在过去一些的时候，Web开发人员并没有太多的去关注内存泄露问题。那时的页面间联系大都比较简单，并主要使用不同的连接地址在同一个站点中导航，这样的设计方式是非常有利于浏览器释放资源的。即使Web页面运行中真的出现了资源泄漏，那它的影响也是非常有限而且常常是不会被人在意的。

51CTO推荐阅读：[早该知道的7个JavaScript技巧](http://developer.51cto.com/art/201005/201607.htm)

今天人们对Web应用有了高更的要求。一个页面很可能数小时不会发生URL跳转，并同时通过Web服务动态的更新页面内容。复杂的事件关联设计、基于对象的JavaScript和DHTML技术的广泛采用，使得代码的能力达到了其承受的极限。在这样的情况和改变下，弄清楚内存泄露方式变得非常的急迫，特别是过去这些问题都被传统的页面导航方法给屏蔽了。

还算好的事情是，当你明确了希望寻找什么时，内存泄露方式是比较容易被确定的。大多数你能遇到的泄露问题我们都已经知道，你只需要少量额外的工作就会给你 带来好处。虽然在一些页面中少量的小泄漏问题仍会发生，但是主要的问题还是很容易解决的。

**泄露方式**

在接下来的内容中，我们会讨论内存泄露方式，并为每种方式给出示例。其中一个重要的示例是JavaScript中的Closure技术，另一个示例是在事件执行 中使用Closures。当你熟悉本示例后，你就能找出并修改你已有的大多数内存泄漏问题，但是其它Closure相关的问题可能又会被忽视。

现在让我们来看看这些个方式都有什么：

1、循环引用(Circular References) — IE浏览器的COM组件产生的对象实例和网页脚本引擎产生的对象实例相互引用，就会造成内存泄漏。这也是Web页面中我们遇到的最常见和主要的泄漏方式；

2、内部函数引用(Closures) — Closures可以看成是目前引起大量问题的循环应用的一种特殊形式。由于依赖指定的关键字和语法结构，Closures调用是比较容易被我们发现的；

3、页面交叉泄漏(Cross-Page Leaks) — 页面交叉泄漏其实是一种较小的泄漏，它通常在你浏览过程中，由于内部对象薄计引起。下面我们会讨论DOM插入顺序的问题，在那个示例中你会发现只需要改动 少量的代码，我们就可以避免对象薄计对对象构建带来的影响；

4、貌似泄漏(Pseudo-Leaks) — 这个不是真正的意义上的泄漏，不过如果你不了解它，你可能会在你的可用内存资源变得越来越少的时候极度郁闷。为了演示这个问题，我们将通过重写 Script元素中的内容来引发大量内存的"泄漏"。

**循环引用**

循环引用基本上是所有泄漏的始作俑者。通常情况下，脚本引擎通过垃圾收集器(GC)来处理循环引用，但是某些未知因数可能会妨碍从其环境中释放资源。对于 IE来说，某些DOM对象实例的状态是脚本无法得知的。下面是它们的基本原则：

**Figure 1: 基本的循环引用模型**

本模型中引起的泄漏问题基于COM的引用计数。脚本引擎对象会维持对DOM对象的引用，并在清理和释放DOM对象指针前等待所有引用的移除。在我们的示例 中，我们的脚本引擎对象上有两个引用：脚本引擎作用域和DOM对象的expando属性。

当终止脚本引擎时第一个引用会释放，DOM对象引用由于在等待脚 本擎的释放而并不会被释放。你可能会认为检测并修复假设的这类问题会非常的容易，但事实上这样基本的的示例只是冰山一角。你可能会在30个对象链的末尾发 生循环引用，这样的问题排查起来将会是一场噩梦。如果你仍不清楚这种泄漏方式在HTML代码里到底怎样，你可以通过一个全局脚本变量和一个DOM对象来引发并展现它。

