17.3 使用变量

变量与值是两个不同的概念：变量相当于容器，而值相当于容器中的内容，为容器贴个标识符，就是变量名。程序能够根据变量名找到内容所在的位置，然后存取值。

17.3.1 声明变量

在 JavaScript 中，声明变量使用 var 语句：

```
var a;                  // 声明一个变量
var a, b, c;            // 声明多个变量
```

当声明多个变量时，应使用逗号运算符进行分隔变量名。

【示例 1】可以在声明中为变量赋值。未赋值的变量，则初始值为 undefined（未定义）值。

```
var a;                  // 声明但没有赋值
var b = 1;              // 声明并赋值
alert(a);               // 返回 undefined
alert(b);               // 返回 1
```

【提示】

变量命名应遵循 JavaScript 标识符命名规则。在计算机编程中，比较经典的变量命名法有三种。

1）匈牙利命名法

这种命名方法是由微软公司的一位程序员查尔斯•西蒙尼提出来的，匈牙利命名法被广泛应用于 Microsoft Windows 编程环境中。它通过在变量名前面加上相应的小写字母的符号标识作为前缀，标识出变量的作用域、类型等，前缀后面是一个或多个单词组合，单词描述了变量的用途。例如，i 表示整数，s 表示字符串，命名示例如下：

```
var sUserName = "css8", iCount = 0;
```

下表列举了匈牙利命名法定义 JavaScript 变量的前缀字符与数据类型对照表，如表 E17.4 所示。

表 E17. 4 匈牙利命名法前缀与变量类型

类型	前缀	示例
整数	i	iValue
浮点数	f	fValue
布尔型	b	bFound
字符串	s	sValue
数组	a	aValues
对象	o	oType
函数	fn	fnMethod
正则表达式	re	rePattern
泛型	v	vValue

2）骆驼式（Camel）命名法

骆驼式命令法是混合使用大小写字母来构成变量的名称。例如，下面分别用骆驼式命名法和下画线命名法定义同一个函数。

```
function printLoadTemplates(){}
function print_load_templates(){}
```

第一个函数名使用了骆驼式命名法，这种命名方法规定每一个单词首字母应使用大写字母来标记，而名称的首字母要小写，这与匈牙利命名法的名称首字母类似，第二个函数名使用了下画线法，函数名中的每一个单词都用一个下画线来标记。

骆驼式命名法近年来很流行，在很多新的语言和编程环境中，它应用得比较多。下画线命名法是 C 语言出现后开始流行起来的，在许多旧的程序和 UNIX 这样的环境中，它的使用非常普遍。

3）帕斯卡（Pascal）命名法

帕斯卡命名法与骆驼式命名法类似，只不过骆驼式命名法是第 1 个单词首字母小写，而帕斯卡命名法是第 1 个单词首字母要大写，例如，MyFunction 就是一个帕斯卡命名的示例，而 myFunction 是一个骆驼命名法。在 C#中，帕斯卡命名法和骆驼命名法使用比较多。

【拓展】

JavaScript 的变量没有类型之分，检测变量的类型，实际上是检测变量包含的值的类型。所以，同一个变量名，它的类型可能会随时变化。用户应该根据开发需要先检测变量类型，再根据类型进行处理。

在 JavaScript 中，可以重复声明同一个变量，也可以反复初始化变量的值。例如：

```
var a = 1;
var a = 2;
var a = 3;
alert(a);                    // 返回 3
```

JavaScript 允许用户不声明变量，而直接为变量赋值，这是因为 JavaScript 解释器能够自动隐式声明变量。但是隐式声明的变量总是作为全局变量而存在的。

【示例 2】当在函数中不声明就直接为变量赋值时，JavaScript 会把它视为全局变量进行处理。由于是全局变量，函数外代码可以访问该变量的值。

```
function f(){
    a = 1;                   // 未声明直接赋值
    var b = 2;               // 声明并赋值
}
f();                         // 调用函数，实现变量初始化
alert(a);                    // 返回 1
alert(b);                    // 提示语法错误，找不到该变量
```

