Flex的垃圾回收机理及预防内存泄露

兰天 2008 年 11 月 QQ:595534098 欢迎拍砖。EmaiL: ltian.cn@gmail.com

本文主要通过对互联网上的一些资料进行收集和整理，然后结合自己做的一些试验写出，未必全面准确，欢迎改正或者补充。

内存问题一直是程序员比较关心的问题，每个程序员都希望自己开发的程序足够健壮，在运行过程中不会因内存泄露而导致程序运行变慢或者崩溃。

现在，较新出现的面向对象语言（比如 Java）增强了内存管理机制，能够自动回收不被使用的内存，或者说能够回收垃圾内存，这种内存管理机制通常被称为“garbage collection（垃圾回收）”，简称 GC。

Flex 开发中所使用的 ActionScript 语言，简称 AS，也是支持 GC 的一种语言，经过编译后的 AS 代码运行在 AS 虚拟机（简称 AVM）中，由 AVM 自动完成垃圾内存回收的工作。Flash Player 就是一个 AVM，所以有时候我们将二者混为一谈。

既然 AVM 能够自动完成垃圾回收的功能，那么是不是 Flex 程序员就可以认为所开发的 Flex 应用不存内存泄露问题呢？答案是否定的。在某些情况下，处理不妥当的代码仍然会导致内存泄露。如何才能避免内存泄露？应该说，AS 程序员在清楚了解 Flash Palyer 垃圾回收的基本原理，并且高度重视内存泄露这个问题才能有效避免内存泄露情况的发生。

Flash Player垃圾回收机制

Flash Player 垃圾回收工作是由垃圾回收器（garbage collector）完成的。垃圾回收器是运行在后台的一个进程，它释放那些不再被应用所使用对象所占用的内存。不再被应用所使用的对象是指那些不再会被那些活动着（工作着）的对象所“引用”的对象。在 AS 中，对于非基本类型（Boolean, String, Number, uint, int）的对象，在对象之间传递的都是对象引用，而不是对象本身。删除一个变量只是删除了对象的引用，而不是删除对象本身。一个对象可以被多处引用，通过这些不同的引用所操作的都是同一个对象。

通过以下两段代码可以了解基本类型和非基本类型对象的差异：

基本类型的值传递：

```
private function testPrimitiveTypes():void
{
    var s1:String="abcd"; //创建了一个新字符串s1，值为"abcd"
    var s2:String=s1; //String是基本类型，所以创建了一个新的字符串s2，s2的值拷贝自s1。
    s2+="efg"; //改变s2的值s1不会受影响。
    trace("s1:",s1);  //输出abcd
    trace("s2:",s2); //输出abcdefg
    var n1:Number=100; //创建一个新的number,值为100。
    var n2:Number=n1; //Number是基本类型,所以又创建一个新number n2,n2的值拷贝自n1。
    n2=n2+100;   //改变n2对n1不会有任何影响。
    trace("n1",n1); //输出100
    trace("n2",n2); //输出200
}
```

非基本类型对象的引用传递：

```
private function testNonPrimitiveType():void
       {
          // 创建一个新对象, 然后将其引用给变量 a :
               var a:Object = {foo:"bar"}
             //将上面所创建对象的引用拷贝给变量 b （通过变量 b 建立对对象的引用）:
               var b:Object = a;
            //删除变量a中对对象的引用:
               delete(a);
            // 测试发现对象仍然存在并且被变量b所引用:
              trace(b.foo); // 输出"bar", 所以对象仍然存在
       }
```

对于非基本类型对象，AS3 采用两种方法来判定一个对象是否还有活动的引用，从而决定是否可以将其垃圾回收。一种方法是引用计数法，一种方法是标记清除法。

Reference Counting

引用计数法是判定对象是否有活动引用的最简单的一种方法，并且从 AS1 就开始在 Flash 中使用。当创建一个对对象的引用后，对象的引用计数就加一，当删除一个引用时，对象的引用技术就减一。如果对象的引用计数为 0 ，那么它被标记为可被 GC(垃圾回收器)

删除。例如：

```
var a:Object = {foo:"bar"}
// 现在对象的引用计数为 1 (a)
var b:Object = a;
// 现在对象的引用计数为 2(a 和 b )
delete(a);
// 对象的引用计数又回到了 1 (b)
delete(b);
// 对象的引用计数变成 0,现在，这个对象可以被 GC 释放内存。
```