1. **<html>**
2. **<head>**
3. **<script** language="JavaScript"**>**
4. var myGlobalObject;
5. function SetupLeak()
6. {
7. // First set up the script scope to element reference
8. myGlobalObject = document.getElementById("LeakedDiv");
10. // Next set up the element to script scope reference
11. document.getElementById("LeakedDiv").expandoProperty = myGlobalObject;
12. }
14. function BreakLeak()
15. {
16. document.getElementById("LeakedDiv").expandoProperty = null;
17. }
18. **</script>**
19. **</head>**
20. **<body** onload="SetupLeak()" onunload="BreakLeak()"**>**
21. **<div** id="LeakedDiv"**></div>**
22. **</body>**
23. **</html>**

你可以使用直接赋null值得方式来破坏该泄漏情形。在页面文档卸载前赋null值，将会让脚本引擎知道对象间的引用链没有了。现在它将能正常的清理引用 并释放DOM对象。在这个示例中，作为Web开发员的你因该更多的了解了对象间的关系。

作为一个基本的情形，循环引用可能还有更多不同的复杂表现。对基于对象的JavaScript，一个通常用法是通过封装JavaScript对象来扩充DOM对象。在 构建过程中，你常常会把DOM对象的引用放入JavaScript对象中，同时在DOM对象中也存放上对新近创建的JavaScript对象的引用。你的这种应用模式 将非常便于两个对象之间的相互访问。这是一个非常直接的循环引用问题，但是由于使用不用的语法形式可能并不会让你在意。要破环这种使用情景可能变得更加复 杂，当然你同样可以使用简单的示例以便于清楚的讨论。

1. **<html>**
2. **<head>**
3. **<script** language="JavaScript"**>**
5. function Encapsulator(element)
6. {
7. // Set up our element
8. this.elementReference = element;
10. // Make our circular reference
11. element.expandoProperty = this;
12. }
14. function SetupLeak()
15. {
16. // The leak happens all at once
17. new Encapsulator(document.getElementById("LeakedDiv"));
18. }
20. function BreakLeak()
21. {
22. document.getElementById("LeakedDiv").expandoProperty = null;
23. }
24. **</script>**
25. **</head>**
26. **<body** onload="SetupLeak()" onunload="BreakLeak()"**>**
27. **<div** id="LeakedDiv"**></div>**
28. **</body>**
29. **</html>**

更复杂的办法还有记录所有需要解除引用的对象和属性，然后在Web文档卸载的时候统一清理，但大多数时候你可能会再造成额外的泄漏情形，而并没有解决你的 问题。

**闭包函数(Closures)**

由于闭包函数会使程序员在不知不觉中创建出循环引用，所以它对资源泄漏常常有着不可推卸的责任。而在闭包函数自己被释放前，我们很难判断父函数的参数以及 它的局部变量是否能被释放。实际上闭包函数的使用已经很普通，以致人们频繁的遇到这类问题时我们却束手无策。在详细了解了闭包背后的问题和一些特殊的闭包 泄漏示例后，我们将结合循环引用的图示找到闭包的所在，并找出这些不受欢迎的引用来至何处。

**Figure 2. 闭包函数引起的循环引用**

普通的循环引用，是两个不可探知的对象相互引用造成的，但是闭包却不同。代替直接造成引用，闭包函数则取而代之从其父函数作用域中引入信息。通常，函数的 局部变量和参数只能在该被调函数自身的生命周期里使用。当存在闭包函数后，这些变量和参数的引用会和闭包函数一起存在，但由于闭包函数可以超越其父函数的 生命周期而存在，所以父函数中的局部变量和参数也仍然能被访问。

在下面的示例中，参数1将在函数调用终止时正常被释放。当我们加入了一个闭包函数后，一个 额外的引用产生，并且这个引用在闭包函数释放前都不会被释放。如果你碰巧将闭包函数放入了事件之中，那么你不得不手动从那个事件中将其移出。如果你把闭包 函数作为了一个expando属性，那么你也需要通过置null将其清除。

同时闭包会在每次调用中创建，也就是说当你调用包含闭包的函数两次，你将得到两个独立的闭包，而且每个闭包都分别拥有对参数的引用。由于这些显而易见的因 素，闭包确实非常用以带来泄漏。下面的示例将展示使用闭包的主要泄漏因素：

