

AS2 转 AS3

文: iiley

关键字: AS2 转 AS3, AS3 教程, AS3 学习, AS3 入门, AS3 新手

目录

AS2 转AS3	1
第一章 语言相关的转换.....	3
1. 类 (Class) 的转换.....	3
2. 方法 (Method) 的转换	4
3. 变量 (Variable) 的转换.....	5
4. 语言上其他方面的转换注意事项	6
第二章 可视元素 (MovieClip, Button...) 的转换.....	7
1. 主要机制的变化.....	7
2. 主要的可视元素类介绍.....	7
3. 使用注意.....	8
第三章 事件的转换.....	10
1. 监听事件的方式 (Handling the events)	10
2. 事件类 (Event classes)	11
3. 事件流 (Event flow)	11
第四章 其他转换.....	14
1. 数 (Number) 的转换	14
2. 映射 (Map) 的更好实现方法.....	14
3. Interval timer的实现有更多选择	15
4. 更多强大的新功能.....	15
第五章 总结与建议.....	16

第一章 语言相关的转换

AS3 相对 AS2 的语法变化其实并不太大，比较需要注意的有下面这些。

1. 类（Class）的转换

差异点介绍：

AS3 的类修饰符在 AS2 的基础上增加了 **internal** 和 **final**。

internal 是默认的修饰，即缺省设定，它表示此类访问范围为同一个包以内（和 Java 的缺省的类修饰一样）。

final 代表此类不可被继承。

另外，AS3 还可以使用内部类，内部类在主类同一个文件里 **package** 大括弧的外面申明，它只能被申明为 **internal** 的。并且，这个内部类只能被主类或者同文件的其它内部类访问。比如这样：

```
package{
    public class MainClass{
        var inter1:InterClass1;
        var inter2:InterClass2;
    }
}

class InterClass1{
}
class InterClass2{
    var inter1:InterClass1;
}
```

转换建议：

对类的转换注意的不多，因为 AS3 只是增强了功能，所以如果直接转，几乎没有任何问题，如果想要通过新关键字来增强原来类的结构，则需要花时间考虑采用这几个新的关键字对结构进行更好的组织，比较有用的可能是内部类，它可以把一些只能被一个类使用的辅助类隐藏起来，不干扰外界的注意力。另外，interface 跟 AS2 一样，只是方法不能显式申明为 **public**，缺省并且只能是 **public** 但是不能用 **public** 关键字，这有点别扭，你说只能用 **public** 那咋就用，但是你又不让写上 **public** 这几个字.....必须得缺省的用.....这有点不爽。

2. 方法 (Method) 的转换

差异点介绍:

AS3 的方法修饰符在 AS2 的基础上增加了 **protected**, **internal**, **final** 和 **override**, 并且 **private** 的语义有变化。无返回的关键字 **Void** 变成了 **void**。方法参数的设定也有一些变化。

private 关键字修饰后代表此方法只能被本类访问。这变得和 Java 语言一样了。而在 AS2 中, **private** 相当于下面要说的 AS3 的 **protected**。

protected 关键字修饰后代表此方法只能被自己和自己的子类访问, 这和 Java 语言的 **protected** 关键字意义一样, 学过 Java 的朋友应该很熟悉。

internal 关键字修饰后代表此方法只能被同一包里的类访问, 这和 Java 语言的缺省方法权限也一样。

final 关键字修饰后代表此方法不能被覆盖。同样, 和 Java 语言也是一样的。

override 关键字修饰那些覆盖父类方法的方法, C#程序员可能非常熟悉这个关键字, 但是 Java 程序员可能有些不太理解, 覆盖.....还需要关键字? 我这里讲一个我以前在 Java&AS2 时代遇到的麻烦, 有一次, 我写一个 Text 组件, 继承自 JSDK 的 JTextComponent 类, 我这个组件功能比较多, 因此方法也比较多, 其中有一个方法名叫 updateText (具体名字我忘记了, 这里随使用一个)。这个组件虽然功能多, 但是都很简单, 逻辑也很清晰, 但是始终会发生很奇怪的错误, 反复检查了三千七百八十六遍代码, 还是找不到错误。最后你猜错误在什么地方? 就在 updateText 这里, 原来 JTextComponent 也有这个方法, 我又写了同名的这个方法, 因此就错误地覆盖了 JTextComponent 的那个方法, 当然, 不能正常工作了。你想想, 要是覆盖的方法都必须加上 **override** 关键字才能编译通过, 那么我在编译的时候就会知道这里不该用这个名字了, 也就不会浪费那么多时间了。简单总结就是: **override** 必须出现在覆盖父类方法的方法前面, 以避免无意的错误覆盖。

override 必须出现在 **protected**, **internal**, **public** 的前面。按照逻辑, 你可以分析知道 **override** 不能出现在 **private** 和 **final** 的前面。