引用计数法很简单，并且不会增加 CPU 开销,可惜的是，当出现对象之间循环引用时它就不起作用了。所谓循环引用就是指对象之间直接或者间接地彼此引用，尽管应用已经不再使用这些对象，但是它们的引用计数仍然大于 0 ，因此，这些对象就不会被从内存中移除。请看下面的范例：

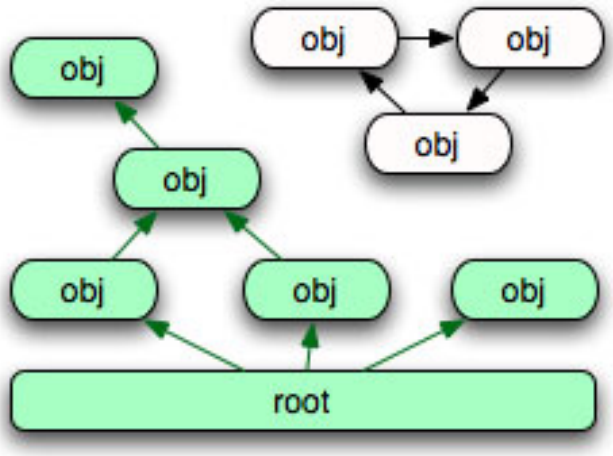
```
创建第一个对象：
var a:Object = {};
// 创建第二个对象来引用第一个对象：
var b:Object = {foo:a};
//使第一个对象也引用第二个对象：
a.foo = b;
// 删除两个活动引用：
delete(a);
delete(b);
```

上面的例子中两个活动引用都已被删除，因此在应用程序中再也无法访问这两个对象。但是它们的引用计数都是 1 ，因为它们彼此相互引用。这种对象间的相互引用可能会更加复杂（a 引用 b,b 引用 c,c 又引用 a,诸如此类），并且难以在代码中通过删除引用来使得引用技术为变为 0 。Flash player 6 和 7 中就因为 XML 对象中的循环引用而痛苦，每个 XML 节点既引用了该节点的子节点，又引用了该节点父节点。因此，这些 XML 对象永远不会释放内存。好在 player 8 增加了一种新的 GC 技术，叫做标记清除。

标记清除（Mark Sweeping）

AS3 使用的第二种查找不活动对象的 GC 策略就是标记清除。 Player 从应用的根节点开始（在 AS3 中通常被称为”根(root)”）,遍历所有其上的引用，标记每个它所发现的

对象。然后迭代遍历每个被标记的对象，标记它们的子对象。这个过程第归进行，直到 Player 遍历了应用的整个对象树并标记了它所发现的每个东西。在这个过程技术的时候，可以安全地认为，内存中那些没有被打标记的对象没有任何活动引用，因此可以被安全地释放内存。可以通过下图可以很直观地了解这种机制（绿色的引用在标记清除过程中被遍历，绿色对象被打上了标记，白色对象将被释放内存）



标记清除机制非常准确，但是这种方法需要遍历整个对象结构，因此会增大 CPU 占用率。因此，Flash Player9 为了减少这种开销只是在需要的时候偶尔执行标记清除活动。

注意：上面所说的引用指的是“强引用（strong reference）”，flash player 在标记清除过程中会忽略“弱引用（weakness reference ）”，也就是说，弱引用在标记清除过程中不被当做引用，不会阻止垃圾回收。

垃圾回收的时机

Flash Player 在运行时请求内存的速度受限于浏览器。因此，Flash Player 采用小量请求大块内存，而不是大量请求小块内存的内存请求策略。同样，Flash Player 在运行时释放内存速度也相对较慢，所以 Flash Player 会减少释放内存的次数，只有在必要的时候才释放内存。也就是说，Flash Player 的垃圾回收只有在必要的时候才会执行。

当前，Flash Player 的垃圾回收发生在 Flash Player 需要另外请求内存之前。这样，Flash Player 可以重新利用垃圾对象所占用的内存资源，并且可以重新评估需要另外请求的内存数量，也会节省时间。

