```JavaScript
let result1 = (null === null)  //true
let result2 = (undefined === undefined)  //true
```

5.3. 区别

相等操作符（==）会做类型转换，再进行值的比较，全等运算符不会做类型转换

```JavaScript
let result1 = ("55" === 55); // false，不相等，因为数据类型不同
let result2 = (55 === 55); // true，相等，因为数据类型相同值也相同
```

`null` 和 `undefined` 比较，相等操作符（==）为 `true`，全等为 `false`

```JavaScript
let result1 = (null == undefined ); // true
let result2 = (null  === undefined); // false
```

5.4. 小结

相等运算符隐藏的类型转换，会带来一些违反直觉的结果

```JavaScript
'' == '0' // false
0 == '' // true
0 == '0' // true

false == 'false' // false
false == '0' // true

false == undefined // false
false == null // false
null == undefined // true

' \t\r\n' == 0 // true
```

但在比较 `null` 的情况的时候，我们一般使用相等操作符 `==`

```javascript
const obj = {};

if(obj.x == null){
  console.log("1");  //执行
}
```

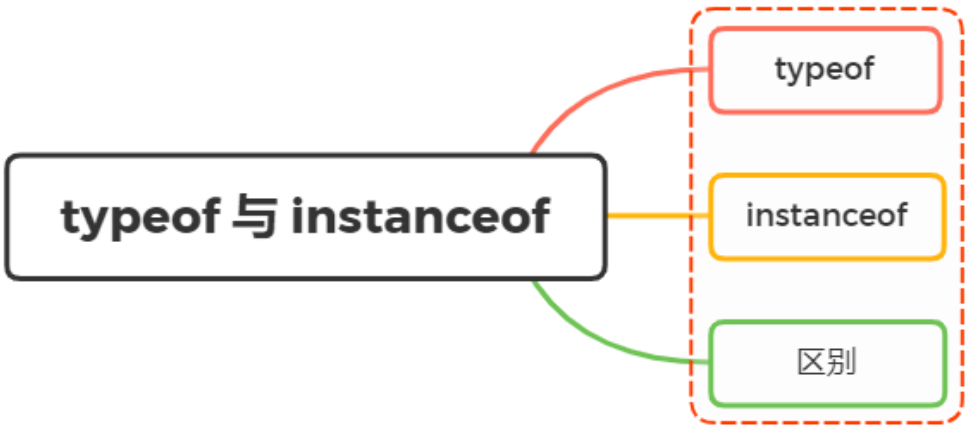
等同于下面写法

```javascript
if(obj.x === null || obj.x === undefined) {
    ...
}
```

使用相等操作符（==）的写法明显更加简洁了

所以，除了在比较对象属性为 `null` 或者 `undefined` 的情况下，我们可以使用相等操作符（），其他情况一律使用全等操作符

6. typeof 与 instanceof 区别



6.1. typeof

`typeof` 操作符返回一个字符串，表示未经计算的操作数的类型

使用方法如下：

```
typeof operand
typeof(operand)
```

`operand` 表示对象或原始值的表达式，其类型将被返回

举个例子

```
typeof 1 // 'number'
typeof '1' // 'string'
typeof undefined // 'undefined'
typeof true // 'boolean'
typeof Symbol() // 'symbol'
typeof null // 'object'
typeof [] // 'object'
typeof {} // 'object'
typeof console // 'object'
typeof console.log // 'function'
```

从上面例子，前6个都是基础数据类型。虽然 `typeof null` 为 `object`，但这只是 `JavaScript` 存在的一个悠久 `Bug`，不代表 `null` 就是引用数据类型，并且 `null` 本身也不是对象

所以，`null` 在 `typeof` 之后返回的是有问题的结果，不能作为判断 `null` 的方法。如果你需要在 `if` 语句中判断是否为 `null`，直接通过 `===null` 来判断就好

同时，可以发现引用类型数据，用 `typeof` 来判断的话，除了 `function` 会被识别出来之外，其余的都输出 `object`

如果我们想要判断一个变量是否存在，可以使用 `typeof`：(不能使用 `if(a)`， 若 `a` 未声明，则报错)

```
if(typeof a != 'undefined'){
    //变量存在
}
```

6.2. instanceof

`instanceof` 运算符用于检测构造函数的 `prototype` 属性是否出现在某个实例对象的原型链上

使用如下：

```
object instanceof constructor
```

`object` 为实例对象， `constructor` 为构造函数

构造函数通过 `new` 可以实例对象， `instanceof` 能判断这个对象是否是之前那个构造函数生成的对象

```
// 定义构建函数
let Car = function() {}
let benz = new Car()
benz instanceof Car // true
let car = new String('xxx')
car instanceof String // true
let str = 'xxx'
str instanceof String // false
```

