最近在看 GC（垃圾收集器）相关的东西，发现了几篇好文，虽然比较老了（06 年的），但是很值得看；翻译了一下，这里的 GC（垃圾收集器）主要是说 flash player9 的。偶英文比较糟糕，有非常少量的语句没有翻译，因为偶看不懂囧 大家多担待，欢迎拍板砖^_^

本文大部分是来自：http://www.gskinner.com/blog/archives/2006/06/as3_resource_ma.html；其中 part1 的译文来自：http://bbs.airia.cn/ActionScript/thread-3656-1-1.aspx。其他为本人翻译大概集合了 5 篇文章。
建议先将“前奏”看完，这样会对理解本文有很大的帮助。

前奏：

理解 delete 关键字

在我的一篇博文中 how the garbage collector works in Flash Player 9，塞德里克尼希米记（原名：Cédric Néhémie，名字是 google 的囧）小盆友提了一个非常好的问题：为什么 delete 一个非动态类的属性时会抛异常呢？delete 关键字是否是真的将对象从内存中删除了呢？

提及到这个话题文档并不是很多，即使有也是非常难找，大部分解答都是在根据相关的测试来猜测或推测。但是我的解释肯定是完全正确的（译者：突然想起 疯狂的赛车 里面的一句话：不烧不专业）。如果不是，请告诉我，我不想误导任何人（=。=！）

我的理解是：d elete 关键词删除的不仅仅是变量的值，还包括实际变量的定义。无疑这样会释放其拥有的所有引用，借此将其交给 GC（垃圾收集器）完成释放。也就是说：delete 不会直接从内存删除引用所指向的对象。

ActionScript 2

在 AS2 中，这种行为的效果并不明显，因为播放器不支持非动态类运行。这就意味着：删除一个变量的定义等同于设置变量的值为 undefined。正是因为这点，编译器永远也不会抛出与 delete 相关的异常。他们之间仅仅有一点点不同。例如：在 AS2 中试图去删除 MovieClip 实例 onPress 和 onMouseOver 事件的的处理函数,不论你删除哪一个操作都会返回一个"undefined"，假如你设置其一为 undefined（不用 delete 操作），你会发现直接设置为 undefined 的对象还是会出现对应操作的光标，而 delete 操作则不会出现。我灰常确信这点，当判断是否要现实光标时，播放器会确认 onPress 事件处理函数的定义是否存在，而非根据其值（value）。

另一方面 AS3 则支持非动态类的运行。所有的类都是事先封装好的，除非他们用 explicitly 来声明动态类。在运行时，你是不可以修改一个已经封装好的类或者其实例的成员。正因为如此，并且根据 ECMAScript 说明，**delete 仅仅是删除动态类的动态属性，**不会删除非动态的变量（或方法）的。

在 ActionScript 2.0 中，你可以使用 delete 来删除一个对象或对象的属性。在 ActionScript 3.0 中，delete 操作符是遵循 ECMAScript 的，这就意味着 delete 只能用在删除对象的动态属性上。

如果你试图删除一个对象的非动态属性，他就会触发编译器的一个错误提示：

1189 > Delete removes dynamically defined properties from an object. Declared properties of a class can not be deleted. This error appears only when the compiler is running in strict mode.

在 AS3 中，delete 将会返回一个布尔值来说明它是否删除成功，你可以试试下面的代码：

```
var t:* = new Timer(15); // no typing to get around the compiler error
trace(delete(t.delay)); // traces false, object is sealed so can't delete
trace(t.delay); // 15 - delete never occurred

var o:* = {fun:"stuff"};
trace(delete(o.fun)); // traces true, object is dynamic so can delete
trace(o.fun); // undefined - delete occurred
```

正题

part1

目前我暂时在研究 ActionScript3.0，它的能力让我很激动。它的原生执行速度带来诸多可能（此句原文 The raw execution speed by itself provides so many possibilities. raw 本意未加工，原始的，这里的意思是指引入 AVM2 之后，ActionScript3.0 在执行速度上有了很大提高，所以使支持更复杂的组件成为可能，译者注）。它引入了 E4X、sockets、byte 数组对象、新的显示列表模型、正则表达式、正式化的事件和错误模型以及其它特性，它是一个令人炫目的大杂烩。