在程序的实际运行中验证了以上的说法，并不是每次应用申请内存时都会导致垃圾回收的执行，只有当 Flash 占用的内存紧张到一定程度时才会执行真正的垃圾回收，如果应用中内存开销增长是匀速的，那么计算机物理内存越大，则垃圾回收触发周期越长。在我的测试环境中，计算机有 2G 的物理内存，直到打开 FLSH 应用的浏览器占用 700M 物理内存之后才会导致 Flash Player 回收垃圾内存。

来自 Adobe 公司的 Alex Harui 总结了两点：

1. 何时真正执行垃圾回收**不可预知**。
2. 垃圾回收总是在请求内存的时候触发，而不是在对象删除时发生。

最后，有关 FP9 中垃圾回收一件非常重要的事情就是：垃圾回收不是立即进行的，而是延迟的。当没有任何对象引用某个对象后，那个对象也不会立即被从内存中清除，相反，它们会在将来某个不确定的时候被移出（从开发者的角度来看）。但这些对象在没有被垃圾回收之前，它们仍然在工作（占用内存和 CPU），尽管这不是我们所希望的。

虽然我们不能指令 Flash Player 立即回收内存，但在程序确定不再引用一个正在工作的对象之前，应终止其工作。比如，停止已经启动的 Timer，停止正在播放的视频或声音，以防止其继续占用 CPU。

强制执行垃圾回收的技巧

很多程序员都想能够在程序中指令计算机进行垃圾回收。目前，Adobe 官方没有公布相关 API 能够强制执行垃圾回收操作。不过互联网上流传一种技巧，见：
http://blogs.eyepartner.com/adrian/flex/flex-tip-6-garbage-collection-in-flex/

该方法利用 player 的 bug 来实现强制回收内存，主要是通过人为抛出某种特别的异常会让 Flash Player 回收内存，代码如下：

```
try
 {
     var lc1: LocalConnection = new LocalConnection();
     var lc2:LocalConnection = new LocalConnection();

      lc1.connect( "gcConnection" );
      lc2.connect( "gcConnection" );
  }
  catch (e:Error)
 {
 }
```

这种方法强制回收内存并不稳定。因此，在开发应用时不要依赖于这种方法来回收内存，只能将其视为辅助方法。

开发中导致内存泄露的常见情况

通过上面的讨论我们可以知道，只要对象被其他活动对象（仍在运行的）所引用，那么这个对象就不会被垃圾回收，从而可能造成内存泄露。

在我们的开发中，如下的一些情形会导致内存泄露：

(一) 被全局对象所引用的对象在它们不再使用时，开发者忘记从全局对象上清除对它们的引用就会产生内存泄露。常见的全局对象有 stage,主 Application，类的静态成员以及采用 singleton 模式创建的实例等。如果使用第三方框架，比如：PureMvc,Cairongorm 等，要注意这些框架的实现原理，尤其要注意框架里面采用 singleton 模式创建的 controler 和 Model。

(二) 无限次触发的 Timer 会导致内存泄漏。无论无限次触发的 Timer 是否为全局对象，无限次触发的 Timer 本身以及注册在 Timer 中的监听器对象都不会被垃圾回收。请看下面的代码：

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml"
    width="400" height="300" creationComplete="this.doIni()"
backgroundColor="#8A85AC" borderColor="#F25488" cornerRadius="0"
borderStyle="solid" borderThickness="3">
    <mx:Script>
        <![CDATA[
            import mx.events.ResizeEvent;
             [Bindable]
             private var timer:Timer=new Timer(1000);

             private var memoryBlocks:Array=new Array();