1. **<html>**
2. **<head>**
3. **<script** language="JavaScript"**>**
5. function AttachEvents(element)
6. {
7. // This structure causes element to ref ClickEventHandler
8. element.attachEvent("onclick", ClickEventHandler);
10. function ClickEventHandler()
11. {
12. // This closure refs element
13. }
14. }
15. function SetupLeak()
16. {
17. // The leak happens all at once
18. AttachEvents(document.getElementById("LeakedDiv"));
19. }
21. function BreakLeak()
22. {
23. }
24. **</script>**
25. **</head>**
26. **<body** onload="SetupLeak()" onunload="BreakLeak()"**>**
27. **<div** id="LeakedDiv"**></div>**
28. **</body>**
29. **</html>**

如果你对怎么避免这类泄漏感到疑惑，我将告诉你处理它并不像处理普通循环引用那么简单。"闭包"被看作函数作用域中的一个临时对象。一旦函数执行退出，你 将失去对闭包本身的引用，那么你将怎样去调用detachEvent方法来清除引用呢？在Scott Isaacs的MSN Spaces上有一种解决这个问题的有趣方法。

这个方法使用一个额外的引用(原文叫second closure，可是这个示例里致始致终只有一个closure)协助window对象执行onUnload事件，由于这个额外的引用和闭包的引用存在于 同一个对象域中，于是我们可以借助它来释放事件引用，从而完成引用移除。为了简单起见我们将闭包的引用暂存在一个expando属性中，下面的示例将向你 演示释放事件引用和清除expando属性。

1. **<html>**
2. **<head>**
3. **<script** language="JavaScript"**>**
4. function AttachEvents(element)
5. {
6. // In order to remove this we need to put
7. // it somewhere. Creates another ref
8. element.expandoClick = ClickEventHandler;
10. // This structure causes element to ref ClickEventHandler
11. element.attachEvent("onclick", element.expandoClick);
13. function ClickEventHandler()
14. {
15. // This closure refs element
16. }
17. }
19. function SetupLeak()
20. {
21. // The leak happens all at once
22. AttachEvents(document.getElementById("LeakedDiv"));
23. }
25. function BreakLeak()
26. {
27. document.getElementById("LeakedDiv").detachEvent("onclick",
28. document.getElementById("LeakedDiv").expandoClick);
29. document.getElementById("LeakedDiv").expandoClick = null;
30. }
31. **</script>**
32. **</head>**
33. **<body** onload="SetupLeak()" onunload="BreakLeak()"**>**
34. **<div** id="LeakedDiv"**></div>**
35. **</body>**
36. **</html>**

在这篇文章中，实际上建议我们除非迫不得已尽量不要创建使用闭包。文章中的示例，给我们演示了非闭包的事件引用方式，即把闭包函数放到页面的全局作用 域中。当闭包函数成为普通函数后，它将不再继承其父函数的参数和局部变量，所以我们也就不用担心基于闭包的循环引用了。在非必要的时候不使用闭包这样的编 程方式可以尽量使我们的代码避免这样的问题。

最后，脚本引擎开发组的Eric Lippert，给我们带来了一篇关于闭包使用通俗易懂的好文章。他的最终建议也是希望在真正必要的时候才使用闭包函数。虽然他的文章没有提及闭包会使用 的真正场景，但是这儿已有的大量示例非常有助于大家起步。

**页面交叉泄漏(Cross-Page Leaks)**

这种基于插入顺序而常常引起的泄漏问题，主要是由于对象创建过程中的临时对象未能被及时清理和释放造成的。它一般在动态创建页面元素，并将其添加到页面 DOM中时发生。一个最简单的示例场景是我们动态创建两个对象，并创建一个子元素和父元素间的临时域（译者注：这里的域(Scope)应该是指管理元素之 间层次结构关系的对象）。