Void 变成了 **void**, 这个只是字符变了, 语义没有变, 查找替换即可。

参数设置方面, 变化的主要有:

参数数量严格检查, 在 AS2 中, 方法申明中如果有 $n(n \geq 0)$ 个参数, 在调用的时候, 可以传 $x(x \geq 0)$ 个参数, 当 $x < n$ 的时候, $[x, n)$ 这些参数会被视作 **undefined**, 而当 $x > n$ 的时候, 多于的参数则被忽略, 也就是说, 传入的参数个数可以不跟申明的个数一致。而在 AS3 中, 则必须一致。

参数缺省值设定, 在 AS3 中可以有这种用法, 申明 `function method(x:int=1,`

`y:int=2`), 调用时 `method()` 代表 `method(1, 2)`, `method(x)` 代表 `method(x, 2)`。这和 C++ 语言是一样的, 熟悉 C++ 的朋友可以跳过这一段了。也就是说, 缺省参数可以让调用的时候少传入一些参数, 没有传入的参数被缺省值代替。注意缺省参数必须是从后往左连续的, 也就是说所有的缺省参数必须在必备参数的后面。这样的用法 `method(param1:int, param2:String="", param3:Number=6)` 可以, 而 `method(param1:int=1, param2:String="", param3:Number)` 不可以。

任意数量参数, 为了实现任意数量参数的方法, AS3 增加了一种模式, `method(...args)` 模式,

`...args` (`args` 可以是任意合法变量名) 代表一个数组, 你可以通过遍历数组的方式得到 `args` 内所有参数。注意 `...args` 必须是最后一个参数, 即是说 `method(param:int, ...args)` 合法, 而 `method(...args, param:int)` 不合法。使用例子:

```
package {
    import flash.display.MovieClip;
    public class RestParamExample extends MovieClip {
        public function RestParamExample() {
            traceParams(100, 130, "two"); // Output: 100,130,two
            trace(average(4, 7, 13));    // Output: 8
        }
    }
}

function traceParams(... rest) {
    trace(rest);
}

function average(... args) : Number{
    var sum:Number = 0;
    for (var i:uint = 0; i < args.length; i++) {
        sum += args[i];
    }
    return (sum / args.length);
}
```

3. 变量 (Variable) 的转换

和类、方法的变化类似, 变量也增加了 **protected**, **internal** 修饰符, 作用和类、方法的同名修饰作用相同, 这里就不重复说明了。**final** 关键字不能用来修饰变量, 这和 Java 不同, 取而代之的是一个替代 **var** 的 **const**, 关键字, 用来标识常量, 比如 `const N:int = 100`; 这和 C/C++ 类似。

还有一个变化是, AS3 引入了新的类型 *****, ***** 代表任意类型 (即不作类型检查), 虽然 AS3 中, 也可以对变量以及方法返回类型作不声明的设置, 但是 `FlexBuilder` 会出警告, 因为 AS3 相对 AS2 来说更静态, 所以不建议省略类型声明, 如果你实在需要不做类型检查,

那么就使用`*`，比如 `var dynamicProperty:*; dynamicProperty` 将可以被赋予任何类型的值，`function method():*` 将可以返回任何类型的值。

另一个重要的地方是，变量的初始值与 AS2 不同，这在转换的时候要小心，对于 AS2，变量在赋值之前，都等于 **undefined**，而 AS3 中，不同的类型初始值分别如下：

Data type	Default value
Boolean	false
int	0
Number	NaN
Object	null
String	null
uint	0
未申明类型（等同于 * 类型）	undefined
其他的类型，包括用户自定义类。	null

所以，这里得小心了，**int**、**Number**、**uint**、**Boolean** 等类型的变量将不可能具有 **null** 或者 **undefined** 的值。

4. 语言上其他方面的转换注意事项

instanceof 应该用 **is** 替换。

强制转换有两种方式，**1** 旧的方式 `YourClass(value)`，如果转换不成功将抛出异常，而不是返回 **null**；**2** `value as YourClass` 转换不成功会返回 **null**。