但是，如果尝试读取一个未声明的变量的值，JavaScript 会提示语法错误。为变量赋值的过程，实际上 JavaScript 也会隐式进行声明。在使用变量时，用户养成良好习惯：先声明，后读写；先赋值，后运算。

【示例 3】下面示例设计：如果变量 a 不存在，就为其赋值为 0。但是变量 a 在未声明前就直接读取，将会抛出语法错误。

```
if(!a) {                     // 如果变量 a 不存在
   a = 0;                    // 则为变量 a 赋值为 0
}
alert(a);
```

针对上面代码，可以考虑使用属性法来读取。因为当读取一个未声明的属性时，JavaScript 不会报错。又有变量 a 是全局变量，作为全局变量，它应该是 window 对象的一个属性，所以可以这样来设计：

```
if(!window.a) {              // 如果 window 对象的属性 a 不存在
   a = 0;                    // 则为变量 a 赋值为 0
}
alert(a);
```

但是上述方法只适用全局变量，如果是局部变量，就只能够通过类型检测法来判断了：

```
(function f(){
    if(typeof a == "undefined"){    // 如果变量 a 的类型为不可知
        a = 0;                      // 则为变量 a 赋值为 0
    }
    alert(a);
})()
```

17.3.2 变量的作用域

变量的作用域（scope）是指变量在程序中可供访问的有效范围，也称为变量的可见区域。在 JavaScript 中，变量作用域可以分为全局作用域和局部作用域。

- 全局作用域是指变量在整个页面脚本中都是可见的，可以自由访问。
- 局部作用域是指变量仅能在声明的函数内部可见的，函数外是不允许访问的。

1．变量优先级

在函数体内，局部变量的优先级要比同名的全局变量高。此时，局部变量会覆盖同名全局变量。例如：

```
var a = 1;                    // 全局变量
(function f(){
    var a = 2;                // 局部变量
    alert(a);                 // 返回 2
})();                         // 直接在函数体上调用函数
```

如果在函数体内存在同名参数变量和全局变量，则参数变量的优先级要比同名全局变量高。例如：

```
var a = 1;                    // 全局变量
(function f(a){               // 参数变量
    alert(a);                 // 返回 3
})(3);                        // 直接调用函数，并传递参数值为 3
```

如果在函数体内存在同名参数变量和局部变量，则局部变量的优先级要比同名参数变量高。例如：

```
var a = 1;                    // 全局变量
(function f(a){               // 参数变量
    var a = 2;                // 局部变量
    alert(a);                 // 返回 2
})(3);                        // 直接调用函数，并传递参数值为 3
```

2．局部作用域嵌套

在 JavaScript 中，函数可以嵌套，也就形成了多个局部作用域嵌套的现象。

```
var a = 1;                    // 全局变量
(function(){
    var a = 2;                // 第 1 层局部变量
    (function(){
        var a = 3;            // 第 2 层局部变量
        (function(){
            var a = 4;        // 第 3 层局部变量
            alert(a);         // 返回 4
        })()                  // 直接调用函数
    })()                      // 直接调用函数
})()                          // 直接调用函数
```

在上面代码中，内层的局部作用域要比外层局部作用域的变量优先级高。在上面示例中，alert(a);语句最终显示的是 4，而不是其他局部变量值。

内层函数可以访问外层函数的变量，而外层函数却不能够访问内层函数的变量，这就是变量作用域链。

【示例 1】在 JavaScript 中，函数具有独立、封闭的作用域，用户可以利用这个特性使用函数封装代码，下面代码是 jQuery 框架的基本结构：

```
(function(){                  // 定义封装的独立作用域
    var jQuery = window.jQuery = window.$ = function( selector, context ){
        return new jQuery.fn.init( selector, context );
    };
    jQuery.fn = jQuery.prototype ={
        // 详细代码
```

```
    }
    jQuery.fn.init.prototype = jQuery.fn;
    jQuery.extend = jQuery.fn.extend = function(){
      // 详细代码
    }
})()                    // 直接调用函数
```