关于 `instanceof` 的实现原理，可以参考下面：

```
function myInstanceof(left, right) {
    // 这里先用typeof来判断基础数据类型，如果是，直接返回false
    if(typeof left !== 'object' || left === null) return false;
    // getProtypeOf是Object对象自带的API，能够拿到参数的原型对象
    let proto = Object.getPrototypeOf(left);
    while(true) {
        if(proto === null) return false;
        if(proto === right.prototype) return true;//找到相同原型对象，返回true
        proto = Object.getPrototypeof(proto);
    }
}
```

也就是顺着原型链去找，直到找到相同的原型对象，返回 `true` ，否则为 `false`

6.3. 区别

`typeof` 与 `instanceof` 都是判断数据类型的方法，区别如下：

- `typeof` 会返回一个变量的基本类型， `instanceof` 返回的是一个布尔值
- `instanceof` 可以准确地判断复杂引用数据类型，但是不能正确判断基础数据类型

- 而 `typeof` 也存在弊端，它虽然可以判断基础数据类型（`null` 除外），但是引用数据类型中，除了 `function` 类型以外，其他的也无法判断

可以看到，上述两种方法都有弊端，并不能满足所有场景的需求

如果需要通用检测数据类型，可以采用 `Object.prototype.toString`，调用该方法，统一返回格式 `“[object Xxx]”` 的字符串

如下

JavaScript | 复制代码

```
Object.prototype.toString({})       // "[object Object]"
Object.prototype.toString.call({})  // 同上结果，加上call也ok
Object.prototype.toString.call(1)    // "[object Number]"
Object.prototype.toString.call('1')  // "[object String]"
Object.prototype.toString.call(true)  // "[object Boolean]"
Object.prototype.toString.call(function(){})  // "[object Function]"
Object.prototype.toString.call(null)   //"[object Null]"
Object.prototype.toString.call(undefined) //"[object Undefined]"
Object.prototype.toString.call(/123/g)    //"[object RegExp]"
Object.prototype.toString.call(new Date()) //"[object Date]"
Object.prototype.toString.call([])       //"[object Array]"
Object.prototype.toString.call(document)  //"[object HTMLDocument]"
Object.prototype.toString.call(window)   //"[object Window]"
```

了解了 `toString` 的基本用法，下面就实现一个全局通用的数据类型判断方法

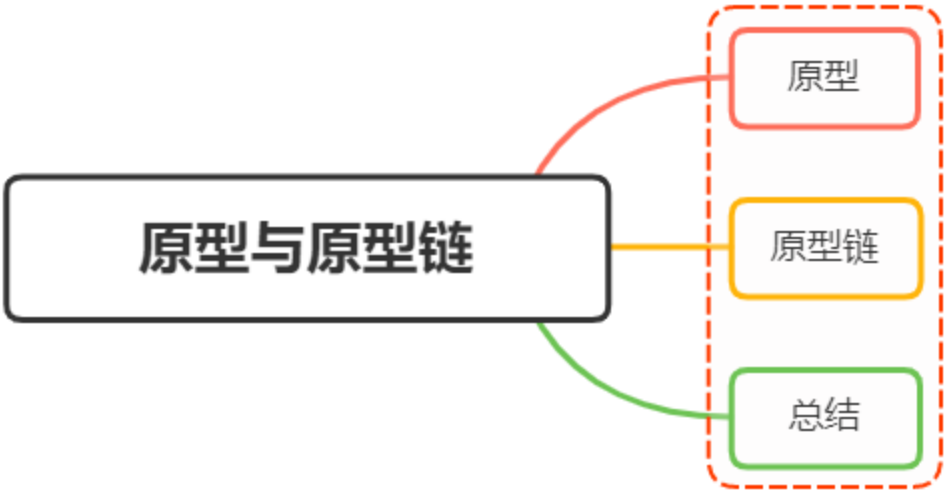
JavaScript | 复制代码

```
function getType(obj){
  let type  = typeof obj;
  if (type !== "object") {    // 先进行typeof判断，如果是基础数据类型，直接返回
    return type;
  }
  // 对于typeof返回结果是object的，再进行如下的判断，正则返回结果
  return Object.prototype.toString.call(obj).replace(/^\[object (\S+)\]$/,
'$1');
}
```

