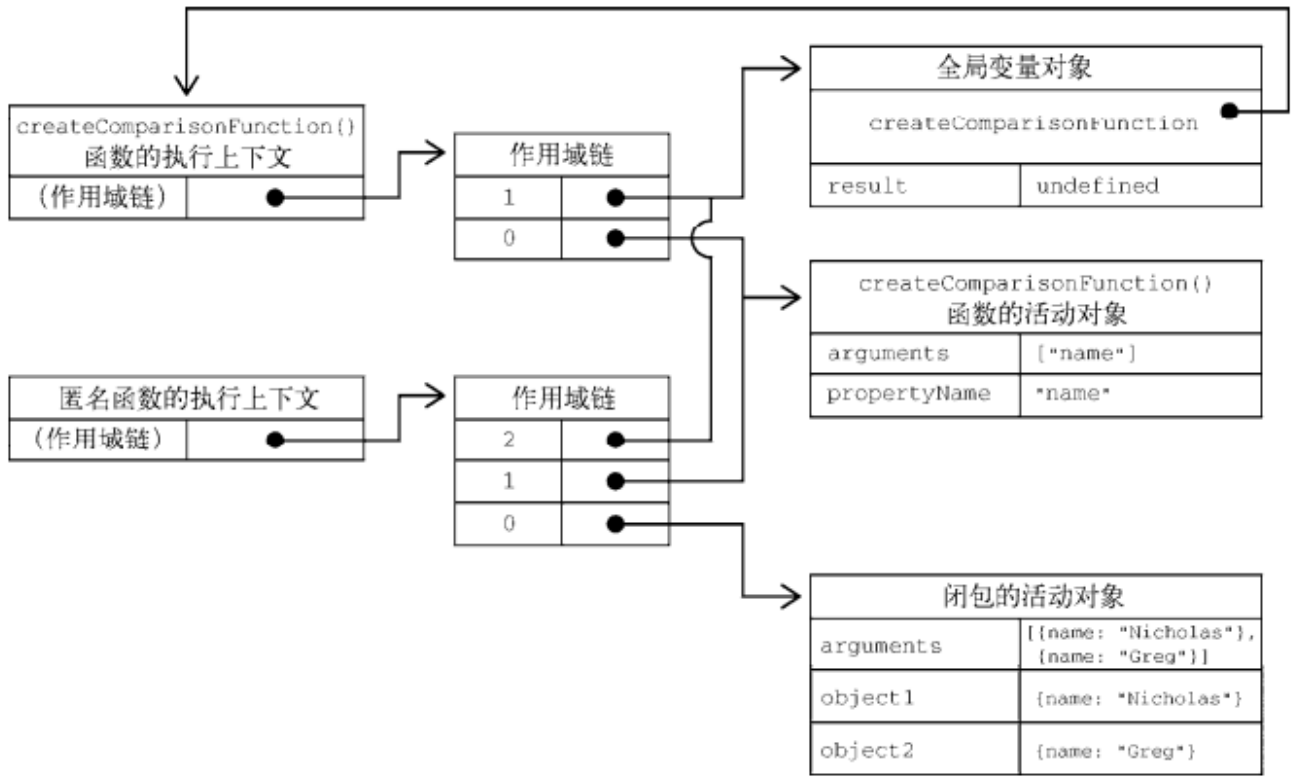
```
    if (n === 0) {
        return a;
    }
    return fibImpl(b, a + b, n - 1);
}
```

闭包

闭包 引用了另一个函数作用域中变量的函数

```
function createComparisonFunction(propertyName) {
    return function (object1, object2) {
        let value1 = object1[propertyName];
        let value2 = object2[propertyName];
        if (value1 < value2) {
            return -1;
        } else if (value1 > value2) {
            return 1;
        } else {
            return 0;
        }
    };
}

let compare = createComparisonFunction('name');
let result = compare({ name: 'Nicholas' }, { name: 'Matt' });
```



createComparisonFunction()的
活动对象并不能在它执行完毕后销毁，因为匿名函数的作用域链中仍然有对它的引用（引用计数），直到匿名函数被销毁后才会被销毁（设置为null）

```
// 创建比较函数
let compareNames = createComparisonFunction('name');
// 调用函数
let result = compareNames({ name: 'Nicholas' }, { name: 'Matt' });
// 解除对函数的引用，这样就可以释放内存了
compareNames = null;
```

this

闭包中没有使用箭头函数，this会在运行时绑定到执行函数的上下文
严格模式，全局 undefined
非严格模式，全局 window

非全局，指向使用的对象

```
identity = 'The Window';
let object = {
    identity: 'My Object',
    getIdentity() {
        return this.identity;
    }
};

console.log((object.getIdentity = object.getIdentity)());
```

