创建对象、数据类型，过程式与面向对象

1. 数据类型

javascript 变量包含两种不同数据类型的值：基本类型和引用类型

1.1.基本类型

值类型(基本类型)：基本类型值指的是简单的数据段，包括 es6 里面新增的一共是有 6 种，

字符串（String）、数字(Number)、布尔(Boolean)、对空（Null）、未定义（Undefined）、

Symbol。

```
var str = '123';
var num = 12;
var bol = true;
var und;
var nul = null;
var sym = Symbol('name');
```

1.2 引用类型

引用数据类型：对象(Object)、数组(Array)、函数(Function)。

```
var obj = {};
var arr = [];
var fun = function(){ }
```

引用类型（object）是存放在堆内存中的，变量实际上是一个存放在栈内存的指针，这个指针指向堆内存中的地址。每个空间大小不一样，要根据情况来进行特定的分配，例如。

```
var person1 = {name:'jozo'};
var person2 = {name:'xiaom'};
var person3 = {name:'xiaoq'};
```

引用类型的值是可变的：

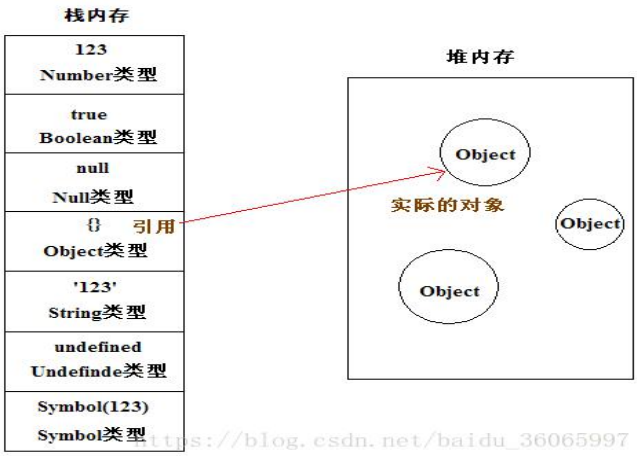
```
person1['name'] = 'muwoo'
console.log(person1) // {name: 'muwoo'}
```

1.3 基本类型 和 引用类型 的区别

javascript 的变量的存储方式：栈内存（stack）和堆内存（heap）

栈：自动分配内存空间，系统自动释放，里面存放的是基本类型的值和引用类型的地址

堆：动态分配的内存，大小不定，也不会自动释放。里面存放引用类型的值。



基本数据类型是按值访问的，因为可以操作保存在变量中的实际的值。
引用类型的值是保存在内存中的对象。JavaScript 不允许直接访问内存中的位置，也就是说不能直接操作对象的内存空间。 在操作对象时， 实际上是在操作对象的引用而不是实际的对象。

传值与传址

基本类型与引用类型最大的区别实际就是 传值与传址 的区别

值传递：基本类型采用的是值传递。

地址传递：引用类型则是地址传递，将存放在栈内存中的地址赋值给接收的变量。

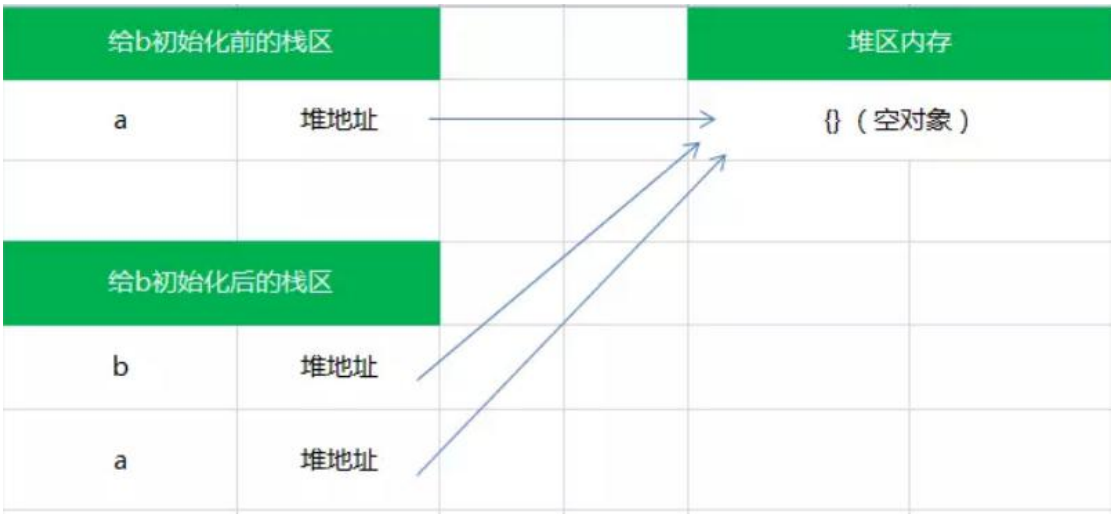
在我们进行赋值操作的时候，基本数据类型的赋值（=）是在内存中新开辟一段栈内存，然后再把再将值赋值到新的栈中。例如:

```
var a = 10;
var b = a;
a ++ ;
console.log(a); // 11
console.log(b); // 10
```