使用如下

```javascript
getType([])     // "Array" typeof []是object，因此toString返回
getType('123')  // "string" typeof 直接返回
getType(window) // "Window" toString返回
getType(null)   // "Null"首字母大写，typeof null是object，需toString来判断
getType(undefined)   // "undefined" typeof 直接返回
getType()            // "undefined" typeof 直接返回
getType(function(){}) // "function" typeof能判断，因此首字母小写
getType(/123/g)      //"RegExp" toString返回
```

7. JavaScript原型，原型链 ？有什么特点?



7.1. 原型

`JavaScript` 常被描述为一种基于原型的语言——每个对象拥有一个原型对象

当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾

准确地说，这些属性和方法定义在Object的构造器函数（constructor functions）之上的 `prototype` 属性上，而非实例对象本身

下面举个例子：

函数可以有属性。 每个函数都有一个特殊的属性叫作原型 `prototype`

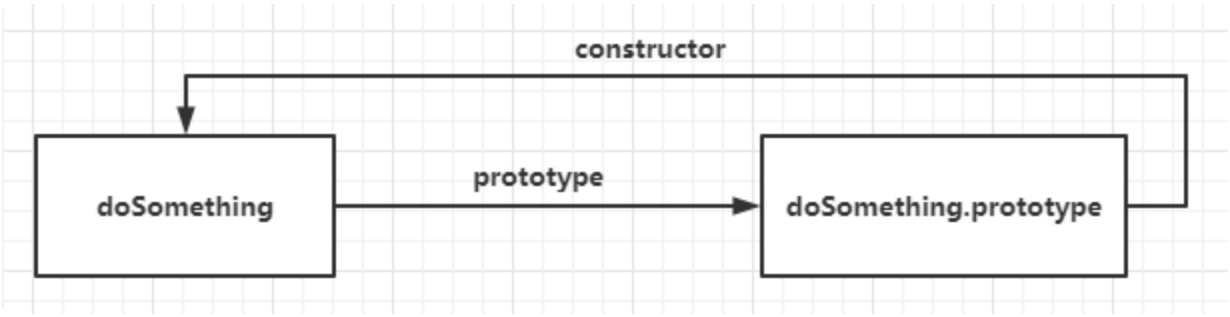
```JavaScript
function doSomething(){}
console.log( doSomething.prototype );
```

控制台输出

```JavaScript
{
    constructor: ƒ doSomething(),
    __proto__: {
        constructor: ƒ Object(),
        hasOwnProperty: ƒ hasOwnProperty(),
        isPrototypeOf: ƒ isPrototypeOf(),
        propertyIsEnumerable: ƒ propertyIsEnumerable(),
        toLocaleString: ƒ toLocaleString(),
        toString: ƒ toString(),
        valueOf: ƒ valueOf()
    }
}
```

上面这个对象，就是大家常说的原型对象

可以看到，原型对象有一个自有属性 `constructor` ，这个属性指向该函数，如下图关系展示



7.2. 原型链

原型对象也可能拥有原型，并从中继承方法和属性，一层一层、以此类推。这种关系常被称为原型链(prototype chain)，它解释了为何一个对象会拥有定义在其他对象中的属性和方法

