Flash和Flex在编程时会有一些不同之处。因为Flash和Flex的编译器（compilers）不同，所以同一段代码无法保证在Flash和Flex环境下都能够编译成功。此外，Flash IDE可以独立制作库元件和显示对象（如MovieClip）；Flex可以使用Embed标签而Flash不支持（应开发者要求，目前最新版的Flash CS4也已经支持Embed标签）；Flash和Flex拥有不同的组件库。

在Flash CS3中按【F1】键，在Flex Builder 3中按【Shift】+【F2】组合键，都可以打开相应的帮助文档。帮助文档中提供了一些基本的教程，同时也可以作为工具书，检索类的属性、方法和事件并提供必要的实例。帮助文档是学习ActionScript 3.0最好和最全面的工具书、参考书。读者在阅读本书时，有必要经常查看帮助文档，了解细节。

**对基本背景有所了解之后，首先我们来运行第一个ActionScript程序。**

**【例1-1】调用trace函数输入字符串 。**

**（1）运行Flash CS3程序。**

**（2）按快捷键【Ctrl】+【N】新建一个Flash文档。**

**（3）按快捷键【Ctrl】+【S】，将文档存为“例1-1\_trace.fla”。**

**（4）按快捷键【F9】，打开ActionScript面板，输入如下代码。**

**//在输出栏中输出字符"ActionScript"**

**trace("ActionScript");**

**（5）按【Ctrl】+【Enter】组合键测试，可以看到在输出面板中输出了字符串“ActionScript”，如下图所示。**

**//声明一个Number数据类型变量n，并赋予值10**

**var n:Number = 10;**

**//输出n的16进制字符串表示形式**

**trace(n.toString(16));**

**输出结果为：a**

**变量n是Number类的一个实例。Number类有一个公共方法（函数）toString()，可以返回变量的字符串表示形式，使用n.toString()的语法形式调用函数。由于toString()方法允许添加参数，以注明字符串的转换基数（允许使用的参数为2～36），本例中使用十六进制，则变量的值10被转换为字符串“a”。**

## 常量

**常量也是变量，只不过只能对其赋值一次，且无法再改变。常量一般通过关键字const进行声明，通常使用大写字母为其命名，单词之间使用下划线（\_）分隔，语法如下。**

**const STAGE\_HEIGHT:uint = 550;**

**由于常量无法再次赋值，因此所有试图向常量赋值的语句会引发编译错误。**

**注意：对常量的赋值仅限于在声明时或构造函数中，其他情况全部会产生编译错误，即使是在声明后马上赋值也不允许 。**

## 数据类型

概述

**ActionScript 3.0存在两类数据类型。一类称为“基元值”，包括Boolean、int、Null、Number、String、uint和void；另一类称为“复杂值”，包括Object、Array、Date、Error、Function、RegExp、XML和XMLList。**

**其中，基元值是数据类型构成的基本单位，基元值的处理速度通常比复杂值要快，因为ActionScript按照一种能够尽可能优化内存和提高速度的特殊方式来存储基元值。**

### Boolean

**Boolean数据类型又称为布尔型数据类型或布尔值。Boolean数据类型只包含两个值：true和false，其他任何值均无效。布尔值在程序中常充当条件判断句的分支条件，或作为一些属性的状态开关来使用。**

**布尔值在未初始化时默认值为false。可以用学过的trace函数验证，代码如下。**

**//声明一个新Boolean型变量myVariable**

**var myVariable:Boolean = new Boolean;**

**//执行trace函数，输出myVariable的值**

**trace(myVariable);**

### int数据类型

**int数据类型是一个32位整数，它的值介于-2,147,483,648和+2,147,483,647之间。使用整数进行计算可以大幅度提高计算效率。ActionScript 3.0将数值型变量细分，为代码优化提供了便利条件。int型变量常常作为计数器的变量类型，也会在一些像素操作中作为坐标进行传递。**

