\*. 数据类型

基本类型：Undefined / Null / Boolean / Number / String

引用类型：Object / Array / Date / RegExp / Function

内置对象：Global / Math

1. 使用 typeof 检测变量类型： typeof variable //也可以是 typeof(variable) ，取值有以下几种：

undefined //变量未初始化 或 未声明

boolean 、number 、string

object //变量值为 对象 或者 null

function //变量值是函数

注：正则表达式 typeof 取值 -- function（safari5及之前、chrome7及之前版本）

-- object （其它浏览器）

2. ECMAScript 中，函数是对象，不是数据类型。

3. null 空对象指针，通常用来标志对象变量未给定值 或者用来解除引用。

undefined 派生自 null：if ( undefined == null ) //true

4. 属性和方法

Boolean: 转型函数 Boolean(var)

Number: 转型函数 Number(var) 适用于任何类型

而 parseInt(var) 和 parseFloat(var) 专门用于字符串转换成数值

最值

Number.MIN\_VALUE / Number.MAX\_VALUE / isFinite(var)

-Infinity / Infinity / Number.NEGATIVE\_INFINITY / Number.POSITIVE\_INFINITY

非法数值

inNaN(var) 注NaN与任何值都不相等，包括NaN本身： if(NaN == NaN) //false

String: 转型函数 String(var)

String() 调用 toString()；值为 null 则返回 null；值为 undefined 则返回 undefined。

Boolean/Number/String/Object 都有 toString(); 但 Null 和 Undefined 没有这个方法。

toString() 转换数值时可以带参（输出数值的基数）。

Object: constructor 构造函数 Object()

valueOf() / toLocaleString()/ toString()

5. 操作符

\*. 一元加和减：相当于 Number()；对象则先调用 valueOf() / toString() 再转换得到的值。

\*. 按位非 NOT：操作数的负值减1。

\*. 左移 << ：不影响操作数的符号位。

\*. 右移 >> ：有符号的右移，用符号位的值来填充所有空位。

\*. 右移 >>>：无符号的右移，用 0 填充所有空位。

\*. 全等 === ，不全等 ！==

\*. 逗号操作符：返回表达式中的最后一项。var num = ( 5, 1, 4, 8 ); // num值为8

6. 语句

\*. for-in 语句 // for(var i in arr) { arr[i] = i; }

\*. label 语句 // 标签为了之后引用

\*. with 语句 // 将作用域添加到某对象中 with(exp) statement; 则 statement 作用域在 exp 对象中。

\*. try-catch 语句

7. 函数

\*. arguments

\*. 没有函数签名，不能重载。

\*. 方法链 （方法中 return this. 即返回对象自身，则可继续 .引用自身的其它方法）

8. 基本类型 vs 引用类型

\*. 给引用类型动态地添加属性和方法，如下：

var person = new Object();

person.name = "Nic";

但不能给基本类型添加属性和方法，如下：

var name = "Nic";

name.age = 21; //这是不对的

\*. 基本类型变量赋值，新建内存，值复制，两个变量之间相互独立：

var age1 = 21; var age2 = age1; //age1和age2分配在不同的内存，互不影响

引用类型变量赋值，传递的是指针，指向堆内存中的对象，两个变量引用同一个对象，相互影响：

var obj1 = new Object(); obj1.age = 21; var obj2 = obj1; //obj1和obj2占同一个内存相互影响。

\*. 参数传递（P70~71）

基本类型作参数，是按值传递的，外部不起作用；

引用类型作参数，即传送地址，外部起作用，局部作用域的修改会在全局作用域中反映出来。

\*. 检测类型

基本类型 typeof variable

引用实例 variable instanceof constructor //所有引用类型的变量都是 Object 的实例

9. 执行环境 作用域链

当执行环境中的所有代码执行完毕后，该环境被销毁，保存在其中的所有变量和函数定义也随之销毁

\*. 变量对象：环境中定义的所有变量和函数都保存在这个对象中。

\*. 全局执行环境：window对象，所有全局变量和函数都是作为window对象的属性和方法创建的。

\*. 函数执行环境：环境是函数，则将函数的活动对象如 arguments 等 作为环境的变量对象。

\*. 作用域链：对执行环境中所有变量和函数的有序访问，作用域链的前端，是当前执行的代码所在的环境的变量对象，下一个变量对象来自包含（外部）环境，而再下一个变量对象则来自更外层的包含环境，这样一直延续到全局执行环境；全局执行环境的变量对象始终是作用域链中的最后一个变量对象。