能力越大责任越大（译者：出自蜘蛛侠），这对 ActionScript3.0 来说一点没错。引入这些新控件带来一个副作用：垃圾收集器不再支持自动为你收集垃圾等假

设。也就是说 Flash 开发者转到 ActionScript3.0 之后需要对关于垃圾收集如何工作以及如何编程使其工作更加有效具备较深入的理解。没有这方面的知识，即使创建一个看起来很简单的游戏或应用程序也会出现 SWF 文件内存泄露、耗光所有系统资源（CPU/内存）导致系统挂起甚至机器重启。

要理解如何优化你的 ActionScript3.0 代码，你首先要理解垃圾收集器如何在 FlashPlayer 9 中工作。Flash 有两种方法来查找非活动对象并移除它们。本文解释这两种技术并描述它们如何影响你的代码。 本文结尾你会找到一个运行在 FlashPlayer9 中的垃圾收集器模拟程序，它生动演示了这里解释过的概念。

关于垃圾收集器 垃圾收集器是一个后台进程它负责回收程序中不再使用的对象占用的内存。非活动对象就是不再有任何其他活动对象引用 它。为便于理解这个概念，有一点非常重要，就是要意识到除了非原生类型（Boolean, String, Number, uint, int 除外），你总是通过一个句柄访问对象，而非对象本身。当你删除一个变量其实就是删除一个引用，而非对象本身。 以下代码很容易说明这一点： ActionScript 代码

```
// create a new object, and put a reference to it in a:
var a:Object = {foo:"bar"}
// copy the reference to the object into b:
var b:Object = a;
// delete the reference to the object in a:
delete(a);
// check to see that the object is still referenced by b:
trace(b.foo); // traces "bar", so the object still exists.
```

如果我改变上述示例代码将 b 也删除，它会使我创建的对象不再有活动引用并等待对垃圾收集器回收。ActionScript3.0 垃圾回收器使用两种方法定位无引用的对象 ：引用计数法和标识清除法。

引用计数法 引用计数法是一种用于跟踪活动对象的较为简单的方法，它从 ActionScript1.0 开始使用。当你创建一个指向某个对象的引用，该对象的引用计数器加 1；当你删除该对象的一个引用，该计数器减 1。当某对象的计数器变成 0，该对象将被标记以便垃圾回收器回收。 这是一个例子： ActionScript 代码

```
var a:Object = {foo:"bar"}
// the object now has a reference count of 1 (a)
var b:Object = a;
// now it has a reference count of 2 (a & b)
delete(a);
// back to 1 (b)
```

```
delete(b);
// down to 0, the object can now be deallocated by the GC
```

引用计数法简单，它不会非 CPU 带来巨大的负担；多数情况下它工作正常。不幸地是，采用引用计数法的垃圾回收器在遇到循环引用时效率不高。循环引用是指对象 交叉引用（直接、或通过其他对象间接实现）的情况。即使应用程序不再引用该对象，它的引用计数器仍然大于 0，因此垃圾收集器永远无法收集它们。下面的代码 演示循环引用是怎么回事： ActionScript 代码

```
var a:Object = {}
// create a second object, and reference the first object:
var b:Object = {foo:a};
// make the first object reference the second as well:
a.foo = b;
// delete both active application references:
delete(a);
delete(b);
```

上述代码中，所有应用程序中活动的引用都被删除。我没有任何办法在程序中再访问这两个对象了，但这两个对象的引用计数器都是 1，因为它们相互引用。循环引用 还可以更加负责 (a 引用 c, c 引用 b, b 引用 a, 等等) 并且难于用代码处理。FlashPlayer 6 和 7 的 XML 对象有很多循环引用问题：每个 XML 节点被它的孩子和父亲引用，因此它们从不被回收。幸运的是 FlashPlayer 8 增加了一个叫做标识-清除的新垃圾回收技术。

标识-清除法 ActionScript3.0 (以及 FlashPlayer 8) 垃圾回收器采用第 2 种策略**标识-清除法**查找非活动对象。FlashPlayer 从你的应用程序根对象开始(ActionScript3.0 中简称为 root)直到程序中的每一个引用，都为引用的对象做标记。 接下来，FlashPlayer 遍历所有标记过的对象。它将按照该特性递归整个对象树。并将从一个活动对象开始能到达的一切都标记。该过程结束后，FlashPlayer 可以安全的假设：所有内存中没有被标记的对象不再有任何活动引用，因此可以被安全的删除。图 1 演示了它如何工作：绿色引用（箭头）曾被 FlashPlayer 标记过程中经过，绿色对象被标记过，白色对象将被回收。

