第十三章 字符串：String

字符串（String）的基本概念，以及创建、连接和比较等基础操作在“字符串：String”（第 2 章）中已经介绍过。如果读者对这些概念和操作尚有模糊，请先复习后再阅读本章。

本章主要讲解 String 的一些较为高级的话题，包括 String 在 ActionScript3 中的实现，常用的字符串操作，以及一些实用的字符串函数实例讲解。

本章导读

对于初学者，如果本章的 13.1 节和 13.3 节不明白，可以暂时跳过不看，不影响后面的学习。

对于 ActionScript2 读者，请注意阅读 13.6 节和 13.7 节。

对于使用其他 OOP 语言的读者建议通读本章。

13.1 CHAR 在那里

我们先从字符串在 ActionScript3 的实现和原理讲起。如果你有一定的其他语言编程基础，那么刚刚接触到 ActionScript3 的字符串时，一定会感到有些奇怪：ActionScript3 中怎么没有 Char（字符）类型。我们就从这里开始。

在 C#中字符串是有序的 System.Char 类的实例集合：在 Java 中 String 也是 Character 的一个可读序列。那么，为什么在 ActionScript3 中并没有看到 Char 或者 Character 的影子呢？ActionScript3 并不打算支持 Char 吗？那么和其他语言又有什么区别呢？

在 ActionScript3 官方语言参考文档中，String 也定义为字符串的有序排列。而且，实际上，ActionScript3 并非没有考虑 Char 这个类型。从“ActionScript3 的保留字”这一节可以看到，Char 仍然作为 ActionScript3 的保留字，留待将来只用。

在 ECMAScript edition 4 draft 中，将 String 定义为：String = Char[16]。其含义是，String 是一组有序的 Char16 字符的集合。举个例子，也就是说，字符串“Kingda<<LF>>”与['K','i','n','g','d','a',<<LF>>]等价。

而 Char16 所指的语意范围（semantic domain）是指处于{'<<u0000>>'…'<<uFFFF>>'}之间的 65 536 个 Unicode 字符。而目前关于 ECMAScript edtion 4 draft 的讨论和提取中，还包括是否要添加进对 Char21 的支持。而 Char21 是指处于{'<<u00000>>'…'<<U0010FFFF>>'}的 Unicode 字符。

在这份标准中，规定了 Character 作为系统的内置类（Built-in Class）。但由于 ECMAScript edition 4 draft 对 Char 的标准尚未完全定下来，作为其标准实现的 ActionScript3 语言推迟对 Char 或者 Character 类的实现还是情有可原的。

笔者估计，ActionScript 对 Char 的支持迟早会加入。在目前，除了涉及到某些特殊运用，比如文本加密等运用时，可能需要我们做一些额外工作。一般来说，现有的 String 类可以满足我们绝大部分的需要。

总结一下，ActionScript3 中的 String 也是字符的有序集合，目前尚没有 Character 内置类。

除此以外，String 的操作和实现与 ECMAScript edition 4 draft 描述相同，与 Java、C#相似。

13.2 在 ActionScript3 中如何针对字符操作

在 ActionScript3 中目前没有对 Character（字符）的实现。那么我们如何对一个字符串中的字符进行操作呢？ActionScript3 给我们提供了如下函数，可以方便地操作 String 中的每一个字符。

? 使用 charAt()访问目标位置的字符。

? 使用 charCodeAt() 得到目标位置字符串的 Unicode 字符的整数字符代码。

? 使用 fromCharCode() 从指定的 Unicode 值得到相应的字符串。

在这之前我们先要明确目标位置是什么。

前面说过，我们可以把一个字符串看成一个由字符组成的有序“数组”。那么这个“数组”虽然不能使用数组访问运算符([])来操作，但它的索引如同普通数组一样，都是从 0 开始，到数组的 length-1 为止的。String 对象是有 length 这个属性的，表示这个对象拥有多少个字符。那么我们所说的目标位置其实就是这个“数组”的索引。

13.2.1 运用 charAt()和 char CodeAt()

看示例 13-1 所示的代码例子。

示例 13-1 charAt()和 charCodeAt()的运用

```
var foo:String = "ActionScript3 殿堂之路";

for (var i:int = 0; i<foo.length; i++) {

trace (foo.charAt(i)+ "tunicode 字符整型值为: " + foo.charCodeAt(i)

+ "t 十六进制为:"+foo.charCodeAt(i).toString(16));

//注：toString(16)，意思为将数值转换成十六进制数来表示

}

/*输出:

A unicode 字符整型值为: 65 十六进制为:41

c unicode 字符整型值为: 99 十六进制为:63
```

t unicode 字符整型值为: 116 十六进制为:74

i unicode 字符整型值为: 105 十六进制为:69

o unicode 字符整型值为: 111 十六进制为:6f

n unicode 字符整型值为: 110 十六进制为:6e

S unicode 字符整型值为: 83 十六进制为:53

c unicode 字符整型值为: 99 十六进制为:63

r unicode 字符整型值为: 114 十六进制为:72

i unicode 字符整型值为: 105 十六进制为:69

p unicode 字符整型值为: 112 十六进制为:70

t unicode 字符整型值为: 116 十六进制为:74

3 unicode 字符整型值为: 51 十六进制为:33

殿 unicode 字符整型值为: 27583 十六进制为:6bbf

堂 unicode 字符整型值为: 22530 十六进制为:5802

之 unicode 字符整型值为: 20043 十六进制为:4e4b

路 unicode 字符整型值为: 36335 十六进制为:8def

*/

复制代码

可以看到 charAt()返回了索引 i 指向的字符，而 charCodeAt()返回了指向的字符的 Unicode 整型值。

在 Unicode 值中，第 33～126 号（共 94 个）是字符，其中第 48～57 号为 0～9 十个阿拉伯数字。第 65～90 号为 26 个大写英文字母，97～122 号为 26 个小写英文字母，其余为一些标点符合，运算符号等。

中文字符的十六进制数值范围是[u4e00-u9fa5]，所有的全角字符的十六进制数值范围是[uFF00-uFFFF]。

常用字符的 ACSII 码表见附录“A.1 常用字符的 ASCII 码表”。

13.2.2 运用 fromCharCode()

fromCharCode()是 String 类的静态方法，它会返回一个由参数中的 Unicode 值表示的字符组成的字符串。参数个数不限，fromCharCode 方法将参数 Unicode 值所代表的字符按顺序组合成一个字符串。

注意，formCharCode 方法对其他参数默认转换成整数型，即，如果给定的参数并不能转换成有效的整数型，那么 fromCharCode 返回的是一个字符串“x00”，即一个仅含 ASCII 码为 0 的空字符的字符串。

看示例 13-2，使用 String.fromCharCode()来拼出中文字符串，以及在该方法参数中使用十六进制整数。

示例 13-2 fromCharCode 的运用

```
//使用中文字符的 Unicode 值就可以拼出中文字符串

var foo:String = String.fromCharCode(27583,22530,20043,36335);

trace (foo);

//输出：殿堂之路

//在 formCharCode 中使用十六进制整数

foo = String.fromCharCode(0x41, 0x53, 0x33);

trace (foo);

//输出： AS3;
```

复制代码

技巧点

运用 String.fromCharCode()可以发现 ActionScript3 和 ActionScript2 的一个不同点：ASCII 码为 0 的字符串在 ActionScript2 中被当成了字符串的标志，在 ActionScript3 中则不会。

使用下面的代码：

```
var str:String = String.fromCharCode(97,0,0,97);
trace (str.length);
//在 ActionScript 3 中输出为 4，在 ActionScript2 中输出为 1
```

trace (str.charCodeAt(3));
//在 ActionScript 3 中输出为 a，在 ActionScript 2 中输出为 NaN

13.3 字符串是不变对象（immutable object）
字符串在 ActionScript3 中是以不变对象的形式存在的。一个字符串对象一旦被创建，其值就被确定不会被更改。这种原理，使得一个字符串对象可以被共享，以降低对内存的消耗。

举个例子，看如下代码。

var a:String = “Kingda”;

var b:String = a;

a += “love actionscript”;

trace (a); //输出：Kingda7 love actionscript

trace (b); //输出：Kingda
复制代码

第一行创建了一个新的 String 对象 A，值为“Kingda”，赋值给变量 a。第二行新建了一个变量 b，将 a 的引用赋给 b。此时 a 和 b 都指向同一个字符串对象“Kingda”。第三行将一个新的字符串对象“love actionscript”添加到变量 a 所指向的字符串上。
如果字符串不是不变对象，由于 a 和 b 指向同一个对象，既然变量 a 指向字符串 A 变化了，那么 trace (b)出来的结果应该和 a 一样。但事实上 b 和 a 不一样。因为第三行并没有改变不变对象 A 的值，而是新创建出来一个字符串对象“Kingda love actionscript”，将其赋值给了变量 a。也就是说，从这一刻起，其实 a 和 b 指向的字符串对象已经不一样了
13.4 ActionScript3 中的 StringBuilder 在哪里
StringBuilder 类是什么？它对于很多中级爱好者而言，并不熟悉。但是在 Java、NET 中都有这个类的存在。字符串在频繁修改时，效率低下。通俗地说，StringBuilder 是为了改进这种情况而加进的工具类。下面来详细解释产生这种状况的情形、原因和解决方式。
由于字符串一旦生成，就是一个不变的对象，其中的字符集也是一个不变的序列。固然，这样好处很多，但带来的一个大麻烦就是：每次修改字符串对象时，都要在内存中创建一个新的字符串对象，这就需要为该新对象分配新的空间。
而其他基元数据类型，每个对象的值最多占几个字节，比如 int、number、boolean 等。而字符串可不一样，你赋给它多少字节，它就能占多少字节的内存。
当我们执行的字符串修改并不多时，那么可能感觉不到这种差异。但在需要对字符串执行重

复修改的情况下，创建新的 String 对象导致的相关的系统开销可能会非常昂贵。在.NET 开发时，如果要修改字符串而不创建新的对象，则可以使用 System.Text.StringBuilder 类。例如，当在一个循环中将许多字符串连接在一起时，使用 StringBuilder 类可以提升性能。而在 Java 中提供了 java.lang.StringBuilder，以及供多线程使用的 java.lang.StringBuilder。

在 ActionScript3 中为什么没有 StringBuilder 类呢？其实在 ActionScript3 的设计过程中，曾经有过 StringBuilder 类。但后来，ActionScript3 对加法运算符（＋）做了修改，使得它在实现复杂字符串（Compound String）时效率非常高，从而 StringBuilder 类没有必要再加进去了。底层原理是因为 ActionScript3 中的复杂字符串实现是一种绳状结构，当“＋”时，只是把现有的 String 对象链接到原有对象后面，不需要频繁生成新的不变对象。

13.5 +、+=及<、<=、>、>=在字符串操作中的重载

ActionScript3 的运算符重载和 Java 相似，加法运算符（＋）和加法赋值运算符（＋＝）也被赋予了重载的功能，用于连接字符串。关系运算符（<、<=、>、>=）也被重载了，可以对字符串按字母排列顺序进行比较。

与其他语言对比

C#、C++和 C 的用户要注意，ActionScript3 中是不可以通过编程对运算符进行重载的。事实上，笔者认为运算符重载会使语言变得复杂，对 ActionScript3 的语言设计也表示赞同。

13.5.1 在 ActionScript3 中运算符的重载：+和+=的重载

加法运算符（＋）和加法赋值运算符（＋＝）可以被用来连接字符串，此时它们被称为字符串运算符。看下面的例子。

```
var a:String = “aaaaa”;

var b:String = “bbbbb”;

var c:String = a + b;

trace (c); //输出:”aaaaabbbbb”。两个字符串连接起来了

c += “ccc”; //使用“+=”让字符串 c 再在末尾增加字符串“ccc”

trace (c); //输出: “aaaaabbbbbbccc”
```

复制代码

当“＋”、“＋＝”被用做字符串运算符时，它们的运算对象都会被看做字符串。如下例。

```
var a:String = “aa”;

var b:int = 3;

var c:String = a + b;

trace (c); //输出:aa3。整型变量 b 被转换成字符添加在最后了

var d:Boolean = true;

c += d;

trace (c); //输出:aa3true。布尔变量 d 的值 true 也被转换成字符串添加在最后了
```

复制代码

实际上，字符串运算符并没有把非字符串运算对象转换成字符串，而是对这些运算对象的 toString()函数所返回的字符串进行拼接。
像 int、Boolean 等这些基本类型，也是执行了其包装类相应的 toString()函数。我们通过如下的例子可以证明。

```
var a:String = “aaa”;

var f:Object = {}; //建立一个空对象，将其引用赋值给变量 f

//人为地将 f 的 toString 函数改成只返回字符串”hohaha”

f.toString = function() {

return “hohaha”;

};

trace (a+f); //输出:aaahohaha，而不是 aaa。可见是将 f.toString()的返回值添加在后面了
```

复制代码

13.5.2 关系运算符（<、<=、>、>=）的重载
如果关系运算符（<、<=、>、>=）两边的运算对象都是字符串的话，那么将从左到右按字母顺序来挨个进行比较。
看下面的例子。

var a:String = “bbc”;

var b:String = “axz”;

trace (a>b); //输出:true。因为字符串 b 的首字母比字符串 a 的首字母顺序靠后

var c:String = “actionscript3.cn” //创建一个长一点的字符串变量 c

trace (b>c); //输出:true。因为 b 的第二个字母比 c 的第二个字母靠后。与字符串长度无关

var e:String = “actionscript”;

trace (e>c); //输出:false。注意，e 的所有字母和 c 相同。此时与字符串长度有关
复制代码

如果是中文字怎么办？
这就涉及到了这个字符串比较运算的实质原理，其他书中都没有提到，包括官方的文档。实际上，这个比较算法是从左到右挨个取字符的相对应的 unicode 数值，然后进行比较。

var a:String = “中”;

var b:String = “国”;

trace (a>b); //输出：false

trace (a<b); //输出：true

trace (a.charCodeAt (0)); //输出：20013。这就是“中”字的 unicode 字符值

trace(b.charCodeAt (0)); //输出：22269。这就是“国”字的 unicode 字符值
复制代码

13.6 常用的字符串操作
常用的字符串操作一般有：查找、匹配、替换、分割子串、大小写转换。
查找、匹配和替换是最常见用的 3 种字符串操作。在 ActionScript3 中，查找、匹配和替换分别对应 search()、math()、replace()3 种字符串方法。这 3 种字符串操作方法，都与正则表达式紧密结合，功能强大。可以说，与正则表达式的结合，使得 ActionScript3 的字符串操作可以与任何一门标准 OOP 语言相媲美。而且，在某些方面，其灵活性、直观性、方便程度甚至超过了 Java 和 C#。这 3 种方法的详细运用请参见“正则表达式与字符串的结合使

用详细”。

除了以上 3 种操作外，还有以下几种不需要正则表达式的常用操作。

? 需要知道一个子串的位置时，可以用子字符串的查找定位。查找定位，除了使用正则表达式来实现以外，还可以使用 indexOf()、lastIndexof()来实现。这对于 ActionScript2 用户一定不陌生，在 ActionScript2 时代这是我们唯一的武器。往事不堪回首。

? 需要根据起始和终了位置来提取了一个字符串时，可以用 substring()、slice()方法来实现。根据起始位值和长度来提取时，可以使用 substr()方法。

? 需要根据特定的标识符字符串分割成几个字符串时，就要用到 split()方法。

? 需要将一个字符串全部转换成小写字母时用 toLowerCase()，全转换成大写时用 toUpperCase()。

下面用代码实例来分别讲解这几种字符串操作的用法。

13.6.1 indexOf()的使用及与 search()的区别

lndexOf()方法是用来判断一个字符串是否存在于一个更长的字符串中。从长字符串左端到右端来搜索，如果存在该子字符串就返回它所处的位置（即索引）。如果在被搜索的字符串没有找到要查找的字符串返回-1。注意，这里的位置应当填写索引值。所有的字符串索引都是从零开始，第一个字符的位置就是 0，终点位置就是字符串的长度减去 1。

该方法等价于 C 语言中的 strstr 函数及 Visual Basic 语言中的 inStr 函数。这个方法也有一个相应的函数，即 lastIndexOf()，从长字符串的右端搜索。

那么问题来了，search()方法也是同样返回目标自字符串索引值的。indexOf()和 search()有什么区别呢？为什么时候该使用它，什么时候该使用 search()这个方法呢？

首先要明确 search()的参数必须是正则表达式，而 indexOf()的参数只是普通字符串。

indexOf()是比 search()更加底层的方法。

如果只是对一个具体字符串来查找，那么使用 indexOf()的系统资源消耗更小，效率更高；如果是查找具有某些特征的字符串（比如查找以 a 开头，后面是数字的字符串），那么 indexOf()就无能为力，必须要使用正则表达式和 search()方法了。

很多时候用 indexOf()不是为了真的想知道子字符串的位置，而是想知道长字符串中没有包含这个子字符串。如果返回索引值是-1，那么说明没有：不等于-1，那么就是有。

13.6.2 substring()、slice()和 substr()的使用及区别

substring()、slice()和 substr()都是从长的字符串中提出的子字符串。三者相同之处在于，都不会改变原来长字符串的内容，只是返回符合条件的子字符串。

所不同的是，substing()和 slice()是根据起点和终点位置来提取的，substr()是根据起点和要截取的字符串长度来提取的。注意，这里的位置应当填写索引值。所有的字符串索引都是从零开始的，第一个字符的位置就是 0，终点位置就是字符串的长度减去 1。

它们的用法如下。

? substing：长字符串变量.substing（起点位置，终点位置）

? slice：长字符串变量.slice（起点位置，终点位置）

? substr：长字符串变量.substr（起点位置，要截取的字符串长度）

而 substring()和 slice()也有一些小区别。

substing()的起点和终点不可以为负数，即使用了负数，也会被当成 0 来处理。slice()的起点和终点不仅可以写正整数，也可以写负数。当设为负数时，意味着是从字符串右端到左端来数。比如说-1，就是指倒数第一个字符。

substring()中如果起始位置大于终点位置，执行时会自动互换位置。slice()中如果起始位置大于终点位置，那么返回的将是一个空字符串。

substring()和 slice()还有一个共同点就是：如果只填写一个参数，那么该参数会默认文起始位置，终点位置自动设为最后一个字符（即字符串长度减去 1）。

至于 substr()，起点位置可以使用正整数，也可以使用负数。长度理论上只能用正整数，但实际上也可以用负数。但是，用负数长度时，与我们通常的思维习惯不符，不能算做阅读性好的代码。因此，应尽量避免用负数。

13.6.3　split()与正则表达式结合运用

split()可以将字符串按指定的分隔符分开。在 ActionScript3 中，分隔符参数甚至可以用正则表达式对象，见示例 13-3。

示例 13-3 split()：用正则表达式分割字符串

```
var str:String = “Sky sky6 ask778 wu99.”;

var delimiter:RegExp = /(d+)/; //表示数字

trace (str.split(delimiter));

//Sky sky,6, ask,778, wu,99,.

//果然以数字分隔开来了
```

复制代码

13.6.4　如何一次性输入多行文本

在编程时，往往需要在代码中写入多行文本（比如，从外部复制而来），这样的话如果使用字符串形式，则需要我们自己手工去掉每行的换行，再加上很多个“＋”号，非常不便。

与大家分享笔者的一个实用的小技巧，以轻松输入多行文本。秘诀就在于使用 XML 的 CDATA 来配合。见示例 13-4。

示例 13-4 一次输入多行文本

```
//建立一个只含有一个 CDATA 节点的临时 XML，将多行文本直接复制进 CDATA 中
var tmpXML：XML =
<txt>
<![CDATA[
运行时异常 （Runtime Exceptions）处理机制
运行时类型 （Runtime types）
密封类（Sealed Classes）
闭包方法（Method closure）
使用 E4X 理论处理 XML 数据
正则表达式
命名空间
新基元数据类型
]]>
</txt>
//将临时 XML 转换成字符串赋值给 myString，就得到了多行文本内容
var myString:String = tmpXML.toString();
trace (myString);
import mx.utils.StringUtil;
//如果发现输出的字符串首尾有空白，可以使用这个方法去掉首尾空白
```

```
myString = StringUtil.trim(myString);

trace (myString);
```

复制代码

13.7 实用的 mx.utils.StringUtil 工具类

Flex 中提供了非常实用的字符串工具类函数 mx.utils.StringUtil。这是个纯工具类，非 Flex 环境也可以正常使用。

它里面提供了 4 种静态方法。

? 截断字符串首尾空白:StringUtil.trim(str:String):void

? 替换字符串:StringUtil.substitute(str:String,…rest):String

? trim 分隔符分割的字符串:StringUtil.trimArrayElements(value:String delimiter:String):String

? 检测空白字符:StringUtil.isWhitespace(character:String):Boolean

trim()函数是使用频率很频繁的字符串处理函数之一。它的用途是把一个字符串的首尾两端的空白去掉，经常用来删掉用户输入的字符串两端多余的空白，以及从 XML 或其他外部数据中获得的文本中的两端空白。

substitute()函数则是 Flex 架构自己使用最多额字符串函数之一了。它的作用是将字符串中含有“{0}”、“{1}”…“{n}”的地方，替换成参数(…rest)传入的参数数组对应元素。

trimArrayElements()函数字而是将字符串中以 delimiter 参数隔开的部分的空白一一去掉。

StringUtil 的使用方法见示例 13-5。

示例 13-5　StringUtil 使用示例

```
import mx.utils.StringUtil;

var str1:String = “ Welcome to Kingda.org! ”

trace (“|” + StringUtil.trim(str1)+ “|”);

//输出：|Welcome to Kingda.org！|

var str2:String = “n”;

trace (StringUtil.isWhitespace(str2));

//输出：true

var str3:String = “ Kingda.org | ActionScript3.cn | Adobe.com ”;
```

```
trace (StringUtil.trimArrayElements(str3, “|”));

//输出：Kingda.org|ActionScript3.cn|Adobe.com

var str4:String = “{0} is a {1} o {2}”;

trace (StringUtil.substitute(str4, “Kingda.org”, ”blog” , “ActionScript3”));

//输出：Kingda.org is a blog of ActionScript3
```

复制代码

13.8 本章小结

本章介绍了一些字符串的中高级应用，请读者熟练掌握。

本章到此结束了，下面将进入 ActionScript3 激动人心的一章??正则表达式。大家将看到用正则表达式来处理字符串是多么优雅、高效、便利。正则表达式，作为大部分语言的通用规则，是每一个 ActionScript3 用户都应该必须了解和掌握的。

第十四章 强大的正则表达式：RegExp

ActionScript3 引入了强大的正则表达式，这对于所有的 ActionScript 开发人员来说是一个绝好的消息和不小的挑战。正则表达式的引入，使得 ActionScript3 的字符串处理功能空前强大。

正则表达式是 ActionScript3 的重大特色之一。

本章导读

建议初学者和所有的 ActionScript 读者都要仔细阅读本章。

对于 Java、C#其他语言中对正则表达式已经能熟练运用的读者，为了了解 ActionScrpt3 正则表达式的不同之处，对以下章节要留心阅读：14.2.2 节，14.2.3 节，14.2.6 节，14.4.节，14.3.9 节。

本章 14.5 节“常用的正则表达式”提供几十个实用的正则表达式，供读者学习和在实际编程中使用。

14.1 什么是正则表达式？

正则表达式的名声很响。它功能强大，而且貌似外星文字，让许多程序员常学常忘，常忘常学。实际上，在复杂的外表下，它有一颗简单的心。

编程人员日常打交道最多的就是字符串，从而字符串的操作也是最繁多的工作之一。在字符串的相关操作中，查找含有特定特征的字符串并进行相关操作，是经常碰到的。比如说：

a．在一大堆文本中找“Windows”，把它替换成“Linux”。这是最简单的替换操作了。

b. 稍微复杂一点的操作有，将文本中所有 html 标签，如“<p>”、“<font>”等所有标签去掉。

c. 查找所有“1272006”这样类似格式的日期，并累计一下有多少个这样的日期。

大家先花 5 分钟思考一下，这些操作用上一章中的方法该如何解决。

再看一下，大家有没有注意到凡是涉及到字符串操作，都大致分为两步：

（1）描述出我们要操作的字符串的特征。比如例子 a 中字符串特征就是“windows”；b 中就是由尖括号起来的 html 标签；c 中就是“1272006”类似格式的日期。

（2）对符合描述特征的字符串（术语称为匹配）进行相关操作。比如例子 a 中是将匹配的字符串替换“Linux”，b 中是将匹配的字符串去掉，c 中是计算匹配的字符串去掉，c 中是计算匹配的字符串一共有多少个。

没有第一步，第二步根本无法进行。如果只用上一章中字符串相关的基础方法 indexOf()等，那么虽然可以大费周折完成第二个和第三个例子，但效率是低下的。写出来的代码其重用性是很低的。

而我们可以发现，第一步的工作是相对独立的，完全可以单独抽出作为一共模块来完成这些特征描述工作。但是问题又来了，对于特征的字符串，可以有好多种匹配的程序语言描述方式，那么用怎样的语言可以简捷明了地描述出来呢？于是，救世主出现了，正则表达式就是

为了解决描述字符串特征而出现的。

正则表达式，英文为 Regular Expression，意思就是“规范的表达式”。准确地说，正则表达式用来描述字符串特征的标准方式。我们通过正则表达式就可以准确地告诉电脑我们需要寻找的字符串是什么样子，进而进行替换、计数等其他后续字符串操作。

14.2 ActionScript3 中的正则表达式

14.2.1 第一个正则表达式

正则表达式的写法初看上去是很复杂的。但实际上了解了它的规律，你会发现，其实它的表述是很直白易懂的，而且还很难设计出比它更简单而全面的字符串描述方式。

我们先看第一个简单的字符串查找和替换例子。在一个字符串中查找“Windows”字符串并替换成“Linux”。

```
var myPattern:RegExp = /Windows/g;

var str:String = “Windows is a excellent operation system. I love Windows.”

var repStr:String = “Linux”;

trace (str.match(myPattern, repStr));

//输出：Linux is a excellent operation system. I love Linux.

trace (str.match(myPattern));

//输出：Windows,Windows
```

复制代码

其中，第一行 myPattern 就是一个正则表达式对象。在 ActionScript3 中的写法非常简便，在两个斜杠中写入要匹配的字符串正则表达式，在第二个斜杠后面写上匹配模式标签。本例中由于就是匹配“Windows”，因此正则表达式很简单，就是“Windows”，我们希望全文匹配，而不是匹配第一个后就停止，因此我们加上一个 g，即 global 的缩写，表示从头到尾匹配。

