```javascript
var a = 10;
var obj = {
  a: 20
}

function fn() {
  this = obj; // 修改this，运行后会报错
  console.log(this.a);
}

fn();
```

## 9.2. 绑定规则

根据不同的使用场合，`this` 有不同的值，主要分为下面几种情况：

- 默认绑定
- 隐式绑定
- new绑定
- 显示绑定

### 9.2.1. 默认绑定

全局环境中定义 `person` 函数，内部使用 `this` 关键字

```javascript
var name = 'Jenny';
function person() {
    return this.name;
}
console.log(person());  //Jenny
```

上述代码输出 `Jenny`，原因是调用函数的对象在游览器中位 `window`，因此 `this` 指向 `window`，所以输出 `Jenny`

注意：

严格模式下，不能将全局对象用于默认绑定，this会绑定到 `undefined`，只有函数运行在非严格模式下，默认绑定才能绑定到全局对象

### 9.2.2. 隐式绑定

函数还可以作为某个对象的方法调用，这时 `this` 就指这个上级对象

```javascript
function test() {
  console.log(this.x);
}

var obj = {};
obj.x = 1;
obj.m = test;

obj.m(); // 1
```

这个函数中包含多个对象，尽管这个函数是被最外层的对象所调用， `this` 指向的也只是它上一级的对象

```javascript
var o = {
    a:10,
    b:{
        fn:function(){
            console.log(this.a); //undefined
        }
    }
}
o.b.fn();
```

上述代码中， `this` 的上一级对象为 `b` ， `b` 内部并没有 `a` 变量的定义，所以输出 `undefined`

这里再举一种特殊情况

```JavaScript
var o = {
    a:10,
    b:{
        a:12,
        fn:function(){
            console.log(this.a); //undefined
            console.log(this); //window
        }
    }
}
var j = o.b.fn;
j();
```

此时 `this` 指向的是 `window` ，这里的大家需要记住， `this` 永远指向的是最后调用它的对象，虽然 `fn` 是对象 `b` 的方法，但是 `fn` 赋值给 `j` 时候并没有执行，所以最终指向 `window`

### 9.2.3. new绑定

通过构建函数 `new` 关键字生成一个实例对象，此时 `this` 指向这个实例对象

```JavaScript
function test() {
  this.x = 1;
}

var obj = new test();
obj.x // 1
```

上述代码之所以能过输出1，是因为 `new` 关键字改变了 `this` 的指向

这里再列举一些特殊情况：

`new` 过程遇到 `return` 一个对象，此时 `this` 指向为返回的对象

```javascript
function fn()
{
    this.user = 'xxx';
    return {};
}
var a = new fn();
console.log(a.user); //undefined
```

如果返回一个简单类型的时候，则 `this` 指向实例对象

```javascript
function fn()
{
    this.user = 'xxx';
    return 1;
}
var a = new fn;
console.log(a.user); //xxx
```

注意的是 `null` 虽然也是对象，但是此时 `new` 仍然指向实例对象

```javascript
function fn()
{
    this.user = 'xxx';
    return null;
}
var a = new fn;
console.log(a.user); //xxx
```

### 9.2.4. 显示修改

`apply()、call()、bind()` 是函数的一个方法，作用是改变函数的调用对象。它的第一个参数就表示改变后的调用这个函数的对象。因此，这时 `this` 指的就是这第一个参数

```JavaScript
var x = 0;
function test() {
  console.log(this.x);
}

var obj = {};
obj.x = 1;
obj.m = test;
obj.m.apply(obj) // 1
```

关于 `apply、call、bind` 三者的区别，我们后面再详细说

## 9.3. 箭头函数

在 ES6 的语法中还提供了箭头函语法，让我们在代码书写时就能确定 `this` 的指向（编译时绑定）

举个例子：

```JavaScript
const obj = {
  sayThis: () => {
    console.log(this);
  }
};

obj.sayThis(); // window 因为 JavaScript 没有块作用域，所以在定义 sayThis 的时候，里面的 this 就绑到 window 上去了
const globalSay = obj.sayThis;
globalSay(); // window 浏览器中的 global 对象
```