\*. 标识符解析：沿着作用域链，从作用域链的前端开始，逐级向后搜索，直到找到标识符为止。

\*. 延长作用域链：with语句 和 try-catch语句

\*. 没有块作用域，语句块内部定义的变量会添加到语句块所在的执行环境，语句块外可访问。

10. 内存管理 垃圾收集

\*. 解除引用，手工级给变量赋值 null，让变量脱离执行环境，以便垃圾收集器下次运行时将其回收。

11. 引用类型

\*. Object 类型

a. 创建

new Object()

对象字面量 var person = { 属性：值 }; / var person = {}; person.属性 = 值;

b. 引用：点引用法 . /方括号法 [ ]

c. 对象都有 toString() / toLocaleString() / valueOf() 方法；

d. object 是基础类型，任何其它所有类型（引用类型和基本包装类型）都继承了 object 的方法。

\*. Array 类型

a. 创建

new Array() / new Array(length) / new Array(a1, a2)

数组字面量 var colors = [ "red", "blue" ];

b. 检测数组 Array.isArray(variable);

c. 排序 array.sort()

\*. 此方法使用 toString() 比较的是字符串， 即使数组中每一项均为数值，如：

var arr = [0, 1, 5, 10, 15] 用 arr.sort() 排序 后 [0, 1, 10, 15, 5]

\*. 使用 比较函数 作为参数，以确定升序还是降序

function compare(value1, value2){ // 小-1升序

if ( value1 < value2 ){

return -1;

} else if (value1 > value2 ){

return 1;

}

else return 0;

}

或者

function compare(value1, value2){

return value1 - value2; // 升序 反过来是降序

}

arr.sort(compare); //[0, 1, 5, 10, 15]

\*. Date 类型

\*. RegExp 类型

a. 创建

var expression = /pattern/flags; // g i m

new RegExp(pattern, flags)

\*. Function 类型

a. 创建

函数声明 function fun(arg) { ... } //随处声明定义，不影响调用

函数表达式（匿名函数） var fun = function(arg) { ... }; //先定义，后调用

命名函数表达式 var fun = function functionname(arg){ ... }; //调用 fun();

b. 函数内部属性

\* arguments --- callee 指向拥有 arguments 对象的函数 arguments.callee

另： caller 指向调用当前函数的函数，与 arguments 没有关系

\* this --- 引用当前函数执行的环境对象

\* length --- 函数接收的命名参数的个数 （是否与 arguments.length 一致？）

\* prototype --- 用来保存实例的所有方法？ 如 toString() 、valueOf() 等

\* obj.apply(obj的作用域对象，参数数组) / call(obj的作用域对象，参数逐个列举)

\* bind 绑定函数到变量对象返回函数实例 var fun2 = fun1.bind(obj);

\*. 基本包装类型

基本包装类型为 Boolean / Number / String，不建议显式创建基本包装类型的对象，

基本类型可以使用其对应的基本包装类型的属性和方法，但不能创建自己的属性和方法。

\*. 基本类型不是谁的实例，var num = 12; //typeof为number；instanceof Number 为 false

\*. var num = Number(val); //转型函数（typeof num 为 number）

var obj = new Number(val); //构造函数（typeof val 为 object）

\*. 内置对象

\*. Global：所有在全局作用域中定义的属性和方法，都是 Global 对象的属性和方法。（window）

encodeURI() / decodeURI() / encodeURIComponent() / decodeURIComponent() / eval()

特殊值 undefined / NaN / Infinity 及其它原生引用类型 Object 等都是 Global 的属性。

\*. Math：min() / max() / ceil() / floor() / round() / random()

\*. 创建对象

1. 工厂模式 （使用函数封装，无法识别对象细节）

2. 构造函数模式 （每个实例有自己的副本，方法重建；若方法抽出构造函数体外则对象没有封装性）

3. 原型模式 （凡 prototype 的属性和方法，则所有实例共享；没 prototype 的则创建自己的副本）

（原型对象/对象实例、 实例中的指针仅指向原型，而不指向构造函数；

重写原型对象相当于新建一个原型而不是修改原先的原型，并且对象指向新的原型，

即切断了构造函数与最初原型之间的联系 --- P157）

（对象实例的属性，会屏蔽原型对象中的同名属性，但不会修改它，使用 delete 可恢复访问）

4. 组合使用构造函数模式和原型模式

（构造函数定义属性，原型模式定义方法和需要共享的属性）

5. 动态原型模式

（所有信息封装在构造函数内部，通过判断是否已创建来创建原型内容，仍然是只创建一次）

\*\* 构造函数 VS 原型 VS 实例

（new 后面的函数是为构造函数；prototype 定义的属性和方法是为原型的属性和方法；

每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，

实例都包含一个指向原型对象的内部指针

）如：function Person(arg){ ... }

var per = new Person(...);