然后，当你将这两个父子结构元素构成的的树添加到页面DOM树中时，这两个元素将会继承页面DOM中的层次管理域对象，并泄漏之 前创建的那个临时域对象。下面的图示示例了两种动态创建并添加元素到页面DOM中的方法。在第一种方法中，我们将每个子元素添加到它的直接父元素中，最后 再将创建好的整棵子树添加到页面DOM中。当一些相关条件合适时，这种方法将会由于临时对象问题引起泄漏。在第二种方法中，我们自顶向下创建动态元素，并 使它们被创建后立即加入到页面DOM结构中去。由于每个被加入的元素继承了页面DOM中的结构域对象，我们不需要创建任何的临时域。这是避免潜在内存泄漏 发生的好方法。

**Figure 3. DOM插入顺序泄漏模型**

接下来，我们将给出一个躲避了大多数泄漏检测算法的泄漏示例。因为我们实际上没有泄漏任何可见的元素，并且由于被泄漏的对象太小从而你可能根本不会注意这 个问题。为了使我们的示例产生泄漏，在动态创建的元素结构中将不得不内联的包含一个脚本函数指针。在我们设置好这些元素间的相互隶属关系后这将会使我们泄 漏内部临时脚本对象。由于这个泄漏很小，我们不得不将示例执行成千上万次。

事实上，一个对象的泄漏只有很少的字节。在运行示例并将浏览器导航到一个空白页 面，你将会看到两个版本代码在内存使用上的区别。当我们使用第一种方法，将子元素加入其父元素再将构成的子树加入页面DOM，我们的内存使用量会有微小的 上升。这就是一个交叉导航泄漏，只有当我们重新启动IE进程这些泄漏的内存才会被释放。如果你使用第二种方法将父元素加入页面DOM再将子元素加入其父元 素中，同样运行若干次后，你的内存使用量将不会再上升，这时你会发现你已经修复了交叉导航泄漏的问题。

1. **<html>**
2. **<head>**
3. **<script** language="JavaScript"**>**
5. function LeakMemory()
6. {
7. var hostElement = document.getElementById("hostElement");
9. // Do it a lot, look at Task Manager for memory response
11. for(i = 0; i **<** **5000**; i++)
12. {
13. var parentDiv =
14. document.createElement("**<div** onClick='foo()'**>**");
15. var childDiv =
16. document.createElement("**<div** onClick='foo()'**>**");
18. // This will leak a temporary object
19. parentDiv.appendChild(childDiv);
20. hostElement.appendChild(parentDiv);
21. hostElement.removeChild(parentDiv);
22. parentDiv.removeChild(childDiv);
23. parentDiv = null;
24. childDiv = null;
25. }
26. hostElement = null;
27. }
29. function CleanMemory()
30. {
31. var hostElement = document.getElementById("hostElement");
33. // Do it a lot, look at Task Manager for memory response
35. for(i = 0; i **<** **5000**; i++)
36. {
37. var parentDiv =
38. document.createElement("**<div** onClick='foo()'**>**");
39. var childDiv =
40. document.createElement("**<div** onClick='foo()'**>**");
42. // Changing the order is important, this won't leak
43. hostElement.appendChild(parentDiv);
44. parentDiv.appendChild(childDiv);
45. hostElement.removeChild(parentDiv);
46. parentDiv.removeChild(childDiv);
47. parentDiv = null;
48. childDiv = null;
49. }
50. hostElement = null;
51. }
52. **</script>**
53. **</head>**
55. **<body>**
56. **<button** onclick="LeakMemory()"**>**Memory Leaking Insert**</button>**
57. **<button** onclick="CleanMemory()"**>**Clean Insert**</button>**
58. **<div** id="hostElement"**></div>**
59. **</body>**
60. **</html>**

这类泄漏应该被澄清，因为这个解决方法有悖于我们在IE中的一些有益经验。创建带有脚本对象的DOM元素，以及它们已进行的相互关联是了解这个泄漏的关键 点。这实际上这对于泄漏来说是至关重要的，因为如果我们创建的DOM元素不包含任何的脚本对象，同时使用相同的方式将它们进行关联，我们是不会有任何泄漏 问题的。示例中给出的第二种技巧对于关联大的子树结构可能更有效（由于在那个示例中我们一共只有两个元素，所以建立一个和页面DOM不相关的树结构并不会 有什么效率问题）。