虽然箭头函数的 `this` 能够在编译的时候就确定了 `this` 的指向，但也需要注意一些潜在的坑

下面举个例子：

绑定事件监听

```javascript
const button = document.getElementById('mngb');
button.addEventListener('click', ()=> {
    console.log(this === window) // true
    this.innerHTML = 'clicked button'
})
```

上述可以看到，我们其实是想要 `this` 为点击的 `button` ，但此时 `this` 指向了 `window`

包括在原型上添加方法时候，此时 `this` 指向 `window`

```javascript
Cat.prototype.sayName = () => {
    console.log(this === window) //true
    return this.name
}
const cat = new Cat('mm');
cat.sayName()
```

同样的，箭头函数不能作为构建函数

# 9.4. 优先级

## 9.4.1. 隐式绑定 VS 显式绑定

JavaScript | 复制代码

```javascript
function foo() {
    console.log( this.a );
}

var obj1 = {
    a: 2,
    foo: foo
};

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

显然，显示绑定的优先级更高

### 9.4.2. new绑定 VS 隐式绑定

JavaScript | 复制代码

```
function foo(something) {
    this.a = something;
}

var obj1 = {
    foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
console.log( bar.a ); // 4
```

可以看到，new绑定的优先级 `>` 隐式绑定

### 9.4.3. `new` 绑定 VS 显式绑定

因为 `new` 和 `apply、call` 无法一起使用，但硬绑定也是显式绑定的一种，可以替换测试

JavaScript | 复制代码

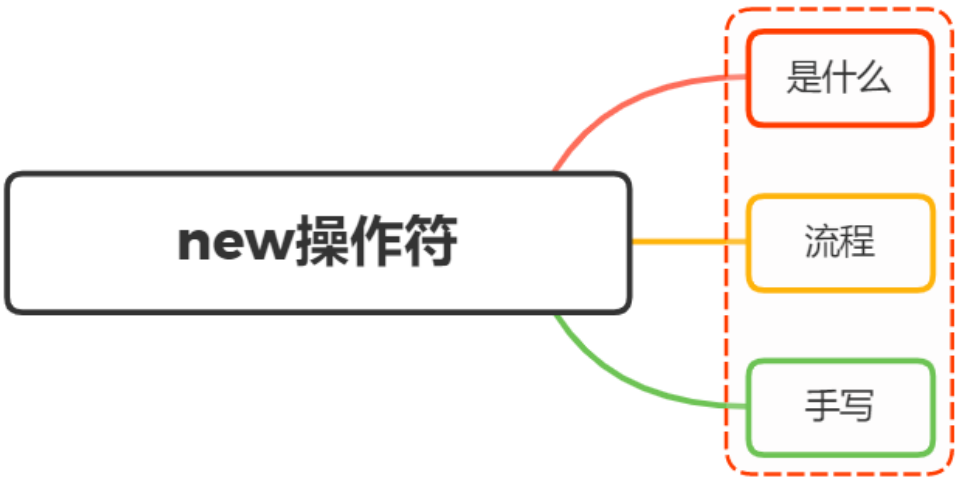
```
function foo(something) {
    this.a = something;
}

var obj1 = {};

var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar( 3 );
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

`bar` 被绑定到obj1上，但是 `new bar(3)` 并没有像我们预计的那样把 `obj1.a` 修改为3。但是， `new` 修改了绑定调用 `bar()` 中的 `this`

我们可认为 `new` 绑定优先级 `>` 显式绑定

综上，new绑定优先级 > 显示绑定优先级 > 隐式绑定优先级 > 默认绑定优先级

# 10. 说说new操作符具体干了什么?




## 10.1. 是什么

在 `JavaScript` 中， `new` 操作符用于创建一个给定构造函数的实例对象

例子

JavaScript | 复制代码

```
function Person(name, age){
    this.name = name;
    this.age = age;
}
Person.prototype.sayName = function () {
    console.log(this.name)
}
const person1 = new Person('Tom', 20)
console.log(person1)  // Person {name: "Tom", age: 20}
t.sayName() // 'Tom'
```