             private function doIni():void
             {
                var mBlock:Array=this.allocateMemory();
                 memoryBlocks.push(mBlock);
                 this.timer.addEventListener(TimerEvent.TIMER,onTimer);
                 this.timer.start();
             }
             private function onTimer(event:TimerEvent):void
             {
               trace(this.toString();
             }
             private function allocateMemory():Array
             {
                 var memoryBlock:Array=new Array(25600000);
                 for(var i:uint=1;i<=25600000;i++)
                 {
                    memoryBlock[i-1]=i;
                 }
                 trace("allcate 100M memory!");
                 return memoryBlock;
             }
        ]]>
    </mx:Script>
    <mx:Label x="119" y="50" text="内存测试组件" width="139" color="#C14717"
fontSize="19" fontWeight="bold"/>
</mx:Canvas>
```

上面的代码自定义了一个测试内存泄露的 Canvas 组件，这个组件在初始化时开辟了 100M 内存（为了方便查看内存的回收情况），同时创建了一个每隔 1 秒钟，无限次数触发的 Timer，并且启动了这个 Timer。

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute" width="579">
    <mx:Script>
        <![CDATA[
            private var memoryTester:LargeMemoryCanvas=null;
            private function addMemoryTester():void
            {
                this.memoryTester=new LargeMemoryCanvas();
                this.container.addChild(memoryTester);
    //this.stage.addEventListener(Event.RESIZE,this.memoryTester.doOnResize,
false,0,true);
            }
            private function removeMemoryTester():void
            {
              this.container.removeChild(this.memoryTester);
              this.memoryTester=null;
            }
            private function gc():void
            {
              try
              {
                  var lc1:LocalConnection = new LocalConnection();
                  var lc2:LocalConnection = new LocalConnection();

                  lc1.connect( "gcConnection" );
                  lc2.connect( "gcConnection" );
              }
              catch (e:Error)
              {
              }
            }
        ]]>
    </mx:Script>
    <mx:VBox id="container" x="0" y="0" width="100%" height="100%">
        <mx:Button label="强制回收内存" fontSize="12" width="130"
click="this.gc();"/>
        <mx:Button label="创建内存消耗组件" fontSize="12"
click="this.addMemoryTester();"/>
        <mx:Button label="移出内存消耗组件" fontSize="12"
click="this.removeMemoryTester();"/>
    </mx:VBox>
</mx:Application>
```

这是一个测试应用，该应用上有三个按钮，分别是“强制回收内存”，“创建内存

消耗组件”和“移出内存消耗组件”。点击“创建内存消耗组件”按钮就会执行创建一个用于内存泄露测试的 Canvs 对象，并将其作为 container 的子对象显示到界面上，点击“移出内存消耗组件”按钮则会将“创建内存消耗组件”按钮所创建的 Canvs 对象从 container 的子对象列表中删除，并且永远不再使用。应用运行后，我们先“创建内存消耗组件”按钮，然后再点击“移出内存消耗组件”按钮，重复这样的操作，我们发现，由于 Canvs 对象上的无限次数触发的 Timer 对象已经启动，导致 Canvs 对象所占用的内存无法被回收，内存会一直增加，最终会导致浏览器崩溃。如果我们将 Canvs 初始化代码中启动 Timer 的语句注释掉，重复上述的测试操作，内存会在某个时候减少，说明占用内存的 Canvs 对象已经被垃圾回收。

通过上面那个简单的测试程序测试了 Timer 的情况，当然，将其稍加改造也可以用来测试其他情况，我在这里所列举的情况内存泄露的情况都是利用上面那个测试程序得到的结论。

(三)通过隐式方式建立的对象之间的引用关系更容易被程序员所忽略，从而导致内存泄露。 最常见的以隐式方式建立对象之间的引用就是“绑定”和“为对象添加事件监听器”。**通过测试我们发现“绑定”不会造成内存泄露，对象可以放心地绑定全局对象**。而调用 addEventListener()方法“为对象添加事件监听器”则可能产生内存泄露，大多数内存泄露都因此而来：下面代码：

```
a.addEventListener(Event.EVENT_TYPE,b.listenerFunction)
```

使得 a 对象引用了 b 对象，如果 a 对象是一个全局对象（全局对象在应用运行期间始终存在），则 b 对象永远不会被垃圾回收，可能会造成内存泄露。比如下面的代码就有造成内存泄露的可能：

```
this.stage.addEventListener(Event.RESIZE,onResize);
```

