第6章 WMLScript语言规范

6.1 范围

无线应用协议（Wireless Application Protocol，WAP）是WAP论坛经过不断努力得到的成果，它提供了一个业界技术规范，以便开发出适用于各种无线通信网络的应用和业务。 WAP论坛的工作范围就是为各种业务和应用制定一系列的技术规范。无线市场正在快速增长，新的用户不断增多，新的业务不断涌现。为了使运营商和生产者能够从容的面对先进业务、多种类业务和快速、灵活的业务生成等诸多的挑战， WAP规定了一系列传输层、会话层和应用层协议。有关 WAP体系结构更多的信息，请参阅“ 无线应用协议体系结构规范 ” (Wireless Application Protocol Architecture Specification) [WAP]。

本规范定义了 WMLScript语言，它属于 WAP的 应用层，能够增加客户端操作的逻辑性。WMLScript以ECMAScript [ECMA262]为基础，并通过适当的修改使其能够支持窄带通信和瘦客户端。它可以和无线标记语言 [WML]一起应用，使客户端具备智能 ，同时也可以作为一个单独工具使用。

ECMAScript和WMLScript的不同之处在于 WMLScript定义了字节码并且是采用了解释器参考体系结构，通过这种方式，使目前通信系统中的窄带信道得到了最佳的利用，并且减少了客户端对存储空间的要求。另外，为了使该脚本语言小而易学，并利于编译成字节码，WMLScript去掉了ECMAScript中一些先进的功能。例如， WMLScript是一种程序语言，支持局部安装的标准库。

6.2 概述

6.2.1 为什么需要脚本

WMLScript使WAP 体系结构能够支持通用的脚本，并且补充了无线标记语言 [WML]在某些应用方面的不足。 WML是一种标记语言，建立在可扩展性标记语言 [XML]的基础上，它用于定义窄带设备（例如蜂窝移动电话和寻呼机）的应用内容，这些内容可以表示成文字、图像、选择表等。它们的形式简单，使用户接口更友好，同时能够显示在用户常用的显视屏上。但是这些显示的内容都是静态的，如果不修改 WML就不可能把它扩展成支持动态的语言。下面列出了WML所不能支持的功能：

- 用户输入的有效性检验。
- 访问设备资源，如：允许程序者在电话上呼出电话、发送消息、将电话号码记录在号码簿中、访问SIM卡等。
- 本地产生消息条或对话框，这样可以避免在告警、错误提示和证实等操作中来回地传递信息，从而节省了宝贵的传输资源。
- 允许扩展设备软件，以及配置新开发出的设备。

为了解决这些问题，人们设计了 WMLScript，旨在使能力有限的窄带通讯设备具有可编程能力。

6.2.2 使用WMLScript的好处

移动瘦客户端的许多业务都可以用 WML 实现，而脚本语言进一步增强了其标准浏览的功能，同时使现有的WML 具有更强的能力。使用脚本语言可以支持更先进的用户界面、增加客户端的智能性、提供自身设备和外围设备的访问能力，同时减少服务器和客户端之间的数据传输带宽。

WMLScript基本上以ECMAScript [ECMA262]为基础，因此开发者在开发新的移动业务时，不必学习新的概念。

6.3 WMLScript的核心

在建立WMLScript 语言时，人们就努力地使它的核心部分尽可能地靠近 ECMAScript语言规范[ECMA262]中定义的核心。在ECMAScript 语言规范中，核心部分包括了基本的数据类型的定义、变量的定义、表达式的定义和语句的定义，这些几乎都可以用在 WMLScript 语言规范中。本节将概括介绍WMLScript的核心内容。

在第6.5节（WMLScript 语法）中将介绍WMLScript的语法规则和具体的语法细则。

6.3.1 词汇结构

本节介绍采用WMLScript书写程序的基本规则。

1. 区分大小写

WMLScript区分大小写，描述中的关键字、变量名、函数名都必须使用恰当的大写字母。

2. 空白区域和换行

WMLScript忽略空格、TAB空格以及空行，除非它们作为字符串的一部分出现在程序中。

语法：

```
WhiteSpace ::
    <TAB>
    <VT>
    <FF>
    <SP>
    <LF>
    <CR>
LineTerminator ::
    <LF>
    <CR>
    <CR><LF>
```