使用函数结构体的封装，可以有效避免了在同一个文档中多个技术框架或其他 JavaScript 代码之间的相互影响。如果用户在文档全局域中又定义了同名变量 jQuery 或$，不会覆盖 jQuery 框架。

```
(function(){            // 定义封装的独立作用域
    var jQuery=window.jQuery=window.$=function(selector, context ){ };
})()                     // 直接调用函数
var jQuery = 1;
var $ = 2;
```

在全局作用域中使用变量，可以不用 var 语句，但是在函数中声明局部变量时，一定要使用 var 语句。

【示例 2】下面示例演示了如果不显式声明局部变量所带来的后果。

```
var jQuery = 1;
(function(){
    jQuery = window.jQuery = window.$ = function(){};
})()
alert(jQuery);          // 结果读取了函数内部封装的代码
```

因此，在函数体内使用全局变量是一种很危险的行为，很可能函数就会改变程序中其他部分的使用值。为了避免此类问题的发生，应该养成在函数体内使用 var 语句声明局部变量。

【拓展】

JavaScript 在预编译期会先预处理声明的变量。但是，变量的赋值操作发生在 JavaScript 执行期，而不是预编译期。看下面这个示例：

```
function f(){
    a = 1;                  // 全局变量 a 赋值为 1
    var b = 2;              // 局部变量 b 赋值为 2
}
try{
    alert(a);               // 尝试读取全局变量 a
}
catch(e){
    alert(e.message);       // 显示错误信息：变量 a 未定义
}
f();                        // 调用函数
alert(a);                   // 读取全局变量 a，返回值为 1
```

通过上面示例，可以看出，在函数未调用之前，函数内部定义的全局变量是无效的，这是因为在 JavaScript 预编译期，仅对函数名、函数内各种标识符进行检索，建立索引。

只有当在 JavaScript 执行期时，才按顺序为变量进行赋值，并初始化。而在执行期，如果函数未被调用，则函数内代码是不被解析的，所以才有了上面看到的示例演示效果。

【示例 3】根据 JavaScript 解析过程，再看下面这个示例：

```
var a = 1;                  // 声明并初始化全局变量
(function f(){
    alert(a);               // 返回 undefined
    var a = 2;              // 声明并初始化局部变量
    alert(a);               // 返回 2
})()
```

上面代码显示，由于在函数内部声明了一个同名局部变量 a，所以在预编译期，JavaScript 就使用该变量覆盖掉全局变量对于函数内部的影响。而在执行初期，局部变量 a 未赋值，所以在函数第 1 行代码中读取局部变量 a 的值也就是 undefined 了。当执行到函数第 2 行代码时，则为局部变量赋值 2，所以在第 3 行中就显示为 2。

17.3.3 变量的作用域链

变量的作用域是基于词法结构来确定，属于静态概念，而不是根据执行顺序来确定。作用域链是 JavaScript 提供的一套解决变量访问的机制。JavaScript 规定每一个作用域都有一个与之相关联的作用域链。

作用域链就是一个对象列表，并根据对象的结构层次被串在一起，提供访问变量的优先顺序。当 JavaScript 访问变量时，会查询当前作用域中是否存在同名变量，当前作用域是一个调用对象，作用域中的变量为调用对象的属性。

如果当前调用对象存在同名属性，则访问该属性值。否则，会沿着作用域链向上继续查询上一级作用域中的调用对象。如果上一级调用对象仍然没有同名属性，那么就继续向上查询，依此类推。

最后，查询到作用域链的顶端（全局对象），如果在全局对象中仍然没有找到同名属性，则返回 undefined 的属性值。

【示例】在下面示例中，通过多层嵌套的函数设计一个多层作用域的上下文环境，在最内层函数中可以分别访问外层函数的私有变量。

```
var a = 1;                    // 全局变量
(function(){
    var b = 2;                // 第 1 层局部变量
    (function(){
        var c = 3;            // 第 2 层局部变量
        (function(){
            var d = 4;        // 第 3 层局部变量
            alert(a+b+c+d);   // 返回 10
        })()                  // 直接调用函数
    })()                      // 直接调用函数
})()                          // 直接调用函数
```

在这个示例中，JavaScript 解释器首先在最内层调用对象中查询属性 a、b、c 和 d，其中只找到了属性 d，并获得它的值（4），然后沿着作用域链，在上一层调用对象中继续查找属性 a、b 和 c，其中找到了属性 c，获得它的值（3），依此类推，直到找到所有需要的变量值为止。

