用 ActionScript 开发高级可视化组件

我们可以用ActionScript开发在Adobe® Flex™应用中使用的高级可视化组件，这个主题，包括以下方面的内容：

译者相关说明：

翻译：兰天　陕西新东方信息自动化有限责任公司

QQ:5995534098

Blog：http://ltian.javaeye.com

目的：在工作中需要使用Flex，在阅读文档时总是怕遗忘，所以记录下来，以备遗忘。另外，分发给同行共享，为快速提高国内Flex　Web应用开发水平贡献微薄之力，如有不妥之处，请指正。另外期望更多人加入到对Flex help翻译的行列中来。

1. 关于创建高级组件

简单组件一般都是现存 Flex 组件的子类，它们通过设置 skpins 或 styles 属性来修改父组件的外观，或者添加一些新的功能。比如，为 Button 控件增加一个新的事件类型，或者更改 DataGrid 控件缺省的 styles 和 skins。更多信息参见，Simple Visual Components in ActionScript.

在高级组件中，通常会执行以下动作：

- 更改已有组件的可视化外观或者可视化特性。
- 创建复合组件，将两个或者多个组件包装在其中。
- 通过继承 UIComponent 类来创建组件。

我们通常用继承现存类的方式来创建组件。比如，要创建基于 Button 的控件，我们就创建 mx.controls.Button 类的一个子类。要开发自己的组件，则需要创建 mx.core.UIComponent 类的子类。

关于重载 UIComponent 类的 protected 方法

Flex 所有的可视化组件都是 UIComponent 类的子类。因此，可视化组件继承了 UIComponent 类所定义的 methods，properties，events，styles 和 effects。

要创建高级的可视化组件，必须实现一个构造器（constructor）。另外要有选择性地重载下表中 UIComponent 类的一个或者多个 protected 方法：

UIComponent 方法	描述
commitProperties()	提交组件所有的属性变化。要么使属性同时更改，要么确保属性按照特定顺序设置。 更多信息参见：实现 commitProperties()方法。
createChildren()	创建组件的子组件。比如，ComboBox 控件包含了一个 TextInput 控件和一个 Button 控件作为它的子组件。更多信息参见：实现 createChildren() 方法.

layoutChrome()	定义 Container 类的子类容器的 border 区域。 更多信息参见：实现 layoutChrome() 方法.
measure()	设置组件的缺省 size 和缺省的最小 size。 更多信息参见：实现 measure()方法 .
updateDisplayList()	根据以前所设置的属性和样式来确定组件的子组件在屏幕上的大小（size）及位置（position），并且画出组件所使用的所有皮肤（ skins）及图形化元素。组件的父容器负责确定组件本身大小（size）。 更多信息参见：实现 updateDisplayList() 方法.

组件的使用者不会直接调用所有这些方法；Flex 调用这些方法。更多信息见 “关于组件实例化生命周期”。

关于失效(机制)的方法

在组件的生命周期中，应用可能会改变组件的大小和位置，更改组件的属性来控制组件的显示，或者更改组件的样式（styles）和皮肤（skin）属性。比如，可以更改组件中所显示的文本(Text)的字体(Font)大小。 作为变更字体大小工作的一部分，组件的大小也可能随之发生变化，这就需要 Flex 去更新应用的布局。布局操作需要 Flex 调用自定义组件的 commitProperties()，measure()，layoutChrome()，以及 updateDisplayList()方法。

应用通过程序来更改字体大小的执行速度大大快于 Flex 更新应用的速度。因此，你应该在确定最终要更改的字体之后再更新布局。

另外一个场景就是，当你设置了组件的多个属性后，比如 Button 控件的 label 和 icon 属性，你肯定会想让所有属性被设置后一次性执行 commitProperties()，measure()，和 updateDisplayList()方法，而不是设置过 label 属性后执行一遍这些方法，然后在设置 icon 属性后又执行一次这些方法。

另外，可能有几个组件同时改变了它们的字体大小。这时应该让 Flex 去协调布局操作，以消除任何冗余过程，而不是在每个组件更新它的字体大小之后都执行一次布局操作。

Flex 使用一种“失效机制（invalidation mechanism)”来同步组件的更改。Flex 用一系列方法的调用来标记组件的某些东西已经发生变化，并需要 Flex 去调用组件的

commitProperties(), measure(), layoutChrome(),或者 updateDisplayList()方法。通过这种办法，Flex 实现了“失效机制”。

下表描述了有关“失效(invalidation)”方法：

失效（Invalidation） 方法	描述
invalidateProperties()	通知组件，以使下次屏幕更新时，它的 commitProperties()方法能被调用。
invalidateSize()	通知组件，以使下次屏幕更新时，它的 measure()方法能被调用。
invalidateDisplayList()	通知组件，以使下次屏幕更新时它的 layoutChrome()方法和 updateDisplayList()方法能被调用。

当组件调用一个“失效”方法，它就通知 Flex，该组件已经被更新。当多个组件调用失效方法，Flex 会协调这些更新，以使这些更新操作下一次屏幕更新时一起执行。

通常，组件使用者不必直接调用这些“失效”方法。这些”失效”方法被组件的 setter 方法或者组件的其他方法在需要的时候调用.更多信息参见实现 commitProperties()方法.

关于组件实例化的生命周期

组件实例化生命周期描述了用组件类创建组件对象时所发生的一系列步骤.作为生命周期的一部分,Flex 自动调用组件的方法,发出事件,并使组件可见。

下面例子用 ActionScript 创建一个 Button 控件，并将其加入到容器之中：

```
//创建一个 Box 容器。
var boxContainer:Box = new Box();
//设置 Box 容器
…
//创建 Button 控件。
var b:Button = new Button()
// 设置 button 控件。
b.label = "Submit";
...
// 将 Button 添加到 Box 容器中。
boxContainer.addChild(b);
```

下面的步骤显示了用代码创建一个 Button 控件，并将这个控件添加到 Box 容器中时所发生的一切：

1. 调用了组件的构造函数，如下面代码所示：

```
// Create a Button control.
var b:Button = new Button()
```

2. 通过设置组件的属性对组件进行了设置,如下面代码所示：

```
// Configure the button control.
b.label = "Submit";
```

组件的 setter 方法将会调用 invalidateProperties(), invalidateSize(), 或者 invalidateDisplayList() 方法.