3. 分号的使用

在WMLScript中，以下的语句必须加分号 ⊖ ：

- 空语句 (见6.3.5节中“ 空语句 ”)

⊖ 兼容性注解：ECMAScript支持可选的分号。

- **表达式语句**(见6.3.5节中“**表达式语句**”)
- **变量声明语句**(见6.3.5节中“**变量声明语句**”)
- **中断语句**(见6.3.5节中“**中断语句**”)
- **继续语句**(见6.3.5节中“**继续语句**”)
- **返回语句**(见6.3.5节中“**返回语句**”)

4. 注释

该语言定义了两种注释结构：行注释（以“//”行结束作为注释行的开始和结束标记）和块注释（以“/*和“*/作为开始和结束的标记，中间包括多行注释）。块注释中不可以再嵌套另一个注释块[⊖]**。**

语法：

```
Comment ::
    MultiLineComment
    SingleLineComment
MultiLineComment  ::
    /* MultiLineCommentChars opt */
SingleLineComment ::
    // SingleLineCommentChars opt
```

5. 文字

(1) **整型文字**

整型文字可以用三种方式表示：十进制、八进制和十六进制整数。

语法：

```
DecimalIntegerLiteral::
    0
    NonZeroDigit DecimalDigits opt
NonZeroDigit :: one of
    1  2  3  4  5  6  7  8  9
DecimalDigits ::
    DecimalDigit
    DecimalDigits DecimalDigit
DecimalDigit :: one of
    1  2  3  4  5  6  7  8  9
HexIntegerLiteral::
    0x HexDigit
    0X HexDigit
    HexIntegerLiteral HexDigit
HexDigit :: one of
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  A  B  C  D  E  F
OctalIntegerLiteral::
    0 OctalDigit
    OctalIntegerLiteral OctalDigit
OctalDigit :: one of
    0  1  2  3  4  5  6  7
```

⊖ **兼容性注解：**ECMAScript**也支持**HTML**注释。**

整型文字的长度和取值范围在 6.3.2节“ 整型数的长度 ”中定义。如果定义的整型文字超出取值范围，编译过程中将给出错误指示。

(2) 浮点文字

浮点文字包括一个十进制小数和一个指数部分。

语法：

```
DecimalFloatLiteral::
   DecimalIntegerLiteral . DecimalDigits opt ExponentPart opt
   . DecimalDigits ExponentPart opt
   DecimalIntegerLiteral ExponentPart
DecimalDigits::
   DecimalDigit
   DecimalDigits DecimalDigit
ExponentPart   ::
   ExponentIndicator SignedInteger
ExponentIndicator:: one of
   e  E
SignedInteger::
   DecimalDigits
   + DecimalDigits
   - DecimalDigits
```

浮点文字的长度和取值范围在 6.3.2节“ 浮点数的长度 ”中定义。如果定义的浮点文字超出取值范围，编译过程中将给出错误指示。如果浮点文字溢出将得到 0 (0.0)。

(3) 字符串文字

字符串是用双引号""或单引号''引起来的一串字符，字符个数不限，可以是 0到多个。

语法：

```
StringLiteral::
   "DoubleStringCharacters opt "
'SingleStringCharacters opt '
```