**当声明一个变量为整型变量后再对其赋值时，将舍掉小数点后的部分，代码如下。**

**//声明一个新int型变量myVariable**

**var myVariable:int = 3.1415926;**

**//执行trace函数，输出myVariable的值**

**trace(myVariable);**

**上述代码的输出结果为：3。**

**注意：如果处理范围超出32位时，可以使用Number数据类型，Number类型可以处理53位整数值。**

### Null数据类型

**Null数据类型只有一个值——null。null通常表示不存在或未提供数据。**

### Number数据类型

**在ActionScript 3.0中，Number数据类型表示所有类型数值，包括整数和浮点数。Number数据类型能够表示的最小值为4.9406564584124654e-324，可使用常量Number.MIN\_VALUE表示；最大值可表示到1.79769313486231e+308，可使用常量Number. MAX\_VALUE表示。**

**//输出Number类型数据的最小值和最大值**

**trace(Number.MIN\_VALUE,Number.MAX\_VALUE);**

**以上代码输出结果为：4.9406564584124654e-324 1.79769313486231e+308**

### String数据类型

**String 数据类型表示的是一个字符序列。在ActionScript内部被存储为Unicode字符，并使用UTF-16 格式。字符串数据用单引号或双引号引用，代码如下。**

**//声明一个字符串变量myString，并将值"中华大仙"赋予该变量**

**var myString:String = "中华大仙";**

**trace(myString); //输出myString的值**

**myString = ""; //将空字符串赋予myString**

**trace(myString) //输出myString的值**

**trace(‘----结束----‘);**

### uint数据类型

**uint数据类型与int数据类型一样，都是32位整数，不过它仅限于无符号整数，因此其取值范围在0～4,294,967,295之间。uint数据类型常用于存储非负整数的属性值，如表示像素的颜色值，代码如下。**

**var tf:TextField = new TextField(); //建立一个新的TextField实例tf;**

**tf.background = true; //使用布尔值,设置tf的背景可以填充颜色**

**tf.backgroundColor = 0xFFFF00; //指定tf背景为黄色**

**tf.text = "Hello!"; //设置tf的文本内容为"Hello!"**

**addChild(tf); //添加tf到显示列表**

### uint数据类型

**上述代码运行输出时将显示一个黄色的文本区域。**

**如果对tf.backgroundColor设置负数，则编译时会发生误，在编译错误面板会输出：“Warning: 1092: Negative value used where a uint (non-negative) value is expected.”提法不能设置uint数据类型参数为负数。**

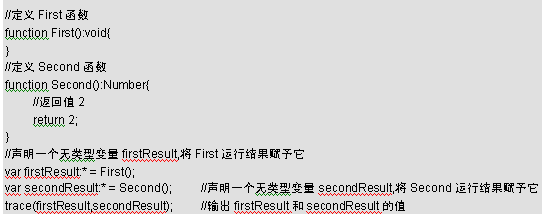
### void数据类型

**void类型只有一个值：undefined。void数据类型的唯一作用是标注返回注释。指定函数不会返回任何值，如以下两个函数。**



**其中First函数无须返回值，而Second函数必须返回一个数字。**

**可以使用trace函数上述两个函数的返回值，将代码改写为如下形式。**



**上述代码运行结果为：undefined 2**

### 类型转换

**类型转换是指将某个值转换为其他数据类型。类型转换分为“隐式”和“显式”两种。隐式转换一般由ActionScript运行时自动执行，有时也被称为“自动转换”。比如将数值2.4赋予int数据类型的变量n时，ActionScript强制将值转换为指定的整数类型：**