上面代码中的 stage 是 UIComponent 的 stage 属性，表示当前 Flex 应用运行的“舞台”。

不过，通过以下三种方式使用 addEventListener 方法不会造成内存泄露：

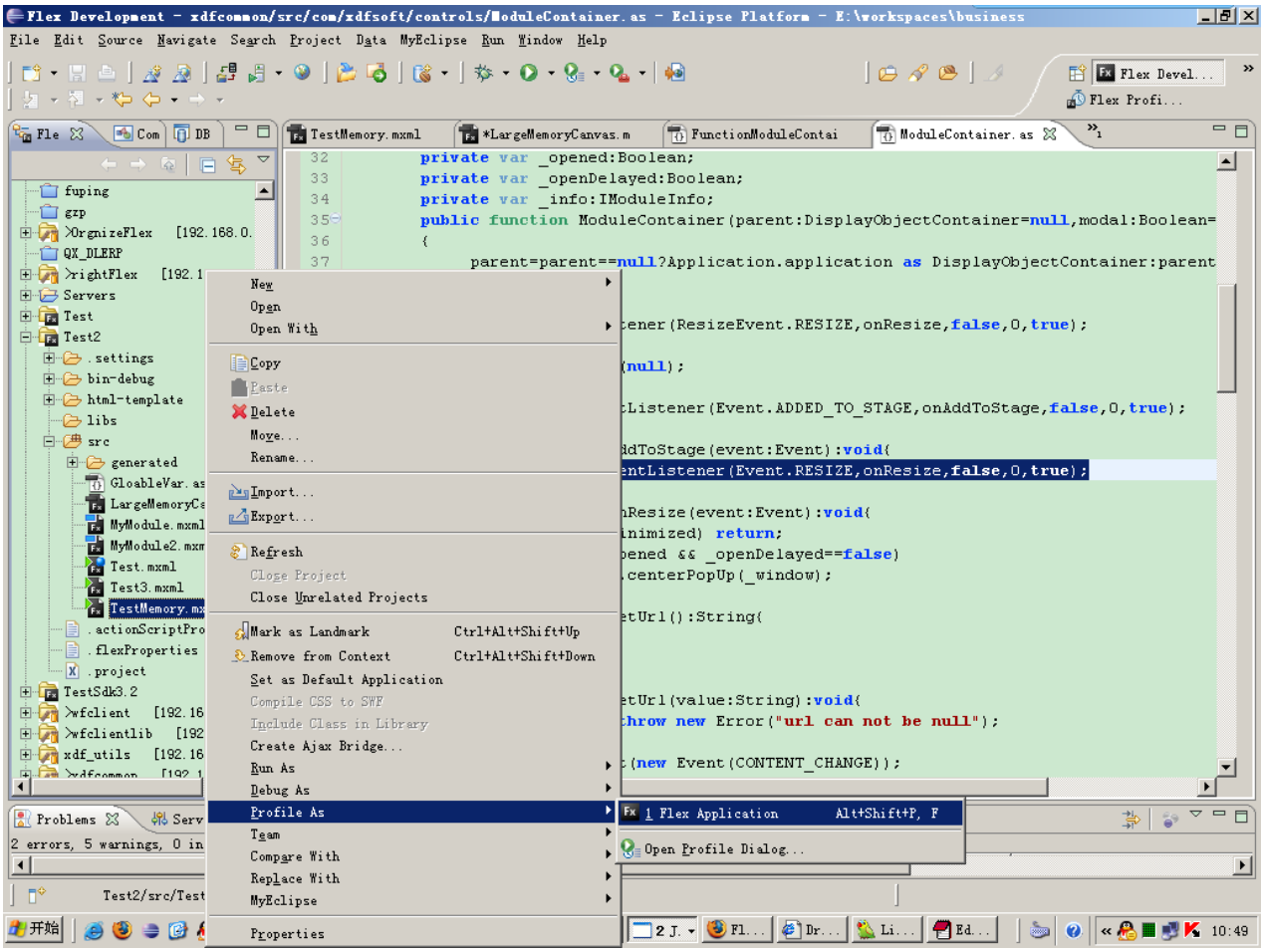
1. **用弱引用方式注册监听器。**就是调用时将 addEventListener 的第五个参数置为 true，例如：someObject.addEventListener(MouseClick.CLICK, otherObject.handlerFunction, false, 0, true);
2. **自引用的方式**。即：为对象添加的监听处理函数是对象本身的方法。例如：this.addEventListener(MouseClick.CLICK, this. handlerFunction);