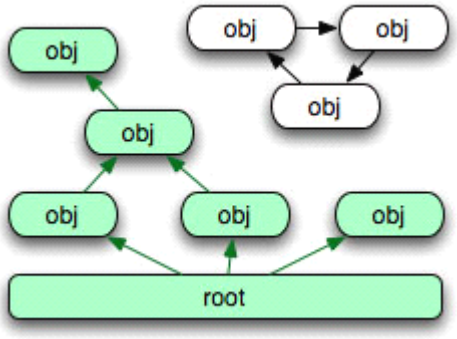

Figure 1. FlashPlayer 采用标记清除方法标记不再有活动引用的对象 标记-清除法非常准确。但是，由于 FlashPlayer 遍历你的整个对象结构，该过程对 CPU 占用太多。FlashPlayer 9 通过调整迭代标识-清除缩减对 CPU 的占用。该过程跨越几个阶段不再是一次完成，变成偶尔运行。

延期（执行）垃圾回收器和不确定性

FlashPlayer 9 垃圾回收器操作是延期的。这是一个要理解的非常重要的概念：当你的对象的所有引用删除后，它不会被立即删除。而是，它们将在未来一个不确定的时刻被 删除（从开发者的角度来看）。垃圾收集器采用一系列启发式技巧诸如查看 RAM 分配和内存栈空间大小以及其他方法来决定何时运行。作为开发者，你必须接受这 样的事实：不可能知道非活动对象何时被回收。你还必须知道非活动对象将继续存在直到垃圾收集器回收它们。所以你的代码会继续运行(enterFrame 事件会继续)、声音会继续播放、装载还会发生、其它事件还会触发等等。

记住，在 FlashPlayer 中你无权控制何时运行垃圾收集器去回收对象。作为开发者，你需要尽可能把你的游戏或应用程序中无用的对象应用清除。

两个例子：
http://www.adobe.com/devnet/flashplayer/articles/garbage_collection.html#
http://www.adobe.com/devnet/flashplayer/articles/garbage_collection.html#

part 2

AS3 给开发者带来了更快的代码运行速度和更多功能更强大的 API。但不幸的是，伴随着功能的增强，对开发者的专业能力要求也越来越高。这篇文章将重点讲述 AS3 在资源管理方面的特性和这些特性可能会引起的一些让人头疼的问题。下一篇文章，我将介绍一些可供我们使用的对策。

在 AS3 中对资源管理影响最大的是其新引入的 display list 模型。在 flash8 和其以前的版本中，当一个 display object 被移除后（用 removeMovie 或

unloadMovie），那么这个对象和其所有的子节点将会被立即从内存中删除，并且终止运行该对象的所有代码。Flash Player 9 引入了更灵活的 display list 模型，把 display objects（Sprites, MovieClips, etc）看做一般的对象。这就意味着开发者可以做很多有趣的事情，比从新定义 display object 的容器（将一个 display object 从一个 display list 移到另一个中），从已生成的 swf 文件中读取已实例化的 display object。不幸的是：这些 display object 将会被垃圾收集器等同于一般对象来处理。这就会引发一些潜在的问题。

问题 1：动态的内容

其中一个比较明显的问题出现在与 Sprite 相关或其他容器的动态实例中，当你想要在某段时间后移除该对象时，Sprites（或其他容器）就会出现一个比较明显的问题：当你从舞台（stage）上将其移除后，此时 display object 已经不在 display list 上了，但事实上它仍然在内存中没有被清除。如果你没有清空这个剪辑的所有的引用或监听器，它可能用远也不会被从内存中删除。

灰常值得我们注意的是：display object 不仅仅仍然占用着内存，而且它仍然在执行其内部的代码，例如 Timer，enterFrames 和其相关的外部监听器。

现在有一个应用于游戏的 sprite 正在执行一个内部的 enterFrame 事件，每一帧它都会执行一些运算，并判断出它附近的其他游戏元素。在 AS3 中，即使你将其从 display list 中删除（removeChild）并且将其引用全部设置为 null，在其被垃圾收集器回收之前，它将继续执行 enterFrame 内的代码。在删除 sprite 之前，你必须明确：“enterFrame 的监听器已经被删除”。

