函数

函数声明与函数表达式

JavaScript引擎加载数据区别对待
JS在执行任何代码前，先读取函数声明，并在执行上下文中生成函数定义
函数表达式必须等代码执行到它那一行才会在执行上下文中生成函数定义

先读取函数声明----->函数声明提升
函数表达式就算使用var也会报错,

```
console.log(sum(10, 10)); // TypeError: sum is not a function
var sum = function(num1, num2) {
 return num1 + num2;
};
```

函数作为值

函数可以作为参数，函数可以返回值

函数内部

ES5 函数内部有两个特殊对象 arguments 和 this
ES5 还有一个属性 caller，ES3没有
ES6 新增 new.target属性

arguments

- 类数组的对象，只有function关键字定义的函数才有
- 参数
- callee属性,指向arguments对象所在函数，一半用来解耦函数名与函数逻辑

```
function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * factorial(num - 1);
    }
}
// 使用arguments.callee实现函数逻辑与函数名解耦
function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * arguments.callee(num - 1);
```

```
    }
}
```

this

普通函数：

- 纯粹的函数调用 全局对象
- 作为对象方法调用 该对象
- 作为构造函数 新对象
- apply，参数对象
  箭头函数，指向最近的作用域
  setTimeout是window的方法，但不是全局函数

```
function King() {
    this.royaltyName = 'Henry';
    // this 引用 King 的实例
    setTimeout(() => console.log(this.royaltyName), 1000);
}
function Queen() {
    this.royaltyName = 'Elizabeth';
    // this 引用 window 对象
    setTimeout(function () { console.log(this.royaltyName); }, 1000);
}
new King(); // Henry
new Queen(); // undefined
```

caller

ES5函数对象有，ES3无
引用 调用当前函数的函数，全局作用域则null
ES5 arguments.caller也存在，但是一直是undefined。为了区分函数的caller

作用与arguments.callee一样。
严格模式 arguments.callee，arguments.caller报错
严格模式不能给函数的caller赋值

new.target

正常函数调用，undefined
new调用，构造函数

```
function King() {
 if (!new.target) {
 throw 'King must be instantiated using "new"'
 }
 console.log('King instantiated using "new"');
}
new King(); // King instantiated using "new"
King(); // Error: King must be instantiated using "new"
```

函数属性与方法

函数是对象，有属性和方法
每个函数都有：length 、prototype (箭头函数没有)
length：表示参数个数
prototype：原型对象，不可枚举

方法：apply(),call()
call一个个传参，apply传数组
修改函数体内this指向

es5 bind(),返回函数，可以用apply，call实现bind。

```
function polyfillBind (fn: Function, ctx: Object): Function {
  function boundFn (a) {
    const l = arguments.length
    return l
      ? l > 1
        ? fn.apply(ctx, arguments)
        : fn.call(ctx, a)
      : fn.call(ctx)
  }

  boundFn._length = fn.length
  return boundFn
}

function nativeBind (fn: Function, ctx: Object): Function {
  return fn.bind(ctx)
}

export const bind = Function.prototype.bind
  ? nativeBind
  : polyfillBind
```

函数表达式

匿名函数
函数表达式不会提升