赋值表达式，这算是普通函数调用而不是对象函数调用

内存泄漏

闭包
垃圾回收机制
在 IE9 之前JScrip对象，COM对象引用计数

立即调用的函数表达式

立即调用的匿名函数
模拟块级作用域
ES6没必要，因为有let，const

私有变量

JS没有私有成员的概念，所有属性都对外公开
私有变量：任何定义在函数或块中的变量
不包括块中的var

私有变量包括函数参数、局部变量，以及函数内部定义的其他函数

特权方法：能够访问函数私有变量（私有函数）的公有方法

实现方法：

- 构造函数中实现（闭包）

```
function MyObject() {
    // 私有变量和私有函数
    let privateVariable = 10;
    function privateFunction() {
        return false;
    }
    // 特权方法
    this.publicMethod = function () {
        privateVariable++;
        return privateFunction();
    };
}
```

- 静态私有变量
  私有作用域定义私有变量

```
(function () {
    let name = '';
    Person = function (value) {
        name = value;
    };
    Person.prototype.getName = function () {
        return name;
    };
    Person.prototype.setName = function (value) {
        name = value;
    };
})();
let person1 = new Person('Nicholas');
console.log(person1.getName()); // 'Nicholas'
person1.setName('Matt');
console.log(person1.getName()); // 'Matt'
let person2 = new Person('Michael');
console.log(person1.getName()); // 'Michael'
console.log(person2.getName()); // 'Michael'
```

模块模式

在一个单例对象上实现了相同的隔离和封装
JavaScript 是通过对象字面量来创建单例对象

```
let singleton = {
    name: value,
    method() {
        // 方法的代码
    }
};
```

模块模式是在单例对象基础上加以扩展，使其通过作用域链来关联私有变量和特权方法

```
let singleton = function () {
    // 私有变量和私有函数
    let privateVariable = 10;
    function privateFunction() {
        return false;
    }
    // 特权/公有方法和属性
    return {
        publicProperty: true,
        publicMethod() {
            privateVariable++;
            return privateFunction();
        }
    };
}();

// 如果需要初始化
let application = function () {
    // 私有变量和私有函数
    let components = new Array();
    // 初始化
    components.push(new BaseComponent());
    // 公共接口
    return {
        getComponentCount() {
            return components.length;
        },
        registerComponent(component) {
            if (typeof component == 'object') {
                components.push(component);
            }
        }
    };
}();
```

模块增强模式

在返回对象之前先对其进行增强
适合单例对象需要是某个特定类型的实例，但又必须给它添加额外属性或方法的场景

```
let singleton = function () {
    // 私有变量和私有函数
    let privateVariable = 10;
    function privateFunction() {
        return false;
    }
    // 创建对象
    let object = new CustomType();
    // 添加特权/公有属性和方法
    object.publicProperty = true;
    object.publicMethod = function () {
        privateVariable++;
        return privateFunction();
    };
    // 返回对象
```

```
    return object;
}();
```

小结

- 函数表达式与函数声明是不一样的。函数声明要求写出函数名称，而函数表达式并不需要。没有名称的函数表达式也被称为匿名函数
- ES6 新增了类似于函数表达式的箭头函数语法，但两者也有一些重要区别
- JavaScript 中函数定义与调用时的参数极其灵活。arguments 对象，以及 ES6 新增的扩展操作符，可以实现函数定义和调用的完全动态化
- 函数内部也暴露了很多对象和引用，涵盖了函数被谁调用、使用什么调用，以及调用时传入了什么参数等信息
- JavaScript 引擎可以优化符合尾调用条件的函数，以节省栈空间
- 闭包的作用域链中包含自己的一个变量对象，然后是包含函数的变量对象，直到全局上下文的变量对象
- 通常，函数作用域及其中的所有变量在函数执行完毕后都会被销毁
- 闭包在被函数返回之后，其作用域会一直保存在内存中，直到闭包被销毁
- 函数可以在创建之后立即调用，执行其中代码之后却不留下对函数的引用
- 立即调用的函数表达式如果不在包含作用域中将返回值赋给一个变量，则其包含的所有变量都会被销毁。
- 虽然 JavaScript 没有私有对象属性的概念，但可以使用闭包实现公共方法，访问位于包含作用域中定义的变量
- 可以访问私有变量的公共方法叫作特权方法
- 特权方法可以使用构造函数或原型模式通过自定义类型中实现，也可以使用模块模式或模块增强模式在单例对象上实现