第二行的 str 字符串对象就是我们要匹配的原文。第三行的 repStr 就是我们用来替换的字符串“Linux”。然后调用 String 对象的 replace 方法。Replace 方法需要两个参数：第一个是要替换的字符串的特征描述，即特征字符串的正则表达式；第二个是用来替换的字符串。

String 对象的 match 方法也是需要搭配正则表达式使用的。它返回的是与正则表达式描述想符合的，即匹配的字符串数组。本例中 str 字符串一共出现了两个“Windows”，因此有两个字符串，输出了两个“Windows”。关于字符串和正则表达式的结合，笔者将在 14.4 节“正则表达式与字符串的结合使用详解”中详细讲述。

14.2.2 正则表达式的两个构成部分

先看一下上一个例子中的正则表达式

```
var myPattern:RegExp = /Windows/g;
```

复制代码

正则表达式分为两个部分：一部分是在双斜杠里面的，这部分叫做匹配模式（Pattern），用来描述字符串特征的；另一部分是在第二个斜杠后面的字母，叫做正则表达式的标志位，是 gimsx 这五个字母的组合。参见 14.3.9 小节“正则表达式的 5 个标志”。

ActionScript3 中除了提供这种方便和标准的写法外，还可以使用正则表达式类的构造函数来生成一个正则表达式的实例。下面的代码中生成的正则表达式实例与上例中的正则表达式完全相同。

```
var myPattern:RegExp = new RegExp(“Windows”,”g”); //构造函数写法
```

复制代码

可以看出正字表达式的构造函数只不过是将匹配模式和标志位分开成两个字符串，当成参数，来生成一个正则表达式实例。标志位是可选参数，如果有为空或者不合法标志位字母出现，那么标志位参数将被忽略。所谓不合法的的标志位字母，意识是不属于 gimsx 的字母。

平时手写代码生成正则表达式时，使用双斜杠写法比较明快直白。构造函数一般是用在运行时生成正则表达式实例。比如，我们通过某个类的实例返回了一个我们需要用来匹配的字母串，那么就可以使用构造函数。例如：

```
var myPattern:RegExp =

new RegExp(someObj.getPatternString(), someObj.getRegFlag());
```

复制代码

使用构造函数生成正则表达式实例，要注意由于匹配模式为 String 型，要注意相关的字符转义，比如单引号和双引号。

注意，本书中举例和途标中的正则表达式都采用双斜杠的表达式。如有需要，要转成构造函数的双字符串表达方式，请自行转换。

14.2.3 ActionScript3 中正则表达式的工作机制

知道正则表达式引擎是如何工作的，有助于你很快理解为何某个正则表达式没有达到期望的结果。

正则表达式的范围定了，但各种语言对它的实现有所不同。对正则表达式的实现一共有两种类型的引擎：DFA 引擎和 NFA 引擎。NFA 引擎比 DFA 引擎应用广泛，因为它支持回调引用（backreference）和惰性量词（lazy quantifiers）。DFA 引擎的执行效率比 NFA 引擎的执行效率高，但更耗内存；NFA 引擎在没有使用回调引用时，与 DFA 引擎的执行效率差不多。

ActionScript3 中使用的是功能更加强大的 NFA 引擎。

在执行结果上，NFA 引擎是一种贪婪（Greedy）的执行方式，一旦找到匹配立刻返回。NFA

引擎总是返回最左边的匹配。这一点很重要。所以即使在当前匹配位置后有可能发现一共其他的匹配，NFA 引擎也总是最先返回最左边的匹配。

当把/cat/应用到“He captured a catfish for this cat”，引擎先比较 c 和“H”，结果失败了。于是引擎再比较 c 和“e”，也失败了。引擎再继续从第五个字符重新检查匹配性。直到第十五个字符开始，cat 匹配上了“catfish”中的“cat”，正则表达式引擎急切地返回第一个匹配的结果，而不会再继续查找是否有其他更好的匹配。

14.3 正则表达式语法

14.3.1 正则表达式中的文字符号

文字符号，就是包括字母、符号和单词。中文也属于广义的文字符号，只不过它在正则表达式中是用 Unicode 码表示的，与字符串一一。单由文字符号组成的正则表达式是最简单的类型。

举例如下：

```
//4 个简单的正则表达式

var simplestPattern:RegExp = /a/;

var singleWordPattern:RegExp = /com/;

var stringPattern:RegExp = /a computer/; //注意：正则表达式中可以直接输入空格

var notExistPattern:RegExp = /lalala/;

//用来查找匹配模式的目标字符串

var targetStr:String = “There is a computer at foo.com”;

trace (targetStr.match(simplelestPattern));

//输出:a  是”is a”的 a,不是”at”中的 a

trace (targetStr.match(singleWordPattern));

//输出:com,注意这里的 com 是 computer 的前三个字母，不是 foo.com 的 com。

trace (targetStr.match(stringPattern));

//输出:a computer

trace (targetStr.match(notExistPattern));
```

//输出:null 因为 targetStr 字符串中根本没有”lalala”这样的字符组合。

复制代码

注意，正则表达式总是返回最左边的匹配结果，而不是最合适的匹配结果。比如以上代码中与 singleWordPattern 匹配的更好的应当是“foo.com”中的“com”，但正则表达式引擎一旦找到“computer”中前 3 个字母匹配“com”，就停止查找并返回这个匹配。原因见第 14.2.3 节：ActionScript3 中正则表达式的工作机制。

文字符号分为 3 种，如表 14-1 所示。

表 14-1 正则表达式中的普通字符、元字符、不可见字符

普通文字符号

A 到 Z，a 到 z，数字 0 到 9，以及不是元字符的其他字符（比如#、＝）

元字符

11 个被保留做特殊用途的符号：[]^$.|?*+()

不可见字符

cx 匹配由 x 指明的字符串。例如，cM 匹配一个 Control-M 或回车符。X 的值必须为 A~Z 或 a~z 之一。否则，将 c 视为一个原义的’c’字符

f 匹配一个换页符。等价于 x0c 和 cL

n 匹配一个换行符。等价于 x0a 和 cJ

r 匹配一个回车符。等价于 x0d 和 cM

s 匹配任何空白字符，包括空格、制表符、换页符等等。等价于[fnrtv]

S 匹配任何非空字符。等价于[^fnrtv]

t 匹配一个制表符。等价于 x09 和 cI

v 匹配一个垂直制表符。等价于 x0b 和 cK

在本节开头的代码例子中，就都是普通文字和符号的例子。

对于元字符，由于它们在正则表达式中有特殊的用途，所以如果字符串中有和元字符相同的字符，那么要用反斜杠“”来进行转义。比如下例中，由于“？”号是元字符，所以要匹配问号时，要加上反斜杠转义：

```
var fooPattern:RegExp = /?/;

var targetStr:String = http://www.kingda.org/index.html?random=1;

trace (targetStr.match(fooPattern));
```

复制代码

另外，由于 ActionScript3 中正斜杠被用来书写表达式，所以，如果我们要匹配正斜杠时，也需要进行转义。

```
var fooPattern:RegExp = ///g;

//注意，加上了 g 标签，表示全字符串匹配，而不是只匹配第一个就返回

var targetStr:String = http://www,kingda.org/index.html?random=1;

trace (targetStr.match(fooPattern));

trace (“length:” + targetStr.match(fooPattern).length);

//输出：3 表明有 3 个正斜杠被匹配了
```

复制代码

但是，如果我们是使用正则表达式构造函数来匹配正斜杠的话，就不需要转义了，如下行代码，但元字符还是需要转义的。

```
var fooPattern:RegExp = new RegExp(“/”,”g”);
```

复制代码

对于，不可见字符，它们代表的是字符串进行排版的字符，是不可见的。因此我们要匹配它们，必须使用这些由反斜杠开头的字符。比如，下例中我们要匹配字符串中的制表符“t”。

```
var fooPattern:RegExp = /t /g;

//参见表 14-1 正则表达式中的普通字符、元字符、不可见字符，也是可以写成/cl/g，/x09/g;

var targetStr:String = “There is a t computer t at foo.com t”;

trace (targetStr.match(fooPattern));

//输出：3 但从这里我们可以看到匹配使成功进行了的，一共有 3 个
```

复制代码

说了这么多，我们来看看下面这个例子，如何使用已学的正则表达式知识，来匹配单个的“com”，而不是单词里面包含的“com”。

```
var fooPattern:RegExp = /scoms/g;

//输出：s 表示任意空白字符，所以我们在 com 前后各加上一个

var targetStr:String = “computer complete com abc.com tcom rcom notcome”;

trace (targetStr.match(fooPattern));

/*

输出:

com, com,

com

注意，最后一个 com 换了一行，因为最后一个 com 前是换行符 r。

*/

trace (targetStr.match(fooPattern).length);

//输出：3 表示有 3 个匹配
```

复制代码

14.3.2 正则表达式中的字符集

1. 字符集的概念和最简单的字符集形式

如果我们想用一个正则表达式匹配一个字符串中的所有前后有空白的“get”和“got”，怎么办？那么用目前的知识无法办到。这就引入了字符集的概念。如下例：

```
var fooPattern:RegExp = /sg[eo]ts/g;

//前后加 s 确保有空白，中间的[eo]就是字符集

var targetStr:String = “geeet got kingdaget kingdagot get geot getter gxt”;

tarce (targetStr.match(fooPattern));

//输出：got,get

trace (“length” + targetStr.match(fooPattern).length);

//输出：2 表示有 2 个匹配
```

复制代码

字符集是指由中括号里定义的字符集合。字符集中的字符只要有一个复合特征描述，那么就会被认为匹配。但是注意了，上面的 targetStr 中还有一个 geot 并没有被匹配。这事因为虽然 e 和 o 都是符合字符集[eo]描述的，但 eo 是不符合[eo]匹配的。字符集只能匹配一个字符，初学者要记住这一点。

像[eo]、[szwd]、[abcd1234]、[“#]等由字符和字符简单罗列而形成的字符集，是最简单的字符形式。

2. 连字符在字符集中的运用

如果我们想描述一个字符集，字符集里包括所有的小写字母，那怎么办？将 24 个小写字母全写到方括号中？当然不会这样拙劣。我们可以使用连字符“-”来定义一个范围。表 14-2 列出了常用的 3 个字符集。

表 14-2 常用的 3 个字符集

[A-Z]	表示从 A 到 Z 的所有大写字母。也可以定义[B-H]这样的小范围的大写字母集合

[a-z]

表示从 a 到 z 的所有小写字母。也可以定义[w-z]这样的小范围的小写字母集合

[0-9]

表示从 0 到 9 的所有数字。也可以定义[3-6]这样的从 3 到 6 的小范围数字字符集合

例子：在字符串中查找日期，样式如 24/05/2007。

```
var foo:RegExp = /[0-3][0-9]/[0-1][0-9]/[0-2][0-9][0-9][0-9]/g;

//注意：正斜杠要转义，在前面加了一个反斜杠

var targetStr:String = “Date Formats: 2006/12/25 2006-12-25 12/25/2007 25/12/2007”;

trace (targetStr.match(foo));

//输出：25/12/2007

trace (“length:” + targetStr.match(foo).length);

//输出：length:1
```

复制代码

由于我们定义了日期各个位置的有效数字范围，所以整个字符串中匹配的只有 25/12/2007。相似的 12/25/2007，由于 25 不符合月份特征“[0-1][0-9]”的描述，因此被排除了。

但我们也可以发现这个正则表达式对于验证还是不够正确的，比如 39/19/2999 这样的错误日期也是被认为匹配的。如果要解决这个问题，还需要后面的知识。这里暂不讨论。

举个常用的例子：

匹配 0xFF 这样的两位数十六进制的数字。正则表达式如下：

```
var foo:RegExp = /0x[A-F0-9][A-F0-9]/g;
```

```
var targetStr:String = “Search hexadecimal:0bEE 0xFF 0x09,0x2e is not a number”;

trace (targetStr.match(foo));

//输出：0xFF，0x09
```

复制代码

上例中，我们默认十六进制数是以大写字母表示的。如果我们允许可以使用小写字母表示，就要将正则表达式修改成“0x[a-fA-F0-9][a-fA-F0-9]”。一个[a-fA-F0-9]表示一个字符，该字符可以是大写或小写的字母，也可以是数字。所以一个严格的十六进制两位数表示应该是“0[xX][a-fA-F0-9][a-fA-F0-a]”。

3. 取反符合在字符集中的运用

先看一个取反符号运用的例子：

```
var foo:RegExp = /g[^eo]t/g;

var targetStr:String = “gt get got geet g t gtt gst seet”;

trace (targetStr.match(foo));

//输出：g t ,gtt.gst

trace (“length” + targetStr.match(foo).length);

//输出：length：3
```

复制代码

取反符号（^）的作用是对方括号里面描述的字符取反。换句话说，使用了取反符号，那么只要是不符合字符集描述的字符就符合匹配。但有两点要注意：

? 取反字符必须紧跟在“[”号后面，不然它将作为普通的字符“^”，没有取反作用。

? 运用了取反字符的字符集，必须匹配一个不符合字符集描述的字符，但不能匹配空的字符。需要注意的是，强调的是空的字符，而不是空白字符。表示空白的字符，是可以匹配的，比如空格、制表符（r）、换行符（n）。如上例中“g[^eo]t”，匹配了“gt”，而不匹配“gt”一样。

4. 字符集中需要转义的特殊字符

我们来比较一下目前已经碰到的 3 个说到转义的地方，如表 14-3 所示。

表 14-3 字符串、正则表达式、正则表达式字符集转义字符对比

字符串

双引号（”）或者单引号（’），反斜杠（）

正则表达式

[]^$.|?*+()如果使用//

正则表达式字符集

]^-

可以看到正则表达式字符集中的元字符，既有特殊用途的字符，比正则表达式中的元字符少。换句话说，我们在定义字符集时，可以不必对“[]^$.|?*+()”这几个符合进行转义，它们将被当成普通符号。而“]^-”在使用时要用“”进行转义。
但是，如果将“^”、“-”符号放在不会有特殊用途的地方，那么不需要用“”来转义，自然会被当成普通字符来处理。比如“^”不紧跟在“[”后面，“-”前面或者后面没有字符。例如：

```
var foo:RegExp = /g[eo^-]t/;
```

复制代码

14.3.3 特殊的点号（.）
在正则表达式中，如果我们要简要地去匹配一个字符，而不用关心被匹配的字符是什么，怎么办？比如说，想匹配一个由 3 个字符组成的字符串，开头为 g，结尾为 t，中间一个随便什么字符。难道要我们写一个字符集，将字母、数字、符号、转义了的元字符，还有不可见字符统统放进去？这将会多么麻烦。所以，正则表达式语法中准备了一个“.”号，来表示任何符号。这种用法很常见，所以“.”成为了最常见的符号之一。然后，它也最容易被误用。
1. 点好不匹配新行符（n）
但是，记住，“.”号不匹配新行符（n）。不仅是 ActionScript3 中的正则表达式引擎，事实上大多数语言的正则表达式在默认情况 下都是不匹配新行符的。Windows 下文本的换行符是“rn”，UNIX 下文本换行符是“n”。因此，在 windows 下，“.”等于是字符集[^rn]；UNIX 下等于是[^n]。

这个例外是历史造成的。早期由于硬件资源的限制和软件系统的设计，使用正则表达式的工具是基于行的。即都是一行一行地读入一个文件，再将正则表达式分别应用到每一行上去。因此，这些字符串是不会包含新行符的，所以“.”也就从不匹配新行符（n）。到了现代，正则表达式往往要应用到多行字符串甚至应用到整个文件中，那么点号不匹配新行符成了一个麻烦。所以，与大多数语言一样，ActionScript3 也提供了一个单行模式标志符（s），打开它可以让点号匹配新行符。如下例：

```
var foo:RegExp = /g.t/sg; //注意：增加了单行标签 s

var targetStr:String = “g5t gnt grt g-t gtt gst seet”;

trace (targetStr.match(foo));

/*

输出：

g5t,g

t,g

t,g-t,gtt,gst

注意，有两个 g 后面换行了，一个是 r，一个是 n

*/
```

复制代码

2. 尽量少使用点号“.”

除了需要使用点号来表示一个任意字符，在其他情况下使用它都要三思而后行。因为点号在匹配几乎所有的字符的同时，它也常常匹配不该匹配的字符。而作为中文用户，更要记住，点号可以匹配任意一个中文字。事实上，点号匹配所有的 Unicode 字符（除了默认情况下的新行符）。

比如，我们想找到一个字符串中所有以符号对隔开的 a，比如：找出一篇文章中所有<1>，|2|，（3），-4-这样形式的标题。

那么最省力的方式看起来就是这样做。

```
var foo:RegExp = /.[0-9]./sg;
```

```
var targetStr:String = “<1> |2| (3) -4- $5* c6d 2advance  想 1 想”;

trace (targetStr.match(foo));

//输出：<1>, |2|, (3) , -4- , $5* , c6d , 2a ,  想 1 想
```

复制代码

如上，结果却是虽然<1>, |2|, (3) , -4-这些标题已经找出来了，但也多了很多我们没有想到的结果。比如一个莫名其妙的“$5*”，连“2advance”也因为有一个数字 2，2 前面是一个空格，符号点号的匹配，2 后面是一个字母也符号点号的匹配所以入选。而点号连汉字“想”也匹配，所以也没有放过“想 1 想”。

那么“/[(-[<][0-9][(-[<]/g”这样的表达式明显要比上例中的表达式严谨的多。所以在不需要点号时，尽量不要有点号。除非你知道要分析的字符串中不会有例外情况出现，才可以偶尔偷一次懒。

14.3.4 选择符

选择符，即“|”，表示选择，用来匹配多个可能的正则表达式中的一个。

一个常用的例子，就是用来匹配英语动词和其过去时。

```
var foo:RegExp = /saw|see/g;

var targetStr:String = “he saw the truth. I see it.”;

trace (targetStr.match(foo));

//输出：saw，see
```

复制代码

由于选择符的优先级在正则表达式中最低，所以它把左右都当成一个整块来处理。如果有多个选择符，那么它会把由选择符隔开的表达式都当成一个整的表达式。

但这就容易带来问题，我们往往错估了选择符的作用范围。比如我们想匹配前后有空白的单词 saw 和 see。往往我们会这样写。

```
var foo:RegExp = /ssaw|sees/g;
```

复制代码

但是，实际上，选择符执行的选择是，要么“ssaw”，“sees”。这就违背了我们的本意。此时，就要使用括号来限制选择符的作用范围。正确的表达式应当这样写：

```
var foo:RegExp = /s(saw|see)s/g;
```

复制代码

关于括号的作用我们在下小节详述。

14.3.5 括号与分组

用圆括号将表达式的一部分括起来，就会将这部分表达式定义成组。有人也将组称为子表达式。组将被当成一个整体来进行操作。这样的操作，被称为分组。

分组一般是为了如下几个运用：

? 和选择符搭配使用。用来限定选择符的作用范围。例：/he(got|get)it/。

? 和限定符搭配使用。可以指定括号中的内容重复次数。例：/(get)*/。详见 14.3.6 节“用正则表达式描述字符串的重复”。

一旦命名了组，就可以通过“向后引用”调用它。详见 14.4.3 节：String 类的 replace()与正则表达式向后引用。

14.3.6 用正则表达式描述字符串的重复

描述字符重复是最常见额字符串特征描述之一。于是，正则表达式提供了一种简捷的描述方式，如表 14-4 所示。

正则表达式

字符串特征描述

能与它相匹配的字符串部分例子

/k{2}a/

K 重复 2 次，后面跟着一个字母 a

Kka,kkabc,kkkabc,xkkabc

/k{1,}ingda/

K 至少重复一次，后面跟字符串 ingda

Wkingda、kingda、lovekingda、kkkkingda

/k{1,2}ingda/

K 重复 1 次到 2 次，后面跟着字符串 ingda

Wkingda、kingda、sskkingda

与其他语言比较

ActionScript3 并不支持/k{,2}/这样的显示限定符写法。

这种使用花括号{}及其中的数字值表示模式出现次数的上下限的方式，称为显式限定符。例如，k{2}将准确匹配两个 k 字符（kk）。如果数字后跟一个逗号，如 k{4}，表示匹配任何出现次数至少为 4 的 k 字符。

除了显示限定符，正则表达式还提供了一些简便的描述方式来方便描述一些常见的重复范式，如表 14-5 所示。

表 14-5 正则表达式的描述方式

正则表达式

字符串特征描述

对应的显示限定符

/wa*/

w 后面紧跟 a，a 重复 0 或多次。匹配：w，wa，waaa，waaaaa

/wa{0,}/

/wa+/

w 后面紧跟 a，a 重复 1 或多次。匹配：wa，waaa，waaaaa

/wa{t,}/

/wa?/

w 后面紧跟 a，a 重复 0 或 1 次。匹配：w，wa

/wa{0,1}/

这些方式又称为非显示限定符。
如果限定符前面跟着的是字符集，那么就限定符合字符集描述的特征字符重复次数。下例中重复的字符集是[eo]，那么只要符合 e 或者 o 的字符重复都会被认为匹配：

```
var foo:RegExp = /g[eo]*gle/g;

var targetStr:String = “asdgle googoogle gooeoeoegle gooooooooogle geeegle”;

tce (targetStr.match(foo));

//输出：google,gooeoeoegle,gooooooooogle,geeegle
```

复制代码

如果限定符前面的是组，那么限定的就是符合组内表达式描述的特征字符串重复次数。下例中，goo 就被当成一个整体来重复。

```
var foo:RegExp = /(goo)*gle/g;

var targetStr:String = “asdgle google googoogle”;

trace (targetStr.match(foo));

//输出：gle,google,googoogle
```

复制代码

14.3.7 注意正则表达式的贪婪性和懒惰性
假设你想用一个正则表达式匹配一个 HTML 标签。当然，HTML 文件必须有效，不含无效的标签。因此，两个尖括号之间的内容，就应该是一个 HTML 标签。
许多正则表达式的新手会首先想到正则表达式/<+>/，他们会很惊讶地发现，对于测试字符串，“What is Reg Exp Greedy”，你可能期望会返回，然后继续进行匹配的时候，返回。但事实是不会的。正则表达式将会匹配“Reg Exp”。很显然这不是我们想要的结果。原因在于“＋”是贪婪的。也就是说，“＋”会导致正则表达式引擎会进行回溯。也就是说，它会放弃最后一次的“重复”，然后处理正则表达式余下的部分。引擎会进行回溯。也就是说，它会放弃最后一次的“重复”，然后处理正则表达式余下的部分。
与“＋”类似，“？*”的重复也是贪婪的。

正则表达式引擎内部原理是怎样的？让我们来看看正则引擎如何匹配前面的例子。第一个记号是“<”，这是一个文字符号。第二个符号是“.”，匹配了字符“a”，然后“＋”一直匹配其余的字符，直到一行的结束。到了换行符，匹配失败（“.”不匹配换行符）。于是引擎开始对下一个正则表达式符号进行匹配。即试图匹配“>”。到目前位置，“<＋”已经匹配了“RegExpGreedy”。引擎会试图将“>”与换行符进行匹配，结果失败了。于是引擎进行回溯。结果是现在“<＋”匹配“Reg ExpGreed”。于是引擎将“>”与“y”进行匹配。显然还是会失败。这个过程继续，直到“<＋”匹配“Reg Exp”，“>”“急切的”，所以它会急着报告所找到的第一个匹配。而不是继续回溯，即使会有更好的匹配，例如“<a href=#>”。所以我们可以看到，由于“＋”的贪婪性，使得正则表达式引擎返回了一个最左边的最长的匹配。

要想得到想要的结果，需要用懒惰性取代贪婪性。在“＋”后面紧跟一个问号“？”就可以强迫正则表达式使用的懒惰方式匹配。“*”、“{}”和“？”表示的重复都可以这样做。因此在上面的例子中我们可以使用“<＋？>”。让我们再来看看正则表达式引擎的处理过程。

正则表达式记号“<”会匹配字符串的第一个“<”。下一个正则记号是“.”。这次是一个懒惰的“＋”来重复上一个字符。这告诉正则引擎，尽可能少地重复上一个字符。因此引擎匹配“.”和字符“a”，然后用“>”匹配字符，结果失败了。引擎会进行回溯，和上一个例子不同，因为是惰性重复，所以引擎是扩展性重复而不是减少，于是“<+”现在被扩展为“<a href=#”。引擎继续匹配一个记号“>”。这次得到了一个成功匹配。引擎于是报告“<a href=#>”是一个成功的匹配。整个过程大致如此。

除了惰性扩展外，还有一个替代方案，即用一个贪婪重复与一个取反字符集：“<[^>]+>”。之所以说这是一个更好的方案，是因为在使用惰性重复时，引擎会在找到一个成功匹配前对每一个字符进行回溯。而使用取反字符集则不需要进行回溯。

再提供一个标签匹配正则表达式，供大家比较/(S*?)[^>]*>.*?|<.*?/>/

最后要记住的是，本教程仅仅谈到的是正则导向的引擎。文本导向的引擎是不回溯的。同时它们也不支持惰性重复操作。

14.3.8 用正则表达式来定位

定位符可以将一个正则表达式固定在一行的开始或结束。也可以创建只在单词内或只在单词的开始或结尾出现的正则表达式。表 14-6 包含了正则表达式及其含义的列表。

表 14-6 正则表达式及其含义

字符

描述

^

匹配输入字符串的开始位置。如果设置了正则表达式的 m 标志位，^也匹配“n”或“r”之后的位置

$

匹配输入字符串的结束位置。如果设置了正则表达式的 m 标志符，$也匹配“n”或“r”之前的位置

b

匹配一个单词边界，也就是指单词和空格间的位置

B

匹配非单词边界

单词边界就是单词和空格之间的位置。非单词边界就是其他任何位置。下面的表达式将匹配单词“Tornado”的前 3 的字符，因为它们出现在单词边界后：

/vTor/

复制代码

这里“b”操作符的位置很重要。如果它位于要匹配的字符串的开始，则将查找位于单词开头处的匹配；如果它位于该字符串的末尾，则查找位于单词结束处的匹配。例如，下面的表达式将匹配单词“Tornado”中的“do”，因为它出现在单词边界之前：

/dob/

复制代码

下面的表达式将匹配“na”，因为它位于“Tornado”中间，但不会匹配‘sina’中的‘na’：

/naB/

复制代码

这是因为在单词“Tornado”中“na”出现在非单词边界位置，而在单词“sina”中位于单词边界位置。非单词边界操作符的位置不重要，因为匹配与一个单词的开头或结尾无关。

var tor:String = “Tornado”;

```
trace (tor.match(/^Tor/));

trace (tor.match(//ado$));

trace (tor.match(/dob/));

trace (tor.match(/Bna));

var sina.String = “sina web”;

trace (sina.match(/naB/));
```