3. 调用 addChild()方法将该组件添加到父组件中,如下代码所示：

```
// Add the Button control to the Box container.
boxContainer.addChild(b);
```

Flex 执行以下动作：

4. 将 component 的 parent 属性设置为对父容器的引用.
5. 计算组件的样式(style) 设置.
6. 在组件上分发 preinitialize 事件.
7. 调用组件的 createChildren() 方法.
8. 调用 invalidateProperties(), invalidateSize()和 invalidateDisplayList()方法以触发后续到来的,下一个“渲染事件”(render event)期间对 commitProperties(), measure(), 或 updateDisplayList()方法的调用。

   这个规则唯一一个例外就是当用户设置组件的 height 和 width 属性时, Flex 不会调用 measure() 方法.
9. 在组件上分发 initialize 事件。此时，组件所有的子组件都被初始化，但是组件没有改更 size 和处理布局。可以利用这个事件在组件布局之前执行一些附加的处理。
10. 在父容器上分发 childAdd 事件.
11. 在父容器上分发 initialize 事件.

12. 在下一个“渲染事件”(render event)中，Flex 执行以下动作:

    a. 调用组件的 commitProperties()方法.

    b. 调用组件的 measure()方法.

    c. 调用组件的 layoutChrome()方法.

    d. 调用组件的 updateDisplayList()方法.

    e. 在组件上分发 updateComplete 事件.

13. 如果 commitProperties(), measure()或者 updateDisplayList()方法调用了 invalidateProperties(), invalidateSize(), 或 invalidateDisplayList()方法,则 Flexh 会分发另外一个 render 事件.

14. 在最后的 render 事件发生后，Flex 执行以下动作:

    a. 通过设置组件的 visible 属性使组件变为可视.

    b. 在组件上分发 creationComplete 事件.组件的大小(size)和布局被确定.这个事件只在组件创建时分发一次.

    c. 在组件上分发 updateComplete 事件.无论什么时候,只要组件的布局(layout),位置,大小或其它可视的属性发生变化就会分发这事件,然后组件被更新，以使组件能够被正确地显示.

当使用 addChild()方法将组件添加到容器中时,大部分工作都是为了设置这个组件.这是因为直到把组件添加到容器中时,Flex 才能确定它的大小(size),设置它所继承样式(style)属性,或者在屏幕上画出它.

也可以用 MXML，在应用中完成上面的组件添加动作,如下面的例子所示:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Box>
        <mx:Button label="Submit"/>
    </mx:Box>
</mx:Application>
```

在 MXML 中创建组件时，Flex 执行的步骤顺序同用 ActionScript 的步骤顺序相同.

使用 removeChild() 方法可将组件从容器中移出,如果对这个组件再没有其他的引用,那就相当于使用 Adobe® Flash® Player 或 Adobe® AIR™的垃圾回收机制将组件从内存中删除.

关于创建组件的步骤

为了实现一个组件,就要重载组件的方法,定义新的属性,分发新的事件,或者执行其他任何应用所需的自定义的逻辑.要想实现自己的组件, 请遵循以下这些常规步骤:

1. 如果有必要，为组件创建所有的皮肤（skins）.
2. 创建 ActionScript 类文件.
   a. 从一个基类扩展，比如 UIComponent 或者其他的组件类.
   b. 指定使用者能够通过 MXML 标记进行设置的属性.
   c. 嵌入（Embed）所有的图片和皮肤文件.
   d. 实现构造器.
   e. 实现 UIComponent.createChildren()方法.
   f. 实现 UIComponent.commitProperties() 方法.
   g. 实现 UIComponent.measure() 方法.
   h. 实现 UIComponent.layoutChrome() 方法.
   i. 实现 UIComponent.updateDisplayList() 方法.
   j. 增加属性(properties) ，方法(methods),样式(styles),事件(events)以及元数据.
3. 以 ActionScript 文件或者 SWC 文件的形式部署组件.

更多关于 MXML 标记属性和嵌入图形以及皮肤文件的信息参见 Simple Visual Components in ActionScript

定义新组件时不一定要重载所有的方法. 只需要重载实现组件功能所需的方法. 如果创建现存组件的子类，比如 Button 控件或者 VBox 容器，那么必须实现组件中所添加的新功能所需的方法.

例如：实现一个自定义的 Button 控件，该控件使用新的机制来定义缺省大小（size）. 在这种情况下，只需要重载 measure()方法. 对于这个例子，参见实现 easure() 方法.

或者，要实现 VBox 容器的一个新子类. 新子类利用 VBox 类已有的所有有关设定大小（sizing）的逻辑，但是变更了 VBox 类的布局逻辑以实现从底部到顶部的方式来布局容器中的子控件，而不是自顶向下的布局. 在这种情况下，只需要重载 updateDisplayList()方法. 关于这个例子，见实现 updateDisplayList()方法.

关于接口

Flex 用接口将组件的基本功能分为离散的单元,这使得它们可以被“一块一块”地实现.例如，使组件可聚焦（有输入焦点），它必须实现 IFocusable 接口；要使它能够参与布局过程，它必须实现 ILayoutClient 接口.

为了简化接口的使用， UIComponent 类实现了下表中定义的所有接口，但是不包括 IFocusManagerComponent 接口和 IToolTipManagerClient 接口. 但 UIComponent 很多子类实现了 IFocusManagerComponent 和 IToolTipManagerClient 接口.

因此，创建 UIComponent 或其子类的子类，就不必实现这些接口. 但是，如果要创建的组件不是 UIComponent 的子类，并且还想在 Flex 中使用这个组件，那么就必须实现一个或者多个接口。

注意：对于 Flex, Adobe 建议所有的组件都应从 UIComponent 及其子类继承.

下表列出了 Flex components 所实现的主要接口:

接口	用途
IChildList	表明容器中子控件的数量.
IDeferredInstantiationUIComponent	表明组件或组件能实施延迟实例化.
IFlexDisplayObject	指定皮肤元素的接口.
IFocusManagerComponent	表明组件或者对象是可聚焦的,这意味着组件可以从 FocusManager 获取焦点. UIComponent 组件没有实现 Ifocusable 接口,因为有些组件不打算接受焦点.
IInvalidating	表明组件或者对象能够使用”失效机制”去执行延迟的,而不是立即的属性提交(commitment), 度量(measurement), 以及画出(drawing) 或布局(layout).
ILayoutManagerClient	表明组件或者对象能够参与 LayoutManager 的提交(commit) ,度量(measure)和更新次序(update

	sequence)
IPropertyChangeNotifier	表明组件支持特定形式的事件传播.
IRepeaterClient	表明组件或者对象能够被用于 Repeater 类.
IStyleClient	表明组件能够从其他的对象继承样式,并且支持 setStyle() 和 getStyle() 方法.
IToolTipManagerClient	表明组件有一个 toolTip 属性,并且因此可以被 ToolTipManager 所监控.
IUIComponent	定义组件能够成为布局管理器或列表的子组件所必须实现的基本 APIs.
IValidatorListener	表明组件能够监听校验(validation)事件,并且因此能够显示校验状态,比如一个红色边框和显示校验错误的提示信息(tooltips)

2. 组件的实现

在用 ActionScript 创建自定义组件时,必须重载 UIComponent 类的一些方法.实现基本的组件结构,构造器,以及 createChildren(),commitProperties(), measure(), layoutChrome()和 updateDisplayList()方法.

基本组件结构

下面例子展示了 Flex 组件的基本结构:

```
package myComponents
{
public class MyComponent extends UIComponent
{
....
}
}
```

必需在包中定义 ActionScript 自定义组件。包能够反映自定义组件在应用的路径结构中的位置.

自定义组件的类定义必须以 public 关键字修饰. 尽管包含类的定义文件中可能还有其他内部类定义,但是,该文件中有且只能有一个 public 类定义.要将所有的内部类定义放在包定义的关闭大括号之下的源文件的底部.

实现构造函器

用 ActionScript 写的 UIComponent 类或其子类的子类，应该定义 public 构造器方法.这里的构造器有以下特点:

- 没有返回类型.
- 应被声明为 public
- 没有参数
- 调用 super()方法以使用父类的构造器.

每个类只能包含一个构造器方法； ActionScript 不支持重载(overloaded) 的构造方法.更多信息参见 Defining the constructor.使用构造器可以设置类属性的初始值,比如,可以设置

属性和样式的缺省值,或者初始化数据结构,比如数组.

不要在构造器中创建“子显示对象”，构造器只应用于设置组件的初始值。如果组件要创建子组件，那么可在 createChildren()方法中创建。

实现 createChildren() 方法

在内部创建其他组件或可视化对象的组件被称为“复合组件（*composite componen*)”。例如，Flex ComboBox 控件包含一个用于定义 ComboBox 文本区的 TextInput 控件和一个用于定义 ComboBox 向下箭头的 Button 控件。组件实现 createChildren()方法，在其内部创建子对象（比如其他的组件)。

应用开发者不要直接调用 createChildren()方法；当开发者调用 addChild()方法将组件添加到父组件中时，Flex 会自动调用 createChildren()方法。注意，createChildren()没有与之相关的失效方法，这意味着组件被添加到父组件中时不会等上一会才调用这个方法。

例如，要定义一个新的组件，这个组件包含一个 Button 控件和一个 TextArea 控件，这里的 Button 控件用于控制用户是否能向 TextArea 控件中输入信息。下面的例子创建了 TextArea 和 Button 控件：

```
// Declare two variables for the component children.
private var text_mc:TextArea;
private var mode_mc:Button;

override protected function createChildren():void {

    // 调用父类的 createChildren()方法.
    super.createChildren();
    // 在创建子组件之前检查这些子组件是否已存在
    // 这是个可选项，但是这样做使得子类可以创建一个不同的子组件
    if (!text_mc)     {
        text_mc = new TextArea();
        text_mc.explicitWidth = 80;
        text_mc.editable = false;
        text_mc.addEventListener("change", handleChangeEvent);
        // 将子组件添加到自定义组件中
```

```
            addChild(text_mc);
        }

        //在创建子组件之前检查这些子组件是否已存在.
        if (!mode_mc)      {
            mode_mc = new Button();
            mode_mc.label = "Toggle Editing";
            mode_mc.addEventListener("click", handleClickEvent);
            //将子组件添加到自定义组件中
            addChild(mode_mc);
        }
}
```

注意，在这个例子中，createChildren()方法调用 addChild()来添加子组件。必须对每个子对象调用 addChild()方法。在创建子对象之后，就能使用子对象的属性来定义子对象的特性。在这个例子中，我们创建了 Button 和 TextArea 控件，初始化它们，然后为它们注册事件监听器。当然，也可以给子组件添加皮肤，更完整的例子参见：例子:创建一个复合组件.

实现 commitProperties() 方法

使用 commitProperties()方法来协调对组件属性的更改。绝大多数情况下，都是对影响组件如何在屏幕上显示的属性使用这个方法。

当 invalidateProperties()方法调用时，Flex 会“安排（schedules）”一个对 commitProperties()方法的调用（这里的“安排（schedules）”指的不是立即执行）。commitProperties()方法在 invalidateProperties()方法调用之后的下一个“渲染事件(render event)”中被执行。当使用 addChild()方法向容器中添加一个组件时，Flex 会自动调用 invalidateProperties()方法。

commitProperties()方法的调用发生在 measure()方法调用之前，这让我们能够设置 measure()方法可能使用的属性值。

定义组件属性的典型模式就是用 getter 和 setter 方法来定义属性，如下面的例子所示：

```
// 为 alignText 属性定义个一个 private 变量。
private var _alignText:String = "right";
// 定义个一个标志来表明_alignText 属性是否发生了变化
```

```
private var bAlignTextChanged:Boolean = false;

// 为属性定义 getter 和 setter 方法
public function get alignText():String {
        return _alignText;
}
public function set alignText(t:String):void {
    _alignText = t;
    bAlignTextChanged = true;
   // 在需要的时候，触发 commitProperties(), measure(),和 updateDisplayList() 方法
   // 在本例的中，不需要去重新度量（remeasure）组件
    invalidateProperties();
    invalidateDisplayList();
}
// 实现 commitProperties() 方法.
override protected function commitProperties():void {
    super.commitProperties();

    // 检查 flag 是否带表 alignText 属性已经变化。
    if (bAlignTextChanged) {
        // Reset flag.
        bAlignTextChanged = false;
        //处理 alignment 变化
        ......
        ......
    }
}
```

正如这个例子中看到的那样，setter 方法更改了属性，调用 invalidateProperties() 和 invalidateDisplayList()方法，然后返回。Setter 方法本身不执行任何基于新属性值的计算。这种设计让 setter 方法能迅速地返回，并把对新属性值的处理留给 commitProperties()方法。

改变控件中的文本（text）对齐（alignment）方式不需要改变控件的大小。但是，一旦

改变了控件的大小，就要在代码加入对 invalidateSize()方法的调用，以触发 measure()方法。

使用 commitProperties()方法的优点如下：

●能协调对多个属性的修改，使得这些变更能够同时生效。

例如，可能定义多个属性来控件组件文本的显示，比如，文本在组件内部的对齐（alignment）属性。Text 或者 alignment 属性的变化都需要 Flex 去更新组件的显示。但是，如果 text 和 alignment 都被改变了，在屏幕更新时，你会希望 Flex 能够一次性地执行所有的有关大小和位置的计算。

因此，需要使用 commitProperties()方法来计算所有与其它属性相关的属性值。通过 commitProperties ()方法来协调属性的变更，可以减少不必要的重复处理。

●能够协对同一个属性的多次修改。

这样就不必每次更新组件的一个属性都执行复杂的计算。比如，用户更改 Button 控件的 icon 属性以更改 Button 控件上显示的图片。根据 icon 的大小或百分比计算 label 的位置是一个开销较大的操作，这样的操作应只在必要时执行一次。

为了避免这样的行为,要使用 commitProperties()方法去执行计算。当更新显示时 Flex 会调用 commitProperties()方法。这意味着不论两次屏幕更新之间属性曾经变化了多少次，Flex 只在屏幕更新时执行一次计算。

下面的例子显示在 commitProperties()方法中如何处理两个相关属性：