String、**Number** 等不存在原始类型和类类型的区分了，也就是说在 AS2 种的 `new String("1") !== "1"` 的情况不再存在了。并且 `"" is String` 也返回 true 了。

namespace 使得你可以自定自己的命名空间，来补充 **public**、**protected**、**private**、**internal** 的不足，比如你可以给你的类定义在 `namespace yourspace` 命名空间中，这样只有同样用了 `namespace yourspace` 的类才可以访问那个类。从而摆脱了 **public**、**protected**、**private**、**internal** 的限制。不过这里我还是不推荐用此东西，几个标准的修饰通常已经够用了，增加 **namespace** 可能会增加结构复杂性从而难以阅读和使用。

第二章 可视元素 (MovieClip, Button...) 的转换

1. 主要机制的变化

在 AS3 种, 可视元素有很大的变化, 在 **flash.display** 包中, 包含了 33 个类/接口, 比起 AS2 时代的 MovieClip, Button, TextField 等简单几个类来说, 是复杂并强大了不少。(注: 下面有的地方简称可视元素为元件)

AS3 的可视元素使用方法和 AS2 不同, AS2 时代, 创建一个可视元素必须通过 *MovieClip.createEmptyMovie()*, *MovieClip.createTextField()* 方法来创建, 不方便不说, 更重要的是, 一个元件从创建时, 就被加到了舞台上, 从舞台上移出后, 这个元件就不可用了。而 AS3 中, 你可用 **new** 来创建任何可视元素, 在需要用的时候把它加到舞台上, 不用的时候移除下来, 等到再用的时候, 还可以再放上去就是。因此非常方便。由于这个特点, 不同 swf 里面的元件可以相互使用了, 比如你在 swf1 的 library 里面做了一个图标, 绑定的类是 *Icon1*, 你可以把 swf1 加载到 swf2 中, 然后通过 *new Icon1()* 创建一个那个图标, 然后通过 *addChild()* 方法把它加到 swf2 的舞台上。也许你已经意识到, 我们再也不需要蹩脚的共享库 (Shared Library) 了。

2. 主要的可视元素类介绍

这里简单介绍一下主要的可视元素, 它们各自的特点和用途。

DisplayObject: 所有可视元素都继承自 **DisplayObject** 类, **DisplayObject** 包含了一系列基本的属性、方法和事件, 比如坐标 *x*, *y*, 透明度 *alpha* 等, 有好些属性和方法和 AS2 的 MovieClip/Button/TextField 类似, 但是有的是有区别的, 比如 *alpha* 的取值范围就不是 0 到 100, 而是 0 到 1, 在使用的时候注意查看文档, 不要犯小错误。

InteractiveObject: 所有可交互的可视元素都继承自 **InteractiveObject** 类, **InteractiveObject** 包含了一系列可交互属性、方法和事件, 比如是否可点击 *mouseEnabled*, 是否可 Tab 接收焦点 *tabEnabled* 等, **InteractiveObject** 自身增加的属性和方法不多, 主要是交互事件上的增加。关于事件将在下一章讲解。

Bitmap: 表示一个位图, 可以和 **BitmapData** 很好的结合使用。

Shape: 表示一个矢量图型, 可以用其 *graphics* 属性代码绘制图形。

DisplayObjectContainer: 可视元件的容器, 可以在其中放置其他可视元件, **Stage** 就是一个容器, 所以可以向 **Stage** 中添加可视元素。**DisplayObjectContainer** 继承自 **InteractiveObject**, 但是不能直接构造它的实例。