复制代码

14.3.9 正则表达式的 5 个标志位

下表列出了可以为正则表达式设置的 5 个标志符。每种标志都可以作为正则表达式对象属性进行访问，如表 14-7 所示。

表 14-7 正则表达式的 5 个标志符

g

global

匹配多个

i

ignoreCase

不区分大小写的匹配。应用于 A～Z 和 a～z 字符，但不能应用于扩展字符，É和é

m

multiline

设置此标志后，$和^可以分别匹配行的开头和结尾

s

dotall

设置此标志后，“.”（点）可以匹配换行符（n）

x

extended

允许扩展的正则表达式。你可以在正则表达式中键入空格，它将作为模式的一部分被忽略。这可使你更加清晰可读地键入正则表达式代码。

14.3.10 正则表达式元字符优先级

相同优先级的元字符从左到右进行运算，不同优先级的元字符先高后低进行运算。各种元字符的优先级从高到低排列如表 14-8 所示。

表 14-8 元字符优先级

元字符

简述

转义符

(),(?:),(?=),[]

分组定义符号，字符集定义符号

*,+,?,{n},{n,}{n.m}

限定符

^,$,anymetacharacter

位置和顺序

|

选择符

14.4 正则表达式与字符串的结合使用详解

正则表达式与字符串结合使用的方式有两类：使用 RegExp 类的两个方法，即 exec()和 test()；或使用 String 类的方法，在字符串中匹配正则表达式 match()、replace()、search()和 splice()。

14.4.1 RegExp 类的 exec()和 test()

RegExp 类的 test()方法只检查提供的目标字符串是否包含正则表达式的匹配内容。如果包含，则返回 true；不包含，则返回 false。如下列所示。

```
var target:String = “This is a test”;

var reg:RegExp = /test/;

trace (reg.test(target));
```

复制代码

RegExp 类的 exec()方法也可用来检查提供的字符串是否有正则表达式的匹配，不过它返回的是一个数组，内容包括匹配的子字符串、同正则表达式中的任意括号组匹配的子字符串。。

这个数组还有 index 属性，指明子字符串匹配起始的索引位置。

```
//查找以 t 为首字母的单词
```

var myPattern：RegExp = /bt([a-z]+)b/;

var str:String = “Ok,the theory is not good”;

var result:Array = myPattern.exec(str);

trace (result);

//输出：the,he

//the 表示匹配的字符串，he 表示表达式中括号组“([a-z]+)”匹配的子字符串

trace(result.index);

//输出：4。匹配字符串起始索引位置
复制代码

如果想多次匹配，可以加上 g 标识符。

var myPattern:RegExp = /bt([a-z]+)b/gi;

var str:String = “Ok, the theory is not good”;

trace (myPattern.exec(str),myPattern.lastIndex);

//输出：the,he 7

trace (myPattern.exec(str), myPattern.lastIndex);

//输出：theory, heory 14
复制代码

14.4.2 String 类的 search()和 match()
search()方法返回与给定模式相匹配的第一个子字符串的索引位置。match()搜索匹配的子字符串，找到了就返回匹配的字符串。如果在正则表达式模式中使用了全局标志 g，match()将返回一个包含匹配子字符串的数组。

var str:String = “Ok, the theory is not good”;

```
var patternA:RegExp = /bt([a-z]+)b/g;

trace (str.search(patternA));

//输出:4

trace (str.match(patternA));

//输出：the,theory
```

复制代码

14.4.3 String 类的 replace()与正则表达式向后引用

String 类的 replace()需要两个参数，第一个参数是传入的正则表达式，第二个是要替换的字符串。如果正则表达式没有设置 g 标识符，那么只替换一个；如果设置了 g 标识符，将替换多个。

```
var str:String = “Ok, the the the theory is not good”;

var patternA:RegExp = /bt[a-z]+b/;

trace (str.replace(patternA,”Foo”));

//输出：Ok，FOO the the the theory is not good

var pattern:RegExp = /bt[a-z]+b/g;

trace (str.replace(pattern,”FOO”));

//输出：Ok，FOO FOO FOO FOO is not good
```

复制代码

正则表达式有一个很重要的特性??向后调用，就是将匹配成功的模式的某部分进行存储，供以后使用。对一个正则表达式模式或部分模式两边添加圆括号就可以将这部分表达式存储到一个临时缓冲区中。可以使用非捕获元字符“？：”、“？＝”或“？！”来忽略对这部分正则表达式的保存。

所捕获的每个子匹配都按照在正则表达式模式中从左至右所遇到的内容存储。存储子匹配的缓冲区编号从 1 开始，连续编号直至最大 99 个子表达式。每个缓冲区都可以使用“n”访问。其中 n 为一个标识特定缓冲区的一位或两位十进制数。

向后调用的一个最简单、最有用的应用是提供了确定文字中连续出现两个相同单词的位置的

能力。请看这个句子：“Ok，we we will do do it.”，里面有多处连续单词重复现象。我们可以使用示例 14-1 所示的方法来删除重复字符。

示例 14-1 利用正则表达式消除连续两个重复单词

```
var str:String = “Ok, we we will do do it.”;

var patternA:RegExp =/b([a-z]+) 1 b/gi;

//括号中的正则表达式就会被下面的$1 所回调引用，“1”用来指定第一个子匹配

trace(str.replace(patternA, “$1”));

//输出：Ok，we will do it.
```

复制代码

再举一个常见例子：

```
var str:String = tom@gmail.com kingda@kingda.org;

var pattern:RegExp = /(w*)@(w*.[org|com]+)/g;

trace (str.replace(pattern, “Hi, $1，你的 E-mail 在$2 上 n”));

/*输出：

Hi，tom，你的 E-mail 在 gmail.com 上

Hi，kingda，你的 E-mail 在 kingda.org 上

*/
```

复制代码

14.5 常用的正则表达式

本节提供了许多常用的正则表达式，一来可以为大家的实际编程提供帮助，二来也可以为大家提供练习正则表达式的机会。比如大家可以看左边正则表达式的目标，自己动手写一写，验证验证，看看自己能否写出同样效果的正则表达式，然后再和右边的标准解决方式对比一下，这样形学习的效果更好。

14.5.1 文本处理和输入限制常用正则表达式

文本处理和输入限制常用正则表达式（见表 14-9）。

表 14-9 文本处理和输入限制常用正则表达式

要匹配的字符串或字符集

正则表达式

中文字符

[u4e00-u9fa5]

双字节字符

[^x00-xff]

全角字符

[^uFF00-uFFFF]

空白行

ns*r 或 n[s|]*r

运用在删除文本空白行时

首尾空白字符

^s*|s*$

可以用来删除行首尾的空白字符（包括空格、制表符、换行符等），非常有用的表达式。如写 trim 函数：

```
function trim(targetStr:String):String
{
return targetStr.replace()
}
```

由 26 个英文字母组成的字符串

^[A-Za-z]+$

由 26 个英文字母的大写组成的字符串

^[A-Z]+$

由 26 个英文字母的小写组成的字符串

^[a-z]+$

由数字、26 个英文字母或下划线组成的字符串

^w+$

验证密码是否安全

/^(([A-Z]*[a-z]*|d*|[-_~!@#$%^&*.()[]{}?\/'"]*)|.{0,5})$|s/

货币数字

/^d+(.d+)14.5.2 网络和 HTML 代码方面的应用

要匹配的字符串或字符集

正则表达式

HTML 标记

<(S*?)[^>]*>.*?|<.*?/>（注意转义）

网上流传的版本太糟糕，上面这个也仅仅能匹配部分，对于复杂的嵌套标记依旧无能为力

网络连接

(h|H)(r|R)(e|E)(f|F) *= *('|")?(w|\|/|.)+('|"| *|>)?

用于提取网页中的链接

图片连接

(s|S)(r|R)(c|C) *= *('|")?(w|\|/|.)+('|" *|>)?

E-mail 地址

w+([-+.]w+)[email=*@w+(%5b-.%5dw+)*.w+(%5b-.%5dw+)*]*@w+([-.]w+)*.w+([-.]w+)*[/email]

表单验证时很实用

URL

[a-zA-z]+://[^s]*

网上流传的版本功能很有限，上面这个基本可以满足需求

[url=http://([/w-%5d+/.)+%5bw-%5d+(/%5b/w-./?%25&=%5d*)]http://([w-]+.)+[w-]+(/[w-./?%&=]*)[/url]?

这个专门用来解析 HTTP 地址，分了组。不过，如果自己有具体应用应当写得更加具体

IP 地址

d+.d+.d+

提取 IP 地址时有用。如将 IP 地址转换成对应数值

```
function IP2V(ip:String):uint{
var re:RegExp = /(d+)(d+).(d+).(d+)/g;
if (re.test(ip)){

var tmp:Array = ip.split(“.”);

return int(tmp[0])*Math.pow(255,3)+int(tmp[1]*Math.pow(255,2)+ int(tmp[2]*255+int[3]);
}else{

throw new Error(“Not a valid IP address!”);
}
}
```

14.5.3 表单验证

表单验证（见表 14-11）

表 14-11 表单验证常用正则表达式

要验证是否合法的字符串

正则表达式

账号

^[a-zA-Z][a-zA-Z0-9_]{4,15}$
以字母开头，允许占用 5～16 字节，允许带有字母、数字和下划线

国内电话号码

d{3}-d{8}|d{4}-d{7}
例：0571-28881088

中国电话号码（包括移动和固定电话）

((d{3,4})|d{3,4}-|s)?d{7,14}

国际电话号码

/^((d{2,3}))|(d{3}-))?((0d{2,3}-)?[1-9]d{6,7}(-d{1,4})?$/

腾讯 QQ

[1-9][0-9]{4,}
腾讯 QQ 号从 10000 开始

中国邮政编码

[1-9]d{5}(?!d)
中国邮政编码为 6 位数字

中国的身份证

d{15}|d{18}

中国的居民身份证号为 15 位或 18 位

身份证这个表达式过于简单，因为出生日期那几位都有限制的

14.5.4 匹配数字

匹配数字（见表 14-12）

要匹配的特殊数字

正则表达式

正整数

?$/

^[1-9]d*$

负整数

^-[1-9]d*$

整数

^-?[1-9]d*$

非负整数

^[1-9]d*|0$

正整数＋0

非正整数

^-[1-9]d*|0$
负整数＋0

正浮点数

^[1-9]d*.d*|0.d*[1-9]d*$

付浮点数

^-[1-9]d*.d*|0.d*[1-9]d*$

浮点数

^?[1-9]d*.d*|0.d*[1-9]d*|0?.0+|0)$

非负浮点数

^[1-9]d*.d*|0.d*[1-9]d*|0?.0+|0$
正浮点数＋0

非正浮点数

^(-([1-9]d*.d*|0.d*[1-9]d*))|0?.0+|0$
负浮点数＋0

注：处理大量数据时有用，具体应用时注意修正

14.6 本章小结

本章讲述了正则表达式的设计原因、思想和重要概念，介绍了正则表达式的语法，并给出了许多具体代码实例。正则表达式与字符串结合后威力巨大，请读者用心掌握。

14.5 节“常用正则表达式”提供了几十个实用的正则表达式，供读者学习和在实际编程中使用。

表 14-10 网络和 HTML 代码方面常用的正则表达式

第十五章 XML 数据处理

如今，XML 在网络和其他领域都有广泛的应用和迅猛的发展。XML 语言已经成为数据格式上的事实标准。作为一名开发人员，应将 XML 语言视为亲密的朋友和得力的助手。

如果要给出放弃 ActionScript2，选择 ActionScript3 的 3 个理由，那么 ActionScript3 对 XML 的近乎完美的支持绝对是其中一个。

ActionScript3 在处理 XML 方面有了长足的进步。由于采用了 E4X[参考文献 2]标准，对 XML 的支持和操作非常方便直观，远远优于传统的 XML DOM，XPath。这是所有 ActionScript3 开发者的福音。

本章导读

ActionScript3 中在 XML 数据处理方面实现了 E4X，是语言的几大特色之一。ActionScript3 对 XML 的操作，不论相对于其他 OOP 语言，还是相对于 ActionScript2 都有了本质的变化。因此，建议所有级别的读者认真阅读本章。

对于初学者，尤其是对于 XML 还不熟悉的读者，15.1 节、15.2 节应仔细阅读。至于 15.7 节可以暂时跳过不读。

对于 ActionScript2 和其他语言读者，如果对 XML 已经很熟悉，15.1、15.2 可以跳过不看。

15.1 XML 的概览

近年来，XML 发展得非常迅速，速度让人惊叹。目前，大部分的软件开发商都采用了 XML 标准。XML 已经成为了事实上的 Web 技术基础。XML 将成为最普遍的数据操纵和数据传输的工具，跨越网络和各种各样的客户端。

XML 究竟是何方神圣？

XML 是一种用来描述数据的语言，是 Extensible Markup Language 的缩写，译为可扩展标记语言。XML 主要关注两点：什么是数据，以及如何存放数据。

那么，这样一种看起来如此强大的语言是否会极其难学呢？答案恰恰相反，非常容易上手。在这之前，我们一起先看看 XML 的来由和简介。

在 XML 出现之前，有成千上万种描述数据的方法。举个例子，您想描述一个网站的名称和 URL，那么可以用 CSV 格式来描述它，见下例。

```
name,url

Kingda's Blog,http://www.kingda.org/

ActionScript3,http://www.actionscript3.cn/
```

复制代码

或者干脆用制表符来分隔数据描述，见下例。

```
name,url
```

Kingda’s Blog http://www.kingda.org/

ActionScript3 http://www.actionscript3.cn/

复制代码

或者你可以用自己创造的方式来描述它，比如下面这种独创的格式。

name: Kingda’s Blog, url: http://www.kingda.org/

name: ActionScript3, url: http://www.actionscript3.cn/

复制代码

就这样，千奇百怪的数据描述格式出现了。不要说跨操作系统的数据会碰到这种数据描述上的转换，甚至连一个应用程序中的不同部分对数据描述也不统一。可想而知，这对整个数据世界的信息交换带来了多么巨大的困难。

XML 语言出现后，以它自由、灵活、易于扩展的特点迅速捕获了所有开发者的心。软件厂商、自由软件全部迅速向它靠拢。那么我们来看看，XML 究竟是什么样的格式，有什么样的优点，会有如此勾人的魅力。

15.2 XML 简要介绍

XML 是一种类似于 HTML 的标记语言，但和 HTML 关注额方向不同。XML 是被设计用来描述数据的，重点是：什么是数据，如何存放数据。HTML 是被设计用来显示数据的，重点是：显示数据及如何更好地显示数据。要注意，XML 不是 HTML 的代替品。

实际上，我们可以把 XML 理解成是一种跨平台的，与软、硬件无关的，专注于处理信息数据描述的工具。

用例子来理解 XML 是最直观的了。我们来看一下，如何使用 XML 来描述我们在上一节中说过的网站数据描述的例子。

15.2.1 一个简单的 XML 示例

示例 15-1 是关于 XML 的例子。

示例 15-1 一个简单的 XML 示例

```
<?xml version=”1.0” encoding=”gbk”?>

<websites>

<site>

<name>Kingda’s blog </name>
```

<url>http://www.kingda.org/</url>

</site>

<site>

<name>ActionScript 3</name>

<url>http://www.actionscript3.cn/</url>

</site>

</websites>

复制代码

看起来，XML 没有什么特别的地方，只是一些用尖括号扩在一起的普通的纯文本。事实上就是如此。XML 其实很平易近人。

第一行，一般都要放置 XML 声明：定义此文档遵循的 XML 标准的版本。在这个例子里是 1.0 版本的标准，使用的是 gbk 字符集。如果 XML 文档中包含非英文的字符，比如简体中文或繁体中文、日文、韩文等，那么推荐使用 UTF-9 字符集，否则，可以略去 encoding 这一栏。

我们可以看到，从第二行开始有成对的尖括号标记出现，如<website></website>、<site></site>、<name></name>。我们把前面的称为开始标记，把后面的含有斜杠的标记称为结束标记。每一对标记，我们称为一个元素（element）。元素里面可以有子元素，比如 name、url 都是 site 的子元素。

例子的第二行告诉我们，这个文档的根元素是什么。在本例中，根元素是 website，这就是表明本文档是用来描述网站的数据。第二行是根元素的开始标记，最后一行就是根元素的结束标记。而中间的两个 site 元素，又可以称为根元素的两个子节点。

要注意，XML 是自描述的语言，非常自由。所有的标记都是由我们自己决定的。你可以随意改变元素的名字，只要符号 XML 语法的合法字符即可。比如，我们可以将根元素改成<web></web>，甚至中文<网站></网站>都可以。

15.2.2 XML 简明语法

XML 的语法也非常简单、直白，所以，下面轻轻松松地看一下 XML 简要语法。对于 Web 前段的 XML 应用来说，一般只要掌握以下 8 条语法就可以应付绝大部分的运用了。

第一条：XML 文档必须有一个根元素，而且只能有一个。

第二条：开始标记和结束标记必须成对出现。

当一个节点为空时，可以简写。比如<name></name>可以简写成<name/>

第三条：所有的 XML 元素必须合理嵌套。

意思是，父元素要清楚包含子元素。比如上例中，如果 site 节点写成如下代码就是不清楚、不合法的嵌套。

```
//不合法的 XML

<site>

<name>ActionScript3</name>

<url>http://www.actionscript3.cn/</site>

</url>
```

复制代码

第四条：XML 标记都是大小写敏感的。比如<name></Name>就不是一个合法的元素。

第五条：XML 元素命名必须遵守下面的规则。

? 元素的名字可以包含字母、数字和其他字符。

? 元素的名字不能以数字或者标点符号开头。

? 元素的名字不能以 XML（或者 xml、Xml、xMl……）开头。

? 元素的名字不能包含空格。

自己创造的 XML 元素还必须注意下面一些简单的规则。

任何的名字都可以使用，没有保留关键字（除了 XML），但是应该使元素的名字具有可读性，名字使用下画线是一个不错的选择。

在 XML 元素命名中不要使用“：”，这是属于 XML 命名空间要用到的特殊字符。

可以使用中文命名，但是考虑到和不同软件的交互可能带来的困难，推荐使用英文命名。

第六条：每个元素可以带有自己的属性。属性值必须使用引号””标识。

比如，我们描述一个苹果元素，有一个属性 color（颜色），那么应该这样写：

```
<apple color=”red” />
```

复制代码

第七条：XML 注释写法：<!?注释内容-->。

第八条：在 XML 文档中的所有文本都会被解析器解析。只有在 CDATA 部件之内的文本会被解析器忽略。

CDATA 部件是什么？有什么用？当一个 XML 文本被解析时，所有的特殊符号都会被解析，以便形成数据对象。但下面 5 种符号都是被 XML 语法保留的：<（小于号）、>（大于号）、&（和）、’（单引号）和”（双引号）。如果我们的节点文本文本中包括这些符号，就可能导致整个 XML 文本解析出错。比如，“示例 15-1”是关于 XML 的例子。

示例 15-1 一个简单的 XML 示例 ”中，如果第一个 site 元素中 name 节点的文本是

Kingda’s<Blog>，就会导致 XML 解析器误认为 Blog 是一个节点开始，这就比较麻烦了。而且如果你要节点文本中写入 HTML 代码、程序代码，或者数学公式，那么这种情况常常出现。

怎么办？XML 为我们提供了方便的 CDATA 部件来解决这类问题。

一个 CDATA 部件以“<![CDATA[”标记开始，以“]]”标记结束，放在节点中。如果在这两个标记之间的文本包含上述特殊符号，那么这些符号都不会被特殊对待，而看成普通文本符号，从而不会导致 XML 解析器误判。

比如，刚刚说的 Kingda’s<Blog>这样的文本就可以写成示例 15-2。

示例 15-2 CDATA 部件的示例

```
<site>

<name>

<![CDATA[Kingda’s <Blog]]>

</name>

<url>http://www.kingda.org/</url>

</site>
```

复制代码

15.2.3 如何编写一个结构优秀的 XML

一个符合 XML 语法的 XML，是一个格式良好有效（well-formed）的 XML。“良好有效”并不等同于它对数据的描述是准确和灵活的，也并不意味着它的描述对于数据解析和处理是友好的。

我们先看一个格式良好的 XML，但其对数据描述却非常糟糕的例子，见示例 15-3。

示例 15-3 结构糟糕的 XML 示例

```
<?xml version=”1.0” encoding=”gbk”?>

<websites>

<url name=”Kingda’s blog”>
```

http://www.kingda.org/

</url>

<web>

<item name=”ActionScript 3”/>

<item name=”http://www.actionscript3.cn/”>

</web>

</websites>

复制代码

虽然这个例子是符合 XML 语法的，也完整描述了该描述的数据，比如网站的 url 和 name，但是结构是非常糟糕的。同一类的数据用了两种元素（url 和 web），并且每个元素中的描述方式也是不同的：url 元素中采用了属性和文本节点的混合，web 元素中采用了两个 item 子元素，而且，item 子元素的属性还是不同的。这样的 XML 如果多了，对于开发人员来说简直就是一场灾难。

如何让一个 XML 结构清晰、优秀，对程序友好是一个比较大的课题，本书不做深入讨论。但对于普通的 XML 运用，按照笔者的经验，一个结构优秀的 XML 应当有如下几个特征。

元素分类清楚，同类元素标识一致，内部结构一致。

父子元素逻辑关系清楚。

不仅考虑目前现有的数据要求，还要考虑到未来可能的数据内容变动。

何时用属性，何时用子元素，使用方式在整个文档中保持一致。

一般来说，如果一个数据元素包括多个同名的项目，那么子元素描述它。如果有大段的文本，尤其是包括空白字符或特殊字符时，子元素。其余时候尽量使用属性描述，降低 XML 层次数目。

所以，示例 15-3 结构糟糕的 XML 示例可以用示例 15-4 更好地描述出来。

示例 15-4 结构简明易于扩展的 XML 示例

当前数据内容的示例

<?xml version=”1.0 ” encoding=”gbk”?>

<websites>

<site name=”Kingda’s blog” url=http://www.kingda.org/ />

<site name=”ActionScript 3” url=http://www.actionscipt3.cn />

</websites>

考虑到今后数据内容的扩展，比如加入网站描述，应尽量少地改变现有的 XML 结构，如下例。

<?xml version= “1.0” encoding=”gbk”?>

<websites>

<site name=”Kingda’s blog” url=”http://www.kingda.org/”>

<description>

<![CDATA[Kingda’s Blog!Welcome to visit]]>

</description>

</site>

<site name=”ActionScript 3” url=http://www.actionscript3.cn/ />

<description>

<![CDATA[ActionScript 3 Chinese Forum. Wonderful resource!]]>

</description>

</site>

</websites>

复制代码

XML 的简明介绍就到此为止了。既然 XML 如此灵活有效，我们来看一下 ActionScript3 是如何重视 XML，如何支持 XML 数据处理的。

15.3 XML 在 ActionScript3 中的地位

XML 非常重要，几乎涵盖了 ActionScript3 程序开发的所有方面，就连 ActionScript3 语言本身也随处可见 XML 的运用，比如 flash.utils 包中对类对象（Class Object）结构的描述就使用了 XML，而 Flex 的 MXML 语言就是基于 XML 而建造的。我们即将要碰到的 Webservice 和诸多协议也全部基于 XML 的。可见 XML 与我们联系之紧密。

在 ActionScript3 中，对 XML 数据的支持怎样呢？在 ActionScript3 中，XML 数据类型已经成功为了 ActionScript3 的内置数据类型，再也不需要使用解析 API 来处理了。XML 对象有自己的运算符，操作起来简单、自然、方便。

这是怎么回事呢？通过顶级的 XML 类和 XMLList 类来实现。这两个类是 ActionScript3 的核心类，放在顶级包中。关于 XML 命名空间部分，则是由 QName 和 Namespace 联合实现的。

那么，这些激动人心的功能是由谁带来的呢？让我们感谢 E4X 吧。

15.3.1 E4X：更好的 XML 处理规范标准

在 E4X 之前（如果以 ActionScript 语言为例，那就是在 ActionScript3 之前），我们和 XML 打交道是很痛苦的。面对的是没完没了的 childNodes、firstChild、nodeValue 这些枯燥的东西。代码可读性差，操作起来极其不方便。在 XML 和程序可用的 Object 之间的转换，要花费大量的时间。虽然后来出现了 mx.xpath.XPathAPI，给我们带来了些许方便，然而并不能根本解决这些问题。而且 XPathAPI 的执行效率也不敢恭维，每次还要在 Fla 中塞进一个数据绑定组件（笔者就经常忘掉，而导致浪费 5～10 分钟不等的时间）。痛苦的当然不止我们 ActionScript 开发人员，其他的语言也差不多。

E4X 是我们的救星，将我们从重复的体力劳动中解救出来。E4X 全称是 ECMAScript for XML 规范，是一套标准。它包括了处理 XML 数据的一系列的类、功能、运算符的规范，从而使得支持这个规范的语言处理 XML 数据时，就像语言原生数据类型一样。Adobe 官方归纳了 3 个优点。

简易：包括代码编写和代码可读性。读者会发现 ActionScript3 中操作 XML 的代码浅白流畅。

统一：其各个方法的设计及背后的逻辑与 ActionScript3 语言其余部分思想一致，易于理解。

熟悉：使用点号（.）来操作 XML 数据，非常容易上手。

E4X[参考文献 2]，无可置疑，是目前最先进的 XML 规范。除了 ActionScript3 已经率先支持它之外，JavaScript2，以及各大浏览器（FireFox 新版本已经支持）都将支持这一标准。其他主流编程语言也都会逐渐支持这一规范。

那么，赶快进入下一节看看在 ActionScript3 中爽快、流畅地操作 XML 吧。在这之前，我们先说说如何将以前的 ActionScript2 方面的 XML 处理部分移植到 ActionScript3 中。

15.3.2 ActionScript2 程序移植时，关于 XML 部分该如何处理

目前使用 ActionScript2 开发的项目已经有一定规模，如果将现有的项目移植到 ActionScript3 中碰到的主要问题之一就是 XML 相关处理。

ActionScript3 保留了之前版本中对 XML 处理的 API，将它们移植到了 flash.xml 包中。原有的 ActionScript2 中的 XML 类，改名为 XMLDocument。其余 XMLNode、XMLParser 和 XMLTag 也都被保留在这个包中以便保持兼容性。

