引用计数

另一种不太常见的垃圾收集策略叫做**引用计数**（reference counting）。引用计数的含义是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型值赋给该变量时，则这个值的引用次数就是 1。如果同一个值又被赋给另一个变量，则该值的引用次数加 1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数减 1。当这个值的引用次数变成 0 时，则说明没有办法再访问这个值了，因而就可以将其占用的内存空间回收回来。这样，当垃圾收集器下次再运行时，它就会释放那些引用次数为零的值所占用的内存。

Netscape Navigator 3.0 是最早使用引用计数策略的浏览器，但很快它就遇到了一个严重的问题：循环引用。**循环引用**指的是对象 A 中包含一个指向对象 B 的指针，而对象 B 中也包含一个指向对象 A 的引用。请看下面这个例子：

```
function problem(){
    var objectA = new Object();
    var objectB = new Object();

    objectA.someOtherObject = objectB;
    objectB.anotherObject = objectA;
}
```

在这个例子中，`objectA` 和 `objectB` 通过各自的属性相互引用；也就是说，这两个对象的引用次数都是 2。在采用标记清除策略的实现中，由于函数执行之后，这两个对象都离开了作用域，因此这种相互引用不是个问题。但在采用引用计数策略的实现中，当函数执行完毕后，`objectA` 和 `objectB` 还将继续存在，因为它们的引用次数永远不会是 0。假如这个函数被重复多次调用，就会导致大量内存得不到回收。为此，Netscape 在 Navigator 4.0 中放弃了引用计数方式，转而采用标记清除来实现其垃圾收集机制。可是，引用计数导致的麻烦并未就此终结。

我们知道，IE 中有一部分对象并不是原生 JavaScript 对象。例如，其 BOM 和 DOM 中的对象就是使用 C++以 COM（Component Object Model，组件对象模型）对象的形式实现的，而 COM 对象的垃圾收集机制采用的就是引用计数策略。因此，即使 IE 的 JavaScript 引擎是使用标记清除策略来实现的，但 JavaScript 访问的 COM 对象依然是基于引用计数策略的。换句话说，只要在 IE 中涉及 COM 对象，就会存在循环引用的问题。下面这个简单的例子，展示了使用 COM 对象导致的循环引用问题：

```
var element = document.getElementById("some_element");
var myObject = new Object();
myObject.element = element;
element.someObject = myObject;
```

这个例子在一个 DOM 元素（element）与一个原生 JavaScript 对象（myObject）之间创建了循环引用。其中，变量 myObject 有一个名为 element 的属性指向 element 对象；而变量 element 也有一个属性名叫 someObject 回指 myObject。由于存在这个循环引用，即使将例子中的 DOM 从页面中移除，它也永远不会被回收。

为了避免类似这样的循环引用问题，最好是在不使用它们的时候手工断开原生 JavaScript 对象与 DOM 元素之间的连接。例如，可以使用下面的代码消除前面例子创建的循环引用：

```
myObject.element = null;
element.someObject = null;
```

将变量设置为 null 意味着切断变量与它此前引用的值之间的连接。当垃圾收集器下次运行时，就会删除这些值并回收它们占用的内存。

为了解决上述问题，IE9 把 BOM 和 DOM 对象都转换成了真正的 JavaScript 对象。这样，就避免了两种垃圾收集算法并存导致的问题，也消除了常见的内存泄漏现象。

myObject.element = null;

element.someObject = null;