```
//为 text 属性定义一个 private 变量.
private var _text:String = "ModalText";
private var bTextChanged:Boolean = false;

//定义 getter 方法.
public function get text():String {
        return _text;
}

//定义 setter 方法以便在属性变化时调用 invalidateProperties()
public function set text(t:String):void {
    _text = t;
    bTextChanged = true;
    invalidateProperties();
    // 改变 text 属性导致控件重新计算缺省大小
```

```
    invalidateSize();
    invalidateDisplayList();
}
//为 alignText 属性定义一个 private 变量。
private var _alignText:String = "right";
private var bAlignTextChanged:Boolean = false;

public function get alignText():String
{
        return _alignText;
}

public function set alignText(t:String):void {
    _alignText = t;
    bAlignTextChanged = true;
    invalidateProperties();
    invalidateDisplayList();
}
// 实现 commitProperties() 方法.
override protected function commitProperties():void {
    super.commitProperties();

    //检查两个属性是否发生变化的标志
    if (bTextChanged && bAlignTextChanged)
    {
        //重置标志
        bTextChanged = false;
        bAlignTextChanged = false;

        //处理两个属性都发生变化的情况
    }
```

```
    //判断是否 text 属性发生变化
    if (bTextChanged) {
        // 重置属性。
        bTextChanged = false;

        // 处理 text 属性的变化。
    }
    // 检查 alignText 属性是否变化
    if (bAlignTextChanged) {
        //重置属性.
        bAlignTextChanged = false;

        // 处理 alignment 属性的变化.
    }
}
```

实现 measure() 方法

measure()方法设置组件的缺省大小，以像素为单位，并且也可以有选择性地设置组件其他属性的缺省值。

当 invalidateSize()方法的调用发生后， Flex 会“安排”一个对 measure()方法的调用。measure()方法在 invalidateSize()调用之后的下一个“渲染事件（render event)”时执行。当使用 addChild()方法将组件添加到容器中时，Flex 会自动调用 invalidateSize()方法。

当为组件设置特定的高和宽后，尽管显示地调用 invalidateSize()方法，但 Flex 不会调用 measure()方法。也就是说，只有当组件的 explicitWidth 和 explicitHeight 属性是 NaN 时 Flex 调用 measure()方法。

在下面的例子中，由于已经显式地设置了 Button 控件的大小，Flex 不会调用 Button.measure()方法：

```
<mx:Button height="10" width="10"/>
```

在已有组件的子类中，只有当正在执行的动作需要更改父类中定义的组件大小设定规则时，才会实现 measure()。因此，要设置一个新的缺省值，或者在运行时执行计算以确定组件大小的规则，就要实现 measure()方法。

在 measure()方法中设置以下有关组建大小的缺省：

属性	描述
measuredHeight measuredWidth	以像素为单位设定组件的缺省高度和宽度。 这些属性被设置为 0，直到 measure()方法被执行。使它们设置为 0，使得组件在缺省情况下不可见。
measuredMinHeight measuredMinWidth	指定组件缺省的最小高度和最小宽度，以像素为单位。Flex 不能将组件的大小设置为比指定的最小值还小。

measure()只设置组件的缺省大小。在 updateDisplayList()方法中，组件的父容器将其实际大小传递给组件，这些属性值与缺省值不同。

组件开发者在应用中用以下列方式也能重载组件的缺省大小：

- 设置 explicitHeight 和 exlicitWidth 属性。
- 设置 width 和 height 属性。
- 设置 percentHeight 和 percentWidth 属性。

例如，定义一个 Button 控件，其缺省的大小为 100 像素宽,50 像素高，并且缺省的最小值为 50 像素宽，25 像素高，如下例所示：

```
package myComponents
{
    // asAdvanced/myComponents/DeleteTextArea.as
    import mx.controls.Button;

    public class BlueButton extends Button {

        public function BlueButton() {
            super();
        }

        override protected function measure():void {
            super.measure();

            measuredWidth=100;
            measuredMinWidth=50;
```

```
            measuredHeight=50;
            measuredMinHeight=25;
        }
    }
}
```

下面的应用中使用了这个 button。

```
<?xml version="1.0"?>
<!-- asAdvanced/ASAdvancedMainBlueButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

    <mx:VBox>
        <MyComp:BlueButton/>
        <mx:Button/>
    </mx:VBox>
</mx:Application>
```

Button 上没有设置任何其它有关 button 大小的约束，VBox 使用 button 的缺省大小，和缺省的最小大小来计算 VBox 在运行时的大小。更多关于设置组件大小的规则，见 Introducing Containers。也可以在应用中重载缺省的大小设置：

```
<?xml version="1.0"?>
<!-- asAdvanced/MainBlueButtonResize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >
    <mx:VBox>
        <MyComp:BlueButton width="50%"/>
        <mx:Button/>
    </mx:VBox>
</mx:Application>
```

在这个例子中，制定 button 的宽度是 VBox 容器宽度的 50%。当容器宽度的 50%小于 button

的最小宽度时，button 使用它的最小宽度。

计算缺省大小

上例子中，实现 measure()方法时用静态的值设置缺省的大小和缺省的最小大小。一些，Flex 组件使用了静态大小，比如 TextArea 的静态大小为 100 像素宽，44 像素高，而不管它所包含的文本什么样。如果文本比 TextArea 控件大，控件就显示滚动条。（***译者注:这里的静态应该指的是绝对布局下的表示像素个数的高宽值，而不是百分比***）.

通常，根据组件特点或者传递给该组件的信息来设置它的缺省大小。比如，Button 控件的 measure()检查它的标签文本，补白（margin）以及字体的特性来决定组件的缺省大小。

在下面的例子中，重载了 TextArea 控件的 measure()方法，这样它就能够检测传递给控件的文本，以及计算 TextArea 控件的缺省大小，以使它能在一行中显示整个文本字符串：