因此，只要做一些导入包和类名变动就可以使老项目中关于 XML 的部分在 ActionScript3 中继续使用，

下面进入正题，如何使用 ActionScript3 处理 XML 数据。

15.4 创建 XML 对象

创建 XML 对象有两种方式：一种是使用构造函数；另外一种是使用简洁明快的 XML 文本直接创建方式。

在一般程序开发中，XML 数据都是从外部读入的，一开始都是字符串形式。那么这时候需要将字符串传入 XML 构造函数。生成 XML 对象。

如果需要在程序中直接写入 XML 文本，那么使用第二种方式最方便。这也是 ActionScript3 的一大创新，可以在代码中直接写入多行 XML 文本。

下面来分别看看这两种方式的运用。

15.4.1 使用构造函数创建 XML 对象

可以传入 XML 类的构造函数并生成 XML 对象的数据类型，不仅仅是 String 类型。任何可以被顶级函数 XML()转换的数据类型都可以作为参数传入 XML 构造函数。

但在实际运用中，绝大多数都是将字符串转换成 XML 对象。例子如下。

```
var foo:String = “<book><name>AS3 殿堂之路</name><pages>500</pages></book>”;

var fooXML:XML = new XML (foo);

trace (fooXML.name);
```

复制代码

15.4.2 直接使用 XML 文本创建 XML 对象

在 ActionScript3 中，在代码中直接使用 XML 文本创建 XML 对象是一种享受。与之相比，在 ActionScript2 时代，在代码中输入 XML 文本是一种痛苦。要么忍受一长串挤在一块的字符串，要么手工一行行地拼接。而在 ActionScript3 中，要简单的多，直接按照 XML 的内容输入即可，想换行就换行，向使用[Tab]键就使用[Tab]键。编译时会自动忽略这些空白。这样就在保证了 XML 美观的同时，也增加了代码的可读性。

如示例 15-5 所示，我们在代码中直接输入一个 XML 的内容，并且使用换行和制表符将 XML 调整地干干净净、清清楚楚。

示例 15-5 直接创建 XML 对象

```
var kingdaXML:XML =

<websites>
```

```
<site name = "Kingda's blog" url = "http://www.kingda.org/">

<discription>

<![CDATA[Kingda's Blog!<b>Welcome to visit</b>]]>

</discription>

</site>

<site name="ActionScript 3" url="http://www.actionscript3.cn/">

<discription>

<![CDATA[ActionScript 3 Chinese Forum.<b>Wonderful resource!</b>]]>

</discription>

</site>

</websites>
```

复制代码

运行代码时，在最后一句 trace()之前的 XML 文本形式就已经被 AVM2 理解成了 XML 对象了。所以，在最后一句 trace()中，我们马上就可以用“.”操作符来访问相应的子元素和属性。本例中访问了 site 节点下 description 的值。输出结果就是第一个子元素 site 下 description 的文本。

大家应该注意到了，居然可以直接用点号（.）来访问 XML 对象的子元素，如同使用点号访问一个 Object 对象的子对象。那么如何访问属性？使用点号又有那些要注意的地方呢？在下一节中介绍点号操作符。

另外，要注意一点，ActionScript3 中的 XML 还可以通过“{}”使用已有的变量来直接构造 XML。使用要点就是要把变量用“{}”括起来。如果设置属性，则不要再加引号了。

```
var rootnNodeName:String = “site”;

var subNodeName:String = “orgin”;

var subNodeContect : String = “Kingda’s Blog”;

var attributeName : String = “url”;
```

```
var attributeValue : String = http://www.kingda.org;

var extXML:XML =

<{rootNodeName} {attributeName}={attributeValue}>

<{subNodeName}>{subNodeContent}</{subNodeName}>

</{rootNodeName}>;

trace (extXML.toXMLString());

/*output

<site url="http://www.kingda.org">

<orgin>Kingda's Blog</orgin>

</site>

*/
```

复制代码

15.5 使用运算符操作 XML 数据

在这一节中讲述如何使用运算符（点号“.”，属性标识符运算符“@”，后裔运算符“..”，通配符“*”）进行常用的 4 类 XML 数据操作。

? XML 对象的创建

? XML 子元素和属性的访问

? 子元素和属性的修改和增删

? 查找 XML 子元素或属性

读者会发现自己读本节的速度会相当快，因为在 ActionScript3 中操作 XML 就是这么简单、爽快和流畅。

15.5.1 使用“.”与“@”直接访问、修改、创建和删除 XML 对象

操作 XML 数据，经常需要我们访问我们访问其子元素或者属性。在 ActionScript2 中，有种解决办法是先将 XML 对象转换成一个和 XML 内容结构像素的 Object 对象，然后再来访问这个 Object 对象。这样非常不方便。

而在上例中可以看出，ActionScript3 已经帮我们完成了这样一步工作，我们可以直接使用“.”号加子元素的名字来访问。那么，ActionScript3 是如何做到这一点的呢？

简单地说，一旦我们生成一个 XML 对象后，我们可以将它看做一个与 XML 文档结构相同

的对象。使用点号来访问各个层级的子元素，使用@号来访问各个元素的属性。同名的子元素成为了一个 XMLList 对象，是 XML 对象的一个集合，可以使用数组访问符“[]”访问。详细的原因和技术细节请见“XML 对象与 XMLList”，

我们接着示例 15-5，继续讲一个例子：访问子元素和属性，见示例 15-6。

示例 15-6 使用运算符访问 XML 子元素和属性

```
var kingdaXML:XML =
<websites>
<site name="Kingda’s blog" url="http://www.kingda.org/">
<pageview>100000</pageview>
</site>
<site name="ActionScript 3" url="http://www.actionscript3.cn/"/>
</websites>
//访问 site 下面的 pageview
trace (kingdaXML.site.pageview);
/*
输出：100000
*/
trace (kingdaXML.site);
/*
输出：
<site name="Kingda’s blog" url="http://www.kingda.org/">
```

```
<pageview>100000</pageview>

</site>

<site name="ActionScript 3" url="http://www.actionscript3.cn/"/>

*/

trace (kingdaXML.site[0]);

/*

输出：

<site name="Kingda’s blog" url="http://www.kingda.org/">

<pageview>100000</pageview>

</site>

*/

trace (kingdaXML.site[1]);

/*

输出为空，因为第二个 site 子元素没有子元素，是一个空节点，所以为空

*/

trace (kingdaXML.site.@name);

/*

由于两个子元素都有 name 属性值，所以两个都输出：

Kingda’s blogActionScript 3

*/

trace (kingdaXML.site.@name[0]);
```

```
/*

输出第一个子元素 name 属性值：

Kingda's blog

*/
```

复制代码

不需要我们写什么 childNodes，也不需要什么 Xpath，如此明白爽快的访问让我们感到一种翻身解放的感觉。而且不仅如此，我们还可以直接用这两个运算符修改子元素的文本节点和属性。

15.5.2 使用“.”与“@”直接修改 XML 对象

本节，我们继续体验如何用运算符直接修改子元素和属性。在修改时，我们必须时刻记住，只有单个的元素才可以使用“.”和“@”来修改；如果有同名的子元素存在，则返回的对象是 XMLList 对象，而不是单个的 XML 对象，此时不能使用“.”和“@”来修改，会发生运行时错误。

详细的原因和技术细节请见“XML 对象与 XMLList”。

先看一个正确的例子（上接示例 15-6），见示例 15-7。

示例 15-7 使用运算符修改 XML 对象

```
kingdaXML.site[0].pageview = 200000;

trace (KingdaXML.site[0]);

/*

输出：

<site name = "Kingda's blog" url = "http://www.kingda.org/">

<pageview>200000</pageview>

</site>

*/
```

```
kingdaXML.site[0].@name = "Kingda's first blog";

trace (kingdaXML.site[0]);

/*

输出：

<site name="Kingda's first blog" url="http://www.kingda.org/">

<pageview>200000</pageview>

</site>

*/
```

复制代码

再看一个比较容易的错误。

```
trace (kingdaXML.site.pageview.length());//输出：1

kingdaXML.site.pageview = 324;

/*

运行时报错：

TypeError:Error #1089:不支持对包含多个项目的列表进行赋值。

*/
```

复制代码

由于 kingdaXML 中只有一个子元素含有 pageview，所以容易想当然地认为直接写 kingdaXML.site.pageview 就可以访问到这个单个的子元素。但是实际上发生的情况却是运行时报错。

这是因为，虽然第二个 site 子元素看起来没有 pageview，但是当我们使用 kingdaXML.site.pageview 访问时，AVM2 实际上生成了一个 XMLList 对象，而不是 XML 对象。对 XMLList 对象使用运算符，当然会报错。所以，请记住，一旦发现这的报错时，就应该知道是由这个原因引起的。正确的修改如下。

kingdaXML.site.pageview[0] = 324;

复制代码

15.5.3 使用“.”与“@”直接添加和删除子元素和属性

XML 类本身就是一个动态类。在一个 XML 对象上添加子元素和属性，就像我们给一个 Object 实例添加属性一样简单。

请看下面的例子（上例示例 15-7），见示例 15-8

示例 15-8 用运算符添加子元素和属性

```
//添加一个空白节点

kingdaXML.site[0].author = new XML();

//以字符串添加一个子节点

kingdaXML.site[0].location = "Hangzhou, China";

//用数组添加一个子节点

kingdaXML.site[0].weekvisit = [1000,2000,3000, 4000];

//添加一个属性

kingdaXML.site[0].@language = "Chinese";

trace (kingdaXML.site[0]);

/*

输出：

<site name="Kingda’s blog" url="http://www.kingda.org/" language="Chinese">

<pageview>100000</pageview>

<author/>

<location>Hangzhou, China</location>
```

```
<weekvisit>1000,2000,3000,4000</weekvisit>

</site>

*/
```

复制代码

输出一个子元素或属性，我们可以使用 delete 关键字。接着示例 15-8，我们把刚刚创建的子元素和属性删除掉，见示例 15-9。

示例 15-9　用运算符删除子元素和属性

```
delete kingdaXML.site[0].author;

delete kingdaXML.site[0].location;

delete kingdaXML.site[0].weekvisit;

delete kingdaXML.site[0].@language;

trace (kingdaXML.site[0]);

/*

输出：

<site name="Kingda's blog" url="http://www.kingda.org/">

<pageview>100000</pageview>

</site>

*/
```

复制代码

15.5.4 简单方便地搜索 XML 子元素和属性

使用 XML，经常要对 XML 进行遍历搜索，以便找到符合搜索条件的子元素。在 ActionScript3 中，这一切变得无比简单。

在本节的示例中展示了如下 4 种用法。

? 如何使用双点号（..），又称为后裔访问符（the descendent accessor），访问当前元素

的所有子元素。

? 如何使用通配符（*）访问。

? 如何使用表达式定制查找子元素的条件。

? 如何使用表达式制定查找属性的条件。

具体代码见示例 15-10。

示例 15-10 常用的搜索 XML 对象方法

```
var kingdaXML:XML =

<websites>

<site name="Kingda’s blog" url="http://www.kingda.org/">

<pageview>150000</pageview>

<child name="Kingda’s flash blog" url="http://www.kingda.org/blog/">

<pageview>100000</pageview>

</child>

</site>

<site name="ActionScript 3" url="http://www.actionscript3.cn/">

<pageview>50000</pageview>

</site>

</websites>

//用法 1：访问所有名为 pageview 的子节点，包括不同层级

trace (kingdaXML..pageview);

/*

输出：
```

<pageview>150000</pageview>

<pageview>100000</pageview>

<pageview>50000</pageview>

*/

//用法 2：查找 site 子元素的属性，返回一个 XMLList 对象，并用 toXMLString()方法输出

trace (kingdaXML.site.@*.toXMLString());

/*

输出：

Kingda’s blog

http://www.kingda.org/

ActionScript 3

http://www.actionscript3.cn/

*/

//用法 3：查找值小于 100 000 的 pageview 子元素

//注意，这里 100 000 已经被自动转换成为值类型了

trace (kingdaXML.site.(pageview <100000));

/*

输出：

<site name="ActionScript 3" url="http://www.actionscript3.cn/">

<pageview>50000</pageview>

</site>

```
*/

//

trace (kingdaXML.site.(@name == "ActionScript 3"));

/*

输出：

<site name="ActionScript 3" url="http://www.actionscript3.cn/">

<pageview>50000</pageview>

</site>

*/
```

复制代码

除了这些基本的方法之外，ActionScript3 还提供给我们更多的惊喜。想想看，如果我们可以自己指定条件验证函数，或者使用正则表达式来设定条件，那是多么美好！ActionScript3 满足了我们的这个奢侈的要求。

15.5.5 *使用正则表达式或自定义函数搜索 XML 子元素和属性

在上一节中我们提到了，可以指定一个表达式来制定对子元素和属性的搜索条件。如果我们将其替换成自定义的函数，那么无疑会带来我们极大的自由。

笔者还发现，表达式中可以直接使用字符串的 match 等函数。这样，正则表达式在 XML 中使用就容易了。

请看示例 15-11 所示代码（上接示例 15-10）。

示例 15-11 使用自定义函数或正则表达式来搜索 XML 对象

```
trace (kingdaXML.site.(check(pageview)));

/*

输出：
```

```
<site name="ActionScript 3" url="http://www.actionscript3.cn/">

<pageview>50000</pageview>

</site>

*/

trace (kingdaXML.site.(pageview.match(/^5d+/)));

/*

输出：

<site name="ActionScript 3" url="http://www.actionscript3.cn/">

<pageview>50000</pageview>

</site>

*/

function check(t:*):Boolean {

trace (t is XMLList);

trace (t.length());

if (t <100000) return true;

return false;
```

复制代码

用法 1 使用了自定义的函数 check。我们会发现实际上 check 执行了两次，因为有两个 site 子元素。使用自定义的函数好处很多：一，可以将复杂的业务逻辑写入进去；二，增加代码的灵活性和健壮性。要注意的是，自定义的函数必须返回 Boolean 值才可以。

用法 2 使用了正则表达式，本例中正则表达式“/^5d+/”的意思是以 5 开头的数字。那么在 site 子元素中符合条件的只有第二个，所以反悔了第二个 site 子元素。正则表达式的强大众所周知，可以如此灵活地嵌入到 XML 搜索中，真的让笔者喜出望外。

好了，到此所有 XML 常用的操作都已经基本讲述完毕。下面讲解 ActionScript3 中 XML 的

高级应用部分和实现原理。

15.6 用 API 实现的一些常用 XML 操作

ActionScript3 对 XML 对象的操作非常全面。光 XML 对象拥有的实例方法就有 42 个。本书限于篇幅，仅介绍 API 中一些常用的 XML 操作。

使用运算符本身也可以创建、组合或改变 XML 对象。但 API 提供的一些功能可以让我们更加精确地控制添加位置。

? appendChild(child:Object):XML:在当前子元素列表之后添加。

? prependChild(value:Object):XML:在当前子元素列表之前添加。

? insertChildAfter(child1:Object,child2:Object):*：在子元素 child1 之后添加。

? insertChildBefore(child1:Object,child2:Object):*：在子元素 child1 之前添加。

代码如示例 15-12 所示。

示例 15-12 用 API 控制插入 XML 子元素的位置

```
var foo:XML = <data/>;

foo.appendChild(<d>ddd</d>);

trace (foo);

/*

<data>

<d>ddd</d>

</data>

*/

foo.prependChild(<b>bbb</b>);

trace (foo);

/*输出

<data>

<b>bbb</b>
```

```
<d>ddd</d>

</data>

*/

var cNode:XML = <c>ccc</c>;

foo.insertChildAfter(foo.b, cNode);

trace (foo);

/*输出

<data>

<b>bbb</b>

<c>ccc</c>

<d>ddd</d>

</data>

*/

var aNode:XML = <a>aaa</a>;

foo.insertChildBefore(foo.b, aNode);

trace (foo);

/*输出

<data>

<a>aaa</a>

<b>bbb</b>

<c>ccc</c>
```

```
<d>ddd</d>

</data>

*/
```

复制代码

除此之外，还有一些较常用的 API，比如：

? contains()对比该 XML 对象与给定 value 参数。

? elements()列出某 XML 对象的元素。

平时，XML 是忽略 XML 注释和 XML 指令的，如果不想忽视，可以使用下述语句打开。

```
XML.ignoreComments = false;

XML.ignoreProcessingInstructions = false;
```

复制代码

打开这两个开关后，就可以使用 children()方法按顺序返回所有子项，包括注释和指令。也可以使用 comments()方法得到含有注释属性的 XMLList 对象。

15.7 *XML 的命名空间

XML 类包含用于处理命名空间的一些方法：addNamespace ()、removeNamespace ()、localName ()、name ()、namespace ()、inScopeNamespaces ()、namespaceDeclarations ()、setLocalName ()、setName ()和 setNamespace ()。

其中 localName ()、name ()和 setName ()方法的用法如下。

```
var xml1:XML = <data xmlns=”http://www.kingda.org/”>12345</data>

trace (xml1.name().localName);

//输出：data

trace (xml1.name().uri);

//输出：http://www.kingda.org/

trace(xml1.localName());

//输出：data
```

```
trace (xml1.namespace());

//输出：http://www.kingda.org/

xml1.setName(*pageview*);

trace (xml1.toXMLString());

//输出：<pageview xmlns=”http://www.kingda.org/”>12345</pageview>
```

复制代码

在示例 15-13 中，将讲述：

? 如何使用 addNamespace()为 XML 加命名空间。

? 如何使用 setNamespace()设置与该 XML 对象关联的命名空间。

? 如何使用 default xml namespace 设置默认命名空间。

? 如何在 XML 中使用命名空间空间类。

示例 15-13 XML 的命名空间运用

```
package org.kingda.book.xml

{

import flash.display.Sprite;

public class SampleXMLNamespace extends Sprite

{

private var enNS:Namespace;

private var cnNS:Namespace;

public function SampleXMLNamespace() {

var dataXML:XML = getData();

trace("=======");
```

```
default xml namespace = cnNS;

trace (dataXML.data.book);

/*输出：

AS3 殿堂之路

*/

trace ("+++" + dataXML.enNS::data.enNS::book);

/*输出：

AS3's road

*/

trace (dataXML.namespace("en"));

//输出：http://books.kingda.org/en/

trace (dataXML.namespace("cn"));

//输出：http://books.kingda.org/cn/

}

private function getData():XML {

var originXML:XML = <data>

<book>AS3 殿堂之路</book>

<site>www.kingda.org</site>

</data>

var copyCN:XML = originXML.copy();

cnNS = new Namespace("cn", "http://books.kingda.org/cn/");
```

```
copyCN.setNamespace(cnNS);
for each (var i:XML in copyCN.elements()) {
i.setNamespace(cnNS);
}
trace (copyCN);
/*输出：
<cn:data xmlns:cn="http://books.kingda.org/cn/">
<cn:book>AS3 殿堂之路</cn:book>
<cn:site>www.kingda.org</cn:site>
</cn:data>
*/
var copyEN:XML = originXML.copy();
enNS = new Namespace("en", "http://books.kingda.org/en/");
copyEN.setNamespace(enNS);
copyEN.book = "AS3's road";
for each (var j:XML in copyEN.elements()) {
j.setNamespace(enNS);
}
var bookXML:XML = <books/>;
bookXML.appendChild(copyCN);
bookXML.appendChild(copyEN);
```

```
bookXML.addNamespace(enNS);

bookXML.addNamespace(cnNS);

trace (bookXML);

/*输出：

<books xmlns:en="http://books.kingda.org/en/" xmlns:cn="http://books.kingda.org/cn/"
xmlns="null">

<cn:data>

<cn:book>AS3 殿堂之路</cn:book>

<cn:site>www.kingda.org</cn:site>

</cn:data>

<en:data>

<en:book>AS3's road</en:book>

<en:site>www.kingda.org</en:site>

</en:data>

</books>

*/

return bookXML;

}

}

}
```

复制代码

15.8 XML 对象与 XMLList 对象

XML 对象可以表示 XML 元素、属性、注释、处理指令或文本元素。它分为两类：“简单内容”XML 对象和“复杂内容”XML 对象。有子节点的 XML 对象属于“复杂内容”的一类：如果 XML 对象没有子节点，而是属性、注释、处理指令或文本节点之中的任何一个，就属于“简单内容”的一类。

如果将 XML 看成树图，那么“复杂内容”XML 属于树枝，“简单内容”XML 属于树叶。

XMLList 对象可以表示一个或多个 XML 对象或元素（包括多个节点或属性）。XMLList 类中包含用于处理一个或多个 XML 元素的方法。因此，可以对作为一个组的多个元素调用方法，也可以对集合中的各个元素分别调用方法。

如果试图对包含多个 XML 元素的 XMLList 对象使用 XML 类方法，可以使用 for…in 和 for each…in 来遍历 XMLList 对象。

如果 XMLList 对象只包含一个 XML 对象，那么可以直接将 XMLList 对象当成 XML 对象，使用 XML 类方法。

15.9 本章小结

本章在开头简介了 XML 的设计思想、原因、优点，还讲述了设计一个结构优秀的 XML 的注意要点。

XML 数据处理在任何一门语言中都是非常重要的。在 ActionScript3 中，实现了 E4X，引入了多个运算符，极大地简化了代码编写，也提供了丰富的 API，对 XML 的方方面面都有支持。

本章最后还讲述了命名空间如何在 XML 中应用。有兴趣的读者可以将这一节和之前的“命名空间”一章结合起来学习。

第十六章 异常和错误的捕捉与处理

将异常处理放到ActionScript3核心类这一部分是经过了仔细斟酌的。在ECMAScript Edition 4 draft[参考文献 1]中规定异常（Error）类要作为内置类。ActionScript3 中的顶级包（Top Level）中，Error 及其子类共占据了 11 个位置，超过了核心类总数的三分之一。ActionScript3 对异常处理的改进是非常显著的。在 AVM2 宣传中，对异常处理机制的良好支持一直是主要宣传点。

本章导读

异常处理对于初学者稍显抽象，平时也比较少碰到，可以先试着学习，如果不明白可以暂时跳过本章。但是日后建议花时间仔细阅读，这是 ActionScript3 的大亮点和大改进，应当好好利用。

对于了解 ActionScript2 的读者，糟糕的异常处理机制已经是历史了。在 ActionScript2 中我们有很多理由不去使用异常处理机制，但是 ActionScript3 中我们没有理由不去好好地掌握它和使用它。本章非常重要，请仔细阅读。

对于其他 OOP 语言读者，本章会显得非常熟悉，语法也基本一致。所要注意的是异步异常的处理稍有不同，请阅读 16.7 节。16.1.2 节中，把 ActionScript3 的异常与 Java、C#略微做了比较。

16.1 什么是异常和错误

异常和错误是指程序执行时遇到的任何错误情况或意外行为。似乎有些抽象，没有关系，了解异常最快的方法就是先看一个出现异常的代码例子。然后，我们再来看一下异常的详细概念，ActionScript3 中的实现，以及与其他语言的不同之处。

16.1.1 一个简单的异常例子

示例 16-1 简单的异常例子

```
package org.kingda.book.error

{

import flash.display.Sprite;

public class ErrorSample extends Sprite

{
```

```
public function ErrorSample() {
var dataSource:OtherData = new OtherData();
try {
trace (dataSource.getArray());
dataSource.getArray().push("kingda");
} catch (error:Error) {
trace (error);
//不管有没有异常抛出,都会执行 finally 中的内容
//TypeError：Error #1009：无法访问空对象引用的属性或方法
}
}
}
}
internal class OtherData {
private var foo:Array;
public function OtherData() {
//...
}
public function getArray():Array {
return foo;
}
```

}

复制代码

在上面的 ErrorSample 中，使用了某个第三方的类的示例（dataSource）所返回的 Array 对象。在本例中，这个第三方的类就是同一个包下的 OtherData 实例。现实开发中，这个第三方的类可能处于不同的包中，甚至是别人以及编译好的组件，甚至不能别我们查看或者修改代码。

在 try{}中的代码试图操作 dataSource 返回 Array 对象，本例中的操作是希望在其中多加入一个字符串“kingda”。结果怎料到，出于某种原因，dataSource 返回的其实是一个 null（本例中的原因是未经初始化）。如果没有 try-catch，那么会导致程序没有办法正常执行下去，一旦执行，Flash Debug Player 就会抛出一个 Error（异常）：“TypeError：Error #1009：无法访问空对象引用的属性或方法。”

这是一个好事儿，至少通知我们出了错。而不是像在 ActionScript2 中那样“蒙混过关”，静悄悄的失败让我们一头雾水，调试时也不知从何查起。

读者可能已经从 try-catch 的字面意思猜出，try{}就是试试 try 下面花括号扩住的代码块，执行一下，如果发生异常就会有 Error 对象抛出，那么就被 catch 语句抓住，使用 catch 语句中的代码来处理这个错误。本例中的处理是将 Error 对象的内容 trace 出来，现实编程中可能会采取一些补救措施或者直接给用户弹出一个窗口提示。

这就是一个很简单的代码发生异常的例子。

异常的具体定义是什么？还有那些时候会引发异常？这就是我们下一节要讲述的内容。

16.1.2 异常的概念和在 ActionScript3 中的实现

可将异常理解成是一类消息，表示程序出错了的消息。

它传递一些 AVM 问题、故障及未按预想的逻辑执行的相关信息。在 ActionScript3 中，使用 Error 类及其子类的实例来统一表示异常和错误。这些实例对象中，一般包括异常信息、异常 ID、堆栈跟踪数据等。通过这些 Error 实例将信息从应用程序的一部分发送到另一部分。

由于 ActionScrpt3 没有对线程的支持，所以异常分为两种：同步异常和异步异常。Error 和其子类一般用来处理同步时的程序异常。在异常处理时，所产生的异常用相关的异常是将（ErrorEvent）表示。

这两种异常的本质和处理机制都不同：对于同步异常来说，使用 try-catch 来处理：对于异步异常，采用的是事件侦听机制来处理。而且，异常事件并不是 Error 的子类，而是 flash.events.ErrorEvent 的子类，这一部分我们将放在的 16.7 节“对异常事件的处理”中介绍。

在本章中，异常一般指同步异常。异步异常，本章会明确写出是异常事件或异步异常，请读者注意。

以下这些情况都可以引发异常。

你的代码或调用的代码中有错误，如操作系统资源不可用、AVM 遇到意外情况，等等。对于这些情况，ActionScript3 应用程序可以从其中一些恢复，而对于另一些，则不能恢复。尽管可以从大多数应用程序异常中恢复，但一般不能从大多数 AVM 异常中恢复。