假设有一个影片剪辑监听来自舞台的鼠标移动事件，在你移除其监听器之前，即使是该剪辑已经被删除（deleted），每有一次鼠标移动事件，其代码都会被执行，也就是这个剪辑将会被永远运行下去，作为一个来自舞台事件派发的引用。

假设：在一些相互关联而且各自处于不同的状态的 sprite（例如一些处于实例化，一些已经被删除），又或是在你删除对象前没能将其所有的引用清空时。你可能不会发觉就是这个细节令你的 cpu 使用率直线上升，降低你的程序或者游戏的执行速度，或者使用户的电脑进入无响应状态。没有方法可以强制 flash 播放器去立即删除一个 display object 并且停止其代码执行。你必须在从一个容器中删除一个 display object 时手动的做一些处理。下一篇文章我将介绍对策。

这里有一个例子（需要 flash 9 播放器），单击 create（创建） 按钮来创建一个新的 Sprite 事例，Sprite 将会带有一个计数器，并会显示出来。单击 remove 按钮并且观察计数器是怎么变化的，事实上 sprite 的所以引用都已经被置空。你可以多创建几个实例来观察这个问题是如何让一个程序恶化的。代码会在本篇文章的最底端。

http://www.gskinner.com/blog/archives/2006/07/as3_resource_ma_1.html

问题 2：加载的内容

设想把加载的 swf 文件也同样当做一般的对象来对待，很容易想到一些你会遇到的问题。正如 display objects 一样，没有一种明确而有效的方法将 swf 的加载内容从内存中删除，或者立即停止其运行。置空其调用的 Loader 对加载的 swf 的所有引用，它最终还是会被 GC（垃圾收集器）收集的。

考虑 2 个这样的情景：

1.你创建了一个 shell 来进行 flash 实验。这个实验是最前沿的，并且它会使 cpu 使用率到达最高。一个用户点击按钮加载了一个实验，观察后，又点击按钮加载了第二个实验。即使是第一个的引用全部被清空，它仍然在后台运行，在第二个实验运行的同时，两个实验的负荷已经超出了 cpu 最大运算能力。

2.一个客户要求你建立一个程序来读取其他开发者的 swf 文件。那些开发者给舞台添加了监听器，或者是用其他方法创建了一些指向 swf 内部的一些引用。你现在没办法删除它的内容，直到关闭应用程序之前，它将一直在内存中并占用 cpu。即使它们没有“内部引用”，它们也将继续执行下去，知道下一次 GC（垃圾收集器）将它们释放。

出于项目安全的考虑，当你加载第三方内容时，你必须要了解的是：你是无法控制其删除或执行的。当一个 swf 被加载后，其很容易伴随你的程序一直在运行，即使是被删除了，当发生交互时被加载的 swf 可能会继续捕捉或干扰用户。

另一个例子和第一个问题一样，只是每次加载 swf 时，不使用动态实例。

上述有两个提及两个情景事例在下面的页面中：
http://www.gskinner.com/blog/archives/2006/07/as3_resource_ma_1.html

问题 3：时间轴

希望这个问题能在最终版本中解决。（译者：此篇文章是 2006 年的）

在 as3 中时间轴是可以用代码来操作的。当执行播放时，它会动态的实例或删除 display object。这就意味着会遇到与问题一相同的问题。在某一帧虽然剪辑从舞台被删除，但是它仍然会继续保存在内存中，在被回收前，它会继续执行其内部的所有代码。这不是程序员所期望的，当然也不是 flash 设计者所期望的。

这里有一个例子，同样的原理，不过这次是在两帧之间删除和实例化。

http://www.gskinner.com/blog/archives/2006/07/as3_resource_ma_1.html

Adobe 在想啥？或者，为什么会出现这个问题？

Java 开发者在看到这一问题时可能会说：“那又如何？”。这种差异是可以理解的， flash 开发人员并不习惯于手动干涉内存管理（因为以前就没这问题），而 Java、C++的开发人员又已经习惯了强大的 GC（无论是自动的还是手动的）。这些问题是最新的内存管理语言自身带来的缺陷，不幸的是这些问题是无法被避免的。

