[**javascript垃圾收集机制与内存泄漏详解**](http://www.cnblogs.com/adforce/archive/2012/12/04/2801200.html)

javascript具有自动垃圾收集机制，也就是说，执行环境会负责管理代码执行过程中的使用的内存。而在C和C++之类的语言中，开发人员的一项基本任务就是手动跟踪内存的使用情况，这是造成许多问题的一个根源。在编写javascript程序时候，开发人员不用再关心内存使用的问题，所需内存的分配 以及无用的回收完全实现了自动管理。这种垃圾收集机制的原理其实很简单：找出那些不再继续使用的变量，然后释放其中占用的内存。为此，垃圾收集器会按照固定的时间间隔（或代码执行中预设的收集时间），周期性的执行这一操作。  
  
下面我们来分析一下函数中局部变量正常的生命周期。局部变量只在函数执行的过程中存在。而在这个过程中，会为局部变量在栈（或堆）内存上分配相应的空间， 以便存储他们的值。然后在函数中是使用这些变量，直至函数执行结束。此时，局部变量就没有存在的必要了，因此可以释放他们的内存以供将来使用。在这种情况下，很容易判断变量是否还有存在的必要；但并非所有情况下都这么容易就能得出结论。垃圾收集器必须跟踪哪个变量有用哪个变量没用，对于不再有用的变量打上标记，以备将来回收其占用的内存。用于标识无用变量的策略可能会因现实而异，但具体到浏览器中的实现，通常有两个策略。  
  
**标记清除**  
  
javascript中最常用的垃圾收集方式是标记清除(mark-and-sweep)。当变量进入环境（例如，在函数中声明一个变量）时，就将这个变量标记为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占的内存，因为只要执行流进入相应的环境，就可能用到它们。而当变量离开环境时，这将其 标记为“离开环境”。  
  
可以使用任何方式来标记变量。比如，可以通过翻转某个特殊的位来记录一个变量何时进入环境，或者使用一个“进入环境的”变量列表及一个“离开环境的”变量列表来跟踪哪个变量发生了变化。说到底，如何标记变量其实并不重要，关键采取什么策略。  
  
垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记（当然，可以使用任何标记方式）。然后，它会去掉环境中变量以及被环境中的变量引用的变量标 记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后，垃圾收集器完成内存清除工作，销毁那些带 标记的值并回收它们所占用的内存空间。  
  
到2008年为止，IE、Firefox、Opera、Chrome和Safari的javascript实现使用的都是标记清除式的垃圾收集策略（或类似的策略），只不过垃圾收集的时间间隔互有不同。  
  
**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。当声明一个变量并将引用类型的值赋给该变量时，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，则说明没有办法访问这个值了，因此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的引用，而对象B中也包含一个指向对象A的引用。请看下面例子：

function () {  
    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。在采用引标记清除略的实现中，由于函数执行之后，这两个对象都离开了作用域。因此这两种相互引用不是个问题。但在采用引用计数策略的实现中，但函数执行完毕后，objectA和objectB还将继续存在，因此他们的引用次数永远不会是0。假如这个函数被重复调用，就会导致大量的内存得不到回收。因此，Netscape在Navigator 4.0中放弃了引用计数器方式，转而采用标记清除来实现对其垃圾回收机制。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，其中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IE中设计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.somObject = myObject;

这里例子在一个DOM元素(element)与一个原生的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，最好是不使用他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.somObject = 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。但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。  
  
(导致循环引用的情况不止这些，其他一些情况将在本书中陆续介绍。)  
  
**性能问题**  
  
垃圾收集器都是周期性运行的，而且如果为变量分配的内存数量很客观，那么回收工作量也是相当大的。在这种情况下，确定垃圾收集的时间间隔是一个非常重要的问题。说到垃圾收集器多长时间运行一次，不禁让人联想到IE因此声名狼藉的性能问题。IE的垃圾收集器是根据内存分配量运行的，具体一点说就是256个变量、4096个对象（或数组）字面量和数组元素（slot）或者64KB的字符串。达到上述任何一个临界值，垃圾收集器就会运行。这种实现的问题在于，如果一个脚本中包含那么多变量，那么该脚本很可能会在其生命中起一支保持那么多的变量。而这样一来，垃圾收集器就可能不得不频繁的运行。结果，由此引发的严重性能问题初始IE7重写了其垃圾收集例程。  
  