无效字符串举例：

```
"Example"     'Specials:\x00\'\b'     "Quote: \""
```

因为有一些字符不可以出现在字符串中，所以 WMLScript定义了一种特殊的转义序列，借助这种序列来表示那些不在字符串中出现的字符。

出现在字符串文字中的转义序列仅仅给该字符串提供了一个字符，该字符不能编译成行结束符或字符串结束符（引号）。

(4) 布尔型文字

真值在 WMLScript中用布尔型文字表示，布尔型文字在 WMLScript中有真值假值两种。

语法：

```
BooleanLiteral   ::
   true
   false
```

(5) 无效型文字

WMLScript支持一种特殊的无效型文字来表示无效的值。表6-1列出了特殊转义序列及其符号。

表6-1 特殊转义序列及其符号

序 列	所代表的字符 ①	UNICODE	符 号
\’	撇号或单引号	\u0027	‘
\”	双引号	\u0022	“
\\	反斜线符号	\u005C	\
\/	斜线	\u002F	/
\b	退一格	\u0008	
\f	换页	\u000C	
\n	新行	\u000A	
\r	回车	\u000D	
\t	横向跳格	\u0009	
\xhh	hh代表两个十六进制数字字符（Latin-1 ISO8859-1)		
\ooo	ooo代表三个八进制数字字符（Latin-1 ISO8859-1)		
\uhhhh	hhhh. 代表四个十六进制数字字符		

① 兼容性注解：ECMAScript支持前面有反斜杠（\）的非转义字符。

语法：

```
InvalidLiteral   ::
    invalid
```

6. 标识符

WMLScript用标识符来标识不同的变量（见6.3.2节）、函数（见6.3.4节）和编译指示（见6.3.7节），标识符㊀的第一个字母不能是数字，但是可以是下划线“ _”。

语法：

```
Identifier   ::
    IdentifierName   but not ReservedWord
IdentifierName ::
    IdentifierLetter
    IdentifierName IdentifierLetter
    IdentifierName DecimalDigit
IdentifierLetter   :: one of
a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
_
DecimalDigit   :: one of
    0  1  2  3  4  5  6  7  8  9
```

非法标识符举例：

```
timeOfDay  speed  quality  HOME_ADDRESS  var0  _myName  ____
```

编译器按最大长度搜索字符串，并且标记出无效的标识符，除“ _”外，标识符不可以包含别的特殊字符 。同时，关键字和保留字不可以作为标识符，例如下面的标识符是非法的：

```
while  for  if  my~name  $sys  123  3pieces  take.this
```

㊀ 兼容性注解：ECMAScript也支持在任何名字处使用$字符。

在标识符中区分大写和小写，也就是说标识符 speed 和 Speed 是不同的。

7. 保留字

WMLScript定义了一系列保留字，它们在编程中具有特殊的意义，不可作为标识符。例如：

```
break  continue  false  true  while
```

8. 命名域

WMLScript使用命名域来区分对象不同的标识符。允许使用如下的命名域：

- 函数名(见6.3.4节)。
- 函数参数名(见6.3.4节) 和变量名(见6.3.2节)。
- 编译指示名 (见6.3.7节)。

这样，在同一个编译单元中，函数名、变量\参数名、编译指示名可以使用同一个标识符。例如：

```
use url myTesit "http://www.host.com/script"

function myTest(myTest) {
  var value = myTest#myTest(myTest)
  return value;
};
```

6.3.2 变量和数据类型

这一节介绍WMLScript中的两个重要的概念：变量和内在的数据类型。变量是一个和数据相关的名字，它可以存储和操作数据。 WMLScript有局部变量 ㊀ ，即仅在函数内部声明或作为函数的参数传递的变量（见6.3.4节）。

1. 变量的声明

变量的声明在WMLScript中是必须的 ㊁ ，它仅仅采用一个关键字“ var ”和所声明的变量名就可以了（有关变量说明的详细内容，见 6.3.5节中“ 变量声明语句 ”）。变量名的命名规则服从标识符的命名规则（见 6.3.1节中“ 标识符 ”），如：

```
var x;
var price;
var x,y;
var size = 3;
```

变量在使用之前必须声明，但可以不进行初始化，未被初始化的变量将自动地置为空字符串。

2. 变量的范围和生命期

变量只可在函数内部使用，并且只能出现在被声明之后，所有变量的名字在函数中必须唯一。块语句中的变量不在此定义范围内 (详见6.3.5节中“ 块语句 ”)。

```
function priceCheck(givenPrice) {
  if (givenPrice > 100) {
    var newPrice = givenPrice;
```

㊀ 兼容性注解：ECMAScript也支持全局变量。
㊁ 兼容性注解：ECMAScript支持自动声明。

```
  } else {
    newPrice = 100;
  };
  return newPrice;
};
```