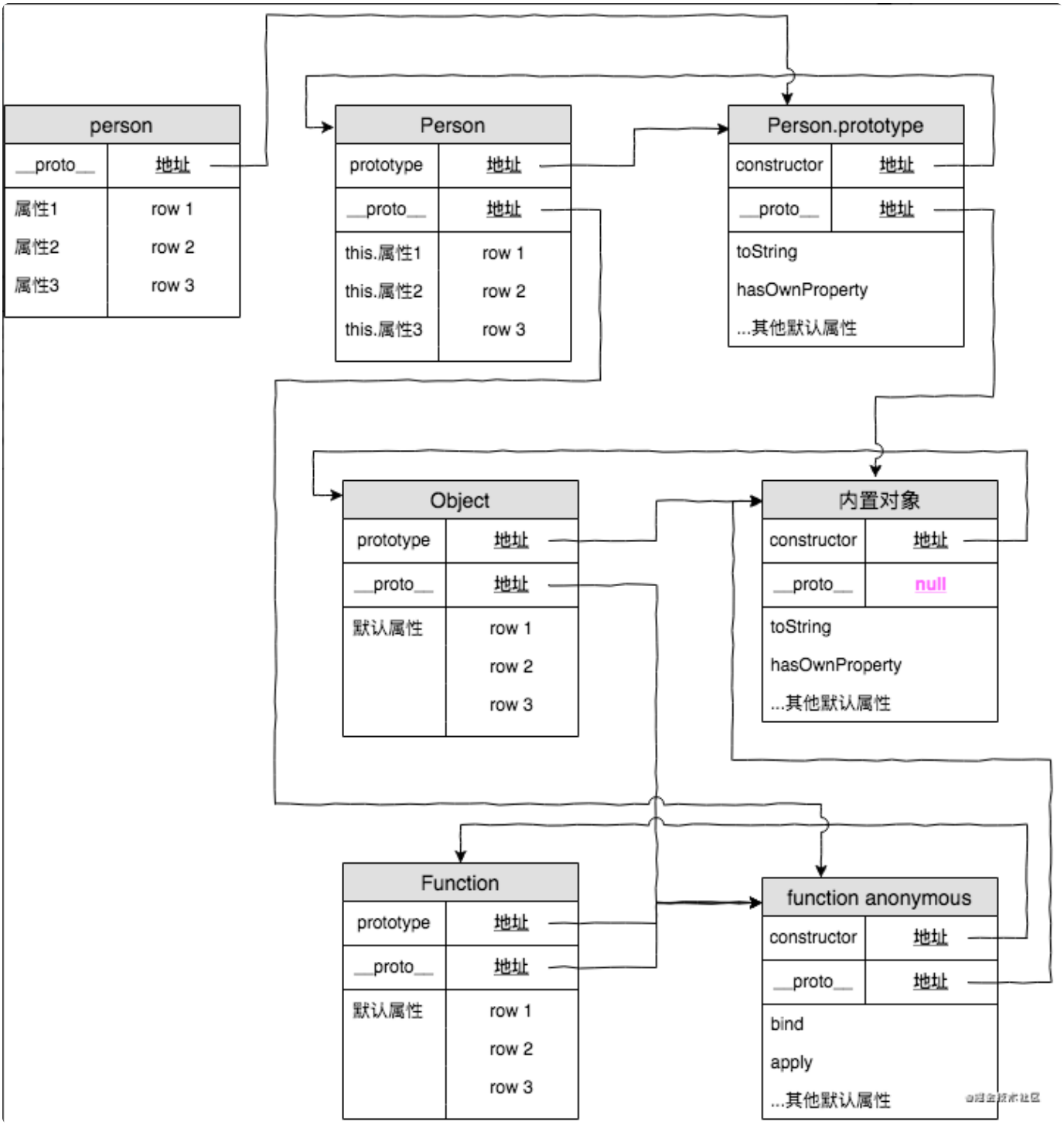
在对象实例和它的构造器之间建立一个链接（它是 `__proto__` 属性，是从构造函数的 `prototype` 属性派生的），之后通过上溯原型链，在构造器中找到这些属性和方法

下面举个例子：

JavaScript | 复制代码

```javascript
function Person(name) {
    this.name = name;
    this.age = 18;
    this.sayName = function() {
        console.log(this.name);
    }
}
// 第二步 创建实例
var person = new Person('person')
```

根据代码，我们可以得到下图



下面分析一下：

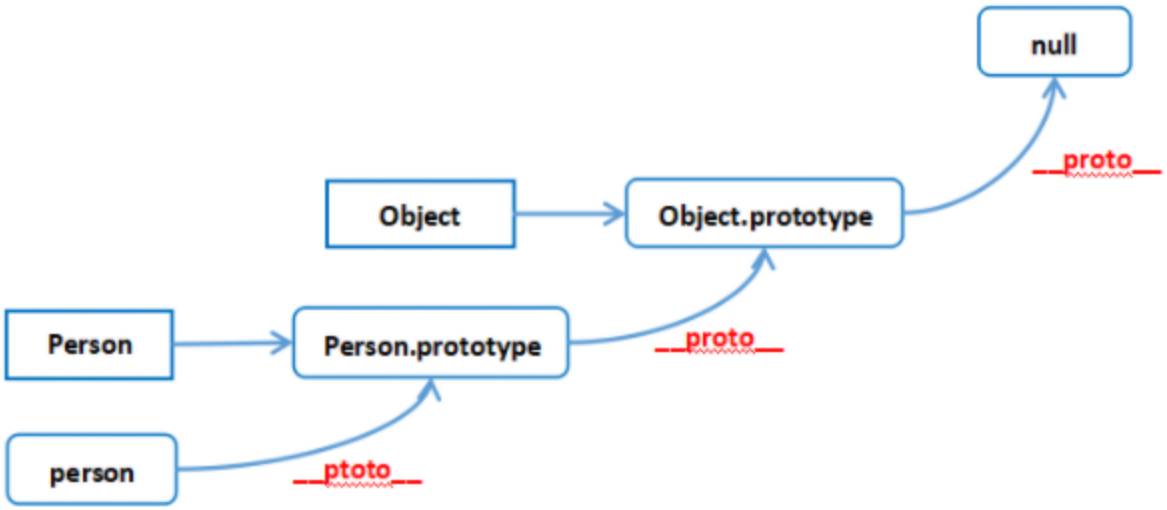
- 构造函数 `Person` 存在原型对象 `Person.prototype`
- 构造函数生成实例对象 `person`， `person` 的 `__proto__` 指向构造函数 `Person` 原型对象
- `Person.prototype.__proto__` 指向内置对象，因为 `Person.prototype` 是个对象，默认是由 `Object` 函数作为类创建的，而 `Object.prototype` 为内置对象
- `Person.__proto__` 指向内置匿名函数 `anonymous`，因为 Person 是个函数对象，默认由 Function 作为类创建