随着IE7的发布，其javascript引擎的垃圾收集例程改变了工作方式：触发垃圾收集的变量分配、字面量和（或）数组元素的临界值被调整为动态修正。IE7中的各项临界值在初始化时与IE6相等。如果例程回收的内存分配量低于15%，则变量 、字面量和（或）数组元素的临界值就会加倍。如果例程回收了85%的内存分配量，则将各种临界重置会默认值。这一看似简单的调整，极大地提升了IE在运行包含大量javascript的页面时的性能。  
  
事实上，在有的浏览器中可以触发垃圾收集过程，当我们不建议读者这样做。在IE中，调用window.CollectGarbage()方法会立即指向垃圾收集，在Opera7及更高版本中，调用widnow.opera.collect()也会启动垃圾收集例程。  
  
**管理内存**  
  
使具备垃圾收集机制的语言编写程序，开发人员一般不必操心内存管理的问题。但是，javascript在进行内存管理及垃圾收集时面临的问题还是有点与众不同。其中最重要的一个问题，就是分配给web浏览器的可使用内存数量通常要比分配给桌面应用程序的少。这样做的目的出要是处于安全方面的考虑，目的是防止运行javascript的网页耗尽全部系统内存而导致系统崩溃。内存限制问题不仅会影响给变量分配内存，同时还会影响调用栈以及在一个线程中能够同时执行语句数量。  
  
因此，确保占用最少内存可以让页面获得更好的性能，最好通过将其值设置为null来释放其引用——这个做法叫做解除引用(dereferencing)。这一做法是用于大多数全局变量和全局对象的属性。局部变量会在他们执行环境时自动被解除引用，如下面这个例子所示：

function createPerson (name) {  
    var localPerson = new Object();  
    localPerson.name = name;  
    return localPerson;  
};  
var gllbalPerson = createPerson("Nicholas");  
  
// 手工解除globalPerson的引用  
globalPerson = null;

在这个例子中，变量globalPerson取得了createPerson()函数返回的值。在createPerson()函数内部，我们创建了一个对象并将其赋给了局部变量localPerson，然后又为该对象添加了一个名为name的属性。最后，当调用这个函数时，localPerson以函数的形式返回并赋给全局变量globalPerson。由于localPerson在createPerson()函数执行完毕后就离开了其执行环境，因此无需我们显示的去为他解除引用。但是对于全局变量globalPerson而言，则需要我们在不使用它的时候手工为它解除引用，这也正是上面例子中最后一 行代码的目的。  
  
不过，解除一个值的引用并不意味着自动回收该值所占用的内存。解除引用的真正作用是让值脱离执行环境，一边垃圾收集器下次运行时将其回收。  
  
**内存泄漏**  
  
由于IE对JScript对象和COM对象使用不同的垃圾收集例程，因此闭包在IE中会导致一些特殊的问题。具体来说，如果闭包的作用域链中保存着一个HTML元素，那么就意味着该元素无法被销毁。来看下面的例子：

function assignHandler () {  
    var element = document.getElementById("someElement");  
    element.onclick = function () {  
            alert(element.id);  
    };  
};

以上代码创建了一个作为element元素时间处理程序的闭包，而这个闭包则有创建了一个循环引用。由于匿名函数保存了一个对assignHandler()的活动对象的引用，因此就会导致无法减少element的引用数。只要匿名函数存在，element的引用数至少也是1， 因此它所占用的内存就永远不会被回收。不过，这个问题可以通过稍微改写一下代码来解决，如下所示：

function assignHandler () {  
    var element = document.getElementById("someElement");  
    var id = element.id;  
      
    element.onclick = function () {  
            alert(id);  
    };  
      
    element = null;  
};

在上面代码中，通过把element.id的一个副本保存在一个变量中，并且在闭包中引用该变量消除了循环引用。但仅仅做到这一步，还是不能解决内存泄漏 的问题。必须要记住：闭包会引用包含函数活动的整个活动对象，而其中包含着element。即使闭包不直接引用element，包含函数的活动对象中也仍 然会保存一个引用。因此，有必要把element变量设置为null。这样就能够解除对DOM对象的引用，顺利地减少其引用数，确保正常回收其占用的内存。