所以说，基本类型的赋值的两个变量是两个独立相互不影响的变量。

但是引用类型的赋值是传址。只是改变指针的指向，例如，也就是说引用类型的赋值是对象保存在栈中的地址的赋值，这样的话两个变量就指向同一个对象，因此两者之间操作互相有影响。例如：

```
var a = {}; // a 保存了一个空对象的实例
var b = a; // a 和 b 都指向了这个空对象
a.name = 'jozo';
console.log(a.name); // 'jozo'
console.log(b.name); // 'jozo'
b.age = 22;
console.log(b.age);// 22
console.log(a.age);// 22
console.log(a == b);// true
```



浅析 JS 中的堆内存与栈内存

栈内存中的变量一般都是已知大小或者有范围上限的，算作一种简单存储。而堆内存存储的对象类型数据对于大小这方面，一般都是未知的。个人认为，这也是为什么 null 作为一个 object 类型的变量却存储在栈内存中的原因。

因此当我们定义一个 const 对象的时候，我们说的常量其实是指针，就是 const 对象对应的堆内存指向是不变的，但是堆内存中的数据本身的大小或者属性是可变的。而对于 const 定义的基础变量而言，这个值就相当于 const 对象的指针，是不可变。

既然知道了 const 在内存中的存储，那么 const 定义的变量不能二次定义的流程也就比较容易猜出来了，每次使用 const 去初始化一个变量的时候，会首先遍历当前的内存栈，看看有没有重名变量，有的话就返回错误。

说到这里，有一个十分容易忽略的点，之前也是自己一直没有注意的就是，使用 new 关键字初始化之后是不存储在栈内存中的。为什么呢？new 大家都知道，根据构造函数生成新实例，这个时候生成的是**对象**，而不是基本类型。再看一个例子

```
var a = new String('123')
var b = String('123')
var c = '123'
console.log(a==b, a===b, b==c, b===c, a==c, a===c)
>>> true false true true true false
console.log(typeof a)
>>> 'object'
```

我们可以看到 new 一个 String，出来的是对象，而直接字面量赋值和工厂模式出来的都是字符串。但是根据我们上面的分析大小相对固定可预期的即便是对象也可以存储在栈内存的，比如 null，为啥这个不是呢？再继续看

```
var a = new String('123')
var b = new String('123')
console.log(a==b, a===b)
>>> false false
```

很明显，如果 a，b 是存储在栈内存中的话，两者应该是明显相等的，就像 null === null 是 true 一样，但结果两者并不相等，说明两者都是存储在堆内存中的，指针指向不一致。

说到这里，再去想一想我们常说的值类型和引用类型其实说的就是栈内存变量和堆内存变量，再想想值传递和引用传递、深拷贝和浅拷贝，都是围绕堆栈内存展开的，一个是处理值，一个是处理指针。

主要区别总结：

1. 值是否可变

基本类型的值是不可变的，除非重新给他赋值，引用类型的值是可变的

```
var name =  ‘jozo';
 name.toUpperCase(); // 输出 'JOZO'
 str[0] = 'mm';
 console.log(name); // 输出  'jozo'

name =’song’;
console.log(name); //song
```

```
//创建个空对象 --引用类型
var person = {};
person.name = 'jozo';
```

2. 比较，是否 2 个数据类型相等

基本类型的比较是值的比较

```
var a = 1;
var b = true;
console.log(a == b);//true
```

引用类型的比较是引用的比较

```
var person1 = '{}';
var person2 = '{}';
console.log(person1 == person2); // true
var person1 = {};
var person2 = {};
console.log(person1 == person2); // false
```

3. 值存放的位置

基本类型的变量是存放在栈区的（栈区指内存里的栈内存），引用类型的值是同时保存在栈内存和堆内存中的对象

```
var person1 = {name:’ jozo'};
var person2 = {name:’ xiaom’ };
```

4. 复制的情况

复制时，基本类型是直接复制了一个新的变量，新旧两个变量之间没有关系

引用类型复制了新的变量，但这个变量是一个指针，新旧两个指针指向同一个对象

基本类型举例：

```
var a = 10;
var b = a;

a ++ ;
console.log(a); // 11
console.log(b); // 10
```