与其他语言对比

在 Java 中，Throwable 类是 Java 语言中所有错误或异常的超类。两个子类的实例：Error 和 Exception，通常用于指示发生了异常情况。Error 用于指示合理的应用程序不应该试图捕获的严重问题。Exception 指示合理的应用程序想要捕获的条件。

但 ActionScript3 并没有这样的划分，所有错误和异常都是 Error 类或其子类的实例。处于顶级包（Top Level）中的 Error 子类们近似于 Java 中的 Exception。flash.errors 包中的 Error 子类代表发生的严重问题，类似于 Java 中的 Error。而且 ActionScript3 中的 Error 不支持异常链，不包含 cause（原因）。

在.NET Framework 中，异常是从 Exception 类继承的对象。从这个方面说，ActionScript3 中的 Error 与.NET Framwork 中的 Exception 地位对等。.NET 中 Exception 分了两个子类，一个是 SystemException，从其中派生出的用户定义的应用程序异常类。在 ActionScript3 中没有这样的划分，所有 AVM 异常类和用户的自定义类都是直接从 Error 中派生的。

ActionScript3 不支持线程的概念，但有同步和异步之分。异步应用产生的异常将发出异常事件（ErrorEvent），而不是异常（Error）。要加以注意。

16.1.3 使用异常处理机制的好处

使用异常处理机制的一个重要的原因是让程序更加强壮、更加易于维护。比如，从一些异常状态中修复程序的运行。比如当异常发生时（所读取资源出错，运行时出现空对象），提示用户或者改变程序的运行路线，确保程序正常运行。异常处理机制，对设计一个“用户友好”的 RIA 程序，至关重要。

但是初一想，这些情况下似乎不用异常也可以处理，直接在异常发生处写入相关代码解决不也很好吗？那么，为什么要引入异常？为什么不在异常出现的位置直接写入相关逻辑来处理呢？

作为大部分语言所采用的异常处理机制，当然有很多原因和优点。

有时，根本不能修改代码来处理异常。原因可能有异常来自于第三方组件或者源代码的缺陷。如果是编译好的组件，则不能访问其代码。如果是源码，可能由于其架构庞大，又没有详细文档，查错困难，不如在自己的代码中集中处理这些异常。

有时，需要在整个程序整个逻辑系统中交流错误信息，以便按统一方式来处理问题。在大型的 RIA 软件设计中，经常要使用该策略。

有时，出现了异常但可能有若干种处理它的方式，但你不知道使用哪一种。这时，可以通过 try-catch-finally 将异常处理委托给异常所在方法的调用者。方法调用者，一般更加了解异常的原因，以及异常发生的环境，可以更加准确地选择处理方法。

这样的总结有点抽象，如果你不太明白也没有关系，后文中有详细的异常处理实例和处理原则。相信你可以在具体的实例应用中体会到异常的好处。

不过在这之前，要先简单介绍一下 try-catch-finally 的使用方法，和 ActionScript3 中对异常的良好支持，以及常用的 Error 子类介绍。

16.2 使用 try-catch-finally 处理异常

在本章开头的例子中已经讲述了一个具体的异常处理的代码例子。一旦我们认为某段代码可能会有潜在的异常发生时，那么这些潜在的异常就应该在这段代码包括在 try 花括号的代码块中。

常见的潜在的异常发生情景有：某对象为 undefined，却对它进行操作；运行时访问了本不属于类定义的方法；运行时调用方法，类型不匹配；使用网络 Socket 连接时，跨越安全沙箱限制或者传入的端口参数不正确，等等。这些在下一节：“ActionScript3 中异常的层次和结构”中会有更加详细介绍。

先看一个简单的代码例子，看如何处理 Socket 连接异常。

在 ActionScript3 中 Socket 的连接只允许连接 65535 以下的端口。示例 16-2 中，端口号居然是 66666，虽然顺口却是非法的，因此会抛出一个 SecurityError 异常。

示例 16-2 Socket 连接异常处理的简单示例

```
package org.kingda.book.error

{

import flash.display.Sprite;

import flash.net.Socket;

public class SecurityErrorExample extends Sprite

{

public function SecurityErrorExample() {

var targetServer:String = "www.kingda.org";

var port:uint = 66666;
```

```
try {
var socket:Socket = new Socket();
socket.connect(targetServer,port);
//注意，一旦异常发生后，下面这句 trace 是不会执行到的
trace(“try end”);
}
catch(e:SecurityError) {
trace(e);
}
finally {
trace (“finally ended!”);
}
}
}
}
```

复制代码

示例 16-2 中依次讲述了 try、catch、finally 这 3 个关键字的含义和用法。

16.2.1 try

将可能会抛出异常的语句，放置在 try 里面，这是告诉编译器，我的这些语句可能会出现异常哦，你要小心点，使用特殊的机制来评估。一旦发生异常，将异常提交给 catch 语句，而不要“蒙混过关”，默默失败（Silent Failure）。

要注意的是，不要一股脑将所有语句都放入 try 块。笔者认为尽可能只放那些可能出错的语句，这样 try 中的内容简洁，一眼就知道调试重点在哪里。

要特别注意的是，异常一旦发生，那么 try 块中的语句就会立即停止执行。换句话说产生异常的语句后所有其他语句都会停止执行。在示例 16-2 中，由于异常发生了，try 块中最后一句 trace()不会被执行。

如果将端口号 65535 以下，异常不会发生，trace 语句也就会执行了。但 try 块外面的语句不受影响，会继续执行。

那么，犯罪嫌疑语句已经被放入 try 后，该找相关负责人来处理它们。这些相关负责人就是 catch 块中的内容。

16.2.2 catch

catch 就是抓住的意思。一个 try 可以跟随几个 catch，以便应付可能抛出的不同异常。

异常一旦发生，如果 try{}后面跟着 catch，那么会有以下几件事发生：

（1） 生成异常对象（Error 类实例或其子类实例），try 块中语句执行立刻中止。

（2） AVM 会按 catch 语句块出现的先后顺序，查找和异常对象应用的 catch 块。

（3）一旦发现异常和 catch()中定义的异常类型相符，就会被 catch“抓住”。抓住之后，就会交给这个 catch 块中的语句处理。

本例中对捕捉到的异常处理，只能简单地将异常对象 trace()出来，在实际运用中，可能是在其中更换端口号，再重新连接一遍：或者提示用户程序异常，需要退出。如果当前没法解决，比如无法决定有效的端口号，那么继续抛出一个异常，希望更高层的代码能捕捉到这个错误。

如果有多个 catch 语句块，那么应当尽量把具体的异常类型放在最前面。比如说 ErrorNetA 是 ErrorA 的子类。如果有 3 个 catch 块分别处理 ErrorNetA、ErrorA、Error 这 3 种异常时，应当吧 catch 异常 ErrorNetA 放在最前面，ErrorA 其次，最普通的 Error 放在最后。

16.2.3 finally

不论抛出的异常是否被 try{}后面跟随的 catch 语句捕捉到，finally 中的语句一定都会执行。这就是 finally 的好处：无论 try 块中的语句是否碰到异常而中止，finally 块中的语句执行不会受到影响。

一般情况下，是在 finally 中做一些释放资源、清除打扫的工作，比如关闭 Socket 的连接等。

finally 是可选的，不是必需的。finally 只能有一个。

16.2.4 try-catch-finally 的语法规则

try-catch-finally 异常处理机制必须符合本节介绍的语法规则，不然编译器会报错。以下是几个典型的语法规则：

? try 后面至少要跟一个 catch 语句块或者 finally 语句块。否则会报错“1073：语法错误：需要 catch 或 finally 子句”。

? try 后面可以跟有多个 catch 语句块。一个 catch 块处理一种异常类型。Catch 块的顺序应根据所处理的具体异常类型来排列。一般来说，从最具体的 Error 子类到最不具体的 Error 类顺序来先后排列。更加详细的原则请参见“处理异常的原则和方式”一节。

? catch 语句块里面可以再嵌套 try-catch-finally 的结构。

try、catch、finally 这 3 个块中可以再次抛出异常。如何抛出异常？请看下一节。

16.3 使用 throw 抛出异常

除了系统 API 可以抛出异常外，我们还可以在自己的代码中指定抛出异常。我们不仅可以抛出系统预定好的异常类实例，还可以扩展 Error 或其子类生成自定义的异常类。

在 ActionScript3 中使用 throw 关键字来抛出异常。

与其他语言比较

Java 用户注意，在 ActionScript3 中，类方法抛出异常无须声明。throw 关键字只用于在语句中抛出异常。

在继续讲述用 throw 抛出异常之前，先必须唠叨两句：要谨慎抛出异常。何时该处理异常，何时该抛出异常，本文在“处理异常的原则和方式”中有详细讲述。先给大家提个醒。

16.3.1 抛出 Error 类或其子类的实例

抛出 Error 类或其子类的异常很简单。Error 类的构造函数接受两个参数：第一个是字符串变量，作为异常的文字信息，默认为空；第二个是异常的数字 ID，int 型的，默认为 0。两个参数都是可选的。从使用上来

说都会定义一下异常的信息，即第一个变量。构造函数如下：

public function Error (message:String = “”,id:int = 0)

复制代码

进阶知识

第二个参数较少用到，如果需要自定义，那么建议避开 1 000~2 152 这段数字。因为这段数字已经被 ActionScript3 内置的 Runtime Error 占用了。为了避免冲突，和考虑到 ActionScript3 日后可能添加更多的内置 Runtime Error，笔者建议保险起见，避开 1 000～2152 这一段。

怎样抛出异常？具体有如下几种常用形式。

生成一个 Error 类或其子类的实例，用 throw 抛出。因为 Error 是动态类，所以某些情况下，我们可能希望让它携带一些信息后，再将其抛出。

生成一个匿名的 Error 或其子类的实例，用 throw 抛出。在只需要抛出异常的情况下，才使用这种方式。这种方式最多只能定义一下异常的信息。

将相关信息传给一个特定的方法（或者函数），由该方法（或者函数）根据这些信息来判断返回什么样的异常。这样的应用比较灵活、代码集中，常见于比较复杂的环境中，值得推荐。

下面的示例 16-3 按顺序给出了这几种常用形式的代码。读者可以更换这几段代码顺序，来查看异常处理的结果。

示例 16-3 抛出异常的几种常用形式

```
package org.kingda.book.error

{

import flash.display.Sprite;

import flash.utils.*;

import flash.net.Socket;

public class ThrowErrorSample extends Sprite

{

public function ThrowErrorSample() {

try {

var host:String = "www.[yourDomain].com";

var socket:Socket = new Socket();
```

```
socket.connect(host, -20);

}

catch(e:SecurityError) {

trace(e);

}

try {

//抛出一个函数返回的 Error 对象

var errorCause:Number = 5000;

throw reportErrorFunc(errorCause);

//抛出一个 Error 对象

var CustomError_1:TypeError = new TypeError("Try the first error:", 1200);

throw CustomError_1;

//抛出一个匿名 Error 对象

throw new Error("Try the second error")

} catch(e:Error) {

trace (e);

}

}

private function reportErrorFunc(eC:Number):Error {

if (!(eC is Number)) return new TypeError("CustemFuncError:Not a Number");

if (eC > 1000) {
```

```
return new Error("CustemFuncError:A big number error.");

} else {

return new Error("CustemFuncError:A small number error.");

}

}

}

}
```

复制代码

16.4 自定义异常

所谓自定义异常，就是通过扩展 Error 类、或者 Error 的子类来创建一个新的异常类。

为什么要自定义异常？因为我们开发具体应用程序时，会碰到如下几种类似情况。

当某种不符合我们程序规定的对象（或者行为）出现时，我们希望抛出一种自定义异常，从而利用异常处理机制来管理它。

这种对象（或者行为），本身符合语法，而且也不会引起任何系统内置异常。我们抛出异常，只是因为它不符合我们自己代码的逻辑。

比如说，我们从第三方数据源接收到一个格式良好的 XML。我们的 RIA 程序要求它包含 3 个名为 action 的子元素，但是这个 XML 并不符合这个要求，而且这很可能会引起后续程序出错。于是我们新建一个 CustomXMLError 类，扩展自 Error 类。一旦发现验证失败，我们就抛出这个 Error 子类的实例，由专门 catch 这个 CustomXMLError 类实例的代码块捕捉到，并执行相关的处理。

某种系统内置异常出现了，但是在我们的程序中的这种异常可能有多种含义。而且不同的含义中，需要在这个异常对象中加入不同的信息。而不同信息又有不同的异常处理方式。那么这时候，一个好的做法就是扩展这个系统内置异常类，生成几个子类，各自代表不同含义下的异常。

比如，在示例 16-2 中，我们得到了一个无效的端口号导致了连接失败。如果这个类中有 3 个方法调用了 socket.connect()，虽然都会抛出 SecurityError 异常。但如果是调用 A 方法引起的，那么可能是由于 A 原因：是调用了 B 方法引起的，可能是由于 B 原因；如果是调用了 C 方法引起的，那么可能是由于 C 原因。而且这 3 种原因各不相同，我们需要有不同的信息加入到 Error 对象中。虽然抛出的都是 SecurityError 异常实例，但已经不能满

足我们的需要了，这样我们就需要扩展 SecurityError 类，生成 3 个子类各自表示不同的原因，具体的过程请见下面的示例 16-4。

示例 16-4 使用自定义异常的例子

```
package org.kingda.book.error

{

import flash.display.Sprite;

import flash.net.Socket;

public class SecurityErrorExample extends Sprite

{

var socket:Socket;

public function SecurityErrorExample() {

socket = new Socket();

var targetServer:String = "www.kingda.org";

var port:uint = 66666;

try {

connectMethodA(targetServer, port);

}

catch(e:CustomBError) {

trace (e);

}
```

```
catch(e:CustomAError) {
trace ("抓到 A 了:"+ e +"t 出错端口号:" + e.getPort());
//输出:抓到 A 了:SecurityError  出错端口号:66666
}
catch(e:SecurityError) {
trace(e);
}
}
private function connectMethodA(host:String, portNum:int):void {
//...A 方法的其他代码....
try {
socket.connect(host, portNum);
} catch (e:SecurityError) {
var customAErr:CustomAError = new CustomAError();
customAErr.record(host, portNum);
throw customAErr;
}
}
private function connectMethodB(host:String, portNum:int):void {
//...B 方法的其他代码....
try {
```

```
socket.connect(host, portNum);
} catch (e:SecurityError) {
throw new CustomBError();
}
}
}
}
class CustomAError extends SecurityError {
//...CustomAError 的一些代码....
private var infoForA:Object = new Object();
public function record (host:String, portNum:int):void {
infoForA.host = host;
infoForA.port = portNum;
//...其他代码....
}
public function getPort():int {
return infoForA.port;
}
}
class CustomBError extends SecurityError {
//...CustomBError 的代码...
```

}

复制代码

我们在 connectMethodA()方法中抛出了特有的 CustomErrorA 类的异常实例，可以看到 CustomErrorA 有自己的属性和方法。由于实际执行的是 connectMethodA()，所以抛出的异常不会当成 CustomErrorB 接受，而被专门管理 CustomErrorA 的 catch 块捕捉到了，并使用了 A 异常特有的方法 getPort()所返回的信息。

在了解了 try-catch-finally 和 throw 异常的应用之后，我们需要深入地了解一下 ActionScript3 中的异常架构，以及常用的异常类。这样我们才可以最好地利用 ActionScript3 强大的异常处理系统。

16.5 ActionScript3 中异常的层次和结构

ActionScript3 对异常的支持是很全面的。下面通过与 ActionScript2 中队异常支持的不同来做详细说明。

与 ActionScript2、ActionScript1 对比

在 ActionScript2 中，并不推荐使用所谓的异常处理机制，主要有以下两个原因。

一是没有系统预定义异常类，需要用户自己定义各种异常类，而且系统自有的 API 不会抛出异常。这使得 ActionScript2 中的异常处理机制很不实用。

另一个重要原因是，ActionScript2 中异常处理机制开销大，对程序的执行效率影响较大，但在 ActionScript3 中，所有 API 都内建对异常的支持，执行效率因为进行过优化从而大大提高，鼓励合理使用异常处理机制。

1. Error 类所包含的信息大大丰富

ActionScript2 的 Error 类只包括 name 和 message。ActionScript3 中不仅包括 name 和 message，还有更加详细的 getStackTrace()。这个方法返回的是堆栈根据数据，不仅包括调用者，还包括一定的上下文。对代码的调试帮助很大。

2. 丰富的系统预定义异常

在 ActionScript2 中，我们只拥有一个 Error 类，没有系统预定义的异常，一切靠我们自己。非常的简陋。

ActionScript3 中，已经给我们定义好了各种常见的异常，已经包含预定义好的 ErrorID（异常 ID）和 Error message（异常消息）。使用起来非常方便。

3. 程序运行出现预定义异常时，自动抛出

这一点和 ActionScript2 大不相同。在以前的 ActionScript2 程序开发中，从来不会自动抛出异常。这对我们调试运行时出现的错误非常不方便，总不能每次都要我们手工来抛出异常吧？ActionScript3 这一点就做得很贴心。一旦程序运行碰到到预定义异常，就会自动抛出。如果是普通的 Flash Player，那么只会抛出异常名称和 ID：如果是供开发用的 Flash Debug Player，那么不仅会抛出异常名称和 ID，而且还会带有异常信息。相关信息如下。

普通的 Flash Player 显示异常为：

TypeError：Error #1009

复制代码

Flash Debug Player 显示异常为：

TypeError：　Error #1009：无法访问空对象引用的属性和方法。

复制代码

4. API 支持更准确的异常定位

在一些老的 ActionScript2 方法中，并没有异常的支持，往往只是用返回值的真假，或者对象的 undefined 或 null 来标识失败。而 ActionScript3 的大部分 API 中，支持精确的异常定位。也就是说，如果该 API 内含不同的几个操作，那么哪个操作出现异常了，就会抛出这个操作所对应的系统预定义异常。这样，开发者可以很快定位是哪个操作出现异常，并加以处理。

5. 异常处理机制的执行不会明显降低程序执行效率

要指出的是，一旦异常发生，异常处理机制开始执行时，任何一门语言在这方面的系统开销都比较大。ActionScript3 也是这样，因此建议尽量少地声明异常。详细的规则请参见“三大忌讳”。

但 ActionScript3 异常机制的执行效率和 ActionScript2 相比已经大大改进了。

16.6 处理异常的原则和方式

本节介绍处理异常的一些基本原则和注意事项。这些规则是历经考验，并且是在成熟的开发实践中总结出来的。希望读者尽可能地遵守，这样编写的代码才可能更加健壮、有效。

16.6.1 三大提倡

? 遇到异常就处理

在代码中碰到异常时，应当尽可能就地解决。如果在当前代码中无法处理，那么再继续抛出，并且在相关的方法之前写好一些注释，告诉自己和代码用户可能本方法会引发哪些异常。

? 针对不同异常提出具体解决方案

对不同异常做出不同的反应，是异常处理机制的优点所在。应当尽可能地对具体的异常给出具体的处理，这也包含着尽量将异常细化的思想。

? 保持对异常的记录，总结经验

当异常发生后，在代码中或者使用的第三方工具（记事本也可以）中，记录下异常的发生原因、次数和发生场景。这对日后的调试非常重要。

16.6.2 三大忌讳

? 尽可能不要使用空代码处理异常

有时为了程序能够运行，我们会不自觉地偷懒，编写一些空的处理块，让 Flash Player 不再弹出烦人的提示。也许，我们会觉得日后有机会再写相关代码来处理，但实际上往往是永

远也不会再来理会。这条忌讳与上一节的第一条提倡相对应。

```
try {

//……这儿是一些可能抛出异常的代码

} catch (err:Error) {} //注意，这儿放了空的处理块
```

复制代码

这种做法实际上相当于认为制造静默失败，基本上摒弃了异常处理的好处。如果此时处理不了，也应当做好记录，以备日后调试。

? 不要用宽泛的异常来对待具体的问题

这条与上一节的第二条提倡相对应。有时，代码中会抛出多种具体异常，但往往我们一偷懒使用了最宽泛的异常类型来处理，这就违背了具体异常具体处理的原则。应当尽量对不同的异常做出不同的处理，避免宽泛化。

最极端的形式莫过于直接使用 Error 类来敷衍。具体内容如下。

```
try {

//……这儿是一些可能抛出多种异常的代码

}catch (err:Error) { //注意，这儿放了最一般的 Error 类

//……一些代码

}
```

复制代码

? 尽量少使用异常处理机制

　在很多时候，异常处理机制不是必需的。比如，避免空对象的引用，完全可以用如下代码来处理：

```
if (obj != null) {

//_

}
```

复制代码

这是因为异常处理机制的运行成本较高，如果能够在当前代码避免使用异常处理机制，应当尽量避免。

16.7 对异常事件的处理

ActionScript3 引入了异常事件的概念，专门处理异步异常。异步事件处理，属于事件处理机制范畴。关于事件处理机制，更详细的知识请参见第 18 章“事件发送和处理”的具体内容。

异常事件一般有两类。

1. 扩展 ErrorEvent 类的异常事件

flash.events.ErrorEvent 类包含用于管理有关网络和通信操作的 Flash Player 运行时错误的属性和方法，AsyncErrorEvent、IOErrorEvent 和 SecurityErrorEvent 类扩展了 ErrorEvent 类。如果使用的是 Flash Player 的调试版，则会出现了一个对话框，向你告知播放器在运行时遇到的没有侦听器函数的异常事件。

2. 基于状态的异常事件

基于状态的异常事件与网络和通信类的 netStatus 和 status 属性有关。如果 FlashPlayer 在读写数据时遇到的问题，netStatus.info.level 或 status.level 属性（取决于使用的类对象）的值将被设置为值“error”。可以通过检查事件处理函数中的 level 属性是否包含值“error”来响应此异常。

异步异常一般发生在网络和通信操作上，本节将重点关注 ErrorEvent 类及其子类的异常事件处理。

异常事件处理分为以下两步：

（1） 添加侦听器，负责侦听可能发出的异常事件，与添加 try 作用类似。

（2） 编写侦听器函数，里面放置处理异常事件的代码，与 catch 作用类似。

下面以示例 16-5 来说明，读取一个不存在的文件发生异常后，如何用侦听器函数来处理。

示例 16-5 异步异常事件的处理

```
package org.kingda.book.error

{

import flash.display.Sprite;

import flash.net.URLLoader;

import flash.net.URLRequest;

import flash.events.IOErrorEvent;

public class SampleErrorEvent extends Sprite

{
```

```
public function SampleErrorEvent() {

var request:URLRequest = new URLRequest("不存在的文件.txt")

var loader:URLLoader = new URLLoader();

loader.addEventListener(IOErrorEvent.IO_ERROR, errorHandler);

loader.load(request);

}

private function errorHandler(evt:IOErrorEvent):void {

trace ("没有找到文件：" + evt);

/* 输出

没有找到文件：[IOErrorEvent type="ioError" bubbles=false

cancelable=false eventPhase=2 text="Error #2032: 流错误。

URL: file:///E|/源代码/不存在的文件.txt"]

*/

//...一些处理这个异常的代码

}

}
```

复制代码

16.8 本章小结

ActionScript3 的异常处理机制是一大亮点，为编写健壮的 ActionScript3 程序提供了有力的保证。对于 ActionScript2 开发人员来说，可能还不习惯异常处理机制，应当慢慢养成习惯。

本章详细介绍了异常处理机制的原因、机制和优点，并提供了相应的指导原则；讲述了自定义异常类，并提供了具体的代码例子加以说明；讲述了同步异常和异步异常杂 ActionScript3 中的处理方式。

第十七章 ActionScript3 目前主要的 API 概览

Application Programming Interface 简称 API，中文意思是应用程序接口，就是软件系统不同组成部分衔接的约定。ActionScript3 的 API 就是 Flash 相关运行时（Runtime）或程序类库提供给用户程序调用的代码。这些代码的主要目的是让开发人员得以调用一组例程功能，而无须考虑其底层的原代码为何或历经其内部工作机制的细节。API 本身是抽象的，它仅定义了一个接口，而不涉及应用程序如何实现的细节。比如，我们非常熟悉的 trace()就是 Flash Player 提供的一个 API，是输出面板或控制台输出代码中指定的字符串。

本章导读

本章短小精悍，扼要介绍了 ActionScript3 相关的 API 分类、架构和来源。并给出了一些有用的第三方资源，建议所有读者通读。

对于 ActionScript3 初学者，对 17.4 节讲述的第三方资源可能不太明白，没有关系，可以跳过不读。

17.1 ActionScript3 的 API 总览

ActionScript3 的 API 大致可以分为 3 个部分：Flash Player 提供的 API，Adobe 官方产品提供的 API，第三方提供的 API。

Flash Player API 是核心的 API，它使得开发人员能够在较低的级别控制 Flash Player 所提供的功能。Flash Player API 的所有成员都在 flash.*包中，包括所有的子包、类、函数、常量、事件和异常。目前正在开发阶段的 Adobe AIR 也是一个运行时，和 Flash Player 运行时相同，会提供一些访问本地资源的 API。由于本书完稿时，Adobe AIR 仍然处于 Beta 阶段，尚未发布正式版，因此不予介绍。

Adobe 官方产品提供的 API 主要分为 Flex API 和 Flash CS3 提供的 API。Flash CS3 提供的 API 都放置在 fl.*包中，Flex API 都放置在 mx.*中。Flex API 大部分是 Flex 架构所独有的，Flex 组件也不可以在 Flex 以外的场合使用。本书不介绍 Flex 部分的 API，只重点将一些 Flash CS3 所提供的 API。

第三方提供的 API，所有官方开发的但没有放入产品包中的一些独立的类库提供的 API，也包括大量开源社区所贡献的不同项目提供的 API，还包括商业组织开发的一些类库所提供的 API。本书中简略介绍前两者中的一些实用的资源。

17.2 Flash Player API 架构和介绍

Flash Player 9 的 API 比以前版本中的分类要科学得多、清楚很多。在 Flash Player 9 之前，即 ActionScript3 产生之前，ActionScript2 和 ActionScript1 中所使用的 Flash Player API 比较混乱。许多类和函数被直接放到了顶级包中，成为全局成员。ActionScript2 中可全局访问的类有 40 多个，全局函数（包括未公开的）有 70 个左右。相比之下，ActionScript3 中处于顶级的核心类只有 28 个，全局函数仅 21 个，另加全局常量 4 个。而且 Flash Player9 提供的 API 的规模和数量总体上要远远超过之前的 API，因此 ActionScript3 中的顶级成员比例是相当小的。