Sprite: 最常用的元件，他继承自 **DisplayObjectContainer**，所以通常用它来做容器或者用来实现可交互的元件。

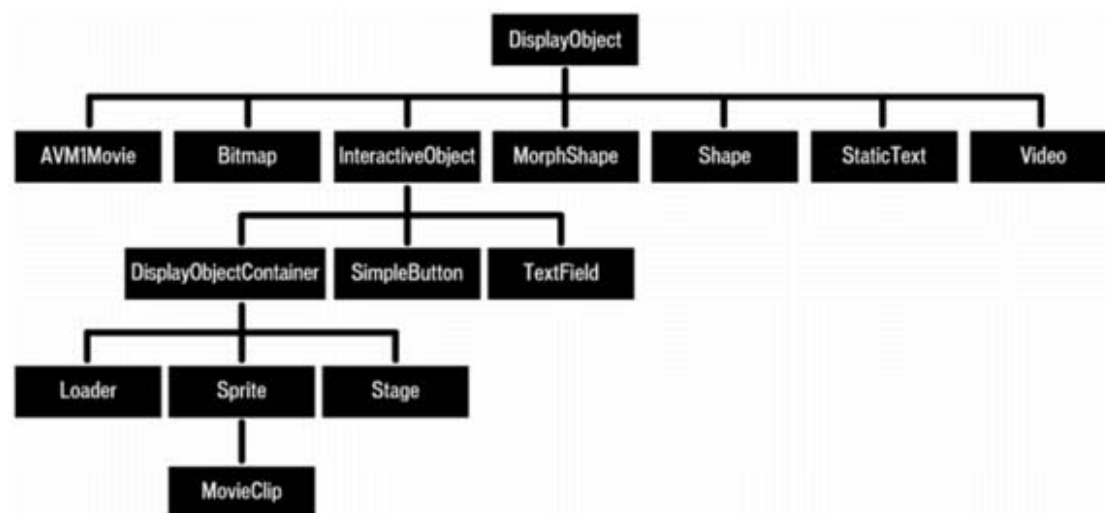
MovieClip: 继承自 **Sprite**，增加了时间轴（帧的设定）。这个类在 AS2 时代是使用最多的可视元件类型了，但是到了 AS3，它被使用的频率将会大大减少，因为通常很多元件是不需要时间轴的，**Sprite** 基本上取代了它的重要位置。

Loader: 加载器元件，你可以把可视内容（图片，swf 等）东西加载到它里面，然后把它放入舞台上的容器里从而显示出来，这比起 AS2 时代必须把东西加载到 **MovieClip** 要方便多了。

TextField: 显示/编辑文字的元件，它其实是在 `flash.text` 包里面的，由于也是非常重要的可视元素，所以这里一并讲解。功能和 AS2 时代基本相同，在关于内容尺寸计算方面有一些增强，多了不少方法。鉴于 AS3 可视元素可以存在于舞台之外的特点，**TextField** 因此可以说功能大大增强，当你不使用一个 **TextField** 的时候，把它从舞台上弄下来，他的各项属性仍然保留着，下次加上舞台的时候，完璧归赵，无任何损失。

Stage: 舞台类。在 AS2 时代，Stage 可控的东西不多，而 AS3 中，**Stage** 其实就是一个 **DisplayObjectContainer**，并且还拥有其他一些属性和方法。功能很强大。当然，由于其特殊性，有的属性和方法是不能使用的，详见帮助文档。

可视元素的继承关系图如下：



3. 使用注意

在 AS2 中的全局变量 `_root` 已经不存在了，因此要给舞台上添加内容，不是那么容易。每个 **DisplayObject** 都有一个 `root` 属性，此属性要在此元件被添加到舞台上之后才有值，因此想直接通过 `root` 往舞台上添加元件是不容易的。`root` 属性表示一个 swf 的根，如果你的影片只是一个 swf，那么所有 **DisplayObject** 的 `root` 值除了 `null` 之外只可能是同一个值。但是如果你加载了另一个 swf 到影片里，那么就有了另一个 `root`。

DisplayObject 的 **stage** 属性将会比较重要，stage 表示 flashplayer 的播放舞台，因此不管你的影片加载了多少个 swf，所有的 **DisplayObject** 的 **stage** 属性除了 **null** 都将是同一个值。**stage** 属性在一个 **DisplayObject** 被添加到舞台上时被自动设置，从舞台上移除后，自动归 **null**，因此，它也是指示一个元件是否在舞台上的一个标志。

没有了全局的 **root**，并且 **stage** 也是被添加到舞台后才被设置，那么如何往舞台上添加元件呢。对于使用 **FlashIDE**，制作的影片在播放的时候自动会被放置到舞台上，因此通过它，间接就可以访问到 **root** 和 **stage**，因此就可以控制放置/移除到舞台的操作了。对于使用 **Flex2** 编译器的用户，编译的主文件/类必须是继承自 **Sprite** 的类，生成的 **swf** 在播放时自动会创建一个那个主类的实例然后放置到舞台上成为 **root**，因此通过它，你也可以控制舞台了。只是，可能用起来不如 AS2 中直接控制 **_root** 来的方便。

第三章 事件的转换

AS3 的事件与 AS2 有很大的不同，形势更统一，功能更强大。

1. 监听事件的方式 (Handling the events)