另一方面，Flash 带来了很多在其他语言中罕见的问题（包括 Flex 中的一部分也是）。Flash 内容中往往会有很多空闲或被动代码在被执行，尤其是 java 和 Flex 在交互时（通常来说：很多密集型运算都是和入户的输入紧密相连的）。Flash 工程会更经常读取外部第三方内容（其代码质量通常是很差的）。Flash 开发人员可利用的工具，资料和框架都比较少。而且据我所知负责开发 flash 的工作者的背景一般都是来自：音乐，艺术，商业，哲学或是其他的什么，但是除了专业程序员。

这种多元化的组合带来了令人惊奇的创造力和内容，但是他们并没有准备去讨论资源管理的问题。

总结

资源管理将会成为 AS3 开发的一个重要部分。忽略这一问题，可能会导致程序运行缓慢，或是完全的拖垮用户的系统。目前为止还没有明确的方法能立即将 display object 从内存中删除并停止其内部代码运行，这就意味这我们有责任去妥善的处理我们创造出来的对象。希望经过交流，可以探索出一个最佳的方法和框架来更轻松解决这个问题。

本篇涉及代码：
http://www.gskinner.com/blog/assets/resMgt2/resourceManagement2.zip

part 3

在第三部分，我们会集中于讲解一些新的工具（AS3 / Flex2）使内存管理更有效。在内存管理方面官方的只有两个函数和内存管理直接相关，但是它们都灰常有用。下面是一些非官方的补充，当然这并不是唯一方法。

System.totalMemory

这是一个简单的工具，但是它灰常重要，因为它是开发者在 flash 中第一个可以实时应用的工具。它可以让你监听到 flash 播放器实时的内存占用大小，更重要的是：你可以利用这个来判断是否抛出异常，来终止即将给用户带来的负面体验 。

下面是一个例子：

```
import flash.system.System;
import flash.net.navigateToURL;
import flash.net.URLRequest;
...
// check our memory every 1 second:
var checkMemoryIntervalID:uint =
setInterval(checkMemoryUsage,1000);
...
var showWarning:Boolean = true;
var warningMemory:uint = 1000*1000*500;
var abortMemory:uint = 1000*1000*625;
...
function checkMemoryUsage() {
  if (System.totalMemory > warningMemory && showWarning)
{
    // show an error to the user warning them that we're
running out of memory and might quit
    // try to free up memory if possible
    showWarning = false; // so we don't show an error
every second
  } else if (System.totalMemory > abortMemory) {
    // save current user data to an LSO for recovery
later?
    abort();
  }
}
function abort() {
  // send the user to a page explaining what happpened:
  navigateToURL(new URLRequest("memoryError.html"));
}
```

这一行为可以通过很多方法来实现，但是至少证明了这一行为的目的是好的。

灰常值得我们注意的是：totalMemory 是一个被一个进程（a single process）使用的“全局变量”（shared value），一个进程可能只有一个窗口，或多个浏览窗口，这些取决与浏览器，操作系统或是有多少个窗口被打开。

弱引用

在 AS3 众多的新特性中，我灰常高兴的看到“weak references”。这种引用不会被垃圾收集器作为判定 object 是否被回收的依据。它的作用是：如果当一个对象仅仅剩下弱引用时，这个对象将会被垃圾收集器在下一轮回收。但是弱引用

只支持两种类型：第一种是经常会因为内存管理机制带来麻烦的事件监听器，我强烈的建议：每当添加监听器时，都将其第五个参数选项，即弱引用设置为 true。下面是其对应的参数设置的例子：

```
someObj.addEventListener("eventName",listenerFunction,use
Capture,priority,weakReference);
stage.addEventListener(Event.CLICK,handleClick,false,0,tr
ue);
// the reference back to handleClick (and this object)
will be weak.
```

更多关于弱引用请浏览： http://www.gskinner.com/blog/archives/2006/07/as3_weakly_refe.html

另一个弱引用支持的是 Dictionary object。一般情况下在初始化时设置其第一个参数为 true，下面是例子：

```
var dict:Dictionary = new Dictionary(true);
dict[myObj] = myOtherObj;
// the reference to myObj is weak, the reference to
myOtherObj is strong
```

更多关于 dictionaries 在 ActionScript 3 的介绍和应用，请点击 http://www.gskinner.com/blog/archives/2006/07/as3_dictionary.html

比较爽的就是可以利用弱引用支持 Dictionary 这个特性，将弱引用“钩”到其他内容上。例如：使用弱引用创建 WeakReference 和 WeakProxyReference 类来实现任何对象都可以创建弱引用。