则：构造函数 Person 内部的prototype 指向-->原型 Person Prototype

原型 Person Prototype 内部的constructor 指向-->构造函数 Person

实例 per 内部的[[Prototype]] 指向-->原型 Person Prototype

\*\* 创建对象 、创建对象实例、创建子类（继承）

已创建好对象A：（原型模型创建对象 prototype属性和方法被实例引用，非prototype不被引用 ）

var a = new A(); //创建对象A的实例

var B.prototype = new A(); //创建对象A的子类B（B继承A）

\*\* 创建实例要不要使用 new（详见 L20\_inherit 比较 app1 和 app2）

工厂模式：return 对象，所以以一般函数方式，不需要 new；

构造函数模式：非 return对象，所以以创建对象方式，需要 new 。

\*. 继承

1. 原型链 （重写原型：SubType.prototype = new SuperType(); 但无法向超类构造函数传递参数）

2. 借用构造函数 （子类构造函数内扩展超类 function SubType() { SuperType.call(this, args); }）

3. 组合继承 （借用构造函数+原型链）

function SubType(argss) {

SuperType.call(this, args); //继承超类的属性，同时传递参数

this.xxx = xxx;

}

SubType.prototype = new SuperType(); //指向超类的原型，实现方法共享

SubType.prototype.constructor = SubType; //指向自身的构造函数，实现参数传递

（2 次调用超类构造函数：一为改写子类原型指向超类时，二为调用子类构造函数时）

（2次调用，是指创建子类时修改子类原型 SubType.prototype = new SuperType() 时调用1次；

而每次创建实例时调用1次超类构造函数和1次子类构造函数。即只在初始2次调用超类构造函数）

（SubType.prototype = new SuperType(); 在子类构造函数的外面，即此句被立即执行；

而工厂模式 var SubType = SuperType(); 在子类构造函数的内部，即函数声明调用时才执行；

这是组合继承多1次调用超类构造函数的原因！详见 L20\_inherit 比较 app1 和 app2）

4. 原型式继承（基于已有对象创建新对象）

5. 寄生式继承

6. 寄生组合式继承 （\*引用类型最理想的继承方式）

（使用超类原型的一个副本，而避免调用超类构造函数，从而改进“组合继承”2次调用的问题）

（使用寄生式继承来继承超类的原型，再将结果指定给子类的原型：

SubType.prototype = new SuperType(); //指向超类的原型，实现方法共享

SubType.prototype.constructor = SubType; //指向自身的构造函数，实现参数传递

将以上修改为：

function inheritPrototype(subType, superType) {

var prototype = object(superType.prototype);

prototype.constructor = subType;

subType.prototype = prototype;

}

inheritPrototype(SubType, SuperType);

\*. 函数

1. 函数调用时，创建执行环境和对应的作用域链，作用域链为指向变量对象的指针链表，

变量对象包括函数本地活动对象（arguments、其它命名参数、局部变量）和全局变量对象。

局部活动对象处在执行环境作用域链的前端最先被搜索；函数执行完毕后，局部活动对象被销毁。

2. 闭包：指有权访问另一个函数作用域中的变量的函数。（函数内部创建一个匿名函数表达式）

内部函数将外部函数的活动对象添加到自己的作用域链中，所以：

当外部函数执行完毕，其执行环境的作用域链会被销毁，但其活动对象不会被销毁；

只有当内部函数的引用被销毁后，外部函数的活动对象才会被销毁。（见P180）

外部函数不能访问内部函数的局部变量（包括参数），

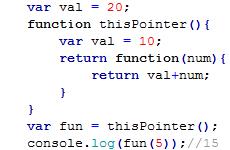
内部函数能访问外部函数的局部变量（包括参数），

但不能通过 arguments 访问外部函数的参数，因为 arguments 只能指向内部函数自身的参数。

3. 闭包中的 this 指针

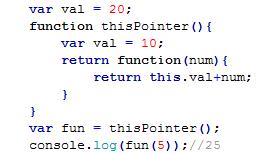
this 对象是在运行时基于函数的执行环境绑定的；由调用该匿名函数的执行环境决定。