在 AS2 时代，有几种监听事件的方式，比如 `onPress = function(){} , addListener(listener)` 等等，而 AS3 时代，统一用一种 `addEventListener(type:String, listener:Function, useCapture:Boolean = false, priority:int = 0, useWeakReference:Boolean = false):void`，这是接口 **IEventDispatcher** 声明的一个方法，**EventDispatcher** 类实现了这个方法（当然还实现了若干其他方法，这里不尽述），这里举个监听元件被按下的 AS2，AS3 方法对照的例子：

AS2:

```
mc.onPress = Delegate.create(this, __onPress);
...
private function __onPress():Void{
    //Do something
}
```

AS3:

```
mc.addEventListener(MouseEvent.CLICK, __onPress);
...
private function __onPress(e:MouseEvent):Void{
    //Do something
}
```

可以看出，首先不同的地方是 AS2 直接覆盖了 `onPress` 属性，让他调用 `__onPress`，而 AS3 是增加一个监听，并没有覆盖什么。说明 AS3 更灵活，因为如果你有两个监听器，AS2 会很难处理。其次，AS2 中为了使得监听函数的作用域（scope）能正确（即是说函数里面的 `this` 要能正确的指到本实例），通常我们需要用 **Delegate**。而 AS3 中，不需要使用它，因为 AS3 对于函数操作会自动进行 `Delegate`，相当于系统或者编译器帮我们调用了 `Delegate`，由此省去了麻烦，避免了忘记调用而引入的错误。AS3 中的每个事件监听函数都必须接受一个事件实例（Event instance，这将在下一节介绍）。

另外，这里看看 `addEventListener` 的其他几个参数，`useCapture` 是否用在捕获时期，这个在第三节会讲解；`priority` 优先级，同一个类型的事件，优先级越大的监听器，会越早被调用，默认值为 0；`useWeakReference` 是否是弱引用，AS3 加入了弱引用的功能，熟悉 Java 的朋友因该知道这是什么含义，即是指这次引用不会计算到引用计数中，垃圾回收器会在一个对象没有被任何其他对象强引用的时候，把它回收掉。也就是说，如果这里你用了 `useWeakReference=true`，那么这个监听器如果没有别的地方存在对它的引用的时候，就会被回收，不过通常我们都用默认值 `false`，因为严格管理的程序，你会自己记得

什么时候加监听，什么时候移除监听的。

在 AS2 时代，移除监听器，对于上面的例子很简单，只需要简单的 `mc.onPress=undefined` 即可，在 AS3 中，需要这样 `mc.removeEventListener(MouseEvent.CLICK, __onPress)`，其实也很方便，你移除刚刚增加的监听，只需要把 `addEventListener` 函数替换为 `removeEventListener` 函数，前三个参数保持不变（我这里的例子第三个参数都用了缺省值）。

AS2 中，还有其他监听事件的方法，比如 `Key.addListener` 这样的方式，这和 AS3 的方式有点相似，不同之处在于它是加入 `object` 作为监听器，然后调用 `object` 的指定名称的方法，因此它很动态，易出错，比如我把 `object` 的 `onKeyDown` 方法写成了 `onkeydown`，编译器不会报任何错误，但是程序运行的时候，可能就不是你想要的效果。

2. 事件类 (Event classes)

前面我们提到了，AS3 的事件监听函数都必须接收一个事件对象，不同的事件可能会接收不同类型的对象（具体是什么类型，请详查帮助文档），但是所有事件类都继承自 `flash.events.Event` 类。不同的事件类型都有他们各自的属性、方法，你可以从这些属性方法得到你想要的事件内容。比如 `MouseEvent.stageX` 让你知道鼠标事件发生时鼠标在舞台上的 x 坐标，`KeyboardEvent.keyCode` 让你知道键盘事件发生时的按下/释放的按键码。

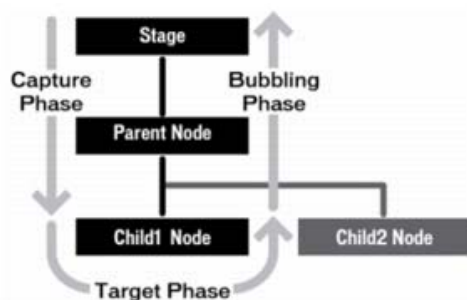
有很多属性、方法是与事件流相关的，因此我们放入下一节讲解。

3. 事件流 (Event flow)