3 子对象引用。即：为子对象添加的监听处理函数是父上对象的方法。例如：

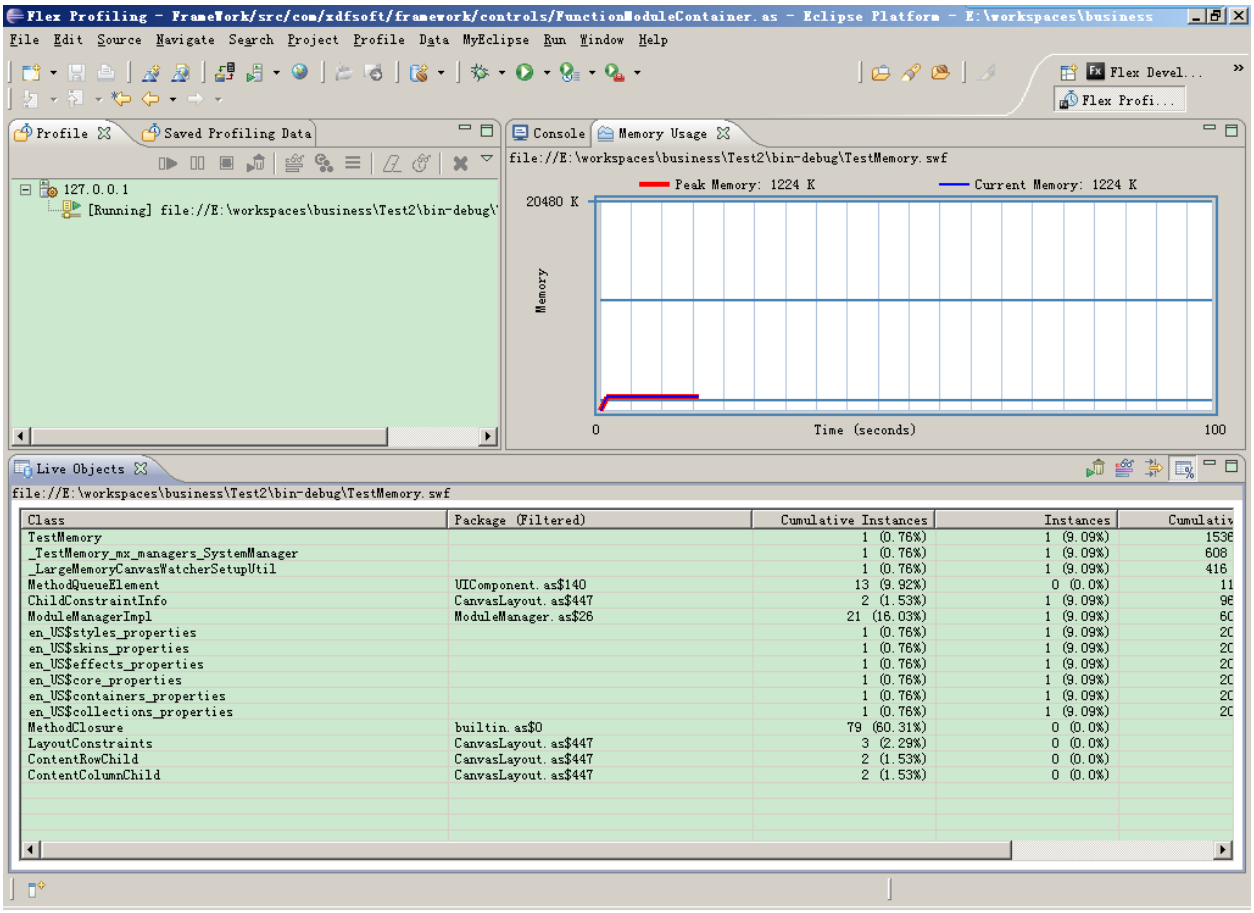
```
private var childObject:UIComponent = new UIComponent;
addChild(childObject);
childObject.addEventListener(MouseEvent.CLICK, this.clickHandler);
```

Flex Builder3 的内存泄露分析工具

Flex Builder3 Pro 带有一个"剖析（Profiler）"工具可以用来帮助我们识别内存泄露。在 Flex Builder3 Pro 的中选择要被"剖析（Profiler）"的应用后点击鼠标右键，在右键菜单中选择"Profile As" 就可以运行"剖析（Profiler）"工具来分析所选择的应用。如下图所示：