Flash Player 9 的 API 按照职责进行了合理的划分，分置在了独立的包中。良好的 API 架构设计降低了系统各部分的相互依赖，提高了组成单元的内聚性，降低了组成单元间的耦合程度，从而提高整个系统的维护性和扩展性。对于其他编程人员来说，Flash Player 9 API 的设计符合标准规范的 OOP，一看就会有很熟悉的感觉。

这些 Flash Player API 都是以本机代码（native code）的形式实现的，执行效率比一般的 ActionScript3 代码要高。查看 AVM2 的开源代码，会发现是基于 C++开发的，代码质量也很高，并且进行过优化。经过测试会发现执行同样的功能，Flash Player API 效率要快一些。在笔者的测试结果中，一般快 20%～30%。因此如果代码中能使用 Flash Player API 就尽量使用它，比我们自己编写出的代码效率高。

可以看出 Flash Player API 涉及面很广，功能比以往版本全很多。其中不少类包涉及的知识已经超出编程的范畴，比如说几何包的一些运用等。如果在一本书中讲解完这些包中的 API，几乎是不可能的事。

本书限于篇幅，只能讲解 ActionScript3 的所有重大特色和重要功能。本部分最主要的讲解 ActionScript3 API 部分的两大特色：事件发送和处理，以及网络通信基础。至于最重要的、改动最大的、官方着力最多的显示编程 API，将专门独立在第 5 部分“ActionScript3 视觉编程”中详细讲述。

17.3 Flash CS3 提供的 API 介绍

Flash CS3 提供的 API 中共含有 17 个包，主要分为 4 大类：Flash CS3 组件、动画效果、Motion XML 和视频。

4 大类中的具体内容请见表 17-1。

其中 Flash CS3 组件架构占了相当的部分，但 Flash CS3 组件架构与 Flex 组件架构相比，不论是在架构设计、控件种类、功能和复杂程度上都是小巫见大巫、难及项背。对于 Flash CS3 组件架构的介绍，请见第 26 章“Flash 创作工具和 Flex 协作开发组件”。

表 17-1 4 大类的具体内容

Flash CS3 组件架构	fl.controls、fl.core、fl.containers、fl.managers、fl.events、fl.data、fl.livepreview、fl.lang、fl.controls.dataGridClasses、fl.controls.listClasses、fl.controls.progressBarClasses、fl.accessibility
动画效果	fl.transitions、fl.transitions.easing
Motion XML	fl.motion、fl.motion.easing
视频	fl.video

17.4 *第三方类库提供的 API

从 ActionScript2 开始，开源社区就慢慢重视起 ActionScript 程序的开发了，出现了一大批第三方的优秀类库。ActionScript3 出现之后，开发热潮更是一波接一波。Adobe 官方又积极组织，由公司原班人马开发了不少有用的类库。

本节将主要介绍一些比较实用和常用的第三方类库。

17.4.1 Adobe 官方提供的产品之外的类库

Adobe 官方开发的这些类库有些会被并入未来的产品中，有些则会持续投放民间继续开发。下文扼要介绍几种比较普及的类库。

1. Corelib

Corelib 下载地址：http://code.google.com/p/as3corelib/。

扼要介绍其中几个实用的类包：

com.adobe.crypto 提供常用的加密算法，包括 MD5、SHA1、SHA224 等。

com.adobe.images 提供将位图转换成 JPG、PNG 的编码器。

com.adobe.serialization.json，顾名思义，是对 JSON 的支持。

com.adobe.utils 是一组实用的数组、字符串、XML 和字典等工具类。

2. Flex Component Kit for Flash CS3

它提供了将 Flash CS3 创作内容转换为 Flex 兼容的 UIMovieClip 实例，这是非常重要的工具，目前已并入 Flex3。Flex2 用户需自行下载。

3. as3flexunitlib

这是官方开发的 ActionScript3 测试驱动框架，非常好用。下载地址为：

http://code.google.com/as3flexunitlib/

17.4.2 ActionScript3 的 3D 应用

目前运用 ActionScript3 开发的 3D 引擎逐渐增多，其中以 Papervision3D 的普及率最高。使用了 Papervision3D 可以比较方便地创建出 3D 的场景、镜头等。安装方法和使用教程可以参见 http://wiki.papervision3d.org/。

17.4.3 开源项目资源

http://www.osflash.org、http://code.google.com/、http://www.riaforge.org 有诸多 ActionScript3 开源项目。著名的 sf.net 上也有一定数量的 ActionScript3 开源项目。

17.5 本章小结

本章介绍了 API 的概念和 ActionScript3 的 API 的几个组成部分和来源。

概述了 ActionScript3 的 Flash Player API|（flash*）、Flash CS3 提供的 API（fl.*），并分类概况描述。

最后，介绍了第三方的实用类库，并给出了 ActionScript3 开源项目的资源链接，供大家使用。

第十八章　事件发送和处理

事件处理代码示例：

```
package{
    import flash.display.Sprite;
    public class SampleOrderDishes extends Sprite{
        public function SampleOrderDishes(){
            //生成一个客人 kingda
            var kingda:Customer = new Customer("Kingda");
            //生成一个 kiki 的服务员
            var kiki:Waiter = new Waiter();
        //注册侦听器,将服务员的 replyOrderFood()方法注册为监听"点菜"事件的侦听器
            kingda.addEventListener(OrderEvent.ORDER_DISHES, kiki.replyOrderFood);
            //点菜,在 order()方法中发出 OrderEvent 事件
            kingda.order();

            kingda.removeEventListener(OrderEvent.ORDER_DISHES, kiki.replyOrderFood);

        }
    }
}

package{
    import flash.events.Event;
    public class OrderEvent extends Event{
        public static const ORDER_DISHES:String = "点菜";
        private var _dishes:Array;  //用来存储菜名
        public function OrderEvent(){
            super(ORDER_DISHES); //将静态属性 ORDER_DISHES 作为事件的默认类型
        }
        //OrderEvent 事件自定义的 getter 和 setter 方法:客人点的菜名清单
        public function set dishes(dishesAry:Array):void{
            _dishes = dishesAry;
        }
        public function get dishes():Array{
            return _dishes;
        }
    }
}

package{
    public class Waiter{
        //replyOrderFood 方法专门用来接受 OrderEvent 对象
        public function replyOrderFood(evt:OrderEvent):void{
```

```
                trace("您好," + evt.target.name + "!     您点的菜是: \r" + evt.dishes);
                trace("我马上吩咐厨房去做.");
            }
        }
}

package{
    import flash.events.EventDispatcher;
    public class Customer extends EventDispatcher{
        public var name:String;
        public function Customer(nS:String){
            name = nS;
        }
        public function order():void{
            //生成 OrderEvent 对象
            var orderDish:OrderEvent = new OrderEvent();
            //点一些菜,放入 OrderEvent 对象中,作为数据
            orderDish.dishes = ["剁椒鱼头","家常豆腐","蛋炒饭"];
            //发送出 OrderEvent 事件
            dispatchEvent(orderDish);
        }
    }
}
```

Event 类的实例属性
type，用来存储事件对象的名称，是字符串类型（推荐使用静态字符串常量）
target，Object 类型，持有引用，指向最初发出事件的那个对象
currentTarget,eventPhase,bubbles 事件与事件流机制相关
cancelable，布尔值，表示这个事件引起的默认动作是否可被取消

Event 类的构造函数
Public function Event(type:String, bubbles:Boolean = false, cancelable:Boolean = false)

Event 类的方法
Clone() 返回当前事件对象的一个拷贝。我们将一个事件对象重复发送时，EventDispatcher 类会自己动调用该方法。
toString() 返回对象表示的值的字符串形式。若自定义事件类，那么重写它时，可使用 formatToString()这个方法在返回的字符串中加入新的事件实例属性。
isDefaultPrevented()、preventDefault()和 cancelable 属性常一起使用，用来阻止事件的默认行为发生，默认行为就是指一些系统内定义的事件发生时自动做出的行为。
stopImmediatePropagation()、stopPropagation()用来阻止事件的传播，使侦听器失去效用。

在类方法（侦听器函数）中的 this 就指向当前的类实例对象，使用在包外定义的 function 做侦听，那 this 就指向 Global

检测事件侦听器，检测当前事件发送者是否为某事件类型注册了侦听器
hadEventListener(type:String):Boolean
willTrigger(type:String):Boolean
区别：如果事件发送者处于显示列表上，那么 willTrigger（）不但检查该显示对象上的侦听器，还会检查该显示列表对象在事件流所在阶段中的所有父级上的侦听器。

自定义类发送事件的 3 种方式
1．　继承 EventDispatcher 类

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.EventDispatcher;

    public class SampleInheritEventDispatcher extends Sprite    {
        public function SampleInheritEventDispatcher() {
         //生成一个 KingdaSampleDispatcher 的实例
            var dispatcher:KingdaSampleDispatcher = new KingdaSampleDispatcher();

            //标准的实现
            dispatcher.addEventListener(KingdaSampleDispatcher.ACTION, actionHandler);
       dispatcher.doSomething();

            //可以用直接字符串标识事件类型，但推荐使用静态加 const 的字符串
            dispatcher.addEventListener("KingdaPlaySC", anotherHandler);

                //直接用 new Event 生成一个新的事件对象，该对象的事件类型为 "KingdaPlaySC"
            dispatcher.dispatchEvent(new Event("KingdaPlaySC"));
            //输出：
                //action 的侦听器: [Event type="action" bubbles=false cancelable=false eventPhase=2]
                //KingdaPlaySC 的侦听器: [Event type="KingdaPlaySC" bubbles=false cancelable=false eventPhase=2]
    }

    private function actionHandler(event:Event):void {
       trace("action 的侦听器: " + event);
    }

        private function anotherHandler(event:Event):void {
       trace("KingdaPlaySC 的侦听器: " + event);
    }
}
```

```
}

import flash.events.EventDispatcher;
import flash.events.Event;

class KingdaSampleDispatcher extends EventDispatcher {
    public static var ACTION:String = "action";

    //如果你需要在自己类中某个方法中发送事件，那么示例如下
      public function doSomething():void {
          //你的代码.....

          //发送事件
      dispatchEvent(new Event(KingdaSampleDispatcher.ACTION));
    }
}
```

2.　复合 EventDispatcher 对象

```
package {
      import flash.display.Sprite;
      import flash.events.Event;

      public class SampleCompositeEventDispatcher extends Sprite{
          public function SampleCompositeEventDispatcher() {
              var kingdaObj:KingdaClass = new KingdaClass();

          //一定要用 kingdaObj.getSender()来返回事件发送对象，才能 addEventListener
              kingdaObj.getSender().addEventListener(KingdaClass.ACTION, lisFunc);

              kingdaObj.doSomething();
          //输出:
          //doSomething
          //listened:yeahyeah

           }
           //侦听器
           private function lisFunc(evtObj:Event):void {
              trace ("listened:"+evtObj.type);
           }
      }
}
      import flash.events.EventDispatcher;
      import flash.events.Event;
```

```
class KingdaClass extends EventDispatcher {
    private var _dispatcher:EventDispatcher;
    public static const ACTION:String = "yeahyeah";

    public function KingdaClass() {
        initSender();
    }

    private function initSender():void {
        _dispatcher = new EventDispatcher();
    }

    //调用一个专门的方法(method)来返回发送事件的 EventDispatcher。
    public function getSender():EventDispatcher {
        return _dispatcher;
    }
    public function doSomething():void {
        trace("doSomething");
        //除了以下两行发送事件，还可以写入其它你要干的事儿。灵活。
        var evtObj:Event = new Event(KingdaClass.ACTION);
        _dispatcher.dispatchEvent(evtObj);
    }
}
```

3． 实现 IeventDispatcher 接口

```
package{
    import flash.display.Sprite;
    import flash.events.*;

    public class SampleIEventDispatcher extends Sprite{
        public function SampleIEventDispatcher() {
            var kingdaObj:KingdaClass = new KingdaClass();
            kingdaObj.addEventListener(KingdaClass.ACTION, lisFunc);
            //用起来和 EventDispatcher 对象一样哦，呵呵。

            var evtObj:Event = new Event(KingdaClass.ACTION);
            trace (kingdaObj is EventDispatcher); //输出：false
            trace (kingdaObj is IEventDispatcher);//输出：true

            kingdaObj.dispatchEvent(evtObj);//确实一模一样
            //输出:listened:yeahyeah
        }

        private function lisFunc(evtObj:Event):void {
```

```
            trace ("listened:"+evtObj.type);
        }
    }
}
    import flash.events.IEventDispatcher;
    import flash.events.EventDispatcher;
    import flash.events.Event;

    class KingdaClass implements IEventDispatcher{
        public var _dispatcher:EventDispatcher;
        public static const ACTION:String = "yeahyeah";

        public function KingdaClass() {
            // other ....
            initSender();
        }

        private function initSender():void {
            _dispatcher = new EventDispatcher(this);
        }
      //在实现接口时还可以乘机干点别的，比如我喜欢吧 useWeakReference 设为 true
        public function addEventListener(type:String,
                                         listener:Function,
                                         useCapture:Boolean = false,
                                         priority:int = 0,
                                         useWeakReference:Boolean = true):void{
                    // do other things;
          _dispatcher.addEventListener(type, listener, useCapture, priority, useWeakReference);
       }

        public function dispatchEvent(evt:Event):Boolean{
          // do other things;
            return _dispatcher.dispatchEvent(evt);
        }

        public function hasEventListener(type:String):Boolean{
                    // do other things;
           return _dispatcher.hasEventListener(type);
        }

        public function removeEventListener(type:String,
                                            listener:Function,
                                            useCapture:Boolean = false):void{
          // do other things;
```

```
            _dispatcher.removeEventListener(type, listener, useCapture);
        }

        public function willTrigger(type:String):Boolean {
            // do other things;
            return _dispatcher.willTrigger(type);
        }
    }
```

第十九章　网络通信基础

AS3 网络通信的流程

1．　构建通信请求对象（URLRequest）

提交带参地址，URLRequest 对象的构建：

```
//访问"http://www.flash-mx.com/mm/greeting.cfm? name=foo&age=28&sex=male "
var varibles:URLVariables = new URLVariables("name=foo&age=28&sex=male");
var request:URLRequest = new URLRequest();
request.url = "http://www.flash-mx.com/mm/greeting.cfm";
request.method = URLRequestMethod.GET;
request.data = variables;
```

提交 XML，URLRequest 对象的构建：

```
var loginXML:XML = <login>
                        <username>Kingda.org</username>
                        <password>ActionScript3</password>
                    </login>;
var request:URLRequest = new URLRequest();
request.url = "http://www.flash-mx.com/mm/greeting.cfm";
request.contentType = "text/xml";
request.data = loginXML.toXMLString();
request.method = URLRequestMethod.POST;
```

2．　使用通信请求对象，构建 URLLoader 对象，并发出数据请求

可以直接把 URLRequest 对象传入 URLLoader 的构造函数，立刻发送请求：

```
var loader:URLLoader = new URLLoader(request);
```

也可以先建立 URLLoader 对象，在合适的时候再发送请求：

```
var loader:URLLoader = new URLLoader();
//...
loader.load(request);
```

如果有数据返回的话，会放在 URLLoader 对象的 data 属性中，由 dataFormat 属性判断其数据格式是文本型（URLLoaderDataFormat.TEXT），二进制型（URLLoaderDataFormat.BINARY），还是变量名值对型（URLLoaderDataFormat.VARIABLES）

URLLoader 对象会发出六种事件

- load()一调用，就发出 Event.OPEN 事件
- 加载过程中，发出 ProgressEvent.PROGRESS 事件，包含下载字节数信息
- 加载完成，发出 Event.COMPLETE
- 加载完成或者失败之前，会发出 HTTP 状态事件 HTTPStatusEvent.HTTP_STATUS
- 加载失败，发出 IOErrorEvent.IO_ERROR 事件
- 发现加载内容不合安全规则，发出 SecurityErrorEvent.SECURITY_ERROR

3．　数据收到后，发出完成事件，调用“读取完成”事件的侦听器处理返回的数据

读取文本格式数据（示例：获取 RSS 种子的文章标题）
AS3 默认使用 Unicode 来解码外部文本，如果是按系统代码页编码（如 GB2312）那么需要先导入 flash.system.System 类，并设置 System.useCodePage = true;

```
package org.kingda.book.net
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.net.URLLoader;
  import flash.net.URLLoaderDataFormat;
  import flash.net.URLRequest;
  import flash.text.TextField;

    public class SampleLoadXML extends Sprite
    {
        private var title_txt:TextField;
        public function SampleLoadXML() {
            //测试网络
            //var targetURL:String = "http://feeds.feedburner.com/Kingda";
            var targetURL:String = "http://www.actionscript3.cn/index.xml";
            //测试本地
            //var targetURL:String = "fla/sample.xml";

            var request:URLRequest = new URLRequest(targetURL);
      var variables:URLLoader = new URLLoader();
      //variables.dataFormat = URLLoaderDataFormat.TEXT; //默认格式
      variables.addEventListener(Event.COMPLETE, completeHandler);
      try
      {
            trace ("loading...");
        variables.load(request);
      }
      catch (error:Error)
      {
        trace("Unable to load URL: " + error);
      }
      //生成一个文本框，用来放置读取的标题
      title_txt = new TextField();
      title_txt.autoSize = "left";
      addChild(title_txt);
    }
    private function completeHandler(event:Event):void
    {
```

```
            //var loader:URLLoader = event.target as URLLoader;
            trace(event.target.data is String);
            var resultXML:XML = new XML(event.target.data);
            var titleList:XMLList = resultXML.channel.item.title;
            for each(var title:XML in titleList) {
                  title_txt.appendText("*" + title + "\n");
            }
        }
        }
}
```

读取值对格式数据

```
package org.kingda.book.net
{
      import flash.display.Sprite;
      import flash.events.Event;
      import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;

      public class SampleLoadVariables extends Sprite
      {
           public function SampleLoadVariables() {
                //测试本地文件
                var targetURL:String = "fla/sample_variables.txt";

                var request:URLRequest = new URLRequest(targetURL);
            var variables:URLLoader = new URLLoader();
            variables.dataFormat = URLLoaderDataFormat.VARIABLES;
            variables.addEventListener(Event.COMPLETE, completeHandler);
            try
            {
                  trace ("loading...");
              variables.load(request);
            }
            catch (error:Error)
            {
              trace("Unable to load URL: " + error);
            }
             }
             private function completeHandler(event:Event):void
        {
            trace(event.target.data);
            //输出：
```

```
            var loader:URLLoader = event.target as URLLoader;
            trace ("=============");
            for (var i in loader.data) {
                  trace (i+"\t："+ loader.data[i]);
            }

        }
       }
}
```

读取二进制格式数据（二进制格式数据以 ByteArray 对象存储在 URLLoader 对象的 data 中）另类加载动画方法

```
package org.kingda.book.net
{
      import flash.display.Sprite;
      import flash.events.Event;
      import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;
    import flash.display.Loader;
    import flash.utils.ByteArray;
    import flash.system.Security;
    import flash.events.ProgressEvent;

      public class SampleLoadBinary extends Sprite
      {
           private var loader:Loader;
           private var loader2:Loader; //请读者自行添加 loader3,loader4.....
           public function SampleLoadBinary() {
                var targetURL:String = "fla/LinkedClass.swf";  //读取本地
                //Security.allowDomain("www.actionscript3.cn");
                //var targetURL:String = "http://www.actionscript3.cn/tmp/LinkedClass.swf";
                var request:URLRequest = new URLRequest(targetURL);
           var variables:URLLoader = new URLLoader();
           variables.dataFormat = URLLoaderDataFormat.BINARY;

           variables.addEventListener(Event.COMPLETE, completeHandler);
           variables.addEventListener(ProgressEvent.PROGRESS, listening);
           try
           {
                trace ("loading...");
             variables.load(request);
           }
```

```
            catch (error:Error)
            {
               trace("Unable to load URL: " + error);
            }

             }
             private function listening(event:ProgressEvent):void {
                  trace ("now:" + event.bytesLoaded + ": " +event.bytesTotal);
             }

             private function completeHandler(event:Event):void {
            trace ("loaded");
            loader = new Loader();
            var content:ByteArray = event.target.data as ByteArray;
            loader.contentLoaderInfo.addEventListener (Event.COMPLETE, convertHandler);
            loader.loadBytes(content);

            //提示，loader2 在此加入:loader2 = new Loader() ...
            loader2 = new Loader();
            loader2.contentLoaderInfo.addEventListener (Event.COMPLETE, convertHandler);
            loader2.loadBytes(content);
        }
        private function convertHandler(event:Event):void {
              //addChild(loader);
              addChild(event.target.loader as Loader);
              event.target.loader.y = 100*Math.random();
        }
        }
}
```

打开网址 navigateToURL()　（getURL()已取消）在 net 包里
public function navigateToURL(request:URLRequest, window:String = null):void
request 包含要跳转的网址
window 显示网页的窗口或 HTML 框架，默认新窗口打开（_self, _blank, _parent, _top）

提交数据，不返回数据使用 sendToURL()
向服务器端提交数据：登录验证例子

```
package org.kingda.book.net
{
      import flash.display.Sprite;
      import flash.events.Event;
      import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;
```

```
import flash.net.URLRequestMethod;
import flash.events.ProgressEvent;
import flash.text.TextField;

  public class SampleSendAndLoad extends Sprite
  {
      private var _result:TextField;
      public function SampleSendAndLoad() {
          //建立一个文本框对象，登录成功后，在其中显示返回的 Session ID。
          _result = new TextField();
          _result.autoSize = "left";
          _result.wordWrap = true;
          _result.width = 300;
          addChild(_result);

          var loginXML:XML =
            <login>
              <username>Kingda.org</username>
              <password>ActionScript 3</password>
            </login>;
                                                    var request:URLRequest = new
URLRequest("http://www.flash-mx.com/mm/login_xml.cfm");
          request.contentType = "text/xml";
          request.data = loginXML.toXMLString();
          request.method = URLRequestMethod.POST;
          var loader:URLLoader = new URLLoader();
          loader.addEventListener(Event.COMPLETE, completeHandler);
          try
          {
            trace ("loading...");
            loader.load(request);
          }
          catch (error:ArgumentError)
          {
            trace("参数错误");
          }
          catch (error:SecurityError)
          {
            trace("发生安全错误");
            _result.text = "发生安全错误。\n" + error;
          }
      }

      private function completeHandler(event:Event):void
```

```
        {
            trace(event.target.data); //服务器端返回的信息
            var resultXML:XML = new XML(event.target.data);
            _result.text = resultXML.@sessionid;
        }
        }
}
```

安全沙箱
当前运行的 SWF 属于哪一个沙箱：使用 Security.sandboxType 只读属性
Security.REMOTE：SWF 文件来自 INTENET，并遵守基于域的沙箱规则。不能读取不同域中的资源，除非另有设置。
Security.LOCAL_WITH_FILE：SWF 是本地文件，可读本地数据，但无法与 INTENET 通信
Security.LOCAL_WITH_NETWORK：SWF 是本地文件，可与 INTENET 通信，但不可读取本地数据。
Security.LOCAL_TRUSTED：SWF 是本地文件，且已使用“设置管理器”或 FLASH PLAYER 信任配置文件受到用户信任。可读本地数据，也可与 INTENET 通信。

跨域文件：crossdomain.xml

