数据类型核心操作步骤和原理

- 值引用
  - 直接按值操作，例如：var a = 12; 直接把12这个值赋值给了变量a（让a变量和12这个值建立了连接关系）

- 引用数据类型
  - 浏览器为其开辟一个新的内存空间，为了方便后期可以找到这个空间，浏览器给空间分配一个16进制的地址
  - 按照一定顺序分别的把对象键值对存储到内存空间当中
  - 把开辟内存的地址赋值给变量（或者其他的东西），以后变量就可以通过地址找到内存空间，然后进行操作

- 函数的操作
  - 创建函数
    - 先开辟一个新的内存空间（为其分配了一个16进制的地址）
    - 把函数体当中编写的js代码当做字符串存储到空间中（函数只创建不执行没有意义）
    - 把分配的地址赋值给声明的函数名（function 和var fn原理其实相同，都是在当前作用域中声明了一个名字，名字是重复的）
    - 函数执行 目的： 执行函数体中的代码
    - 函数执行的时候，浏览器会形成一个新的私有作用域（只能执行函数体中的代码）供函数体中的代码执行
    - 执行代码之前，先把创建函数存储的那些字符串变为真正的js表达式，按照从上到下的顺序在私有作用域下执行

- 一个函数可以被执行N次，每一次执行相互之间互不干扰（后面会学习两者之间创立间接关系） 形成的私有作用域把函数中的私有变量包裹起来了，在私有作用域中操作私有变量和外界没关系，外界也无法直接的操作私有变量，我们把函数执行形成的这种保护机制叫做**闭包**

堆内存和栈内存释放

- 作用域的概念就叫做栈内存
- 存放引用类型的叫做堆内存
- js的堆栈内存
  - 栈内存：
    - 俗称叫做作用域（全局作用域和私有作用域）
    - 目的：
      - 为js提供执行环境（执行代码的地方）
      - 基本数据类型是直接存放在栈内存中的
  - 堆内存
    - 存储应用数据类型值得（相当于一个存储的仓库）
      - 对象存储的是键值对

        - 函数存储的是字符串
    - 内存多好还是少好引出
        - 在项目中，我们的内存越少性能越好，我们需要把一些没用的内存处理掉
        - [堆内存] var o = {};当前对象对应的堆内存被变量o占用着，堆内存是无法销毁的； o= 12的时候相当于销毁了堆内存，这样在栈内存也占用着 所以让o指向null空指针 o= null ；null空对象指针（不指向任何的堆内存）,此时上一次的堆内存就没有被占用,
        - 浏览器销毁时机：
            - 谷歌浏览器销毁会在空闲时间把没有占用的堆内存销毁
            - 计数器

        - [栈内存]
            - 一般情况下，函数执行形成栈内存，函数执行完，浏览器会把形成的栈内存自动释放；

        - 全局作用域在加载页面的时候执行，关闭页面的时候销毁

变量提升（预解释）

基本概念
带var不带var
只对等号左边的提升
不管条件是否成立都变量提升
重名的处理

- 概念
    - 在当前作用域中自上而下执行之前浏览器首先会把所有带var和function的关键字的进行提前的声明和定义 声明(declare)： var num; 在当前作用域中喊一句我有num这个民了 定义(defined)： num = 12; 把声明的名字赋一个值
    - 带var关键字的只是提前声明一下
    - function的关键字提升阶段把声明和定义完成了

```
console.log(num);
console.log(fn);
var num = 1;
function fn() {
    console.log(denum);
    var denum = 10;
    console.log(denum);
}
fn();
console.log(num)
```

作用域链初步讲解

- 定义变量的时候带var和不带var的区别
- 带var
    - 在当前作用域中声明了一个变量，如果当前是全局作用域，也相当于给全局作用域设置了一个属性叫做a

- 不带var
    - 在全局作用域中，如果不带var仅仅是给全局对象设置了一个新的属性名（把window点省略了）
    - 以后项目中，如果你的目的创建变量，最好不要省略var

- 私有作用域下的区别

```
    function fn() {
        console.log(a);
        var a = 12;
        cosnole.log(a);
    }
    fn()
    console.log(a);// a is not defined 闭包机制：私有作用域保护里面的私有变量不受外界干扰
    fn()
    function fn() {
        a = 12;
        console.log(12);
    }
    console.log(12);
```