从上面可以看到：

- `new` 通过构造函数 `Person` 创建出来的实例可以访问到构造函数中的属性

- `new` 通过构造函数 `Person` 创建出来的实例可以访问到构造函数原型链中的属性（即实例与构造函数通过原型链连接了起来）

现在在构建函数中显式加上返回值，并且这个返回值是一个原始类型

```JavaScript
function Test(name) {
  this.name = name
  return 1
}
const t = new Test('xxx')
console.log(t.name) // 'xxx'
```

可以发现，构造函数中返回一个原始值，然而这个返回值并没有作用

下面在构造函数中返回一个对象

```JavaScript
function Test(name) {
  this.name = name
  console.log(this) // Test { name: 'xxx' }
  return { age: 26 }
}
const t = new Test('xxx')
console.log(t) // { age: 26 }
console.log(t.name) // 'undefined'
```

从上面可以发现，构造函数如果返回值为一个对象，那么这个返回值会被正常使用

## 10.2. 流程

从上面介绍中，我们可以看到 `new` 关键字主要做了以下的工作：

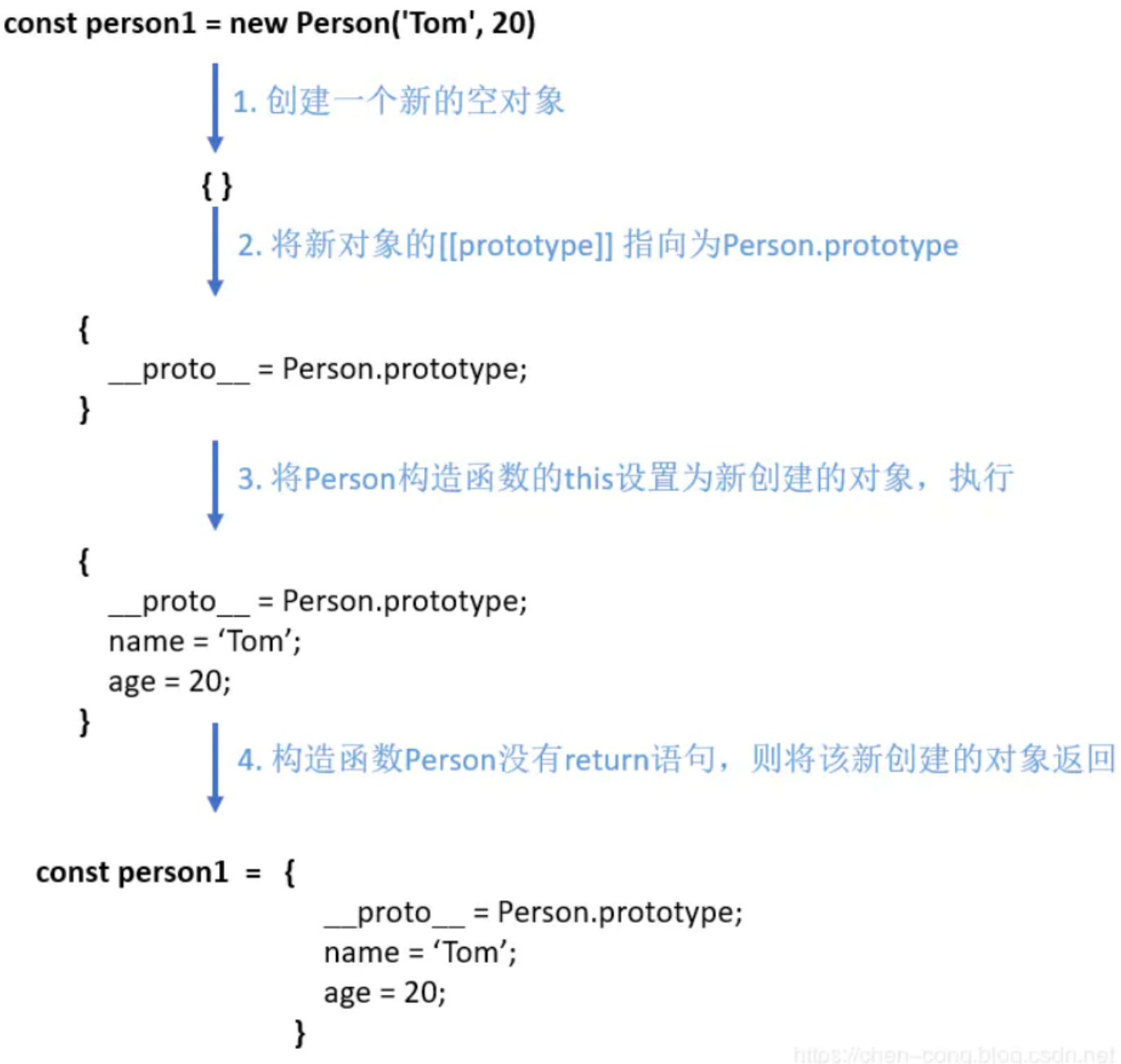
- 创建一个新的对象 `obj`
- 将对象与构建函数通过原型链连接起来
- 将构建函数中的 `this` 绑定到新建的对象 `obj` 上
- 根据构建函数返回类型作判断，如果是原始值则被忽略，如果是返回对象，需要正常处理