事件流是 AS3 引入的新机制，它让可视元素的事件监听更灵活强大。

事件上事件流只对可视元素有效，也就是说 `DisplayObject` 对象和其子类对象才拥有此机制。究竟什么是事件流呢？

事件流是指一个事件不光是触发自己的监听器，它还会触发自己父元件到舞台整个路径中的其他节点的同类监听器，顺序是这样的：第一阶段（Capture 阶段）事件从 **stage** 到发生者的父元件路径中所有元件依次触发事件，然后第二阶段（Target 阶段）事件发生者自己触发，最后第三阶段（Bubbling 阶段），事件从发生者父类再回溯到 **stage** 依次触发。如下图所示：



注意这里的 Flow，是指舞台上的元件层次结构中的流，对应于 `DisplayObject.parent` 和 `DisplayObjectContainer.getChildAt()` 所相关的一个结构。注意这并不是类继承关系的结构。在接触 AS3 的初期，这比较难以理解透，这里我举个例子，应该能够使得你更好理解：

在 AS2 时代，假设我要做一个简单的窗口，这个窗口上面只放置了一个确定按钮，我们设想的功能是，首先这个窗口可以被拖动，然后用户点击确定按钮则关闭这个窗口。那么通常，简单地我们会创建一个 MC 作为窗体，然后这个 MC 里面创建一个 Button，监听 Button 的 `onPress` 事件来关闭 MC，这时没问题，然后拖动 MC 的实现，我们通过监听 MC 的 `onPress` 事件来 `startDrag`，问题出现了，由于 Button 在 MC 内，因此 MC 监听了事件后，Button 就接收不到事件了。通常为此，我们得创建一个背景元件放在那个 MC 里面，Button 的下面，然后监听那个背景元件的事件来启动拖动。这样要创建这个多余的背景元件原因就是，**如果一个元件的父元件监听了事件（这里指鼠标事件，有的事件是不同的），那么这个元件将不会监听到任何事件。**相当于说父元件吃掉了所有子元件的事件。而 AS3 里面，就不同了，回想一下刚才介绍的事件流的过程，如果用户点击了 Button，首先会是 stage 会触发事件，然后是 MC 的父元件们，然后是 MC，然后是 Button，然后再反向循环一次。如果用户点击了 MC（比如 Button 旁边，MC 内），则会是 stage->MC 父元件->MC-> MC 父元件->stage，两种情况，MC 都会得到事件触发，因此，在 AS3 时代，再也不需要创建多余的专门用来监听事件的辅助元件了。代码也会变得简单。并且 `capture` 和 `bubbling` 阶段的不同顺序，可以让你选择是先于事件触发者做动作，还是后于它做动作。
`addEventListener(type:String, listener:Function, useCapture:Boolean = false, priority:int = 0, useWeakReference:Boolean = false):void`
 方法的第三个参数，让你可以指定是否是监听 `capture` 阶段的事件，`false` 代表监听其他两个阶段的事件。在 `removeEventListener` 方法中，你要指定要移除的监听器是哪个阶段的，缺省值是 `false`。

相对于事件流，Event 类有一系列与之相关的属性和方法：

target: 此属性指向事件触发者，比如刚才的例子，鼠标点击了 Button，那么这个 `target` 就是 Button，如果鼠标点击了 Button 旁边，MC 内，那么它就是 MC。

currentTarget: 此属性指向事件流中，当前触发事件的元件。比如刚才的例子，鼠标点击了 Button，如果你监听了 MC 的鼠标点击事件，那么在监听 MC 的监听器里面，这个 `currentTarget` 就指向 MC，而在监听 Button 的监听器里面，这个 `currentTarget` 指向 Button。你可以看看事件流的那个图，`currentTarget` 实际上值的顺序依次是 Stage -> Parent Node -> Child1 Node -> Parent Node -> Stage。比较起 `target`，

对于每个事件，`target` 则始终是指事件发生者，比如图中的 `Child1 Node`。

bubbles: 此属性表示此事件在事件流中是否有 bubbling 阶段，有的事件是没有这一阶段的，比如 `unload` 事件，大部分非交互性事件也都没有 bubbling 阶段。所有事件都拥有 capture 阶段吗？前面讲过了，只有可视元素才拥有事件流的概念，也就是说非可视元素是不可能拥有 capture 和 bubbling 阶段的。那所有元件的事件，都拥有 capture 阶段？对，是这样的，`bubbles` 为 `false` 的事件虽无 bubbles 阶段，但有 capture 阶段。