```
<cross-domain-policy>
<allow-access-from domain=”*.kingda.org” />
<allow-access-from domain=”*.adobe.com” />
</cross-domain-policy>
```

kingda.org 和 adobe.com 这两个域的 SWF 都可以读取本站上的文件。

授予脚本访问权限：Security.allowDomain()
所涉及的权限：
1．SWF 文件之间的跨脚本访问
2．显示列表访问
3．事件检测
4．对 Stage 对象的属性和方法的完全访问

ActionScript 3 视觉编程精要

20.1 什么是显示对象
在舞台上显示的对象，在 AS3 中统一被称为显示对象(Display Object)
显示对象除了包含能看得见的显示对象外,也包括不能看见但却真实存在的显示对象容器(Display Object Container)

20.1.1 ActionScript3 中显示对象等级结构
舞台(Stage)
当前 SWF(文档类或 MainTimeline)
容器
显示对象

20.1.2 显示列表:显示对象模型
AS3 中所有显示对象均使用显示列表(Display List)的方式进行管理,只有在列表中列出的对象才会在舞台上显示。
显示对象有两种：在显示列表中(on-list)和不在显示列表中(off-list)，在显示列表中的对象会被渲染，不在显示列表中的对象依然存在，不被渲染罢了。

20.2 ActionScript3 中显示对象的种类
20.1 ActionScript2 中的 MovieClip
MovieClip 是万能的，但缺点是一旦创建，就拥有了一大堆时间轴等属性，非常浪费资源，有时候只将它作为空白容器。
MovieClip 的地位在 AS3 中减弱了很多。

20.2.2 ActionScript3 显示对象种类划分：一个统一、两个层次
一个统一，AS3 中所有显示对象都统一于 DisplayObject 类
第一大层次：是否可以接受互动事件,可以接受的,称为 可互动的显示对象(InteractiveObject);不可以交互的,称为非互动显示对象.

可互动的显示对象(InteractiveObject):指能够接受鼠标单击、键盘敲击等人机交互事件。
位图、形状、视频等就不能接受这些事件，所以归入不可以交互类。

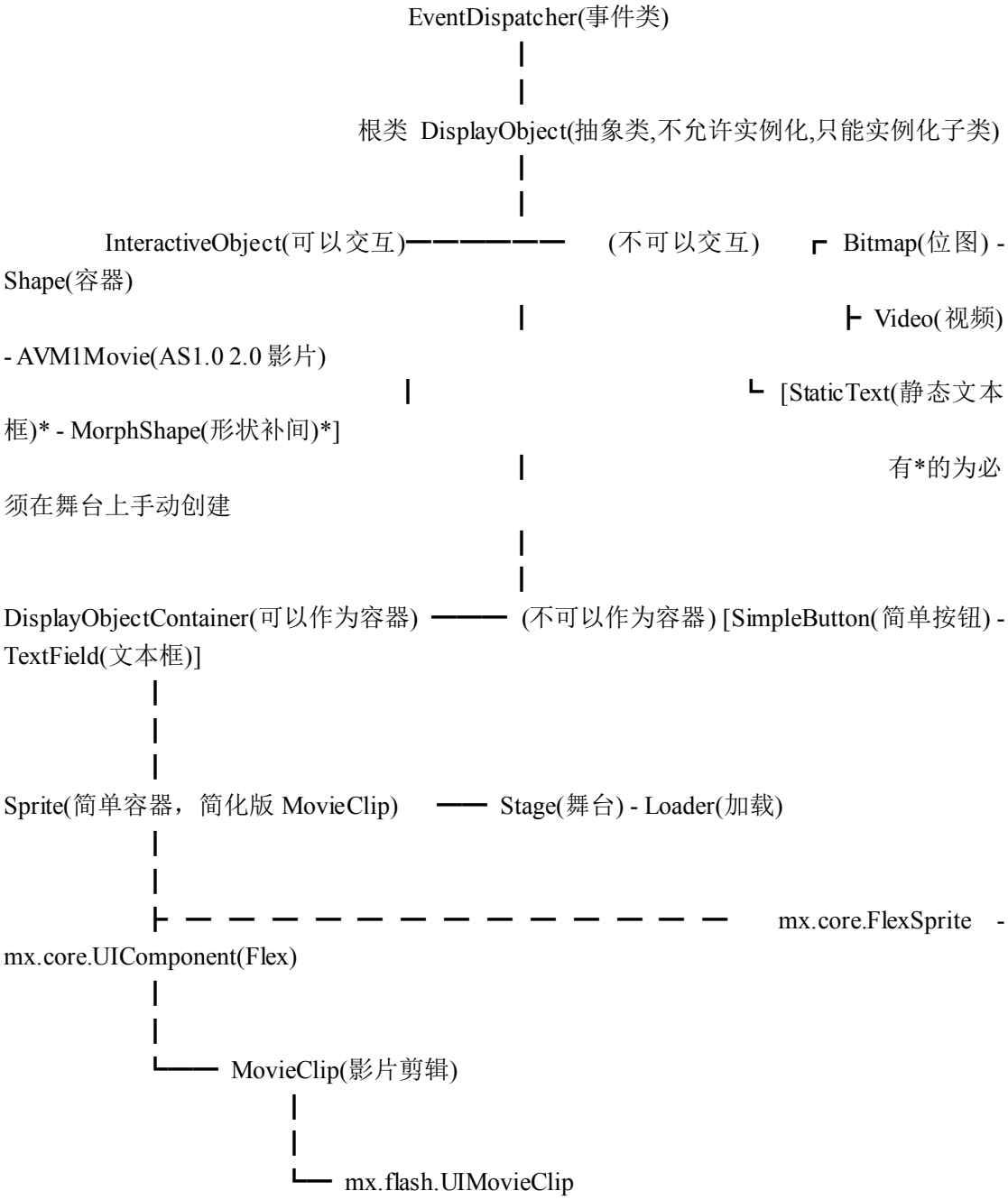
第二大层次：是否可以作为容器，可以容纳其它显示对象的，称为 显示对象容器(Display Object Container)

20.3 显示对象类库架构
以下很好反映了 一个统一、两个层次 这个概念

所有显示对象都是继承自父类 DisplayObject 这个抽象类，而父类 DisplayObject 则继承自 EventDispatcher 类，
说明所有显示对象都能发送事件。

DisplayObject、InteractiveObject、DisplayObjectContainer 是显示对象架构中的三个核心对象，它们都是不能被实例化的抽象类

视觉架构类图:



20.3.1 InteractiveObject 类和非 InteractiveObject 类
InteractiveObject 类　　可以接受人机交互事件
非 InteractiveObject 类 不可以接受人机交互事件
不可以接受人机交互事件的类中有 6 个同级对象 ：
AVM1Movie、Bitmap、MorphShape、Shape、StaticText、Video

其中再细分 MorphShape 和 StaticText 不可以用代码创建
StaticText 是 Flash 编辑环境下用文本工具创建的
MorphShape 是在 flash 中创建形状渐变时自动生成的

AVM1Movie： Actionscript Virtual Machine1(ActionScript 虚拟机 1)，即使用 AS1.0 AS2.0 创建的 Flash 影片，
为了向下兼容，当载入使用以上版本 AS 创建的影片时，会自动创建这个类，以同 AVM2 区分开来。

20.3.2 容器类和非容器类
第二层是 InteractiveObject 类的对象
包含三类，使用容器与非容器的概念来区分。
容器：可以在它里面加载其它的 DisplayObject 的容器
非容器对象：TextField、SimpleButton
TextField 就是动态文本框
SimpleButton 是 Flash API

剩下的就是 DisplayObjectContainer(显示对象容器)类，其下有：
Sprite、Loader、Stage
Stage 是舞台类
Loader 是原有 MovieClip 中加载外部资源的方法集合。

20.3.3 Sprite 和 MovieClip
Sprite 将是我们在 AS3 中接触最多的容器，可以把它理解成去掉时间轴的 MovieClip。
Sprite 中含有 Graphic 对象，可以像 MovieClip 那样直接在自身绘图，但 Sprite 不同于 Shape，区别在于 Sprite 是容器，而 Shape 不是。

MovieClip 是 Sprite 的子类，只保留了一些与时间轴控制相关的 gotoAndStop 方法和 currentFrame 属性等。

20.3.4 非 Flash API 的几个显示对象类
需要在 Flex 中使用到的显示对象，必须是 UIComponent 类的子类或实现 IUIComponent 的接口(Interface)

20.4 ActionScript3 视觉架构的优越性
20.4.1 更高效的渲染，更好的内存利用
AS3 中影片剪辑被弱化成
绘制矢量使用轻量的 Shape 对象，
需要容器使用轻量的 Sprite 对象，
降低了时间轴的使用，也同时降低了内存的浪费。

20.4.2 自动化的深度管理
AS3 中的显示对象深度由程序自动管理。
每个 DisplayObjectContainer 实例都有 numChildren 属性，用于显示对象容器中的子对象数目
显示对象容器列表中对象的索引从 0 开始，到 numChildren-1

20.4.3 完整遍历显示列表

在 AS3 中，可以访问显示列表中的所有对象，包括使用 ActionScript 创建的对象及在 IDE 中绘制的对象。

20.4.4 列表外的显示对象

只有在显示列表中的对象才会被显示在舞台上，添加到显示列表的方法是调用 addChild()或 addChildAt()

第二十一章　DisplayObject 类与矢量图、位图

显示对象的可视属性：x， y， width， height，scaleX， scaleY， mouseX， mouseY(鼠标相对于对象注册点的距离)， rotation， alpha，visible

显示对象的常用非可视属性：

name（显示对象名字，字符串）

parent（指向显示对象父容器的引用，未加入显示列表则为 null）

root（当前 SWF 主类的实例的引用，未加入显示列表则为 null）

stage（指向显示对象所在的舞台的引用）

mask（持有的引用是用来遮罩的显示对象）

DisplayObject 是所有显示对象的抽象父类，有七个子类

InteractiveObject 抽象类

AVM1Movie，Bitmap，MorphShape，Shape，StaticText，Video 具体类

InteractiveObject 抽象类的子类：DisplayObjectContainer，SimpleButton，TextField

DisplayObjectContainer 抽象类的子类：Sprite，Stage，Loader

Sprite 的子类：MovieClip，mx.core.FlexSprite

MovieClip 的子类：mx.flash.UIMovieClip

mx.core.FlexSprite 的子类：mx.core.UIComponent

Shape，Sprite，MovieClip 类中都有 Graphics 对象

创建位图和使用 setPixel()改变位图

```
package org.kingda.book.display
{
    import flash.display.Sprite;
    import flash.display.BitmapData;
    import flash.display.Bitmap;

    public class SampleBitmap extends Sprite
    {
        public function SampleBitmap() {
            //生成两个 BitmapData 对象 dataA, dataB
            //dataA 是 100×100 的深绿色矩形
             var dataA:BitmapData = new BitmapData(100,100,true, 0xff669900);
            //dataB 是 100×100 的橙黄色矩形
            var dataB:BitmapData = new BitmapData(100,100,true, 0xffFF9900);
            //分别生成三个位图显示对象
            var bitmapA:Bitmap = new Bitmap(dataA);
            var bitmapB:Bitmap = new Bitmap(dataB);
            var bitmapC:Bitmap = new Bitmap(dataB.clone()); //将 dataB 复制了一份
            bitmapA.bitmapData = dataB;
            //bitmapA:将 dataB 替换了 dataA，此时 bitmapA 和 bitmapB 持有的都是 dataB
            bitmapB.x = 200;
            bitmapC.x = 400;
```

```
            //加入显示列表
            addChild(bitmapA);
            addChild(bitmapB);
            addChild(bitmapC);
            //改变 dataB 的像素信息，把它中心 20×20 的像素都改成了红色
            for (var i:int = 40; i<60; i++) {
                for (var j:int = 40; j<60; j++) {
                    dataB.setPixel(i,j,0xFF0000);
                }
            }
        }
    }
}
```

创建普通遮罩

```
package org.kingda.book.display
{
    import flash.display.Sprite;
    import flash.display.Loader;
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.display.BitmapData;
```

```
import flash.display.Bitmap;


public class SampleSimpleMask extends Sprite
{
    private var _bitmap:Bitmap;
    private var _circleMask:Sprite;
    public function SampleSimpleMask() {
        initMask();
        startLoadImg();
    }

    private function loaded(evt:Event):void {
        _bitmap = evt.target.content as Bitmap;
        addChild(_bitmap);
        _bitmap.mask = _circleMask;  //设置遮罩
    }

    private function initMask():void {
        _circleMask = new Sprite();
        _circleMask.graphics.beginFill(0xff0000);
        _circleMask.graphics.drawCircle(60,60,60);
        _circleMask.graphics.endFill();
```

```
            addChild(_circleMask);    //由于要拖动，所以要添加到显示列表中
            _circleMask.startDrag(true);
        }

        private function startLoadImg():void {
            var loader:Loader = new Loader();
            var request:URLRequest = new URLRequest("fla/desktop_madebykingda_1024.jpg");
            loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loaded);
            loader.load(request);
        }
    }
}
```

创建带有模糊边缘的 Alpha 圆形遮罩

```
package org.kingda.book.display
{
    import flash.display.Sprite;
    import flash.display.Loader;
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.display.BitmapData;
```

```
import flash.display.Bitmap;
import flash.filters.BlurFilter;
import flash.filters.BitmapFilterQuality;

public class SampleAlphaMask extends Sprite
{
    private var _bitmap:Bitmap;
    private var _circleMask:Sprite;
    public function SampleAlphaMask() {
        initMask();
        startLoadImg();
    }

    private function loaded(evt:Event):void {
        _bitmap = evt.target.content as Bitmap;
        addChild(_bitmap);
        _bitmap.cacheAsBitmap = true;
        _bitmap.mask = _circleMask;
    }

    private function initMask():void {
        _circleMask = new Sprite();
```

```
            _circleMask.graphics.beginFill(0xff0000);
            _circleMask.graphics.drawCircle(60,60,60);
            _circleMask.graphics.endFill();
            //用滤镜模糊化，产生 Alpha 通道渐变效果
            _circleMask.filters = [new BlurFilter(20,20, BitmapFilterQuality.HIGH)];
            //一定要将位图缓存打开，才能有 Alpha 遮罩效果
            _circleMask.cacheAsBitmap = true;
            addChild(_circleMask);
            _circleMask.startDrag(true);
        }

        private function startLoadImg():void {
            var loader:Loader = new Loader();
                                                    var request:URLRequest = new
URLRequest("fla/desktop_madebykingda_1024.jpg");
            loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loaded);
            loader.load(request);
        }
    }
}
```

捕获 HTML 文本超链接的信息

```
package org.kingda.book.display
{
    import flash.display.Sprite;
    import flash.text.TextField;
    import flash.events.TextEvent;
    import flash.net.URLRequest;
    import flash.net.navigateToURL;

    public class SampleTextLink extends Sprite
    {
        private var txt:TextField;
        public function SampleTextLink()
        {
            txt = new TextField();
            txt.width = 300;
            txt.wordWrap = true;
            addChild(txt);
            txt.htmlText =
                "点击<u><a href='event:geturl|http://www.kingda.org/|_blank'>这儿</a></u>
在" +
                "新窗口打开一个到 www.kingda.org 链接。<br/>" +
                "点击<u><a href='event:load|someswf.swf'>这儿</a></u>则是在 Flash 中加载
一个动画。<br/>" +
```

```
            "点击<u><a href='event:move|10'>这儿</a></u>则是将文本框移动 10 个像素。"

            txt.addEventListener(TextEvent.LINK, clickLink);

        }


        private function clickLink(evt:TextEvent):void {

            trace (evt.text);

            var cmdArray:Array = evt.text.split("|");

             switch (cmdArray[0]) {

                case "geturl":

                    geturl(cmdArray[1],cmdArray[2]);

                    break;

                case "load":

                    loadswf(cmdArray[1]);

                    break;

                case "move":

                    movetxt(cmdArray[1]);

                    break;

                default:

                    trace (cmdArray);

            }

        }
```

```
        private function geturl(url:String, target:String):void {
            var tmpRequest:URLRequest = new URLRequest(url);
            navigateToURL(tmpRequest, target);
        }
        private function loadswf(url:String):void {
            trace ("load a swf from :" + url);//用 trace 模拟一下.
        }
        private function movetxt(distance:Number):void {
            txt.x += distance;
        }

    }
}
```

改变动态文本内容可以使用高效率的方法 appendText()

碰撞检测：hitTestObject 和 hitTestPoint

```
package org.kingda.book.display
{
    import flash.display.Sprite;
    import flash.events.Event;
```

```
public class SampleHittest extends Sprite
{
    private var circle:Sprite;
    public function SampleHittest() {
        circle = new Sprite();
        circle.graphics.beginFill(0x669900);
        circle.graphics.drawCircle(0,0,10);
        circle.graphics.endFill();

        var rectA:RectSprite = new RectSprite("A", 0xffcc00);
        rectA.mouseChildren = false;
        rectA.name = "A";
        rectA.x = 100,      rectA.y = 50;

        var pointStar:StarShape = new StarShape();
        pointStar.x = 200,   pointStar.y = 200;
        pointStar.width = 10; pointStar.height = 10;
        addChild(pointStar);

        addChild(rectA);
        addChild(circle);
        circle.startDrag(true);
```

```
            rectA.addEventListener(Event.ENTER_FRAME, isHit);
        }
        private function isHit(evt:Event):void {
            if (circle.hitTestObject(evt.target as RectSprite)) {
                trace ("碰到了 Rect A");
            }
            if (circle.hitTestPoint(200, 200, true)) {
                trace ("碰到了坐标点（200，200）");
            }
        }
    }
}

//绘制圆角矩形
package org.kingda.book.display
{
    import flash.display.Sprite;
    import flash.text.TextField;

    public class RectSprite extends Sprite
    {
        private var _label:TextField;
```

```
        public function RectSprite(labelName:String, color:uint) {
            //以下四行使用 Sprite 内的 graphics 对象画一个圆角矩形背景
            this.graphics.lineStyle(2, 0x85DB18);
            this.graphics.beginFill(color);
            this.graphics.drawRoundRect(0,0,100,50,10,10);
            this.graphics.endFill();
            //生成文本框，并将 labelName 用 html 设成 24 号粗体
            _label = new TextField();
            _label.htmlText = "<font size='24'><b>"+labelName+"</b></font>";
            _label.selectable = false;
            _label.autoSize = "left";  //文本框高度、宽度随文本长度变化
            //将文本框加入 RectSprite 对象的显示列表
            addChild(_label);
        }

    }
}

//绘制星星
package org.kingda.book.display
{
    import flash.display.Shape;
```

```
import flash.display.GradientType;


public class StarShape extends Shape
{
    public function StarShape(x:Number=0, y:Number=0,
                              points:int=5, innerRadius:Number=20,
                              outerRadius:Number=50, angle:Number=0,
color:uint=0xff0000) {

        var count = Math.abs(points);
        this.graphics.lineStyle(2, 0x85DB18);

        //开始填色
        this.graphics.beginFill(color);
        if (count>2) {
            // init vars
            var step, halfStep, start, n, dx, dy;
            //计算两点之间距离
            step = (Math.PI*2)/points;
            halfStep = step/2;
            //起始角度
            start = (angle/180)*Math.PI;
```

```
                                        this.graphics.moveTo(x+(Math.cos(start)*outerRadius),
y-(Math.sin(start)*outerRadius));
                    //画星状图的边
                    for (n=1; n<=count; n++) {
                        dx = x+Math.cos(start+(step*n)-halfStep)*innerRadius;
                        dy = y-Math.sin(start+(step*n)-halfStep)*innerRadius;
                        this.graphics.lineTo(dx, dy);
                        dx = x+Math.cos(start+(step*n))*outerRadius;
                        dy = y-Math.sin(start+(step*n))*outerRadius;
                        this.graphics.lineTo(dx, dy);
                    }
                }
                this.graphics.endFill();
            }
        }
}
```

第二十二章　DisplayObjectContainer　与　Sprite MovieClip等

容器对象.addChild(显示对象);　//在一个容器里添加显示对象，返回显示对象

容器对象.removeChild(显示对象);　//从一个容器中移掉一个显示对象，返回显示对象

容器对象.contains(显示对象); //检测当前容器的显示列表是否包含此显示对象，返回布尔值

当传入的参数不合理时会抛出 ArgumentError 异常。

显示对象独立于显示列表

```
package org.kingda.book.display
{
	import flash.display.Sprite;

	public class SampleIndependentState extends Sprite
	{
		public function SampleIndependentState() {
			//生成一个 RectSprite。由于处于同一包中，不需要再 import
			var foo:RectSprite = new RectSprite("foo", 0xFFCC00);
			//设置好 foo 的状态：坐标、透明度、横向缩放、旋转
			foo.x = 100;
			foo.y = 100;
			foo.alpha = 0.5;
			foo.scaleX = 0.5;
			foo.rotation = 30;//注，添加旋转角度后，动态文本框不会显示。这是正常的。详见“文本”一章
```

```
                    //添加 foo 后，foo 在舞台的样子和状态定义的一样
                    addChild(foo);
                    //再移除 foo
                    removeChild(foo);
                    //再加上 foo，发现 foo 的状态仍然保留
                    addChild(foo);


                    //生成新容器 fooSprite
                    var fooSprite:Sprite = new Sprite();
                    //将 fooSprite 下移 200 像素
                    fooSprite.y = 200;
                    //添加 foo 到 fooSprite 中，此时由于 fooSprite 尚未添加入显示列表，
也不会显示
                    fooSprite.addChild(foo);
                    //fooSprite 加入显示列表，发现 foo 状态也没变，只不过跟着 fooSprite
往下移了 200 像素
                    //这是因为子可视对象的坐标是相对于父容器的，父容器坐标不同，
子可视对象屏幕显示会跟着变
                    addChild(fooSprite);
                }
        }
}
```

多次添加同一个显示对象

（同一个显示对象不论被代码加入显示列表多少次，在屏幕上只会有一个显示对象）

```
package org.kingda.book.display
{
	import flash.display.Sprite;
	import flash.events.MouseEvent;

	public class SampleAdd extends Sprite
	{
		private var conA:Sprite;
		private var conB:Sprite;
		private var buttonA:RectSprite;
		private var buttonB:RectSprite;
		private var buttonX:RectSprite;
		private var star:StarShape;

		public function SampleAdd() {
			star = new StarShape();
			star.y = 100; //相对于父容器的坐标
			conA = new Sprite();
			conB = new Sprite();
			conB.x = 200;
```

```
            addChild(conA);
            addChild(conB);

            buttonA = new RectSprite("A", 0xff9900);
            buttonB = new RectSprite("B", 0x669900);
            buttonX = new RectSprite("X", 0x669900);
            //对按钮位置布局
            buttonA.y = 100;
            buttonB.x = 150;
            buttonB.y = 100;
            buttonX.y = 160;
            buttonX.x = 50;
            addChild(buttonA);
            addChild(buttonB);
            addChild(buttonX);
            //添加鼠标单击事件的侦听
            buttonA.addEventListener(MouseEvent.CLICK, addStarInContainer);
            buttonB.addEventListener(MouseEvent.CLICK, addStarInContainer);
            buttonX.addEventListener(MouseEvent.CLICK, removeStar);
            trace(this.contains(star));
            //输出：false。现在 star 还没有加入显示列表。点击按钮后才会加入。
        }
```

```
//根据事件的发出者，确定那个容器添加星星
private function addStarInContainer(evt:MouseEvent):void {
        if(evt.currentTarget == buttonA) {
                conA.addChild(star);
                trace ("容器 A 加入了星星");
        }else {
                conB.addChild(star);
                trace ("容器 B 加入了星星");
        }
}
//移除星星
private function removeStar(evt:MouseEvent):void {
        //使用 contains 判断
        if (conA.contains(star)) {
                conA.removeChild(star);
                trace ("容器 A 移除了星星");
        } else if (conB.contains(star)) {
                conB.removeChild(star);
                trace ("容器 B 移除了星星");
        } else {
                trace ("星星已经不存在，不需要再移除");
        }
```

```
            }
        }
}
```

每个容器摇篮有的子显示对象总数记在 numChildren 里，从 0 到 numChildren-1。

如果子显示对象深度发生冲突，其它子显示对象会自动做调整。

用 graphics 绘出的矢量图不是 Shape 显示对象，也不在它的容器对象的子对象列表中。它始终处于容器对象所有子对象的下面。因此可以用来做背景图。

深度和背景图

```
package org.kingda.book.display
{
    import flash.display.Sprite;

    public class SampleDepth extends Sprite
    {
        public function SampleDepth() {
            var foo:Sprite = new Sprite();
            //这儿使用同包的 StarShape 类和 RectSprite 类，不需要 import
            var star:StarShape = new StarShape();
            var rect:RectSprite = new RectSprite("Rect", 0xFF9900);
            star.x =100;
            star.y = 100;
            rect.x = 50;
```

```
                rect.y = 50;
                foo.x = 50;
                foo.y = 50;
                //两个都加入 foo 中。读者可以将下面两行顺序互换，查看效果
                foo.addChild(star);
                foo.addChild(rect);
                star.name = "kingda";
                rect.name = "kingda";
                //使用 getChildIndex 方法得到这两个子对象在列表中的索引值
                trace (foo.getChildIndex(star));//输出：0
                trace (foo.getChildIndex(rect));//输出：1
                trace (foo.getChildByName("kingda")); //[object StarShape]
                //使用 foo 绘制灰色圆形，会发现这个矢量图成为背景图
                foo.graphics.beginFill(0xCCCCCC);
                foo.graphics.drawCircle(100,100,100);
                foo.graphics.endFill();
                //将 foo 加入显示列表
                addChild(foo);
                trace (foo.numChildren);    //2
            }
        }
}
```

容器对象.addChildAt(显示对象，深度);　//在指定深度上加入显示对象，返回显示对象

容器对象.removeChildAt(深度); //去掉某个深度的显示对象，返回显示对象

如果传入的显示对象不合理会抛出 ArgumentError 异常对象，如果深度值在容器列表中不存在，如超出 numChildren-1，就抛出 RangeError 异常。

交换不同深度的对象

容器对象.swapChildren(显示对象 A，显示对象 B);　//会抛出 ArgumentError 异常

容器对象.swapChildrenAt(深度 A，深度 B);　//会抛出 RangeError 异常

重设列表中已有对象的深度

容器对象.setChildIndex(显示对象,指定深度);　//会抛出 ArgumentError 和 RangeError 异常

常用方法：

置顶：容器对象.setChildIndex(显示对象 A, (容器对象.numChildren-1));

置底：容器对象.setChildIndex(显示对象 A, 0);

插到 B 前面：容器对象.setChildIndex(显示对象 A, 容器对象.getChildIndex(显示对象 B));

插到 B 后面：容器对象.setChildIndex(显示对象 A, 容器对象.getChildIndex(显示对象 B)-1);

操作深度的具体代码例子

```
package org.kingda.book.display
{
        import flash.display.Sprite;
```

```
public class SampleSwapDepth extends Sprite
{
    public function SampleSwapDepth() {
        //代码段 1：
        //生成五个不同颜色的 RectSprite 矩形对象
        var a:RectSprite = new RectSprite ("A", 0xB9121B);
        var b:RectSprite = new RectSprite ("B", 0x4c1B1B);
        var c:RectSprite = new RectSprite ("C", 0xF6E497);
        var d:RectSprite = new RectSprite ("D", 0xFCFAE1);
        var e:RectSprite = new RectSprite ("E", 0xBD8D46);
        //摆放位置
        a.x = 100, a.y = 100;
        b.x = 120, b.y = 120;
        c.x = 140, c.y = 140;
        d.x = 160, d.y = 160;
        e.x = 180, e.y = 180;
        //加入显示列表
        addChild(a);
        addChild(c);
        addChild(e);

        //代码段 2：
```

```
addChildAt(b,1);

//因为 a 是第一个对象索引为 0，要放在 a 之前，所以设为 1

//原来在 1 的可视对象 c 以及其他可视对象，自动往后移位；

trace (getChildIndex(c));//输出：2

trace (getChildIndex(e));//输出：3


//代码段 3：

addChild(d);

//直接调换可视对象 d 和 e 的深度

swapChildren(d,e);

//将倒数第一位的深度和倒数第二位深度上的可视对象对调

//swapChildrenAt((this.numChildren-2),(this.numChildren-1));


//代码段 4：

setChildIndex(c,(this.numChildren-1));

trace (getChildIndex(d));//输出：2

trace (getChildIndex(e));//输出：3


//代码段 4：

setChildIndex(a,getChildIndex(d));

trace ("=========")

trace (getChildIndex(a));    //2
```

```
                trace (getChildIndex(b));    //0
                trace (getChildIndex(c));    //4
                trace (getChildIndex(d));    //1
                trace (getChildIndex(e));    //3
            }
        }
}
```

访问子显示对象的三种方式

1．　通过深度访问子显示对象（速度最快，效率最高）

容器对象.getChildAt(深度)；//返回值类型是 DisplayObject 抽象类型，要转型

```
var someSprite:Sprite = container.getChildAt(0) as Sprite
package org.kingda.book.display
{
import flash.display.Sprite;
import flash.display.DisplayObject;
import flash.display.DisplayObjectContainer;
import flash.display.Shape;

public class SampleAccessByDepth extends Sprite
{
        public function SampleAccessByDepth() {
```

```
var a:RectSprite = new RectSprite ("A", 0xB9121B);

var b:RectSprite = new RectSprite ("B", 0x4c1B1B);

var c:RectSprite = new RectSprite ("C", 0xF6E497);

var container:Sprite= new Sprite();

a.x = 100, a.y = 100;

b.x = 120, b.y = 120;

c.x = 140, c.y = 140;

//生成一个匿名的 StarShape 对象加入 b 中

b.addChild(new StarShape());