举个例子：

JavaScript | 复制代码

```javascript
function Person(name, age){
    this.name = name;
    this.age = age;
}
const person1 = new Person('Tom', 20)
console.log(person1)  // Person {name: "Tom", age: 20}
t.sayName() // 'Tom'
```

流程图如下：



## 10.3. 手写new操作符

现在我们已经清楚地掌握了 `new` 的执行过程

那么我们就动手来实现一下 `new`

JavaScript | 复制代码

```
function mynew(Func, ...args) {
    // 1.创建一个新对象
    const obj = {}
    // 2.新对象原型指向构造函数原型对象
    obj.__proto__ = Func.prototype
    // 3.将构建函数的this指向新对象
    let result = Func.apply(obj, args)
    // 4.根据返回值判断
    return result instanceof Object ? result : obj
}
```

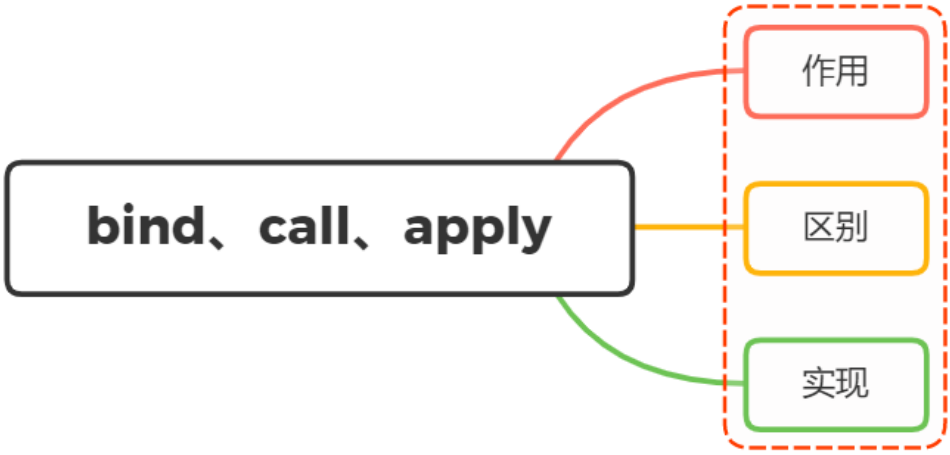
测试一下

JavaScript | 复制代码

```
function mynew(func, ...args) {
    const obj = {}
    obj.__proto__ = func.prototype
    let result = func.apply(obj, args)
    return result instanceof Object ? result : obj
}
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.say = function () {
    console.log(this.name)
}

let p = mynew(Person, "huihui", 123)
console.log(p) // Person {name: "huihui", age: 123}
p.say() // huihui
```