cancelable: 此属性表示此事件是否可以取消它的默认行为。此属性与 `preventDefault()` 方法相关。

eventPhase: 此属性表示此事件处于事件流中的哪个阶段，`capture`，`target`，`bubbling` 三种阶段之一。

type: 此属性表示此事件是什么类型，这个属性与 `addEventListener/removeEventListener` 的第一个参数 `type:String` 对应。

preventDefault(): 此方法取消事件的默认行为，并不是所有事件都有默认行为，也并不是所有事件的默认行为都可以取消。一个事件的默认行为是否可取消，可以通过 **cancelable** 属性得知。这里举个例子，`TextEvent.TEXT_INPUT` 事件的默认行为是把输入的字符加入到 `TextField` 中，此事件行为可以取消，如果你调用它的 **preventDefault()** 方法，那么字符就不会加入到 `TextField` 中。而 `MouseEvent.MOUSE_DOWN` 这样的事件，就没有可取消的默认行为，比如你按下一个按钮，已经行为已经发生，不能取消，实际上这个行为在逻辑上看，也没有什么可取消的东西，看你怎样理解了，反正可否取 **cancelable** 已经表明。

isDefaultPrevented(): 此方法返回事件的默认行为是否已经被取消了。

stopPropagation(): 此方法可停止事件在事件流中的传播，比如上面提到的例子，假如你在 capture 阶段 stage 监听器里面调用了此方法，那么后面的节点中，都不会收到事件了。如果你在中途调用此函数，那么此函数执行后，后面的节点阶段，不会收到事件。

stopImmediatePropagation(): 此方法可停止事件在事件流包括当前节点中的传播，此方法与 **stopPropagation()** 的区别之处在于，**stopPropagation()** 不会停止当前节点的事件触发，你知道，我们可以给同一个节点，比如 stage 加入多个监听器，如果采用 **stopPropagation()**，那么在你调用了它之后，同节点的监听器被触发完全之后，事件停止传播。而 **stopImmediatePropagation()** 会在它被调用后立即停止传播，即使是同节点的事件，也将收不到事件。

总结:

其实 `Event` 类里面的大部分属性和方法都与事件流有关，当然也就是与可视元素有关。用户自己的非可视元素类，也可以有事件，简单的继承 `EventDispatcher` 类就可以拥有事件功能，但是，不会拥有事件流相关的东西，事件也不会有 `capture` 和 `bubbling` 阶段，只有 `target` 阶段。当然，一般来说普通的类，也不需要这些阶段。

第四章 其他转换

1. 数（Number）的转换

在 AS2 中，数就只有一种类型——**Number**，整数浮点数都统一为 **Number**。而在 AS3 中，数有三种不同的类型，分别为 **Number** 表示浮点数，**int** 表示整数，**uint** 表示无符号整数（即非负整数）。其中 **Number** 是 64 位，**int**，**uint** 是 32 位。因此在实际应用中，我们的选择更多了，如果不在乎，可以直接沿用 AS2 的方式全部用 **Number**，除了初始值不同外，没有其他区别（但是不要小看这个初始值不同，可能会引发很大的问题，因为 **Number** 不能为 **undefined** 了，非初始化或转换失败时的值都为 **NaN**）。我想有很多地方还是需要整数的，所以如果方便的话，尽量把只能为整数的地方采用 **int** 或者 **uint** 代替，这样能避免很多麻烦，减少很多 **Math.round()** 这样的调用。

要比较注意的地方可能是，直接书写的数字，默认是 **Number** 型的，比如 `array[2]`，这里的 2 其实是 **Number** 型的，`array[2]` 能工作是因为 `array` 会对 **Number** 进行自动转换，所以这里用起来没有问题。再比如，`var result:Number = 3/2;` 对于熟练的 Java 程序员，可能会认为 `result` 应该会等于 1，在 AS3 中，他会等于 1.5，理由就是 3,2 这里都本身就是 **Number** 型的。如果 `var result:int=3/2;` 则结果会是 1，因为 `result` 是整型的，AS3 会自动对数类型相互进行转换，所以大多不需要写强制转换的语句。比如有个方法为：

```
function traceInt(n:int):void{
    trace("n = " + n);
}
```

如果我这样调用 `traceInt(2.5)`，编译器不会报错，运行结果为 **n=2**。是因为 AS3 自动帮我们把 2.5 转换成了整型（转换方式是去掉小数部分取整）。这一特点使得数的三种类型可以相互很好的通用，但是在使用时也得小心，因为自动取整会在你难以察觉的时候发生值的改变。