- 作用域链
    - 函数执行形成一个私有的作用域（保护私有变量），进入到私有作用域中，首先变量提升（声明过的变量是私有的），接下来代码执行
        - 执行的时候遇到一个变量，如果这个变量是私有的，那么按照私有处理即可
        - 如果当前这个变量不是私有的，我们需要向他的上级作用域查找，上级如果没有继续找 找到全局作用域（window）为止，我们把这种查找查找机制叫做**作用域链**
        - 如果上级作用域有，我们当前操作的事都是上级作用域中的变量（假如我们在当前作用域把值改了，相当于把上级作用域中的这个值给修改了）
        - 如果上级作用域中没有这个变量 window： 变量 = 值：相当于给window设置了一个属性，以后再操作window下就有了

只对等号左边提升

- = : 赋值，左边是变量，右边都应该是值 匿名函数：函数表达式（把函数当做一个值赋值给变量或者其他内容）

```
console.log(fn)
    function fn(){}
```

```
    console.log(fn)
```

- 真实项目中，应用这个原理，我们创建函数的时候可以使用函数表达式的方式；
    - 1.只能对等号左边变量提升，所以变量提升完之后，当前函数只是声明了，没有定义，想要执行函数只能放在赋值代码之后，放在前面执行相当于让undefined执行
    - 2.这样让我们的代码逻辑更加严谨，以后想要知道一个执行的函数做了什么功能，只需要向上查找定义的部分即可（不会存在定义的代码在执行下面的情况）

```
var fn = function a() {
    console.log(a); //只能在这里面调用
}
```

不管条件是否成立都要进行变量提升

```
    console.log(num);
    console.log(fn);
    if(1 !== 1) {
        var num = 12;
        function fn() {};
    }
```

```
console.log(num);
    console.log(fn);
    if(1 === 1) {
        var num = 12;
        function fn() {};
        console.log(num);
        console.log(fn);
    }
    fn();
    console.log(num)
    console.log(fn)
```

- 不管条件是否成立，判断体中出现的var/function都会进行变量提升；但是在最新版浏览器中，function声明的变量只能提前声明不能定义了（前提：判断体中）
- 代码执行到条件判断地方
    - 条件不成立
        - 进入不到判断体的，赋值的代码执行不了， 此时之前声明的变量或者函数依然是undefined
    - 条件成立
        - 进入条件判断体中的第一件事不是diam执行，而是把之前变量提升没有定义的函数首先定义了（进入到判断体中函数就定义了：为了迎合es6块级作用域）
        - 兼容：旧版本的声明和定义，新只会声明

```
f = function(){return ture;}
    g = function(){return false;}
    ~function(){
        if(g() && [] == ![]) {
           f = function() {return false;}
           function g() {return true;}
        }
    }()
    console.log(f())
    console.log(g())
```

重名情况下的处理

- 在变量提升阶段，如果名字重复了，不会重新的进行声明，但是会重新的选择定义（后面赋的值会把前面的替换掉）

```
fn();
function fn() {console.log(1)}
fn();
function fn() {console.log(2)}
fn();
var fn = 3;
fn();
function fn() {console.log(3)}
fn();
function fn() {console.log(4)}
fn();
```

私有变量都有

- 作用域[SCOPE]
- 栈内存 全局作用域： window
- 私有作用域： 函数执行
- 块级作用域： 使用let创建变量存在块级作用域
- [作用域链]
    - 当前作用域代码执行的时候遇到的一个变量，我们首先看一下是否属于私有变量，如果是当前作用域私有变量，那么以后在私有作用域中在遇到都是操作私有的变量，（闭包：私有作用域保护私有变量不受外界干扰）；
    - 如果不是私有的变量，向其上级作用域超找，也不是上级，继续向上查找，一直找到window全局作用域为止，我们把这种查找机制叫做作用域链；
    - 全局下有，操作的就是全局变量，全局下没有（设置：给全局对象window增加了属性名&&获取：报错）

- 查找私有变量

- js中的私有变量有且只有两种
  - 在私有作用域变量提升阶段，声明过的变量（或者函数）
  - 形参也是私有变量

```
function fn(num1, num2) {
        var total = num1+num2;
        return total;
    }
    var result = fn(100, 200)
```

- 函数执行形成一个新的私有作用域
- 形参赋值
- 变量提升
- 代码自上而下执行
- 当前栈内存（私有作用域）销毁或者不销毁

```
    var x=10,y=20,z=30;
    function fn(x,y) {
    console.log(x,y,z);
    var x = 100;
    y=200;
    z=300;
    console.log(x,y,z)
    }
    fn(x,y,z)
    console.log(x,y,z)

    // 2 CASE
    function fn(b) {
        console.log(b);
        function b() {
            alert(b)
        }
        b();
    }
    fn(1);
```