```
package myComponents
{
    // asAdvanced/myComponents/MyTextArea.as
    import mx.controls.TextArea;
    import flash.text.TextLineMetrics;

    public class MyTextArea extends TextArea
    {

        public function MyTextArea() {
            super();
        }

        // The default size is the size of the text plus a 10 pixel margin.
        override protected function measure():void {
            super.measure();

            // Calculate the default size of the control based on the
            // contents of the TextArea.text property.
            var lineMetrics:TextLineMetrics = measureText(text);
```

```
            // Add a 10 pixel border area around the text.
            measuredWidth = measuredMinWidth = lineMetrics.width + 10;
            measuredHeight = measuredMinHeight = lineMetrics.height + 10;
        }
    }
}
```

当 text 字符串长度超过应用的显示区域，通过增加逻辑来增长 TextArea 控件的高度，使文本(text) 能在多行显示。下面应用使用了这个组件：

```
<?xml version="1.0"?>
<!-- asAdvanced/MainMyTextArea.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

    <MyComp:MyTextArea id="myTA" text="This is a long text strring that would normally cause a TextArea control to display scroll bars. But, the custom MyTextArea control calcualtes its default size based on the text size."/>

    <mx:TextArea id="flexTA" text="This is a long text strring that would normally cause a TextArea control to display scroll bars. But, the custom MyTextArea control calcualtes its default size based on the text size."/>

</mx:Application>
```

实现 layoutChrome() 方法

Container 类，以及 Container 类的子类，使用 layoutChrome()方法来定义容器区域的边框（border area)。

当 invalidateDisplayList()方法调用发生时，Flex 将“安排”一个 layoutChrome()方法的调用。layoutChrome()方法在 invalidateDisplayList()方法调用之后的下一个“渲染事件”期间执行。当使用 addChild()方法将一个组件添加到容器中时，Flex 自动调用 invalidateDisplayList()方法。

通常，使用 RectangularBorder 类来定义容器区域的边框。比如，可以创建一个 RectangularBorder 对象，然后在重载的 createChildren()方中,将其作为一个子控件添加到组件中。

当创建容器类的子类时，可以使用 createChildren()方法去创建容器的“内容子控件”。“内容子控件”是指在容器中显示的子组件。用 updateDisplayList()方法来确定“内容子控件”的位置。

使用 layoutChrome()方法通常是用于定义容器的边框区域和确定边框区域的位置，以及确定要在边框区域中显示的附加元素。例如，Panel 容器使用 layoutChrome()方法定义 panel 容器的 title 区域，这个区域用来包含 title 文本和 close 按钮。

将容器的内容区域和容器边框区域分开处理的主要原因是为了处理 Container.autoLayout 属性被设置为 false 的这种情况。当 autoLayout（自动布局）属性被设置为 true 的时候，只要容器子控件的大小和位置发生变化，容器及其子控件就会进行度量和布局。缺省值为 true。

当 autoLayout 属性被设置为 false 的时候，度量和布局只在子控件被添加到容器中或者从容器中移出时执行。然而，Flex 在两种情况下都执行 layoutChrome()。所以，就算在 autoLayout 属性被设置为 false 的情况下，容器仍然能够更新它的边框区域。

实现 updateDisplayList() 方法

updateDisplayList()方法按照前面被调用的方法中设定的属性和样式来设定**子组件**的大小和位置，并画出组件所使用的皮肤和图片元素。而组件本身的大小由组件的父容器决定。

直到组件的 updateDisplayList()方法被调用之后，组件才能在屏幕上显示出来。当 invalidateDisplayList()方法调用发生时，Flex 会“安排”一个对 updateDisplayList()方法的调用。updateDisplayList()方法在 invalidateDisplayList()方法调用之后的下一个“渲染事件”发生时才会被执行。当使用 addChild()方法将组件添加到容器中时，Flex 会自动调用 invalidateDisplayList()方法。

updateDisplayList()方法的主要用途如下：

■用于设置组件中元素的大小和位置，以用于组件的显示。

很多组件由一个或者多个子组件组成，或者有若干属性用于控制组件中信息的显示。比如，Button 控件有一个可选的 icon，并且，使用 labelPlacement 属性可以指定按钮上的文字在相对于 icon 的什么地方显示（左侧还是右侧）。

Button.updateDisplayList()方法使用 icon 和 labelPlacement 属性值来控制 button 的显示。

对于有子组件的容器来说，updateDisplayList()方法控制那些子组件该如何确定位

置。比如，Hbox 容器的 updateDisplayList()方法在一行上按照从左到右的循序确定子组件的位置。VBox 容器的 updateDisplayList()方法在一列上按照从上到下的顺序确定子组件的位置。

要在 updateDisplayList()方法中确定一个组件的大小，应当使用 setActualSize()方法，而不是使用与组件大小相关的属性，诸如 width 和 height。要确定组件的位置，应当使用 move()方法，而不是 x 和 y 属性。

■用于画出组件所需的所有可视元素。

组件支持很多类型的可视元素，比如皮肤，样式，和边框。在 updateDisplayList()方法中，可以添加这些可视元素，使用 Flash 绘画 APIs,以及对组件中这些可视化的显示执行另外一些控制。

updateDisplayList()方法形式如下：

```
protected function updateDisplayList(unscaledWidth:Number,
    unscaledHeight:Number):void
```

属性有以下值：

unscaledWidth

指定组件的宽度，以像素为单位，在组件的坐标系中，不管组件的 scaleX 属性值是多少。这个值就是由父容器所确定的组件宽度。

unscaledHeight

指定组件的高度，以像素为单位，在组件的坐标系中。不管组件的 scaleY 属性值是多少。这个值就是由父容器所确定的组件高度。

缩放发生在 Flash Player 或者 AIR 中，发生时机是在 updateDisplayList()执行之后。比如，一个组件的 unscaledHeight 属性是 100，而其 scaleY 属性是 2.0,那么它在 Flash Player 或 AIR 中出现的高度为 200 像素。

VBox 容器对布局机制的重载（override）：

VBox 容器按照子组件加入到容器中的先后顺序将子组件按照从上到下的方式进行布局。下面的例子重载了它的 updateDisplayList()方法，这个方法使 VBox 容器按照从底向上的方式进行布局：

```
package myComponents
{
    // asAdvanced/myComponents/BottomUpVBox.as
```

```
import mx.containers.VBox;
import mx.core.EdgeMetrics;
import mx.core.UIComponent;
public class BottomUpVBox extends VBox
{

    public function BottomUpVBox() {
        super();
    }

    override protected function updateDisplayList(unscaledWidth:Number,
        unscaledHeight:Number):void {
        super.updateDisplayList(unscaledWidth, unscaledHeight);
        // Get information about the container border area.
        // The usable area of the container for its children is the
        // container size, minus any border areas.
        var vm:EdgeMetrics = viewMetricsAndPadding;