**var n:int = 2.4;**

**trace(n); //输出为2**

**显示转换，有时也称为“强制转换”，是通过代码指示ActionScript将一个数据类型的变量“视为”另一个数据类型。**

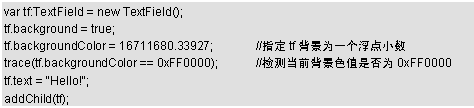
### 自动转换

**在以下场合，ActionScript会在编译过程执行自动转换，以确保操作合法。**

**（1）在赋值语句中，代码如下。**

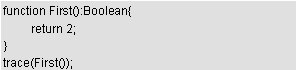


**（2）将值作为函数参数传递时，代码如下。**



**上述代码运行后输出true，表明文本区域的背景颜色值为0xFF0000（十进制数：16711680），而非赋值时的浮点数。在传递参数时，ActionScript已经执行了自动转换。**

**（3）从函数中返回值时，代码如下。**



**上述代码中定义了函数返回值为一布尔型数据类型，当使用return返回2时，ActionScript自动将其转换为布尔值。输出结果为：true。**

**（4）在使用一些运算符时，代码如下。**



**上述代码输出结果为：false。在求n的逻辑反时，ActionScript将值2转换为布尔值true，求反回输出false。**

**说明：“!”运算符对布尔值求反，可理解为“不”或“非”。!true返回false；!false返回true。**

### 强制转换

**强制转换的语法为新类型的名称加上括号组括起的原值，如转换2.4为布尔值类型的写法为Boolean(2.4)。**



**说明：typeof函数用于返回变量的数据类型。**

**在ActionScript 3.0中，如果进行了类型声明，再进行赋值时，如出现类型不匹配时，需要进行强制转换，如以下代码。**



**在编译时会产生错误信息：“1067: Implicit coercion of a value of type int to an unrelated type String.”，提示赋予字符串型变量的值不匹配。此时强制转换则可以完成赋值，代码如下。**



**读者首先需要了解自动转换的内部机制，在无法进行强制转换或转换内容不是自己需要的数据类型时，进行强制转换。**

### 转换规律

**无论自动转换还是强制转换，ActionScript 3.0的方式都是一样的。在进行基元值的相互转换时，遵守一些默认的规则。**

**转换为数值型（int，uint，Number）的规则：**

**（1）布尔值转换为数值型时。True转换为1，false转换为0。**

**（2）Null类型的null值，转换为0。**

**（3）字符串类型转换时，如果字符串包含的字符全部为数字，则结果为该数字，代码如下。**

**trace(int("-3721"));//输出结果为:-3721**

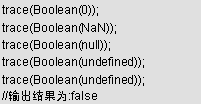
**如果字符串中含有无法转换的其他字符，则转换为Number类型时结果为NaN，转换为整数型（int，uint）时，结果为0。**

