## 预解析

- 一段程序包含有两个步骤：

    - 解析

    - 执行

> 在一段程序执行之前会有一个解析 过程，同时JS在解析的过程中，会 把变量声明和函数声明提升当前作  用域的最开始。\*\*这一动作我们成 为预解析\*\*

        例如：

            \*   var a = 1;

            \*       声明：var a

            \*       赋值：a = 1

            \*   提升的是声明部分

            \*   所以也称：变量（函数）声明预解析

- 函数声明同样也有预解析

    - function 函数名称（）{} => 都属于函数声明部分

    - 另外：函数还有一个创建方式，函数表达式（赋值、运算）

          例如：

          // fn2()

            var fn2 = function() {// 声明了一个匿名函数

                 console.log(2);

             }

          先声明一个变量存储函数

          可以看出来该函数变成了匿名函数

- 函数声明与函数表达式没有太大的  区别，但是需要注意的是：预解析 ，函数声明可以先调用后声明。但 是函数表达式不行

## 作用域

> 变量和函数并不是在程序任何地方任何时候都能访问的，在JS中 变量和函数是存在访问范围的

- 全局-windon

    - 不在任何的函数内部声明都属于全局的

    - 可以在任何地方都能访问到

- 局部（函数）

    - 在函数内部声明的

    - 只能在该函数以及该函数的子函数内部才能使用

\*\*函数外部不能随意访问一个函数内部的数据，但是函数的内部是可以访问其外部数据的\*\*

> \*\*JS中一个内置对象，这个对象是   JS自动创建的--\_windon\_，该    对象是浏览器创建的，不是      ECMAScript创建的，并且是我    们JS其他所有的顶层对象\*\*

- 当我们创建一个全局变量的时候，同时会在windon这个对象下面创建一个同名属性：

        例如：

        var a = 1；

        ==> windon.a

    - 约定：当我们去访问对象是windon的时候，那么windon可以省略不写

          例如：

          console.log(window.a) => console.log(a);

## 作用域链

> 当我们去调用一个变量或函数的  时候，首先程序会在当前作用域内 进行查找，如果查找到该变量或函 数的定义，那么使用就当前查找到 的，如果没有，则会向外（上）的 作用域去查找，一直找到为止，如 果一直找到最外层（全局），还没 有找到，浏览器则会报错，我们把 上面这个称为：作用域链

    例如：

     var c = 'miaov';

        // window.[[Scopes]].c = 'miaov';

        function fn2() {

            // fn2.\_\_Scopes\_\_ = window.[[Scopes]]

            // fn2.[[Scopes]].a = 10;

            // fn2.[[Scopes]].fn3 = function(){};

            var a = 10;

            console.log(a);

            function fn3() {

                console.log(c);

                // fn3.\_\_Scopes\_\_ = fn2.[[Scopes]]

                //

                // fn3.[[Scopes]] = {}

            }

            fn3();

        }

        fn2();

        console.log(a);

## 闭包

- closure/机制

- 函数嵌套函数（这就形成了一个闭包环境），内部函数可以访问外部的数据

    - 可以让外部长久持有一个函数内部数据，让一个函数内部可以访问外部的数据

    - 避免全局变量命名污染

    - \*需要注意的问题:因为可能会导致内部数据长期驻留内存，所以会不小心占用大量内存（内存泄漏）

            例如：

            // function fn1() {

                //     var a = 1;

                //

                //     function fn2() {

                //         console.log(a);

                //     }

                // }

         // function fn3() {

            //     var a = 1;

            //     a += 1;

            //     return a;

            // }

            // 不能直接访问

            // console.log(a)

上面案例说明：虽然return了，但是还不能直接访问a，但是可以调用该函数fn3()，还需要该函数执行完成以后返回内部的值，通过返回值的形式来访问函数内部数据

        // var v = fn3();

        // console.log(v);

### 垃圾回收机制

- 函数中的局部变量，是函数执行过程中创建的，当函数执行完成以后，表示当前这次函数工作已经执行完成了，那么和这个函数有关的数据（在函数中声明的数据）就会被视为垃圾，将会被回收。

        例如：

        // var v1 = fn3();

            // console.log(v1);

            //

            // var v2 = fn3();

            // console.log(v2);

    - 但是当其变量在其他函数使用时，会保留该变量数据

            例如：

           function fn4() {

                   var a = 1;

                   return function() {

                       a += 1;

                       return a;

                   }

               }

               var f1 = fn4(); //返回了fn4内部的匿名函数

               var v1 = f1();   //执行f1，得到fn4中声明的a

               console.log(v1);

               var v2 = f1();

               console.log(v2);

               如果不执行fn4（），那么fn3（）也不会执行

## 函数自执行

        // function fn1() {

            //     console.log(1);

            // }

            // fn1();

            // 函数声明不是表达式，所以不能使用()

            // function fn1() {

            //     console.log(1);

            // }();

            // 函数表达式

            // (function fn1() {

            //     console.log(1);

            // })();

            // !function() {

            //     console.log(1);

            // }();

            //

            // ~function() {

            //     console.log(1);

            // }();

            //

            // +function() {

            //     console.log(1);

            // }();

            //

            // -function() {

            //     console.log(1);

            // }();

            // ~function(a, b) {

            //     // var a = 1;

            //     // var b = 2;

            //     console.log(a + b);

            // }(10, 20);

## 递归--recursion

- 程序调用自身的编程技巧称为递归

- 函数自己调用自己

- 递归也是一种循环

- 最下层的函数执行完才能执行上一层函数

      案例：

      function add(n) {

              if (n == 1) {

                  return n;

              }

              return add(n-1) + n;

          }

          // console.log( add(5) );

          // 5

          // add(4) + 5

          // add(3) + 4 + 5

          // add(2) + 3 + 4 + 5

          // add(1) + 2 + 3 + 4 + 5

          // 1 + 2 + 3 + 4 + 5