可以发现，代码虽然很短，但是能够模拟实现 `new`

# 11. bind、call、apply 区别？如何实现一个bind?




## 11.1. 作用

`call` 、 `apply` 、 `bind` 作用是改变函数执行时的上下文，简而言之就是改变函数运行时的 `this` 指向

那么什么情况下需要改变 `this` 的指向呢？下面举个例子

JavaScript | 复制代码

```
var name = "lucy";
var obj = {
    name: "martin",
    say: function () {
        console.log(this.name);
    }
};
obj.say(); // martin, this 指向 obj 对象
setTimeout(obj.say,0); // lucy, this 指向 window 对象
```

从上面可以看到，正常情况 `say` 方法输出 `martin`

但是我们把 `say` 放在 `setTimeout` 方法中，在定时器中是作为回调函数来执行的，因此回到主栈执行时是在全局执行上下文的环境中执行的，这时候 `this` 指向 `window` ，所以输出 `lucy`

我们实际需要的是 `this` 指向 `obj` 对象，这时候就需要该改变 `this` 指向了

JavaScript | 复制代码

```
setTimeout(obj.say.bind(obj),0); //martin, this指向obj对象
```

## 11.2. 区别

下面再来看看 `apply` 、 `call` 、 `bind` 的使用

### 11.2.1. apply

`apply` 接受两个参数，第一个参数是 `this` 的指向，第二个参数是函数接受的参数，以数组的形式传入

改变 `this` 指向后原函数会立即执行，且此方法只是临时改变 `this` 指向一次

JavaScript 复制代码

```
function fn(...args){
    console.log(this,args);
}
let obj = {
    myname:"张三"
}

fn.apply(obj,[1,2]); // this会变成传入的obj，传入的参数必须是一个数组；
fn(1,2) // this指向window
```

当第一个参数为 `null` 、 `undefined` 的时候，默认指向 `window` (在浏览器中)

JavaScript 复制代码

```
fn.apply(null,[1,2]); // this指向window
fn.apply(undefined,[1,2]); // this指向window
```

### 11.2.2. call

`call` 方法的第一个参数也是 `this` 的指向，后面传入的是一个参数列表

跟 `apply` 一样，改变 `this` 指向后原函数会立即执行，且此方法只是临时改变 `this` 指向一次

```javascript
function fn(...args){
    console.log(this,args);
}
let obj = {
    myname:"张三"
}

fn.call(obj,1,2); // this会变成传入的obj，传入的参数必须是一个数组；
fn(1,2) // this指向window
```

同样的，当第一个参数为 `null` 、 `undefined` 的时候，默认指向 `window` (在浏览器中)

```javascript
fn.call(null,[1,2]); // this指向window
fn.call(undefined,[1,2]); // this指向window
```

### 11.2.3. bind

bind方法和call很相似，第一参数也是 `this` 的指向，后面传入的也是一个参数列表(但是这个参数列表可以分多次传入)

改变 `this` 指向后不会立即执行，而是返回一个永久改变 `this` 指向的函数

```javascript
function fn(...args){
    console.log(this,args);
}
let obj = {
    myname:"张三"
}

const bindFn = fn.bind(obj); // this 也会变成传入的obj ，bind不是立即执行需要执行一次
bindFn(1,2) // this指向obj
fn(1,2) // this指向window
```

### 11.2.4. 小结

从上面可以看到， `apply` 、 `call` 、 `bind` 三者的区别在于：

- 三者都可以改变函数的 `this` 对象指向
- 三者第一个参数都是 `this` 要指向的对象，如果如果没有这个参数或参数为 `undefined` 或 `null`，则默认指向全局 `window`
- 三者都可以传参，但是 `apply` 是数组，而 `call` 是参数列表，且 `apply` 和 `call` 是一次性传入参数，而 `bind` 可以分为多次传入
- `bind` 是返回绑定this之后的函数， `apply` 、 `call` 则是立即执行