17.3.4 变量回收

JavaScript 包含一个垃圾回收的小程序，这个程序能够周期性的遍历 JavaScript 环境中的所有变量列表，并且给这些变量所引用的值做个标记。如果被引用的值是对象或数组，那么对象的属性或者数组的元素就会被递归做个标记。通过递归遍历所有值的树状图，垃圾回收器就能够找到（并标记）仍旧使用的每个值。那些没有标记的值就是无用的存储单元。

当给所有正在使用的变量做完标记之后，垃圾回收器就会开始进行清除。在这个阶段中，它将遍历环境中所有值的列表，同时释放那些没有标记的值。

【示例】在下面这个示例中，变量 a 存储的是字符串"javascript"，然后再给变量 a 赋其他值，这时候在内存中字符串"javascript"所占据的空间就没有被任何变量引用，此时 JavaScript 垃圾回收器就会把这个字符串视为垃圾，并执行回收，释放它占据的内存空间。

```
var a = "javascript";
a = 123456;
```

同时，如果为变量 a 赋值为 null，则 JavaScript 垃圾回收器就知道这个变量也没有用，于是把这个变量视为垃圾一并进行回收。如果一个变量、属性、元素或对象被赋值为 null，也就意味着它们是无用放入垃圾了，JavaScript 垃圾回收器将择机对其进行回收。

```
a = null;
```

17.3.5 变量污染

定义全局变量有 3 种方式：

- 在任何函数外面直接执行 var 语句。

```
var f = 'value';
```

- 直接添加一个属性到全局对象上。全局对象是所有全局变量的容器。在 Web 浏览器中，全局对象名为 window。

```
window.f = 'value';
```

- 直接使用未经声明的变量，以这种方式定义的全局变量被称为隐式的全局变量。

```
f = 'value';
```

JavaScript 最为糟糕的就是对全局变量的依赖。由于在所有作用域中都可见，使用全局变量会降低程序的可靠性。

【示例 1】应该避免使用全局变量，努力减少使用全局变量的方法：在应用程序员中创建唯一一个全局变量，并定义该变量为当前应用的容器。

```
var My = {};,
My.name = {
    "first-name" : " first ",
    "last-name" : " last "
};
My.work = {
    number : 123,
    one : {
        name : " one ",
        time : "2015-9-14 12:55",
        city : "beijing"
    },
    two : {
        name : "two",
        time : "2015-9-12 12:42",
        city : "shanghai"
    }
};
```

只要把多个全局变量都追加在一个名称空间下，将显著降低与其他应用程序产生冲突的几率，应用程序也会变得更容易阅读，因为 My.work 指向的是顶层结构。当然也可以使用闭包体将信息隐藏，它是另一中有效减少“全局污染”的方法。

【示例 2】在编程语言中，作用域控制着变量与参数的可见性及生命周期。这为程序开发提供了一个重要的帮助，因为它减少了名称冲突，并且提供了自动内存管理。

```
var foo = function() {
    var a = 1, b = 2;
    var bar = function() {
        var b = 3, c = 4;        // a=1, b =3, c=4
        a += b + c;              // a=8, b =3, c=4
    };                           // a=1, b =2, c=undefined
    bar();                       // a=21, b =2, c=undefined
};
```

大多数使用 C 语法的语言都拥有块级作用域。对于一个代码块，即包括在一对大括号中的语句，其中定义的所有变量在代码块的外部是不可见的。定义在代码块中的变量在代码块执行结束后会被释放掉。但是，对于 JavaScript 语言来说，虽然该语言支持代码块的语法形式，但是它并不支持块级作用域。

JavaScript 支持函数作用域，定义在函数中的参数和变量在函数外部是不可见的，且在一个函数中的任何位置定义的变量在该函数中的任何地方都可见。

其他主流编程语言都推荐尽可能迟地声明变量，但是在 JavaScript 中就不能够这样使用，因为它缺少块级作用城，最好的做法是在函数体的顶部声明函数中可能用到的所有变量。