引用类型举例：

```
var a = {}; // a 保存了一个空对象的实例
var b = a;  // a 和 b 都指向了这个空对象

a.name = 'jozo';
console.log(a.name); // 'jozo'
console.log(b.name); // 'jozo'

b.age = 22;
console.log(b.age);// 22
console.log(a.age);// 22


console.log(a == b);// true
```

内存分配和垃圾回收

一般来说栈内存线性有序存储，容量小，系统分配效率高。而堆内存首先要在堆内存新分配存储区域，之后又要把指针存储到栈内存中，效率相对就要低一些了。
垃圾回收方面，栈内存变量基本上用完就回收了，而推内存中的变量因为存在很多不确定的引用，只有当所有调用的变量全部销毁之后才能回收。

JavaScript 变量均为对象。当您声明一个变量时，就创建了一个新的对象。

关于值传递　(函数内部的更改不影响外部)

```
var num = 50;
function add(num){
    num=100;
}
add(num);
console.log(num);
```

引用(指针 | 快捷方式)传递 （函数内部的更改会影响外部）

```
var arr = [10,20,30];
function add2(arr){
    arr[0]=99;
}
add2(arr);
console.log(arr);
```

1.4 声明变量类型

当您声明新变量时，可以使用关键词 "new" 来声明其类型：

```
var car=new String;
var x=  new Number;
var y=  new Boolean;
var cars=   new Array;
var person= new Object;
```

1.5 如何判断数据类型呢？

基本数据类型的检测：使用 typeof
typeof 2 // number
typeof null // object
typeof {} // object
typeof [] // object
typeof (function(){}) // function
typeof undefined // undefined
typeof '222' // string
typeof true // boolean

引用类型(对象类型)检测：使用 instanceof （不讲）

instanceof 用于判断一个变量是否某个对象的实例

对象的原生原型扩展函数： Object.prototype.toString.call
Object.prototype.toString.call: 对象的一个原生原型扩展函数,用来更精确的区分数据类型

Object.prototype.toString.call('aaaa') // [object String]
Object.prototype.toString.call(2222) // [object Number]
Object.prototype.toString.call(true) // [object Boolean]
Object.prototype.toString.call(undefined) // [object Undefined]
Object.prototype.toString.call(null) // [object Null]
Object.prototype.toString.call({}) // [object Object]
Object.prototype.toString.call([]) // [object Array]
Object.prototype.toString.call(function(){}) // [object Function]

4. constructor（不讲）

constructor 也能判断数据类型

如： var obj= new Object() obj.constructor==Object

js 里面还有好多类型判断

[object HTMLDivElement] div 对象 ，

[object HTMLBodyElement] body 对象,
[object HTMLCollection]. HTML 元素的集合，它提供了可以遍历列表的方法和属性
各种 dom 节点的判断，这些东西在我们写插件的时候都会用到

2. 对象的创建

2.1 object 类型构造函数

```
var obj1 = new Object();
var arr1 = new Array();
var arr1 = new String();
// 添加属性
obj1 .name = "小明";
// 添加方法
obj.sleep = function () {
    console.log(this.name + "在睡觉");
};
```

2.2 字面量定义 （简单的字面量 ， 嵌套字面量 ）

```
var obj2= {};
// obj2.name='laney';
// obj2.age = '33';
obj2['name'] = 'laney';
obj2['age'] = '22';


var obj3 = {
    name:'laney',
    age:'234',
    action:function(){}
}
var obj4 = {
    name:'song',
```

```
    age:'12',
    action:function(){}
}
```

2.3 工厂方式定义

```
    function createObj(name,age) {
        var obj = new Object();
          obj.name = name;
          obj.age = age;
          obj.action = function(){
              console.log(obj.name);
          }
        return obj;
      }

var c1 = createObj('xiaohong',’20’);
var c2 = createObj('xiaoming',22);
```

2.4 构造函数方式

```
// 注意：此处建议方法名首字母大写

 function Person(name) {
          this.name=name;
          this.age=age;
          this.action = function(){
            console.log(this.name);
          }
    }
var p1 = new Person(‘xiaohong’);
var p2 = new Person(‘xiaoming);
```