        // Get the setting for the vertical gap between children.
        var gap:Number = getStyle("verticalGap");

        // Determine the y coordinate of the bottom of the usable area
        // of the VBox.
        var yOfComp:Number = unscaledHeight-vm.bottom;

        // Temp variable for a container child.
        var obj:UIComponent;

        for (var i:int = 0; i < numChildren; i++)
        {
            // Get the first container child.
```

```
                obj = UIComponent(getChildAt(i));

                // Determine the y coordinate of the child.
                yOfComp = yOfComp - obj.height;

                // Set the x and y coordinate of the child.
                // Note that you do not change the x coordinate.
                obj.move(obj.x, yOfComp);

                // Save the y coordinate of the child,
                // plus the vertical gap between children.
                // This is used to calculate the coordinate
                // of the next child.
                yOfComp = yOfComp - gap;
            }
        }
    }
}
```

在这个例子中，使用 UIComponent.move()方法设置容器中每个子控件的位置。也可以用 UIComponent.x 及 UIComponent.y 属性去设置这些坐标。区别就是，move()方法不仅改变组件的位置，而且在调用这个方法之后**立即分发**了一个 move 事件，设置 x 和 y 属性也更改组件的位置，但却在下一个屏幕更新事件中分发 move 事件。

下面的应用使用了这个组件：

```
<?xml version="1.0"?>
<!-- asAdvanced/MainBottomVBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

  <MyComp:BottomUpVBox>
    <mx:Label text="Label 1"/>
    <mx:Button label="Button 1"/>
```

```
    <mx:Label text="Label 2"/>
    <mx:Button label="Button 2"/>

    <mx:Label text="Label 3"/>
    <mx:Button label="Button 3"/>
    <mx:Label text="Label 4"/>
    <mx:Button label="Button 4"/>
  </MyComp:BottomUpVBox>
</mx:Application>
```

在组件中画图

每个 Flex 组件都是 Flash Sprite 类的子类，并因此而继承了 Sprite.graphics 属性。Sprite.graphics 属性所指定的 Graphics 对象可以用来向组件中添加矢量绘画（vector drawings）。

例如，在 updateDisplayList()方法中，可以使用 Graphics 类去画出边框和水平线以及其他图形元素：

```
override protected function updateDisplayList(unscaledWidth:Number,
unscaledHeight:Number):void
{
    super.updateDisplayList(unscaledWidth, unscaledHeight);
        // Draw a simple border around the child components.
    graphics.lineStyle(1, 0x000000, 1.0);
    graphics.drawRect(0, 0, unscaledWidth, unscaledHeight);
```

3. 使组件具备可访问性

Web 内容日益增长的一个需求就是让那些残障人士能够访问 WEB。视力有问题的人可以通过屏幕阅读软件来使用 Flash 应用中的可视内容，屏幕阅读软件能够对屏幕的内容提供语音描述.

当创建一个组件时,可以引入 ActionScript 使组件和屏幕阅读器可以通过语音进行通信.当开发者使用组件来构建运行在 Flash 中的应用时,他们使用 Accessibility 面板来配置每个组件实例.

Flash 包括下面这些“可访问性”方面的特性:

- 自定义的焦点导航.
- 自定义键盘快捷键.
- 基于屏幕文挡和屏幕制作环境.
- 一个具备可访问能力的类.

想要使组件具备可访问能力(accessibility) ,在组件类文件中添加以下代码行:

```
mx.accessibility.ComponentName.enableAccessibility();
```

例如,下面的代码行使 MyButton 组件具备了可访问能力:

```
mx.accessibility.MyButton.enableAccessibility();
```

关于可访问性的其他信息,参见 Creating Accessible Applications.

4. 为组件增加版本号

当发行组件时，可以定义一个版本号。版本号可让开发者知道他们是否需要更新组件，并为解决技术问题提供帮助。要为组件设置一个版本号，使用静态变量 version，如下例所示：

```
mx.accessibility. static var version:String = "1.0.0.42";
```

注意：Flex 不使用也不解释这个版本号的值。

如果在一个组件包中创建了很多组件，可以在一个外部文件中包含版本号。使用这种方式，只需要在一个地方更新版本号。例如，下面代码引入了一个外部文件的内容，这个外部文件在一个地方了存储版本号：

```
include "../myPackage/ComponentVersion.as"
```

ComponentVersion.as 文件的内容和前面的变量声明类似，如下例所示：

```
static var version:String = "1.0.0.42";
```

5. 组件设计的最佳实践

在设计组件时，使用以下实践：

- 让文件大小尽可能地小。
- 通过提供通用的功能让组件实现最大的重用性。
- 使用 Border 类而不是图片资源去画包围组件四周的边框。
- 使用基于标记（tag-based）的皮肤。

假定组件有一个初始的状态（state），由于 style 属性是在对象上，所以可以为 styles 和属性设置初始值，这样，不用在对象构造的初始化代码中设置他们，除非用户重载了缺省的状态。

译者注：这里的状态(state)不是普通意义上的组件的数据，而是 Flex 框架中的一个类。

6. 例子：创建一个复合组件

*复合组件*是指包含多个组件的组件。这些组件可以是图形资源或者是图形资源同组件类的合并。比如，创建包括按钮和文本域（text field）的组件，或者，组件包含一个按钮，一个文本域以及一个校验器（validator）。

在创建复合组件时，应当在组件类的内部实例化这些控件。假设这些控件有图形资源，那就必须在类文件中规划所引入的控件的布局，以及设置一些属性，诸如缺省值之类。必须确保引入符合组件所需的所有类。

如果组件类是从比较基础的类扩展，比如 UIComponent，而不是从像 Button 这样的控件类扩展的，那么就必须实例化自定义组件中的每个子控件，并管理它们在屏幕上的显示。

复合组件中单个组件的属性不能从 MXML 制作环境中访问，除非设计上允许这么做。例如，如果创建一个从 UIComponent 扩展的组件，并在这个组件中使用了一个 Button 和一个 TextArea 组件，那么就不能在 MXML 标记中设置 Button 控件的 label 文本，原因就是这个组件不是直接从 Button 类扩展的。

创建组件

这个例子组件被称为 ModalText，在 ModalText.as 文件中定义，组合了一个 Button 控件和一个 TextArea 控件。用 Button 控件来控制是否允许在 TextArea 控件中进行输入。

为复合组件定义时间监听器

自定义组件通过实现 createChildren()方法来创建子组件，如下面代码所示：

```
override protected function createChildren():void {
    super.createChildren();