2. 映射（Map）的更好实现方法

在 AS2 时代，相信很多人都用 **Object** 类实现过自己的 Map（比如 [ASWing](#) 里面的 `org.aswing.util.HashMap`），因为 **Object** 的键值特性，使得实现用 **String** 作键（**key**），任何类型作值（**value**）的 Map 很方便，而且效率很高。比如 `obj["key1"] = xxx`，`obj["key2"] = xxx`，这样的用法比比皆是。但是，它的弱点就是只能用 **String** 对象作键（非 **String** 的键会被自动 `toString()`），要实现通用的 Map，必须得能够接受任何类型的值作键，而 AS3 给我们提供了一个能实现此功能的类——**flash.utils.Dictionary**，它能真正接受任意类型的键值，比如 `dic[aObject] = xxx`，`dic[3.5] = xxx`。具体用法参见帮助文档。

3. Interval timer 的实现有更多选择

AS3 增加了 `flash.utils.Timer` 类，多数情况下它可以代替 `setInterval/clearInterval`。由于是类支持，封装性和可用性都比 `setInterval/clearInterval` 要好，推荐使用。

另外，在以前的时代，`onEnterFrame` 事件被大量的使用，在 AS3 时代，推荐 `Timer` 结合 `Event.RENDER` 事件实现更流畅反映更灵敏的交互程序。`Event.RENDER` 的优点在于，它会让你在图形更新前的最后阶段进行操作，使得操作马上得以表现出来。而 `onEnterFrame` 里的操作，是在下一帧才会体现出来的（这里我不能保证，但是经过长期的观察，我感觉它是这样的）。要触发 `Event.RENDER` 事件，你必须调用 `stage.invalidate()`。在一帧之内，如果你调用了 `stage.invalidate()` 那么这帧图形要更新之前，`Event.RENDER` 事件会触发，下一次更新的时候，就不会再触发了，除非你在下一次更新之前又调用了 `stage.invalidate()`。

4. 更多强大的新功能

AS3 不光是语言、可视元素、事件变化了，并且带来更多的强大的东西。

二进制控制：`flash.net.Socket` 类，能够支持二进制传输，`flash.net.ByteArray` 让你存储二进制数据，从而各种网络传输类都支持二进制传输，比如 `flash.net.URLLoader`, `flash.net.ShareObject` 等。可以说 AS3 中，摆脱了 AS2 时代的字符数据做主的形式，现在字符和二进制方式都可以使用了。

更强大的音频控制：这个我尚未研究，但是从网上很多人做出的音频相关的试验证明，AS3 已经可以对音频进行更多的处理，具体可翻翻 `flash.media` 里的类。

异常，断点调试：当然后者这跟 IDE 有关系了，对于异常，FlashPlayer9 能够在运行时弹出异常窗口表示出异常信息。这使得在 AS2 中还是鸡肋的异常机制，得以真正发挥作用了。FlexBuilder2 提供了 Debug 视图，可以给程序设置断点，单步调试，实时查看变量值等等现代 IDE 调试工具有的基本它都有了，对于调试程序来说，这简直比 AS2 时代方便太多了，我估计那些一批一批的 `Tracer/Log` 工具在 AS3 时代都将不会有下文。

第五章 总结与建议

在 AS2 项目往 AS3 转换/移植的过程中，不要想象这只是一个语法转换的过程，实际上，你不得不考虑 AS3 以及其类库中的一些新机制，所能带来的性能以及程序结构良好性的提升。因此，可能很多东西，你得重新设计，特别是可视元素和事件相关的，也就是用户交互方面的东西，我认为是必须得根据 AS3 的特点重新设计的，否则转换是无价值的，甚至会得到更差的程序。一些纯算法方面的代码，可能不需要重新设计，直接替换一些语言层面的东西即可。还有忘了说 FlashPlayer9 的速度提升，据我初步试验推断，代码执行速度提升 10 倍左右，可视元素运行/渲染速度平均提升 2 倍左右，位图渲染速度有比较明显的提升。

一些在 AS2 里面我们需要的东西，在 AS3 里面，我们不再需要了；一些用 AS2 做出来的东西，运行起来效率不够理想，用 AS3 做出来，运行得更快了；一些 AS2 里面不可能做到的东西，在 AS3 里面，我们可以做了。

相关资源:

Adobe Flex 2 Language Reference :

<http://livedocs.macromedia.com/flex/201/langref/index.html>

Flashseer论坛 : <http://www.flashseer.org>

AsWing : <http://www.aswing.org>