//加入显示列表

container.addChild(a);

container.addChild(b);

container.addChild(c);

addChild(container);


var star:Shape;


for (var i:int = 0; i<container.numChildren; i++) {

        for ( var j:int = 0;

                j<(container.getChildAt(i) as Sprite).numChildren;

                j++    )
```

```
                    {
                        var tmpObj:DisplayObject =
                            (container.getChildAt(i) as Sprite).getChildAt(j);
                        if (tmpObj is StarShape) {
                            star = tmpObj as StarShape;
                            trace (i + "|" + j);    //1|1
                        }
                    }
                }
                if (star != null) star.alpha = 0.5;
            }
        }
    }
```

遍历容器的所有子显示对象

```
package org.kingda.book.display
{
import flash.display.Sprite;
import flash.display.DisplayObjectContainer;
import flash.display.DisplayObject;

public class SampleTraversingList extends Sprite
```

```
{
        public function SampleTraversingList() {
                var container:Sprite = new Sprite();
                var rect:RectSprite = new RectSprite ("A", 0xB9121B);
                container.addChild(rect);
                container.addChild(new StarShape());
                addChild(container);
                rect.name = "KingdaRect";
                traverseDisplayContainer(container);



                trace ("----------------");
                traverseDisplayContainer(this);


        }


        public static function traverseDisplayContainer(
container:DisplayObjectContainer,
                                                        indentString:String        =
""):void
        {
                var child:DisplayObject;
```

```
                    for (var i:uint=0; i < container.numChildren; i++) {
                        child = container.getChildAt(i);
                        trace (indentString,"depth:"+i, child, child.name);
                        //如果发现是容器，则递归
                        if (container.getChildAt(i) is DisplayObjectContainer) {
                            traverseDisplayContainer(DisplayObjectContainer(child),
indentString + "    ");
                        }
                    }
        }
}
}
```

实用的遍历工具函数(mx.utils.DisplayUtil 工具类提供)：

DisplayUtil.walkDisplayObjects(displayObject:DisplayObject, callbackFunction:Function);

上例可以简化为：

```
import mx.utils.DisplayUtil;
DisplayUtil.walkDisplayObjects(container, traceContent);
function traceContent(d0:DisplayObject):void{
    Trace(“depth:” + d0.parent.getChildIndex(d0), d0, d0.name);
}
```

2．　通过 name 访问显示对象

容器对象.getChildByName(“显示对象名字”);

```
package org.kingda.book.display
{
	import flash.display.Sprite;

	public class SampleAccessByName extends Sprite
	{
		public function SampleAccessByName(){
			var container:Sprite= new Sprite();
			var rect:RectSprite = new RectSprite ("A", 0xB9121B);
			container.addChild(rect);
			container.addChild(new StarShape());
			addChild(container);
			rect.name = "kingdaRect";
			trace(container.getChildByName("kingdaRect"));    //[object RectSprite]
		}
	}
}
```

3．　通过坐标访问显示对象

容器对象.getObjectsUnderPoint(点对象);

//返回数组，包含该容器在这个坐标点下的所有显示对象。子对象，容器都在内。

这个坐标是全局坐标，如果需要相对坐标要使用 localToGlobal()进行坐标转换。

如果该坐标下的显示对象是从其他域载入的，那么要使用 Security.allowDomain(),才可以使用该方法。可先用 areInaccessibleObjectsUnderPoint()判断是否有这样不可访问的显示对象。

```
package org.kingda.book.display

{

        import flash.display.Sprite;

        import flash.geom.Point;

        import flash.display.Shape;

        import flash.text.TextField;



        public class SampleAccessByCoordinates extends Sprite

        {

                public function SampleAccessByCoordinates(){

                        var container:Sprite= new Sprite();

                        var a:RectSprite = new RectSprite ("A", 0xB9121B);

                        var b:RectSprite = new RectSprite ("B", 0x4c1B1B);

                        var c:RectSprite = new RectSprite ("C", 0xF6E497);

                        a.x = 100, a.y = 100, a.name = "a";

                        b.x = 120, b.y = 120, b.name = "b";

                        c.x = 140, c.y = 140, c.name = "c";

                        //加入显示列表

                        container.addChild(a);
```

```
            container.addChild(b);
            container.addChild(c);
            addChild(container);

            var targetPoint:Point = new Point(150, 125);
            var resultArray:Array = container.getObjectsUnderPoint(targetPoint);
            trace(resultArray);
            //[object RectSprite],[object RectSprite]
            for each(var i in resultArray){
                if(i is RectSprite) trace(i, i.name);
                if(i is TextField) trace(i, i.text);
            }
            //[object RectSprite] a
        //[object RectSprite] b    出来的结果跟书上有出处，没有显示出 TEXTIELD 子对象

            //画出点的位置
            var pointShape:Shape = new Shape();
            pointShape.graphics.beginFill(0xFFFFFF);
            pointShape.graphics.drawCircle(targetPoint.x, targetPoint.y, 3);
            pointShape.graphics.endFill();
            addChild(pointShape);
    }
```

```
        }
}
```

MovieClip 是动态类，Sprite 是密封类，前者比后者多了对时间轴的支持

MovieClip 的实例属性：

currentFrame, currentLabel, totalFrames, currentScene, scenes, currentLabels, enabled

MovieClip 的实例方法：

play(), stop(), gotoAndPlay(帧数或标签字符串，场景), gotoAndStop(帧数或标签字符串，场景), nextFrame(), prevFrame(), nextScene(), prevScene()

在关键帧上添加代码，使用 addFrameScript()

注意点：帧数索引是从 0 开始的；参数传入的帧数超出影片帧数范围则不会被执行；如果指定帧上已有代码，那么 addFrameScript()会替换原有代码；要删除指定的关键帧上代码，使用 addFrameScript(帧数索引，null)；

```
function afunc():void{
            trace(ball_mc.currentFrame);
            trace(“other code here…”);
}
ball_mc.addFrameScript(1, afunc, 3, afunc);
```

加载外部图像和 SWF 文件

标准方式是使用 Loader 类对象，Loader 也是容器，如果加载成功，会保存在 content 实例属性中。Loader 可用 load()读取图像或 SWF，也可用 loadBytes()读取 BytesArray 对象中的数据。

Loader 在加载时，它的 contentLoaderInfo 属性持有的 LoaderInfo 对象会发出加载进度事件。

监听加载进度可以监听它发出的 ProgressEvent.PROGRESS 事件，该事件对象中含有 bytesLoaded 和 bytesTotal 数据。

加载 SWF，访问其中脚本，提取其中的类定义

```
package org.kingda.book.display
{
        import flash.display.Sprite;
        import flash.display.Loader;
        import flash.display.MovieClip;
        import flash.events.Event;
        import flash.net.URLRequest;
        import flash.system.ApplicationDomain;
        import flash.display.LoaderInfo;

        public class SampleLoadSWF extends Sprite
        {
                private const FILE_PATH:String = "fla/MotionXML.swf";
                private const CLASS_NAME:String = "Ball";

                private var loader:Loader;
                private var request:URLRequest;
```

```
                public function SampleLoadSWF() {
                        loader = new Loader();
                        request = new URLRequest(FILE_PATH);
                        loader.contentLoaderInfo.addEventListe
ner(Event.COMPLETE, onComplete);
                        loader.load(request);
                }

                private function onComplete (event:Event):void {
                        var loadedSWFInfo:LoaderInfo =
event.target as LoaderInfo;
                        var domain:ApplicationDomain =
loadedSWFInfo.applicationDomain as ApplicationDomain;
                        //从加载的 SWF 域中提取类定义
                        var BallClass:Class =
domain.getDefinition (CLASS_NAME) as Class;
                        var ballA:MovieClip   = (new
BallClass()) as MovieClip;
                        var ballB:MovieClip   = (new
BallClass()) as MovieClip;
                        ballA.x = 0,ballA.y = 100;
                        ballB.x = 100, ballB.y = 100;
                        ballB.scaleX = 2;
                        addChild (ballA);
```

```
                                        addChild (ballB);



                                        var loadedSWF:MovieClip =
loadedSWFInfo.content as MovieClip;

                                        trace(loadedSWF == loader.content);

                                        //输出:true

                                        //表明 contentLoaderInfo.content 与
loader.content 确实一样，指向被加载 SWF 主类对象

                                        trace (loadedSWF.welcomeString);

                                        //输出：哈哈，你访问到被加载的 SWF 脚本
了。

                                        //这说明访问到了 MotionXML.swf 主时间轴
定义的 var welcomeString

                                        //同理，也可访问其他子显示对象或其他脚
本也是

                                        addChild(loadedSWF);//将加载的 SWF 对象
加入显示列表，这是才会在屏幕显示出来。

                                }

            }

}
```

访问被加载 SWF 文件的参数及访问网页传入的参数

如当前 SWF 被传入参数，“a.swf?name=kingda”，那么可以这样访问：

this.stage.loaderInfo.parameters.name;

parameters 对象是一个 Object 对象，所以可以遍历

```
for(var i in this.stage.loaderInfo.parameters){ trace(i+ “:” +
this.stage.loaderInfo.parameters[i]); }
```

10 月 26 日

==========

第 23 章 Flash CS3：库元件的类绑定与

==========

23.1 类绑定的好处、原理和第一个例子

类绑定使元件由原来只可以使用 attachMovie、createEmptyMovieClip 等创建实例的途径，变成可以直接使用 new 类名()的类。

23.1.1 第一个类绑定的例子：自定义标签按钮

1.打开库元件面板；
2.右键单击要绑定类的剪辑元件，选择“属性”；
3.在属性面板选中“为 ActionScript 导出”；
4.在“类”输入框中填入绑定类的全部路径；
5.基类可根据剪辑元件情况填入 flash.display.MovieClip 或 flash.display.Sprite。

23.1.2 Flash 创作工具中的类绑定

文档类可以看成编译出来的 SWF 为一个大的影片剪辑和一个文档类进行绑定。

23.1.3 帧代码和类绑定的角色比较

应用原则是：与具体剪辑元件相关的代码写在帧代码中；和具体剪辑元件无关的通用代码尽量写在绑定类中。

23.1.4 绑定的选择：MovieClip 子类和 Sprite 子类

Flash 默认创建的元件都是基于 MovieClip 类，如果绑定元件不需要时间轴支持，在元件属性设置基类中把 flash.display.MovieClip 改成 flash.display.Sprite。

23.2 如何在绑定类中访问影片内的子显示元件

在绑定类中可以用名字直接访问舞台上用属性面板命名过的影片剪辑。因为 Flash 默认设置了“自动声明舞台实例”。Flash 在编译时会进行声明，但在其他 AS3 IDE 编译时报错。

取消方法：文件->发布设置->Flash 标签->脚本：AS 边的设置->取消勾选，舞台：自动声明舞台实例。

23.3 绑定类中访问剪辑子元件的两种方法

23.3.1 手工声明 public 属性方法

在 flash 关闭“自动声明舞台对象”选项，访问剪辑子元件：
1.手工增加的属性必须和剪辑子元件的命名一致；
2.属性必须使用 public。

不推荐在标准 RIA 开发项目中使用这种方法。

23.3.2 自字义新属性方法

打开“自动声明舞台对象”选项，或使用手工声明后：

1.新增一个属性，类型可以不同，名字与子元件名不能重复；
2.将子元件引用赋值给新增属性，再对新属性操作。方法是：在构造函数使用**getChildByName(“子元件名”)**将子元件引用赋给新属性，使用 as 关键字将返回显示对象转为新属性的类型。
优点是降低与美术元件耦合度，保护类定义的封装。

比较好的代码编写习惯：
1.新增属性以两个下划线开头，可以表明这个属性与 fla 文档库元件有关；
2.新增属性赋值在 initView()方法中进行，方便进行修改。

23.4 三种不同的类绑定

对影片剪辑元件 3 种绑定类型：
1.自动生成的类绑定；
2.影片剪辑绑定独立的类文件；
3.多个影片剪辑通过自动生成类继承同一个类。

23.4.1 自动生成的类绑定及用代码直接生成库元件对象

代码直接生成库元件对象：
1.在类文件新建一个 Class 类型的属性；
2.在构造函数中使用 getDefinitionByName(“库元件类名”)得到引用，使用 as 转型成 Class 类型，赋值给 Class 类型属性。
例如：

```
…
private var __Spider:Class;                    //预备 Class 类型
private var spider:MovieClip;                  //预备预用的剪辑对象
…构造函数调用 initView()
private function initView():void{              //专门处理显示对象的方法
    __Spider=getDefinitionByName(“Spider_mc”) as Class;     //获得库元件自动绑定类
    spider=new __Spider();                     //通过库元件转为的类新建剪辑对象
    addChild(spider);
}
…
```

23.4.2 元件与独立的类绑定

每个 Fla 文件库中不能有两个或以上的元件绑定同一个类文件。

23.4.3 让不同的库元件继承自共同的类

希望同一个 fla 文件库中多个元件实现相同的行为，可以将要绑定的类文件设为基类，这些元件都继承同一个基类，而绑定类由编译时自动生成。

23.5 舞台与文档类

AS3 中舞台是根容器，根容器下面是 SWF 主类的实例（文档类或 MainTimeLine 类）。

所有显示对象的 stage 属性指向舞台，root 属性指向 SWF 主类的实例。

==========

第 24 章 显示编程与事件、人机交互

==========

24.1 事件的事件流（Event Flow）机制

24.1.1 事件流机制的 3 个阶段

三个阶段：1.捕获阶段、2.目标阶段、3.冒泡阶段；
事件流机制好处是可以选择事件流机制中所经过的任意一个节点（即容器）来添加对事件的侦听（触发到的显示对象所在的各级父容器也会被触发，而可以在父级容器侦听事件）。

24.1.2 事件 currentTarget 属性和 target 属性

target：表示发生事件的显示对象，一般是处于最里层；
currentTarget：表示当前侦听事件的节点，一般是容器。

24.1.3 事件侦听时的事件流阶段

Event 对象的事件流阶段由 eventPhase 属性表示：值为 1，表示在捕获阶段；值为 2，表示在目标阶段；值为 3，表示在冒泡阶段。
侦听器调用一般在冒泡阶段，当事件经过添加了侦听器的容器时才会被侦听到，如果当前侦听显示对象就是事件发生对象，那么就是目标阶段发生了侦听器调用。

24.1.4 事件都能冒泡吗？

1.事件本身是否支持冒泡，由 bubbles 属性设定，默认 false 不支持；
2.如果事件发生对象不在显示列表中，也不能冒泡。
另：事件发生对象在显示列表中，发生事件的捕获阶段始终存在。

24.1.5 说明事件流机制的例子

侦听事件方法中如：

```
…
Private function clickHandler(evt:MouseEvent):void{
      trace(evt.target.name);//获得事件发生目标，直接触发的显示对象
      trace(evt.currentTarget.name);//获得侦听到事件的目标，设置侦听的显示对象
     trace(evt.eventPhase);//获得事件流当前的阶段，值：1 在捕获时触发，2 事件触发和侦听为同对象，捕获后回到目标时就触发，3 冒泡时触发
}
…
```

24.2 鼠标相关事件

鼠标事件对象属于 MouseEvent 类，事件共有 10 种：

- 单击：MouseEvent.CLICK（单击）、MouseEvent.DOUBLE_CLICK（双击）；
- 按键状态：MouseEvent.MOUSE_DOWN、MouseEvent.MOUSE_UP；
- 鼠标悬停或移开：MouseEvent.MOUSE_OVER、MouseEvent.MOUSE_OUT、MouseEvent.ROLL_OVER、MouseEvent.ROLL_OUT；

- 鼠标移动：MouseEvent.MOUSE_MOVE；
- 鼠标滚轮：MouseEvent.MOUSE_WHEEL。

属性分两类：

- 当前鼠标坐标：相对坐标 localX、localY；舞台坐标 stageX、stageY；
- 相关按键是否按下（Boolean 类型）：altKey、ctrlKey、shiftKey、buttonDown。

24.2.1 鼠标事件对象的目标

当鼠标发出事件对象时，这个事件对象的目标属性（target）是鼠标所触发的最里层的显示对象。

将容器的 mouseChildren 属性设为 false，容器内的子显示对象不会参与鼠标互动，而鼠标事件对象 target 属性就会指向该容器。
将容器的 mouseEnabled 属性设为 false，容器自身不接受鼠标互动，与 mouseChildren 相反。

24.2.2 单击和双击、鼠标按下和松开

单击事件 MouseEvent.CLICK；双击事件 MouseEvent.DOUBLE_CLICK；都与鼠标按下与松开有关。
单击事件发生过程：
目标（target）一致
鼠标按下事件 MouseEvent.MOUSE_DOWN -> 鼠标松开事件 MouseEvent.MOUSE_UP -> 单击事件 MouseEvent.CLICK，

双击事件对象在事件发生之前需要将对象 doubleClickEnabled 属性设为 true，默认为 false。
双击事件发生过程：
target 一致
MouseEvent.MOUSE_DOWN -> MouseEvent.MOUSE_UP -> MouseEvent.CLICK -> MouseEvent.MOUSE_DOWN -> MouseEvent.MOUSE_UP -> MouseEvent.DOUBLE_CLICK

24.2.3 鼠标悬停和鼠标移走

悬停和移走两对事件：
1.MouseEvent.MOUSE_OVER、MouseEvent.MOUSE_OUT；
2.MouseEvent.ROLL_OVER、MouseEvent.ROLL_OUT，的 bubbles 属性为 false，这两个事件不参与事件流冒泡。

对于不参与事件流冒泡或鼠标互动的显示对象，可以对容器使用 mouseChildren=false，但既需要悬停和移出效果，又要保留子对象能接受其他人机事件，使用 ROLL_OVER 和 ROLL_OUT 最合适。

24.2.4 用代码模拟鼠标输入

AS3 提供了 MouseEvent 的构造函数来模拟鼠标输入。

模拟鼠标输入有两步：
1.新建鼠标事件对象；
2.选择发送鼠标事件的对象，用它调用 dispatchEvent()。
例如：

```
…
var clickEvt:MouseEvent=new MouseEvent
                    (MouseEvent.CLICK,true,false,0,0,null,true,true,false,t
                rue,0);
var kk:RectSprite=new RectSprite("kk",0xff9900);
addChild(kk);
kk.dispatchEvent(clickEvt);
…
```

24.3 键盘相关事件

键盘交互事件：flash.events.KeyboardEvent，总共两个事件类型：

- 按键按下：KeyboardEvent.KEY_DOWN；
- 按键弹起：KeyboardEvent.KEY_UP。

按了什么键是记录在 KeyboardEvent 对象的 keyCode 属性中，属性 charCode 记录按键输出的字符的字符集值。

KeyboardEvent 只有 6 个属性：
按键信息（uint 型）：keyCode 和 charCode；
辅助键信息（Boolean 型）：altKey、ctrlKey 和 shiftKey；
按键区域：keyLocation。

24.3.1 监听键盘输入的例子

KeyboardEvent 事件是冒泡的。
String.fromCharCode(evt.charCode)//获得按键字符串

24.3.2 全局监听键盘事件

全局监听键盘事件，只要将事件发送者设为舞台即可，使用显示列表中任何一个显示对象的 stage 属性来添加键盘事件侦听。

24.3.3 Tab 键与 tqbChildren

控制容器内的对象可否接受[Tab]键控制，可以使用 tabChildren 属性（Boolean 型）。与 mouseChildren 类似。

24.4 拖曳、鼠标跟随

24.4.1 拖曳

只有 Sprite 及其子类才能使用拖曳方法和属性：
开始拖曳：startDrag()；
停止拖曳：stopDrag()；
放置目标：dropTarget。

24.4.2 鼠标跟随

鼠标跟随是使用 MouseEvent.MOUSE_MOVE 事件来实现；
Mouse.hide()将正常鼠标指针隐藏；
鼠标事件的 updateAfterEvent()是事件处理完成后将指示 Flash Player 呈现结果，让鼠标跟随画面更加流畅。

第 25 章 ActionScript 3 动画编程

==========

25.1 动画编程原理

25.1.1 TimerEvent.TIMER 和 Event.ENTER_FRAME 比较

AS3 中产生动画方式有两种：

1.定时更新：每隔一定时间让显示对象改变一次，使用 Timer 类对象定时发出事件，并侦听该对象的 TimerEvent.TIMER 事件。

2.每帧更新：跟随 Flash 本身播放速度，在每次 Flash 屏幕更新时改变显示对象，侦听该对象的 Event.ENTER_FRAME 事件。

一般使用 Event.ENTER_FRAME 事件。

25.1.2 Event.ENTER_FRAME 事件与 ActionScript2 中的 onEnterFrame

AS3 每次屏幕更新都会发出 Event.ENTER_FRAME 事件，因些会调用到侦听器函数，从而实现调用动画代码。

另外，一旦不需要动画，不需要侦听该事件时，一定要移除侦听。

25.2 代码绘制

代码绘制动画编程，每帧都用代码来绘制一些稍不同的图形，一般为矢量动画绘制，使用 Graphics 对象提供 API 来绘制图形。

25.3 改变显示对象属性

改变显示对象属性动画编程，是在每帧中改变已有显示对象的属性，包括添加滤镜。

25.4 使用 Timer 类实现动画效果

实现方法，生成一个 Timer 类实例，构造函数中传入参数告诉 Timer 对象事件间隔时间、事件次数，当调用 Timer 对象 start()方法后，就开始发出计时事件。

共有两种事件：

1.TimerEvent.TIMER：计时事件，每隔指定时间发出；

2.TimerEvent.TIMER_COMPLETE：计时结束事件，计时结束时发出。

25.5 借助 fl.transitions 包中的现有类创建动画代码

25.5.1 Tween 类的多种用法

Tween 构造函数中指定变化：目标对象（obj）、目标属性（prop）、变化方式（func）、起始值（begin）、终止值（finish）、动画长度（duration）、是否以时间计时（useSeconds）；

最常用的是动画结束事件 TweenEvent.MOTION_FINISH。

格式：

```
Tween(obj:Object,prop:String,func:Function,begin:Number,finish:Number,duration:Number,useSeconds:Boolean=false)
```

例如：

```
…

var xTween:Tween=new
Tween(star,"x",Elastic.easeOut,100,200,2,true);

xTween.addEventListener(TweenEvent.MOTION_FINISH,continueMove);

…
```

25.5.2 TransitionManager 的用法

使用 TransitionManager 可以看成是对一个指它对象执行预定义好的动画效果。

例如：

```
…
```

```
addChild(starMov);
starMov.x=100;starMov.y=100;
var trans:TransitionManager=new TransitionManager(starMov);
trans.startTransition({       type:Fly,
      direction:Transition.IN,
      duration:1,
      easing:Bounce.easeOut});
…
```

25.6 Flash 创作工具与动画编程的结合：Animator 运用

Flash 中可以将自定义的补间动画复制成 AS3 代码，并且可以将这些代码运用在其他视觉对象上。这些补间都是用 XML 描述的。

25.6.1 原理：MotionXML 的构造和 Animator 的使用

功能是在 fl.motion.*包中类所实现，一般只需用到 Animator 类。

MotionXML 中包含着两类子节点：

1.source 节点：是用来记载显示对象初始状态；

2.Keyframe 节点：对应着动画中的关键帧。

根节点是 Motion，对应 fl.motion.Motion 类。

25.6.2 具体演示和代码实例

1.打开动画 fla 文件，选择所有关键帧，用鼠标右键单击“复制为 ActionScript3 代码”，粘贴到文本编辑器，将 XML 对象 XML 代码复制出来另存为 xml 文件；

2.在代码中使用 URLRequest 和 URLLoader 来加载 xml 文件；

3.再通过监听事件读到 evt.target.data 转成 XML 对象，如：_xml=new XML(evt.target.data);

4.生成 Animator 对象，将显示对象和动画 xml 绑定，如：_animator=new Animator(_xml,_star);

5.Animator 提供所有控制动画播放方法，如：_animator.play();

25.6.3 Animator 使用技巧和资源

Animator 对象播放时会发出 fl.motion.MotionEvent 事件类对象，3 个事件是：

- MotionEvent.MOTION_END：动画完成后发出（常用）；
- MotionEvent.MOTION_START：动画开始播放时发出；
- MotionEvent.MOTION_UPDATE：动画更改并屏幕更新发出事件。

Animator 对象可以改变 motion 对象和目标显示对象：

- motion 属性改变动画内容，如：_animator.motion=newMotion;
- target 属性改变目标显示对象，如：_animator.target=newBall;

==========

第 26 章 Flash 创作工具和 Flex 协作开发组件

==========

26.1 浅谈 FlashCS3 组件与类绑定的剪辑元件

Flash 组件也是通过类绑定的方式实现，一个类绑定元件完全符合组件标准。在 AS3 中一般可以直接使用类绑定来做广义上的组件，而不必拘泥于 Flash 组件架构。

26.2 Flex 组件与 FlashCS3 创作内容如何结合

26.2.1 Flex 组件架构对显示对象的要求

在 Flex 框架中，只有 UIComponent 类型或 UIComponent 类型的显示对象才可以被加入到 Flex 程序中去。Flex 中添加纯容器，那么只能使用对应的 UIComponent 类。

26.2.2 Flex Component Kit for Flash CS3 的安装和使用

1.运行 Adobe Extension Manager CS4 安装 Flash 扩展组件，FlexComponentKit.mxp（转换 Flex 组件）和 Flex_Skins_12_05.mxp（转换 Flex 容器）；

2.在 FlashCS4 命令中增加了“将元件转换为 Flex 组件”和“将元件转换为 Flex 容器”，选择要转换的元件使用对应命令即可；

3.右键点击已经转换的元件，选择“导出 SWC 文件”；

4.Flex 项目中导入 SWC 文件（右键点击项目或 Project->Properties->Flex Build Path->Library path->Add SWC）；

5.mxml 中要加入原 SWC 元件的类路径，如：<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute" xmlns:max="org.display.*">这样就可以使用该包内的组件。

26.2.3 Flash CS3 和 Flex 的桥梁：UIMovieClip

UIMovieClip 类处于 mx.flash 包中，继承自 MovieClip。UIMovieClip 实现了 IDeferrendInstantiationUIComponent 接口（IUIComponent 子接口），是 Flex 架构中的基本成员。

26.3 如何开发 UIMovieClip 组件

Flash 中将影片剪辑转换成 UIMovieClip 剪辑后，原先绑定剪辑的类就脱离了，而再将类修改继承自 UIMovieClip，再与剪辑绑定，导出 SWC 文件后导入 Flex 中可以保留原先类中定义被使用。

注意：

1.原类的继承要进行修改，如：

```
…

import mx.flash.UIMovieClip;

public class UISquare extends UIMovieClip{

…
```

2.修改已转的 UIMovieClip 剪辑属性类路径（原绑定类路径）和基类（flash.display.MovieClip）。