    // Create and initialize the TextArea control.
    if (!text_mc)     {
        text_mc = new TextArea();
        ...
        text_mc.addEventListener("change", handleChangeEvent);
        addChild(text_mc);
```

```
    }

    // Create and initialize the Button control.
    if (!mode_mc)     {
        mode_mc = new Button();
        ...
        mode_mc.addEventListener("click", handleClickEvent);
        addChild(mode_mc);
    }
}
```

createChildren()方法中也包含对 addEventListener()方法的调用，用这个方法为 TextArea 控件所产生的 change 事件以及 Button 控件所产生的 click 事件注册监听器。这些事件监听器定义在 ModalText 类中，如下面例子所示：

```
// Handle events that are dispatched by the children.
private function handleChangeEvent(eventObj:Event):void {
        dispatchEvent(new Event("change"));
}

// Handle events that are dispatched by the children.
private function handleClickEvent(eventObj:Event):void {
        text_mc.editable = !text_mc.editable;
}
```

可在复合组件中处理子组件所分发的事件。在这个例子中，Button 控件的 click 事件的事件监听器被定义在复合组件类中，用来控制 TextArea 控件的 editable 属性。

但是，如果子组件分发了一个事件，并且希望在组件之外可以有时机来处理这个事件，那么就要在组件定义中增加逻辑来广播这个事件。注意，TextArea 控件的 change 事件监听器中重新广播了这个事件。这就使运用这个组件的应用也能处理这个事件，如下面的例子所示：

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns:MyComp="myComponents.*">

    <mx:Script>
        <![CDATA[
```

```
            import flash.events.Event;

            function handleText(eventObj:Event)
            {
                ...
            }
        ]]>
    </mx:Script>

    <MyComp:ModalText change="handleText(event);"/>
</mx:Application>
```

创建 ModalText 组件

下面的例子代码实现了 ModalText 组件的类定义。

ModalText 组件是一个包含一个 Button 控件和一个 TextArea 控件的复合组件。这个控件有以下特性：

- 缺省情况下不能在 TextArea 中进行编辑。
- 通过点击 Button 控件去控制 TextArea 控件的编辑。
- 通过使用控件的 textPlacement 属性来控制 TextArea 显示在控件的左边还是右边。
- 编辑控件的 textPlacement 属性将会分发 placementChanged 事件。
- 在程序中使用 text 属性可以向 TextArea 控件写入内容。
- 编辑控件的 text 属性将会分发一个 textChanged 事件。
- 编辑 TextArea 控件的文本将会分发 change 事件。
- textPlacement 和 text 属性都可以作为数据绑定表达式的源。
- 可以有选择性地为 Button 的 up, down 和 over 状态设置皮肤。

下面的例子在 MXML 文件中使用 ModalText 控件，并且将其 textPlacement 属性设置为 left：

```
<?xml version="1.0"?>
<!-- asAdvanced/ASAdvancedMainModalText.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
```

```
    xmlns:MyComp="myComponents.*" >

    <MyComp:ModalText textPlacement="left" height="40"/>

</mx:Application>
```

通过处理 placementChanged 事件，来确定 ModalText.textPlacement 属性何时被修改，如下面的例子所示：

```
<?xml version="1.0"?>
<!-- asAdvanced/ASAdvancedMainModalTextEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*" >

    <mx:Script>
      <![CDATA[
        import flash.events.Event;

        private function placementChangedListener(event:Event):void {
          myEvent.text="placementChanged event occurred - textPlacement = "
              + myMT.textPlacement as String;
        }
      ]]>
    </mx:Script>

    <MyComp:ModalText id="myMT"
        textPlacement="left"
        height="40"
        placementChanged="placementChangedListener(event);"/>
    <mx:TextArea id="myEvent" width="50%"/>

    <mx:Label text="Change Placement" />
    <mx:Button label="Set Text Placement Right"
        click="myMT.textPlacement='right';" />
    <mx:Button label="Set Text Placement Left"
        click="myMT.textPlacement='left';" />
</mx:Application>
```