- `Function.prototype` 和 `Function.__proto__` 同时指向内置匿名函数 `anonymous`，这样原型链的终点就是 `null`

7.3. 总结

下面首先要看几个概念：

`__proto__` 作为不同对象之间的桥梁，用来指向创建它的构造函数的原型对象的



每个对象的 `__proto__` 都是指向它的构造函数的原型对象 `prototype` 的

```
JavaScript | 复制代码
1  person1.__proto__ === Person.prototype
```

构造函数是一个函数对象，是通过 `Function` 构造器产生的

```
JavaScript | 复制代码
1  Person.__proto__ === Function.prototype
```

原型对象本身是一个普通对象，而普通对象的构造函数都是 `Object`

```
JavaScript | 复制代码
1  Person.prototype.__proto__ === Object.prototype
```

刚刚上面说了，所有的构造器都是函数对象，函数对象都是 `Function` 构造产生的

```JavaScript
Object.__proto__ === Function.prototype
```

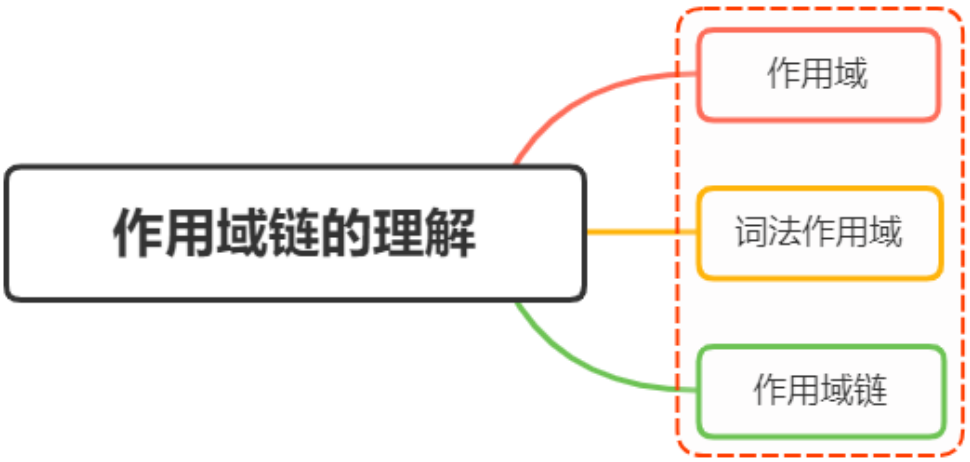
`Object` 的原型对象也有 `__proto__` 属性指向 `null`， `null` 是原型链的顶端

```JavaScript
Object.prototype.__proto__ === null
```

下面作出总结：

- 一切对象都是继承自 `Object` 对象， `Object` 对象直接继承根源对象 `null`
- 一切的函数对象（包括 `Object` 对象），都是继承自 `Function` 对象
- `Object` 对象直接继承自 `Function` 对象
- `Function` 对象的 `__proto__` 会指向自己的原型对象，最终还是继承自 `Object` 对象

8. 说说你对作用域链的理解



8.1. 作用域

作用域，即变量（变量作用域又称上下文）和函数生效（能被访问）的区域或集合

换句话说，作用域决定了代码区块中变量和其他资源的可见性

举个例子

```javascript
function myFunction() {
    let inVariable = "函数内部变量";
}
myFunction();//要先执行这个函数，否则根本不知道里面是啥
console.log(inVariable); // Uncaught ReferenceError: inVariable is not defined
```