## 11.3. 实现

实现 `bind` 的步骤，我们可以分解成为三部分：

- 修改 `this` 指向
- 动态传递参数

JavaScript | 复制代码

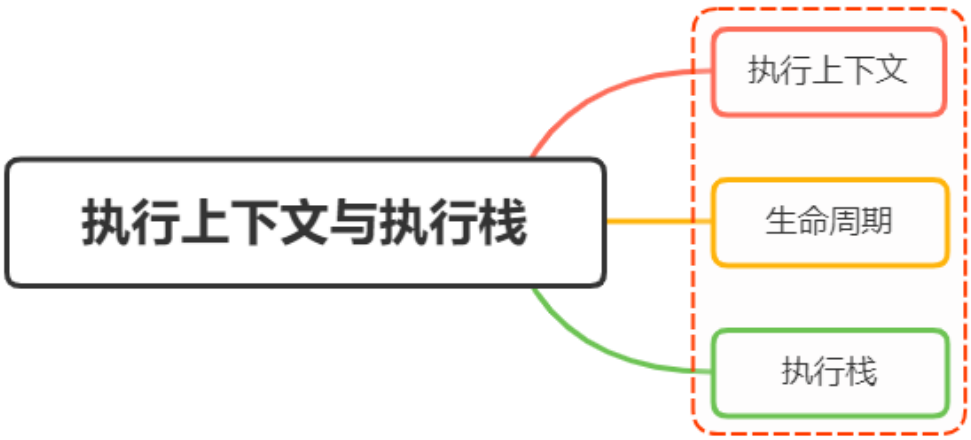
```
// 方式一：只在bind中传递函数参数
fn.bind(obj,1,2)()

// 方式二：在bind中传递函数参数，也在返回函数中传递参数
fn.bind(obj,1)(2)
```

- 兼容 `new` 关键字

整体实现代码如下：

```javascript
Function.prototype.myBind = function (context) {
    // 判断调用对象是否为函数
    if (typeof this !== "function") {
        throw new TypeError("Error");
    }

    // 获取参数
    const args = [...arguments].slice(1),
          fn = this;

    return function Fn() {

        // 根据调用方式，传入不同绑定值
        return fn.apply(this instanceof Fn ? new fn(...arguments) : context, args.concat(...arguments));
    }
}
```

# 12. JavaScript中执行上下文和执行栈是什么?




## 12.1. 执行上下文

简单的来说，执行上下文是一种对 `Javascript` 代码执行环境的抽象概念，也就是说只要有 `Javascript` 代码运行，那么它就一定是运行在执行上下文中

执行上下文的类型分为三种：

- 全局执行上下文：只有一个，浏览器中的全局对象就是 `window` 对象， `this` 指向这个全局对象

- 函数执行上下文：存在无数个，只有在函数被调用的时候才会被创建，每次调用函数都会创建一个新的执行上下文
- Eval 函数执行上下文： 指的是运行在 `eval` 函数中的代码，很少用而且不建议使用

下面给出全局上下文和函数上下文的例子：

```
// global context

var sayHello = 'Hello';

function person() {          // execution context

    var first = 'David',
        last  = 'Shariff';

    function firstName() {   // execution context
        return first;
    }

    function lastName() {    // execution context
        return last;
    }

    alert(sayHello + firstName() + ' ' + lastName());

}
```

紫色框住的部分为全局上下文，蓝色和橘色框起来的是不同的函数上下文。只有全局上下文（的变量）能被其他任何上下文访问

可以有任意多个函数上下文，每次调用函数创建一个新的上下文，会创建一个私有作用域，函数内部声明的任何变量都不能在当前函数作用域外部直接访问

## 12.2. 生命周期

执行上下文的生命周期包括三个阶段：创建阶段 → 执行阶段 → 回收阶段

### 12.2.1. 创建阶段

创建阶段即当函数被调用，但未执行任何其内部代码之前

创建阶段做了三件事：