如何查找上级作用域

```
var n = 10;
function sum( ) {
  console.log(n)
}
sum();

~funciton(){
```

```
  var n = 100;
  sum();
}
```

- 函数执行形成一个私有的作用域（A），A的上级作用域是谁，和他在哪执行的没关系，主要看他是在哪个作用域下定义的，当前A的上级作用域就是谁；

```
var n = 10;
var obj = {
  n:20,
  fn:(function(){
      var n = 30;
      return function(){
        console.log(n)
      }
  })()
}
obj.fn()


var n = 10;
var obj = {
  n:20,
  fn:(function(n){
      return function(){
        console.log(n)
      }
  })(obj.n)
}
obj.fn()
// cannot read  property 'n' of undefined
```

闭包作用之保护

- 形成私有作用域，保护里面的私有变量不受外界干扰
- Query: 常用的js类库，提供了很多项目中常用的方法（兼容所有浏览器） Zepto: 小型JQ，专门为移动端开发准备的

```
(function(window, undefined){
    var jQuery = function(){
        ......
    }
    ......
    window.jQuery = window.$ = jQuery
})(window)
```

- 真实项目中，我们利用这种保护机制，实现团队协作开发（避免了多人同一个命名，导致代码冲突的问题）

闭包作用保存

- 函数执行形成一个私有作用域，函数执行完成，形成的这个栈内存一般情况下都会自动释放
  其他情况：函数执行完成，当前私有作用域（栈内存）中的某一部分内容被栈内存以外的其他东西（变量/元素的事件）占用了，当前的栈内存就不能释放掉，也就形成了不销毁的私有作用域（里面的私有变量也不会销毁）

```
function fn() {
    var i = 1;
    return function(n) {
     console.log(n + i++);
    }
}
var f = fn();
f(10);//11
fn()(10);//11
f(20)//22
fn()(20)//21
```

js中this指向问题

- 当前函数执行的主体（谁执行的函数this就是谁）
- 函数外面的this是window,外面一般都研究函数内的this指向问题
- this是谁和他在哪定义以及在哪执行的没有任何关系

```
    function eat() {

    }
    people.eat()

    // ----------
    var obj = {
        name:'obj',
        fn: function() {
            console.log(this)
        }
    }
    obj.fn();
    var f = obj.fn;
    f();
```

js非严格模式下（默认就是非严格模式）

- 自执行函数中的this一般都是window

```
var obj = {
    fn: (function(){
        //this->window
        return function(){}
    })()
}
```

- 2 给元素的某个事件绑定方法，当事件触发执行对应方法的时候，方法中的this一般都是当前操作的元素本身

```
oBox.onclick = function(){
    //this -> oBox
}
```

- 3 还有一种方式快速区分this;当方法执行的时候，看看方法名前面是否有点，有点，点前面是谁this就是谁，没有点this一般都是window
- 在js严格模式下（让js更加严谨）
- 开启严格模式： 在当前作用域的第一行加上“use strict”，开启严格模式，那么当前作用域下再执行的js代码都是严格模式处理的（包含了函数中的代码） 在js严格模式下，如果执行主体不明确，this指向是undefined(非严格模式下指向的是window)

```
function fn()  {
    console.log(this)
}
fn(); this-> undefined
window.fn(); this -> window
```

- 综合题

```
var num = 1,
        obj = {
            num: 2,
            fn: (function(num) {
                this.num *= 2;
                num +=2;
                return function() {
                    this.num *= 3;
                    num++;
                    console.log(num);
                }
            })(num)
        };
```

```
    var fn = obj.fn;
    fn();
    obj.fn();
    console.log(num, obj.num);
```

综合题2

```
for(var i = 0; i< oList.length;i++) {
        oList[i].onclick = function() {
            onchange(i)
            // => 不行的原因，给当前Li点击事件绑定方法的时候，绑定的方法并没有执行（点击的时候才执行）；循环3次，分别给3个lI的点击事件绑定了方法，循环完成后i=3(全局的)；当点击的时候，执行绑定的方法，形成一个私有的作用域，用到了变量i,i不是私有的变量，向全局查找，此时全局的i已经是最后的循环的3了；
            // -> 自定义属性解决
            // -> 闭包解决
            // -> let
        }
    }
```

闭包汇总

- 函数执行，形成一个私有作用域，保护里面的私有变量不受外界的干扰，这种保护机制叫做闭包 但是现在市面上，99%的IT开发者都认为： 函数执行，形成一个不销毁的私有作用域，除了保护私有变量以外，还可以存储一些内容，这样的模式才是闭包