上述例子中，函数 `myFunction` 内部创建一个 `inVariable` 变量，当我们在全局访问这个变量的时候，系统会报错

这就说明我们在全局是无法获取到（闭包除外）函数内部的变量

我们一般将作用域分成：

- 全局作用域
- 函数作用域
- 块级作用域

8.1.1. 全局作用域

任何不在函数中或是大括号中声明的变量，都是在全局作用域下，全局作用域下声明的变量可以在程序的任意位置访问

```javascript
// 全局变量
var greeting = 'Hello World!';
function greet() {
  console.log(greeting);
}
// 打印 'Hello World!'
greet();
```

8.1.2. 函数作用域

函数作用域也叫局部作用域，如果一个变量是在函数内部声明的它就在一个函数作用域下面。这些变量只能在函数内部访问，不能在函数以外去访问

```javascript
function greet() {
  var greeting = 'Hello World!';
  console.log(greeting);
}
// 打印 'Hello World!'
greet();
// 报错：  Uncaught ReferenceError: greeting is not defined
console.log(greeting);
```

可见上述代码中在函数内部声明的变量或函数，在函数外部是无法访问的，这说明在函数内部定义的变量或者方法只是函数作用域

8.1.3. 块级作用域

ES6引入了 `let` 和 `const` 关键字,和 `var` 关键字不同，在大括号中使用 `let` 和 `const` 声明的变量存在于块级作用域中。在大括号之外不能访问这些变量

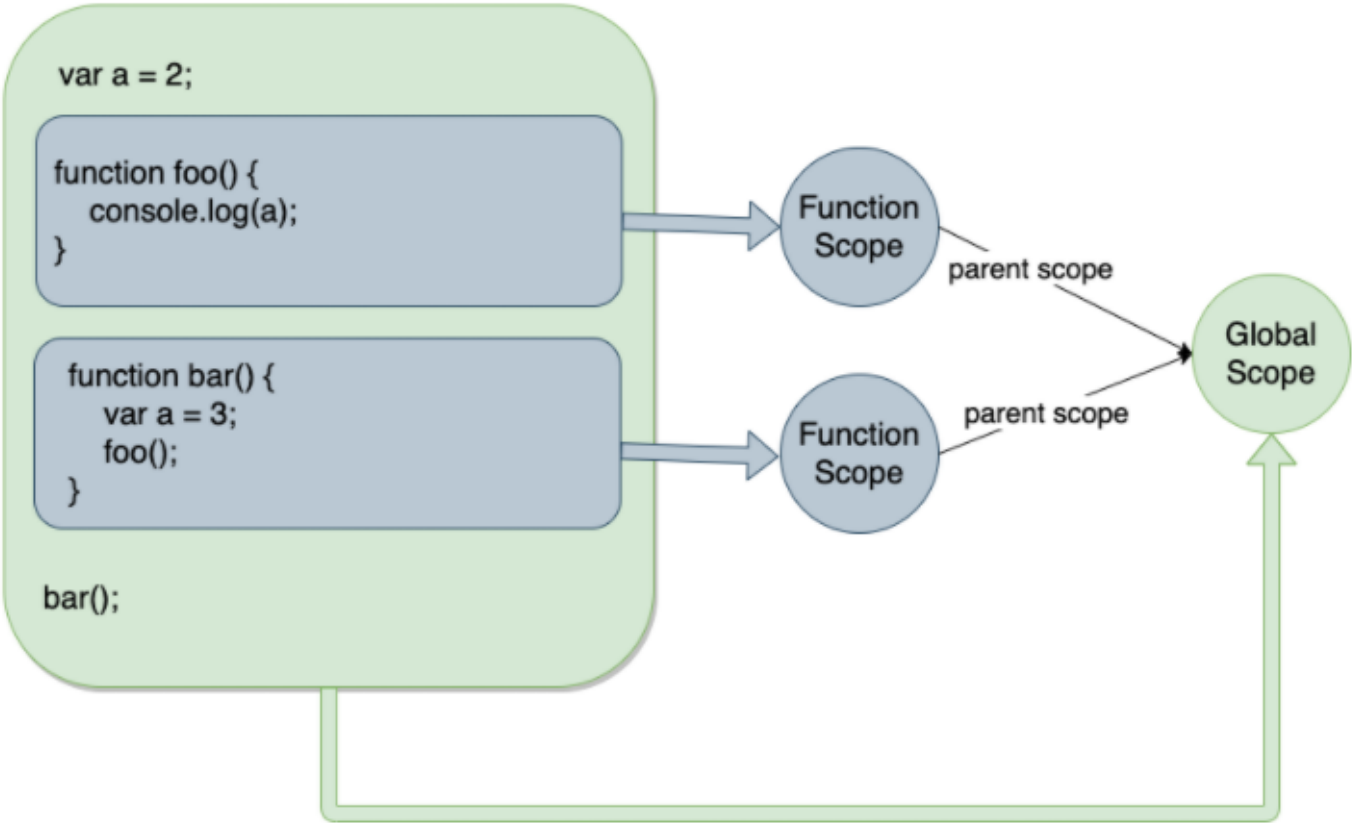
```javascript
{
  // 块级作用域中的变量
  let greeting = 'Hello World!';
  var lang = 'English';
  console.log(greeting); // Prints 'Hello World!'
}
// 变量 'English'
console.log(lang);
// 报错：Uncaught ReferenceError: greeting is not defined
console.log(greeting);
```