2.5 Object.create

```
Object.create(prototype)
```

```
var p = {name:'o4'}
var Obj4 = Object.create(p);
```

4. 过程式开发与面向对象开发

面向过程 （洗衣服）-- 站在一个执行者的角度去做事情

洗衣服作为例子：

```
// 1. 找个盆
// 2. 收集要洗的衣服
// 3. 放水放洗衣液
// 4. 洗一洗
// 5. 晒一晒
```

面向对象 – 站在指挥者的角度，是一种开发思想

```
// 1. 找个对象
// 2. 让他去洗
// 3. 检查是否洗好并晾晒
```

或者扫地 也是，自己亲自拿着扫帚 扫地， 1. 找个扫帚 2. 开始扫 3. 扫地完成 4. 工具放到原位

扫地机器人： 1. 按开关选择程序 2. 让扫地机器人去扫 3. 扫完检查

面向对象的三大特性：

封装

隐藏对象的属性和实现细节，仅对外提供公共访问方式，将变化隔离，便于使用，提高复用性和安全性。

继承

提高代码复用性；继承是多态的前提。

多态

多态性是指相同的操作或函数、过程可作用于多种类型的对象上并获得不同的结果。不同的对象，收到同一消息可以产生不同的结果，这种现象称为多态性。

五大基本原则：

单一职责原则
类的功能要单一，不能包罗万象，跟杂货铺似的。
开放封闭原则
一个模块对于拓展是开放的，对于修改是封闭的，想要增加功能热烈欢迎，想要修改，哼，一万个不乐意。
接口分离原则
设计时采用多个与特定客户类有关的接口比采用一个通用的接口要好。

最后
1、抽象会使复杂的问题更加简单化。
2、从以前面向过程的执行者，变成了张张嘴的指挥者。
3、面向对象更符合人类的思维，面向过程则是机器的思想

实例分析

<!-- 需求： 需要点击一个按钮，可以让所有的正方形变成圆形,以及背景颜色和 字体发生变化 -->

HTML:

```
  <ul class="box-list">
     <li>软谋</li>
     <li>软谋</li>
     <li>软谋</li>
     <li>软谋</li>
  </ul>
 <div class="btn-group">
  <button type="button" class="btn-begin">开始变化</button>
  <button type="button" class="btn-stop">恢复状态</button>
</div>
```

Js：

```
// 面向过程
  // //第一步，找出 class 名为 sign 的所有的正方形,开始按钮， 停止按钮
  var $allOneLi = $('.box-list li');

  //第二步，找到开始按钮
  $('.btn-begin').on('click',function(){
    $allOneLi.css({background:'red','border-radius':'50%',color:'#fff'});
    setTimeout(function(){
      $allOneLi.css({background:'organge','border-radius':'50%',color:'#fff'});
    },5000);
  })

  $('.btn-stop').on('click',function(){
    $allOneLi.removeAttr('style');
  })


//面向对象
  var funObj = {
    beginAction:function(obj){
      var objInit = {
        el:'',
        bg:'red',
        radius:'50%',
        color:'#fff',
        tobg:'#000'
      }

      var endobj = Object.assign({},objInit,obj);
        $(obj.el).css({background:endobj.bg,'border-radius':endobj.radius,color:endobj.color});

        setTimeout(function(){
          $(obj.el).css({background:endobj.tobg,'border-
radius':endobj.radius,color:endobj.color});
        },2000);
    },
    endAction:function(){
      $(obj.el).removeAttr('style');
    }
  }

  $('.btn-begin').on('click',function(){
    funObj.beginAction({
```

```
        el:'.box-list li',
        bg:'blue',
        radius:'50%',
        color:'#fff',
        tobg:'#23a6c0'
    });
})

$('.btn-stop').on('click',function(){
    funObj.endAction();
})
```

Array 扩展

① Array.prototype.indexOf(value) : 得到值在数组中的第一个下标
② Array.prototype.lastIndexOf(value) : 得到值在数组中的最后一个下标
③ Array.prototype.forEach(function(item, index){}) : 遍历数组
④ Array.prototype.map(function(item, index){}) : 遍历数组返回一个新的数组，返回加工之后的值
⑤ Array.prototype.filter(function(item, index){}) : 遍历过滤出一个新的子数组， 返回条件为 true 的值

```
/*
 需求:
 1. 输出第一个 6 的下标
 2. 输出最后一个 6 的下标
 3. 输出所有元素的值和下标
 4. 根据 arr 产生一个新数组,要求每个元素都比原来大 10
 5. 根据 arr 产生一个新数组, 返回的每个元素要大于 4
 */
var arr = [2,4,3,1,2,6,5,4];
console.log(arr.indexOf(4));
console.log(arr.lastIndexOf(4));
arr.forEach(function(item, index){
  console.log(item, index);
})
var arr1 = arr.map(function(item, index) {
  return item + 10;
});
console.log(arr1);
var arr2 = arr.filter(function(item, index){
```

```
  return item > 3;
});
console.log(arr, arr2);
```