"剖析（Profiler）"工具运行后的界面如下图所示：

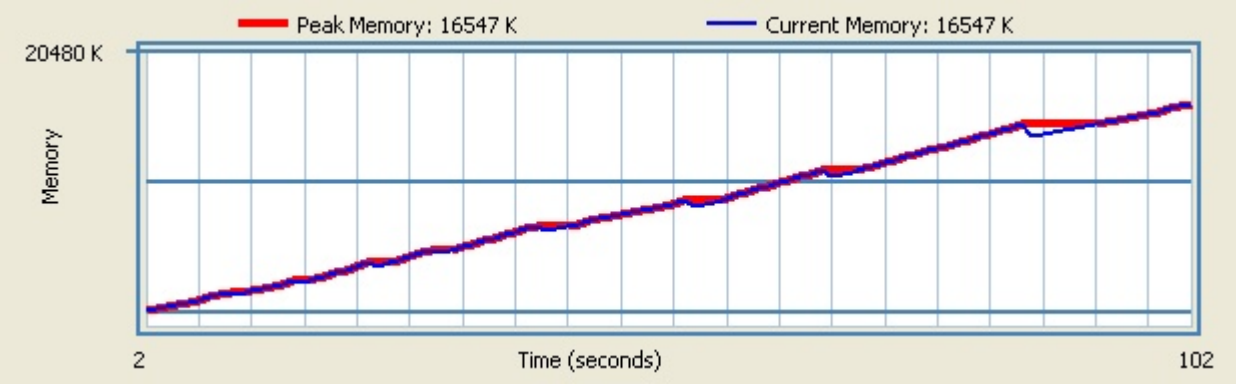


在这个工具中，有一个"Force Garbage Collection（强制垃圾回收）"按钮，当应用被“剖析(Profiler)”时或者说以剖析方式运行时，点击这个按钮然后观察“Live Objects（活动对象列表）”可以帮我们分析内存泄露.如果确信已经完全移除和清除对对象的引用，那么“Live Objects”列表中的"instances "就会减少。

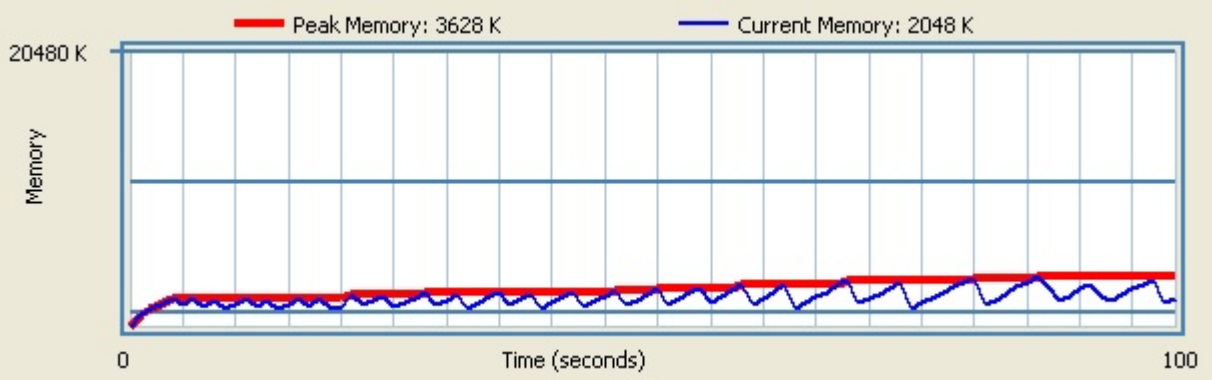
通过"Cumulative Instances（累积的实例）"栏可以看到有多少个对象曾被创建。 而"Instances（当前实例）"栏可以看到当前有存在的对象实例有多少。如果在创建和移除对象之后运行“Force GC（强制垃圾回收）”，"Cumulative Instances（累积的实例）"的数量和"Instances（当前实例）"的数量相同，则可能存在内存泄露。

"Memory Usage（内存使用）"图提供了另一种确定内存泄露的方法，但只适合小应用。红色的线代表最大的内存使用，而兰线则代表当前的内存使用。如果兰线和红线从不分离，则说明有内存泄露。

糟糕的内存使用



良好的内存使用



引用：

http://www.gskinner.com/blog/archives/2006/06/as3_resource_ma.html

http://www.dreamingwell.com/articles/archives/2008/05/understanding_m.php

http://blogs.eyepartner.com/adrian/flex/flex-tip-6-garbage-collection-in-flex/