下面的例子显示定义控件的 ModalText.as 文件源码：

```
package myComponents
{
    // Import all necessary classes.
    import mx.core.UIComponent;
    import mx.controls.Button;
    import mx.controls.TextArea;
    import flash.events.Event;
    import flash.text.TextLineMetrics;

    // ModalText dispatches a change event when the text of the child
    // TextArea control changes, a textChanged event when you set the text
    // property of ModalText, and a placementChanged event
    // when you change the textPlacement property of ModalText.
    [Event(name="change", type="flash.events.Event")]
    [Event(name="textChanged", type="flash.events.Event")]
    [Event(name="placementChanged", type="flash.events.Event")]

    /*** a) Extend UIComponent. ***/
    public class ModalText extends UIComponent {

        /*** b) Implement the class constructor. ***/
        public function ModalText() {
            super();
        }


        /*** c) Define variables for the two child components. ***/
        // Declare two variables for the component children.
        private var text_mc:TextArea;
        private var mode_mc:Button;


        /*** d) Embed new skins used by the Button component. ***/
        // You can create a SWF file that contains symbols  with the names
        // ModalUpSkin, ModalOverSkin, and ModalDownSkin.
        // If you do not have skins, comment out these lines.
        [Embed(source="Modal2.swf", symbol="blueCircle")]
```

```
        public var modeUpSkinName:Class;

        [Embed(source="Modal2.swf", symbol="blueCircle")]
        public var modeOverSkinName:Class;

        [Embed(source="Modal2.swf", symbol="greenSquare")]
        public var modeDownSkinName:Class;


        /*** e) Implement the createChildren() method. ***/
        // Test for the existence of the children before creating them.
        // This is optional, but we do this so a subclass can create a
        // different child instead.
        override protected function createChildren():void {
            super.createChildren();

            // Create and initialize the TextArea control.
            if (!text_mc)
            {
                text_mc = new TextArea();
                text_mc.explicitWidth = 80;
                text_mc.editable = false;
                text_mc.text= _text;
                text_mc.addEventListener("change", handleChangeEvent);
                addChild(text_mc);
            }

            // Create and initialize the Button control.
            if (!mode_mc)
            {   mode_mc = new Button();
                mode_mc.label = "Toggle Editing Mode";
                // If you do not have skins available,
                // comment out these lines.
                mode_mc.setStyle('overSkin', modeOverSkinName);
                mode_mc.setStyle('upSkin', modeUpSkinName);
                mode_mc.setStyle('downSkin', modeDownSkinName);
                mode_mc.addEventListener("click", handleClickEvent);
                addChild(mode_mc);
```

```
        }
    }


    /*** f) Implement the commitProperties() method. ***/
    override protected function commitProperties():void {
        super.commitProperties();

        if (bTextChanged) {
            bTextChanged = false;
            text_mc.text = _text;
            invalidateDisplayList();
        }
    }


    /*** g) Implement the measure() method. ***/
    // The default width is the size of the text plus the button.
    // The height is dictated by the button.
    override protected function measure():void {
        super.measure();

        // Since the Button control uses skins, get the
        // measured size of the Button control.
        var buttonWidth:Number = mode_mc.getExplicitOrMeasuredWidth();
        var buttonHeight:Number = mode_mc.getExplicitOrMeasuredHeight();

        // The default and minimum width are the measuredWidth
        // of the TextArea control plus the measuredWidth
        // of the Button control.
        measuredWidth = measuredMinWidth =
            text_mc.measuredWidth + buttonWidth;

        // The default and minimum height are the larger of the
        // height of the TextArea control or the measuredHeight of the
        // Button control, plus a 10 pixel border around the text.
        measuredHeight = measuredMinHeight =
            Math.max(mode_mc.measuredHeight,buttonHeight) + 10;
```

```
        }


        /*** h) Implement the updateDisplayList() method. ***/
        // Size the Button control to the size of its label text
        // plus a 10 pixel border area.
        // Size the TextArea to the remaining area of the component.
        // Place the children depending on the setting of
        // the textPlacement property.
        override protected function updateDisplayList(unscaledWidth:Number,
                unscaledHeight:Number):void {
            super.updateDisplayList(unscaledWidth, unscaledHeight);

            // Subtract 1 pixel for the left and right border,
            // and use a 3 pixel margin on left and right.
            var usableWidth:Number = unscaledWidth - 8;

            // Subtract 1 pixel for the top and bottom border,
            // and use a 3 pixel margin on top and bottom.
            var usableHeight:Number = unscaledHeight - 8;

            // Calculate the size of the Button control based on its text.
            var lineMetrics:TextLineMetrics = measureText(mode_mc.label);
            // Add a 10 pixel border area around the text.
            var buttonWidth:Number = lineMetrics.width + 10;
            var buttonHeight:Number = lineMetrics.height + 10;
            mode_mc.setActualSize(buttonWidth, buttonHeight);

            // Calculate the size of the text
            // Allow for a 5 pixel gap between the Button
            // and the TextArea controls.
            var textWidth:Number = usableWidth - buttonWidth - 5;
            var textHeight:Number = usableHeight;
            text_mc.setActualSize(textWidth, textHeight);

            // Position the controls based on the textPlacement property.
            if (textPlacement == "left") {
                text_mc.move(4, 4);
```

```
            mode_mc.move(4 + textWidth + 5, 4);
        }
        else {
            mode_mc.move(4, 4);
            text_mc.move(4 + buttonWidth + 5, 4);
        }

        // Draw a simple border around the child components.
        graphics.lineStyle(1, 0x000000, 1.0);
        graphics.drawRect(0, 0, unscaledWidth, unscaledHeight);
    }


    /*** i) Add methods, properties, and metadata. ***/
    // The general pattern for properties is to specify a private
    // holder variable.
    private var _textPlacement:String = "left";

    // Create a getter/setter pair for the textPlacement property.
    public function set textPlacement(p:String):void {
        _textPlacement = p;
        invalidateDisplayList();
        dispatchEvent(new Event("placementChanged"));
    }

    // The textPlacement property supports data binding.
    [Bindable(event="placementChanged")]
    public function get textPlacement():String {
        return _textPlacement;
    }

    private var _text:String = "ModalText";
    private var bTextChanged:Boolean = false;

    // Create a getter/setter pair for the text property.
    public function set text(t:String):void {
        _text = t;
        bTextChanged = true;
```

```
            invalidateProperties();
            dispatchEvent(new Event("textChanged"));
        }

        [Bindable(event="textChanged")]
        public function get text():String {
                return text_mc.text;
        }

        // Handle events that are dispatched by the children.
        private function handleChangeEvent(eventObj:Event):void {
                dispatchEvent(new Event("change"));
        }

        // Handle events that are dispatched by the children.
        private function handleClickEvent(eventObj:Event):void {
                text_mc.editable = !text_mc.editable;
        }
    }
}
```

7. 疑难问题

当我试图在MXML中使用组件时遇到了"don't know how to parse…"这样的错误。

这意味着编译器没有发现SWC文件，或者SWC文件的内容没有列出这个组件。确保SWC文件在Flex的搜索路径中，并且确保xmlns属性指向了正确的位置。试将SWC文件移动到与MXML文件相同的目录，并且设置名字空间为"*"，如下例所示：

<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*">

更多信息参见： Using the Flex Compilers.

当试图在MXML中使用组件时，遇到了"xxx is not a valid attribute…"这样的错误。

保证这个属性拼写正确，并确保它不是私有(private)属性。

我没遇到任何错误，但什么也没出现。

验证组件是否被实例化。一个验证方法就是在MXML应用中放一个Button和一个TextArea控件，并在button点击时将text属性设置为自定义组件的ID。例如：

```
<!-- This verifies whether a component was instantiated. -->
<zz:mycomponent id="foo"/>
<mx:TextArea id="output"/>
<mx:Button label="Print Output" click="output.text = foo.id;"/>
```

组件被正确实例化，但是没有出现(1)

有些情况下，组件所需要的支持类在需要之时并没有准备好。Flex按照必须的实例化顺序（首先是基类，然后是子类）将类添加到应用中。但是，如果类实例化过程中调用了一个静态方法，并且这个静态方法依赖其他类，那么Flex不会知道把依赖类放到其它类之前，因为Flex不知道那个方法即将被调用。

一个可能的补救方法就是添加一个依赖那个类定义的静态变量。在类被实例化前，Flex知道

所有静态变量依赖必须被准备好，这样就保证了正确的类加载顺序。

下面的例子添加了一个静态变量，来告诉连接器（linker），类 A必须在类B之前初始化：

```
public class A {
   static function foo():Number {
     return 5;
  }
}
public class B {
   static function bar():Number {
        return mx.example.A.foo();
   }
  static var z = B.bar();
   // Dependency
  static var ADependency:mx.example.A = mx.example.A;
}
```

组件被正确实例化，但是没有出现(2).

验证measuredWidth和measuredHeight是否为非0。如果它们是0或者NaN，那么要确保measure()方法被正确地实现。

也要验证visible属性是否被设置为true。如果visible属性是false，那么要确保组件调用了invalidateDisplayList()方法。

组件被正确实例化，但是没有出现(3).

可能另外一个类或者SWC文件中重载了自定义组件，或者占用了组件的标识符。确保没有命名冲突。