WeakReference 类

WeakReference 利用了 Dictionary 可以存储弱引用的特点来实现将弱引用“钩”到其他任意对象的功能。这个类在实例化和访问上会有一些开销，所以我建议将其应用在一些可能得不到释放而且较大的对象上。这些代码虽然不能取代那些使对象正常“分解”的代码，但是它可以帮你确保大型数据对象被垃圾收集器正常分解。

```
import com.gskinner.utils.WeakReference;
var dataModelReference:WeakReference;
function registerModel(data:BigDataObject):void {
   dataModelReference = new WeakReference(data);
}
...
function doSomething():void {
```

```
   // get a local, typed reference to the data:
   var dataModel:BigDataObject = dataModelReference.get()
as BigDataObject;
   // call methods, or access properties of the data
object:
   dataModel.doSomethingElse();
}
```

从良好的代码结构来说，这是一个好的解决方案，因为它保证了你的数据类型带来良好的安全性，而且没有二义性（non-ambiguou）。这些代码是那些希望快速实现这一功能的人准备的，我还将其整合到另一个 WeakProxyReference 类（同时也是一个学习代理（Proxy）的好例子 ）中。

WeakProxyReference 类

WeakProxyReference 使用了 Proxy 类来代理弱引用对象。它的效果和 WeakReference 类是基本是一样的，**WeakProxyReference** 可以直接调用弱引用对象的方法并且是直接传递给目标。**WeakProxyReference** 的问题就是失去了类型安全性，并且有一点点二义性代码。也就是说它可能会抛出运行时错误。（特别是当你试图去访问一个对象中不存的属性）但是不会出现编译错误。

```
import com.gskinner.utils.WeakProxyReference;
var dataModel:Object; // note that it is untyped, and not
named as a reference
function registerModel(data:BigDataObject):void {
   dataModel = new WeakProxyReference(data);
}
function doSomething():void {
   // don't need to get() the referent, you can access
members directly on the reference:
   dataModel.doSomethingElse();
   dataModel.length++;
   delete(dataModel.someProperty);

   // if you do need access to the referent, you need to
use the weak_proxy_reference namespace:
   var bdo:BigDataObject =
dataModel.weak_proxy_reference::get() as BigDataObject;
}
```

一种可以强制执行垃圾收集的方法（不推荐使用）

在我的前一篇文章中，我说过在 AS3 中垃圾收集周期是不确定的，没有方法可以知道它下一次什么时候运行。严格的讲这句话也不完全的对，有一个技巧可以强制让 flash 播放器执行一次垃圾收集，这个技巧很方便你去探索垃圾收集和在开发期内测试你的程序，但是它绝不能出现在开发完成的产品中，因为它会破坏处理器的负载能力。同时官方也是不推荐使用的，所以你不能靠它的功能来完成实质功能上提升。

强制执行垃圾收集（表示计数法或引用清除法），你所要做的就是 执行两次相同的 LocalConnection。这样做系统会抛出一个异常，所以你必须为它准备好异常捕捉（try/catch）

```
try {
   new LocalConnection().connect('foo');
   new LocalConnection().connect('foo');
} catch (e:*) {}
// the GC will perform a full mark/sweep on the second
call.
```

再次重复一次：这个方法仅仅可以用于开发周期内的测试。它绝不能出现在开发完成的产品中！

总结

毫无疑问的是：ActionScript 3 给开发者在资源管理方面带来了更多的工作。虽然我们只有刚刚提到的一些应对工具，但是有对策总是比没有的好，而且 Adobe 至少也注意到这个问题了。采取有效的对策和方法并合理的搭配这些工具，相信你可以很好的管理 Flash 9 和 Flex 2 的工程。

Download WeakReference and WeakProxyReference.

http://www.gskinner.com/blog/assets/WeakReference.zip

尾声

补充资料：

原文出处：http://www.tan66.cn/?p=11

在 AIR 程序中发现了一种快速激活 GC sweep 的办法。
那就是将窗口最小化，瞬间就环保了。
当然为了做到悄无声息，还要将窗口还原。
这两句代码写在一起就可以了。前提条件是当前状态不是最小化。

```
stage.window.minimize();
stage.window.restore();
```

问题是，这样会看到窗口缩小又变回来的过程。所以在实际开发程序的时候可以将主程序窗口隐形

```
visible=false;
```

将要显示的内容放在其它窗口中，当然不要忘记加入控制主程序窗口退出的代码。