第二个技巧是在创建元素的开始不关联任何的脚本对象，所以你可以安全的创建子树。当你把你的子树关联到页面DOM上后，再继续处理你需 要的脚本事件。牢记并遵守关于循环引用和闭包函数的使用规则，你不会再在挂接事件时在你的代码中遇到不同的泄漏。

我真的要指出这个问题，因为我们可以看出不是所有的内存泄漏都是可以很容易发现的。它们可能都是些微不足道的问题，但往往需要成千上万次的执行一个更小的 泄漏场景才能使问题显现出来，就像DOM元素插入顺序引起的问题那样。如果你觉得使用所谓的"最佳"经验来编程，那么你就可以高枕无忧，但是这个示例让我 们看到，即使是"最佳"经验似乎也可能带来泄漏。我们这里的解决方案希望能提高这些已有的好经验，或者介绍一些新经验使我们避免泄漏发生的可能。

**貌似泄漏(Pseudo-Leaks)**

在大多数时候，一些APIs的实际的行为和它们预期的行为可能会导致你错误的判断内存泄漏。貌似泄漏大多数时候总是出现在同一个页面的动态脚本操作中，而 在从一个页面跳转到空白页面的时候发生是非常少见的。那你怎么能象排除页面间泄漏那样来排除这个问题，并且在新任务运行中的内存使用量是否是你所期望的。 我们将使用脚本文本的重写来作为一个貌似泄漏的示例。

象DOM插入顺序问题那样，这个问题也需要依赖创建临时对象来产生"泄漏"。对一个脚本元素对象内部的脚本文本一而再再而三的反复重写，慢慢地你将开始泄 漏各种已关联到被覆盖内容中的脚本引擎对象。特别地，和脚本调试有关的对象被作为完全的代码对象形式保留了下来。

1. **<html>**
2. **<head>**
3. **<script** language="JavaScript"**>**
4. function LeakMemory()
5. {
6. // Do it a lot, look at Task Manager for memory response
7. for(i = 0; i **<** **5000**; i++)
8. {
9. hostElement.text = "function foo() { }";
10. }
11. }
12. **</script>**
13. **</head>**
14. **<body>**
15. **<button** onclick="LeakMemory()"**>**Memory Leaking Insert**</button>**
16. **<script** id="hostElement"**>**function foo() { }**</script>**
17. **</body>**
18. **</html>**

如果你运行上面的示例代码并使用任务管理器查看，当从"泄漏"页面跳转到空白页面时，你并不会注意到任何脚本泄漏。因为这种脚本泄漏完全发生在页面内部， 而且当你离开该页面时被使用的内存就会回收。对于我们原本所期望的行为来说这样的情况是糟糕的。

你希望当重写了脚本内容后，原来的脚本对象就应该彻底的从 页面中消失。但事实上，由于被覆盖的脚本对象可能已用作事件处理函数，并且还可能有一些未被清除的引用计数。正如你所看到的，这就是貌似泄漏。在表面上内存消耗量可能看起来非常的糟糕，但是这个原因是完全可以接受的。

**总结**

每一位Web开发员可能都整理有一份自己的代码示例列表，当他们在代码中看到如列表中的代码时，他们会意识到泄漏的存在并会使用一些开发技巧来避免这些问 题。这样的方法虽然简单便捷，但这也是今天Web页面内存泄漏普遍存在的原因。考虑我们所讨论的泄漏情景而不是关注独立的代码示例，你将会使用更加有效的策略来解决泄漏问题。这样的观念将使你在设计阶段就把问题估计到，并且确保你有计划来处理潜在的泄漏问题。

使用编写加固代码（译者注：就是异常处理或清理对象等的代码）的习惯并且采取清理所有自己占用内存的方法。虽然对这个问题来说可能太夸张了，你也可能几乎从没有见到编写脚本却需要自己清理自己占用的内 存的情况；使这个问题变得越来越显著的是，脚本变量和expando属性间存在的潜在泄漏可能。