8.2. 词法作用域

词法作用域，又叫静态作用域，变量被创建时就确定好了，而非执行阶段确定的。也就是说我们写好代码时它的作用域就确定了， `JavaScript` 遵循的就是词法作用域

```javascript
var a = 2;
function foo(){
    console.log(a)
}
function bar(){
    var a = 3;
    foo();
}
bar()
```

上述代码改变成一张图



由于 `JavaScript` 遵循词法作用域，相同层级的 `foo` 和 `bar` 就没有办法访问到彼此块作用域中的变量，所以输出2

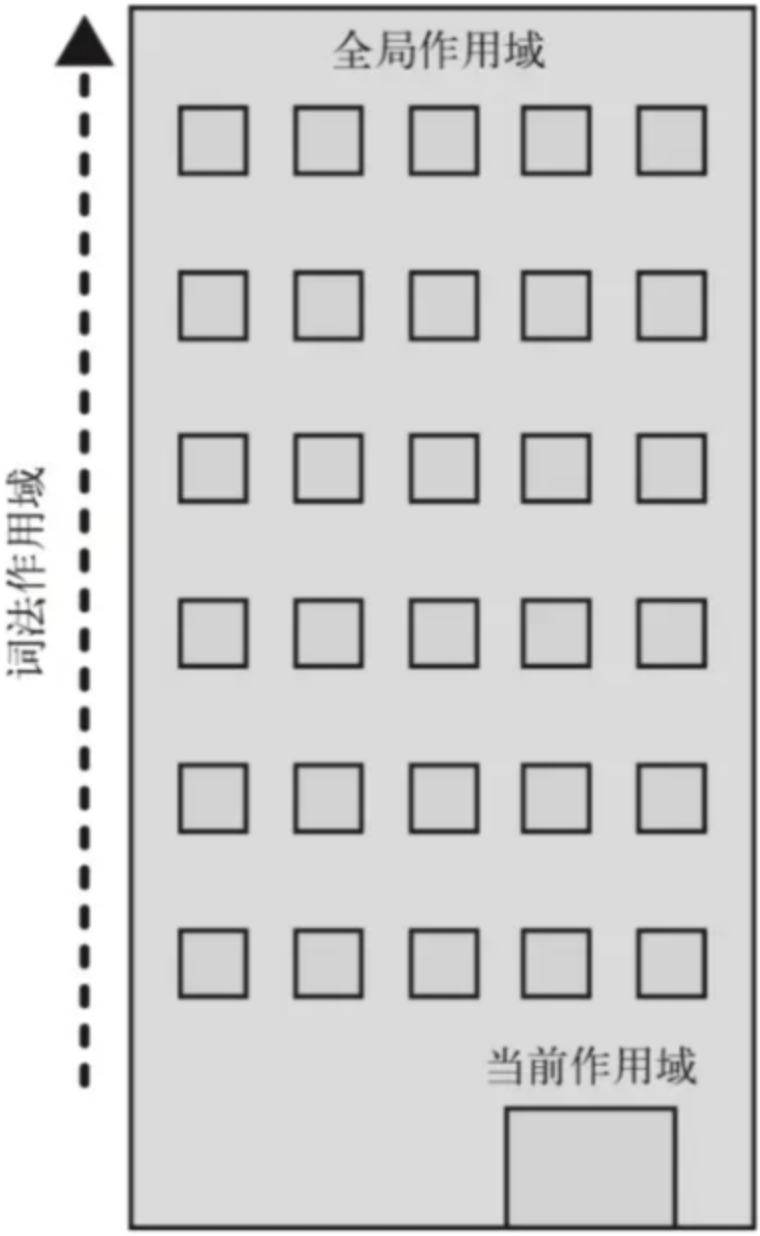
8.3. 作用域链

当在 `Javascript` 中使用一个变量的时候，首先 `Javascript` 引擎会尝试在当前作用域下去寻找该变量，如果没找到，再到它的上层作用域寻找，以此类推直到找到该变量或是已经到了全局作用域