则沿作用域链从前端开始搜索，所以 val 取的是外部函数中活动对象，而不是全局变量对象。



\* 匿名函数 中限定 this

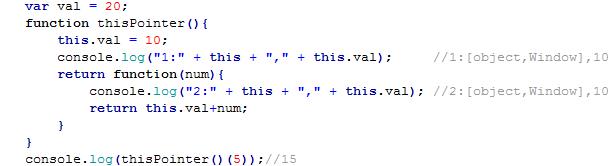
则由调用该匿名函数的执行环境决定，匿名函数返回给 fun 在全局中，故 this 为全局变量对象。



\* 函数声明 中的 this

函数声明引用的 this 为该函数的执行环境对象，本例中为 window；

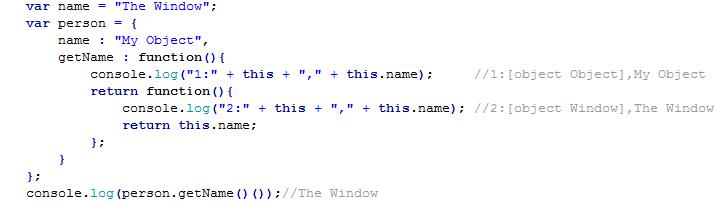
函数声明中定义 this.val 变量被视为一般变量，按作用域链方法搜索，取最前端的属性值。



\* 引用类型 中的 this

引用类型定义中的 this 指向该引用类型本身（这里是“对象字面量”表示法）；

而在匿名函数中，因为其返回到全局并被调用，指向全局执行环境，故 this 为全局变量对象。



4. 闭包与语句块变量作用域不是一回事：

语句块内定义的变量的作用域为语句块所在的执行环境，而不是仅是语句块内部。

闭包作用域包括外部函数的活动对象，即能访问外部函数的局部变量（包括参数）； 当外部函数执行完毕，其执行环境的作用域链会被销毁，但其活动对象不会被销毁；只有当闭包引用被销毁后，外部函数的活动对象才会被销毁。

5. 模仿块级作用域，使其内部变量的作用域私有，外部不能访问：

（function(){ //这也是匿名函数，定义后立即调用执行

... //块级作用域，私有属性

}）(); //立即执行，函数内部的所有变量（除了闭包及其外部活动对象）都会被立即销毁

\* 函数表达式和函数声明一样，并不是立即调用的；函数声明置于函数调用前后都可以，

而函数表达式必须要先定义然后才能调用。

6. 特权方法：有权访问私有变量和私有函数的公有方法。

（封装对象：公有、私有 属性和方法）

\* 在构造函数中定义特权方法：

function MyObject (){

//私有变量

var privateVariable;

//私有函数

function privateFunction(){

...

}

//公有方法

this.publicMethod = function(){

...

};

}

\* 通过原型模式实现特权方法：（使用原型，则变量为静态私有变量，为所有实例共享）

(function(){

//私有变量

var privateVariable;

//私有函数

function privateFunction(){

...

}

//使用函数表达式的方式定义构造函数 （不要 var 声明，即为全局变量，外部可使用）

MyObject = function(){

...

};

//公有方法

MyObject.prototype.publicMethod = function(){

...

};

})();

var obj = new MyObject();

obj.publicMethod();

\* 模块模式：单例创建私有变量和特权方法（以返回对象字面量的形式）

var singleton = function(){

//私有变量

var privateVariable;

//私有函数

function privateFunction(){

...

}

//特权/公有属性和方法

return {

publicVariable : ... ,

publicMethod : function(){

...

}

};

}();

\* 增强的模块模式：单例创建私有变量和特权方法

var singleton = function(){

//私有变量

var privateVariable;

//私有函数

function privateFunction(){

...

}

//创建对象

var object = new CustomType();

//添加特权/公有属性和方法

object.publicVariable = ... ;

object.publicMethod = function(){

...

};

//返回这个对象

return object;

}();

\* 在 (function(){...})(); 内部按一般方式创建对象 A，然后 window.A = A;

以此方式将 A 传递给 window，则外部可访问 A 。