**（4）undefined转换为Number类型时结果为NaN，转换为整数型（int，uint）时，结果为0。**

**（5）null转换为数值型时结果全部为0。**

**转换为布尔值的规则：**

**（1）0、NaN、null、undefined、空字符串转换为false，代码如下。**



**（2）非0值，非空字符串转换为true，代码如下。**



**注意，本例中的“0”为字符串类型数据，并非数值型数据**

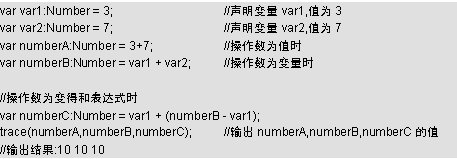
**转换为字符串的规则：**

**转换字符串时，各值转换为其对应的字符串，代码如下。**



## 运算符

**运算符是一种特殊的运算符号，代表一定的函数功能。运算符与操作数按照一定规律排列，以实现不同的运算。操作数一般是值、变量或表达式，如常见的加号（+）就是一种运算符。以下代码使用加号进行运算，并使用赋值运算符（=）将运算结果赋予变量。**

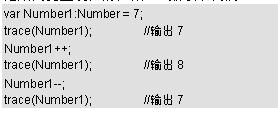


### 算术运算符

**算术运算符用于对操作数进行传统的算术运算，包括加减乘除（+）、（-）、（\*）、（/），递增（++）、递减（--）、求余（%）。**

**加减乘除运算同普通的算术运算符“＋－×÷”。**

**递增符的操作数必须是变量，其运算结果为变量现在的值加1。递减符的操作数也必须是变量，其运算结果为变量现在的值减1，如以下代码。**



### 赋值运算符

**赋值运算符（=）是将值赋予变量的运算符号。赋值运算符是一个二元运算符，包含2个操作数。左侧的是被赋值的变量（不可以是值或表达式），右侧为任意操作数。通过赋值操作，可以将右侧操作数的值赋予左侧的变量。**



**赋值运算符和算术运算符合起来，组成算术赋值运算符，这是一种先运算后赋值的组合运算符。**

**+=：操作数一与操作数二相加后再赋值给操作数一。**

**-=：操作数一与操作数二相减后再赋值给操作数一。**

**\*=：操作数一与操作数二相乘后再赋值给操作数一。**

**/=：操作数一与操作数二相乘后再赋值给操作数一。**

**%=：操作数一与操作数二求余后再赋值给操作数一。**

### 逻辑运算符

**逻辑运算符的操作数全部要求是布尔值（只有true和false），其运算结果也为布尔型数据类型，包括反（！）、或（||）、与（&&）。**

**！：对操作数求逻辑反，即当操作数为true时返回false，为false时返回true。**

**||：对两个操作数求逻辑或，即只要有任意一个为true时即返回true。全为false时才会返回false。**

**&&：对两个操作数求逻辑与，即只要有任意一个为false时即返回false。全为true时才会返回true。**

### 比较运算符

**比较运算符会对两个操作数进行比较，并返回一个布尔值（真：true，假：false）。**

**==：等于，比较两个操作数是否相等，代码如下。**

**trace(7 == 5); //返回false**

**!=：不等于，比较两个操作数是否不相等，代码如下。**

**trace(7 ！= 5); //返回true**

**>：大于，比较操作数一是否大于操作数二，代码如下。**

**trace(7 > 5); //返回true**

**<：大于，比较操作数一是否小于操作数二，代码如下。**

**trace(7 < 5); //返回false**

**其他类似的运算符还包括：**

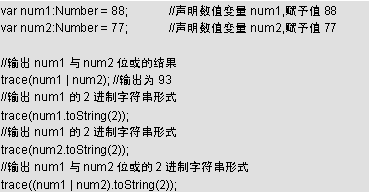
**<=（小于等于）、>=（大于等于）、===（全等运算符）**

**说明：全等运算符与普通的等于运算符类似，但在比较时不进行数据类型的自动转换。**

### 位运算符

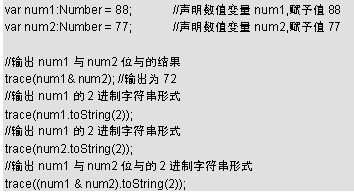
**位运算符将操作数先转化为2进制数（只有0，1），再对每位执行运算。常用的位运算符包括位或（|）、位与（&）和左右位移（<<、>>)。**

**|：位或，将2个操作数都转换为2进制数，再对每位执行逻辑或操作，代码如下。**



**注意：上述代码中num1的二进制形式为1011000，num2的二进制形式为1001101，则两数第一位分别为0和1，进行或操作时返回1；第二位分别为0和0，或操作时返回0……以此类推，则各位执行位操作的结果为1011101，十进制值为93。**

**&：位与，把2个操作数都转换为2进制数，再对每位执行逻辑与操作，代码如下。**



**注意：以上代码中num1的二进制形式为1011000，num2的二进制形式为1001101，则两数第一位分别为0和1，进行与操作时返回0；第二位分别为0和0，与操作时返回0……以此类推，则各位执行位操作的结果为1001000，十进制值为72。**

**<<：按位左移，将操作数转换为2进制数后，按参数左移相应的位数。**

**假设变量boo的二进制值为1001，则其左移2位后为100100（增加位数后补零），以上过程可写为如下代码。**



**>>：按位右移，与<<类似。**