如果在全局作用域里仍然找不到该变量，它就会在全局范围内隐式声明该变量(非严格模式下)或是直接报错

这里拿《你不知道的Javascript(上)》中的一张图解释：

把作用域比喻成一个建筑，这份建筑代表程序中的嵌套作用域链，第一层代表当前的执行作用域，顶层代表全局作用域



变量的引用会顺着当前楼层进行查找，如果找不到，则会往上一层找，一旦到达顶层，查找的过程都会停止

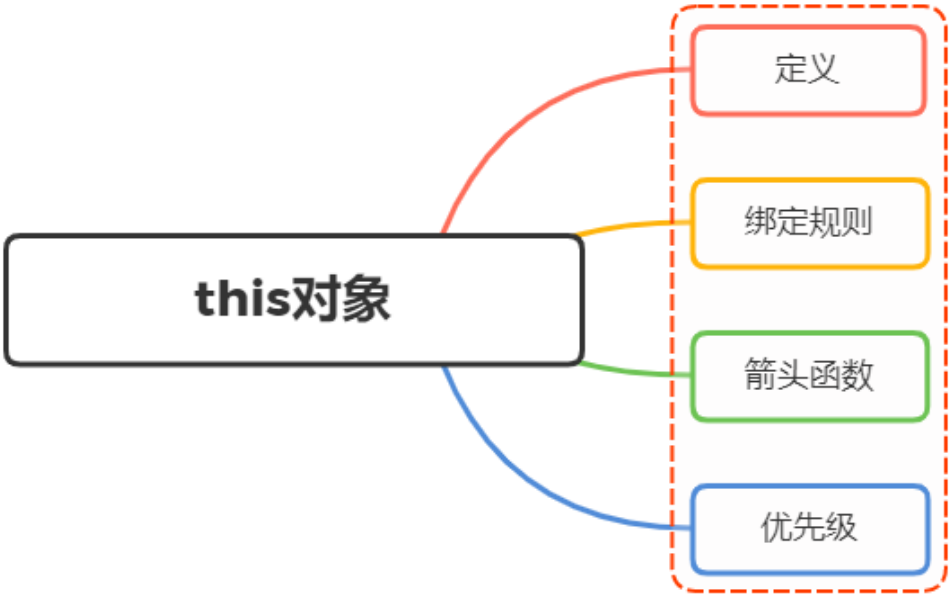
下面代码演示下：

```javascript
var sex = '男';
function person() {
    var name = '张三';
    function student() {
        var age = 18;
        console.log(name); // 张三
        console.log(sex); // 男
    }
    student();
    console.log(age); // Uncaught ReferenceError: age is not defined
}
person();
```

上述代码主要主要做了以下工作：

- `student` 函数内部属于最内层作用域，找不到 `name` ，向上一层作用域 `person` 函数内部找，找到了输出“张三”
- `student` 内部输出 `sex` 时找不到，向上一层作用域 `person` 函数找，还找不到继续向上一层找，即全局作用域，找到了输出“男”
- 在 `person` 函数内部输出 `age` 时找不到，向上一层作用域找，即全局作用域，还是找不到则报错

9. 谈谈this对象的理解



9.1. 定义

函数的 `this` 关键字在 `JavaScript` 中的表现略有不同，此外，在严格模式和非严格模式之间也会有一些差别

在绝大多数情况下，函数的调用方式决定了 `this` 的值（运行时绑定）

`this` 关键字是函数运行时自动生成的一个内部对象，只能在函数内部使用，总指向调用它的对象

举个例子：

JavaScript 复制代码

```javascript
function baz() {
    // 当前调用栈是：baz
    // 因此，当前调用位置是全局作用域

    console.log( "baz" );
    bar(); // <-- bar的调用位置
}

function bar() {
    // 当前调用栈是：baz --> bar
    // 因此，当前调用位置在baz中

    console.log( "bar" );
    foo(); // <-- foo的调用位置
}

function foo() {
    // 当前调用栈是：baz --> bar --> foo
    // 因此，当前调用位置在bar中

    console.log( "foo" );
}

baz(); // <-- baz的调用位置
```

同时，`this` 在函数执行过程中，`this` 一旦被确定了，就不可以再更改