变量的生命期是指从变量被声明开始到函数的结束为止。如：

```
function foo( )  {
  x = 1;            //错误：变量在声明之前使用
  var x,y;
  if (x) {
    var y;          //错误：重复声明
  };
};
```

3. 变量的访问

变量只可以在声明它的函数中使用，通过使用变量名来使用变量的内容，如：

```
var myAge   = 37;
var yourAge = 63;
var ourAge  = myAge + yourAge;
```

4. 变量的类型

WMLScript是一种弱类型语言，它的变量不分类型。但是实际上，它有四种基本的数据类型（布尔型、整型、浮点型）和特别定义的无效型数据类型（用于将一个无效数据和其他类型的数据区分开来）。编程者无须指定变量的类型，而变量也可以随时代表不同类型的数据，同时WMLScript可以在需要的时候自动的进行类型转换。如：

```
var flag          = true;       // 布尔型
var number        = 12;         // 整型
var temperature   = 37.7;       // 浮点型
number            = "XII";      // 字符串型
var except        = invalid;    // 无效型
```

5. 参考变量（L-Values）

有一些运算符(详见6.3.3节中“赋值运算”)要求左操作数是一个参考变量（L-value）而并非一个变量值，所以在WMLScript 中除了上述五种数据类型外，又有第六种类型：变量型，这种类型用来指明必须提供一个变量名。

```
result += 111;    // +=运算符需要一个变量
```

6. 类型的等价

WMLScript 支持不同类型数据之间的运算。各种类型的数据都可以作为所有运算符（见6.3.3节）的操作数，在需要时进行自动的类型转换（见 6.4节）。

7. 数字取值

WMLScript支持两种类型的数字值：整型值和浮点型值。变量可以在初始化时置为整型或浮点型值[⊖]，在运行过程中，一些运算可以修改这个值。浮点型和整型之间地转换在第 6.4节中说明。

⊖ 惯例：当一个变量值可以是整型或浮点型的情况下，使用一个更一般的术语：数字（number）来替代。

```
var pi     = 3.14;
var length = 0;
var radius = 2.5;
length     = 2*pi*radius;
```

(1) 整型数的长度

整型数的长度为32位(2的补码)，也就是说整型数的取值范围[⊖]是－2147483648到2147483647。Lang [WMLSLibs]库函数在运行中可以取这些值（见表6-2)。

表6-2 Lang 库函数的取值

Lang.maxInt( )	最大整型值
Lang.minInt( )	最小整型值

(2) 浮点数的长度

浮点数的最大/最小取值范围[⊜]和精度同[IEEE754]的定义，WMLScript支持32位单精度浮点数格式：

最大值 3.40282347E+38

最小非零值 (普通精度所必须支持的) 1.17549435E-38 或者更小

浮点库函数 [WMLSLibs] 在运行中可以取这些值，见表6-3:

表6-3 浮点库函数的取值

Lang.maxFloat( )	最大浮点型值
Lang.minFloat( )	最小浮点型值

几种特殊的浮点数的使用规则：

- 如果运算结果不属于单精度浮点型有限实数集（如，结果不是一个数，或结果为正无穷大等)，则结果为无效值。
- 如果运算结果为一个溢出的浮点数，则结果为零 (0.0)。
- 正零和负零完全相等。

8. 字符串的取值

WMLScript 中的字符串可以包含字母、数字、特殊字符等。变量可以被初始化为字符串型，WMLScript中的运算和标准String库[WMLSLibs]函数均可以使用字符串型值。如：

```
var msg = "Hello";
var len = String.length(msg);
msg     = msg + ' Worlds!';
```

9. 布尔型取值

布尔型值可以在变量初始化、变量赋值或说明需要布尔型值的参数时使用，它可以是一个文字或者是一个逻辑表达式(详见6.3.3节中“逻辑运算”)。

```
var truth = true;
var lie   = !truth;
```

⊖ 兼容性注解：ECMAScript没有规定最大整型数和最小值整型数，所有的数字都按浮点值描述。

⊜ 兼容性注解：ECMAScript对所有数字使用双精度64比特格式[IEEE754]浮点类型。

6.3.3 运算符和表达式

本节介绍WMLScript语言支持的运算以及如何利用它们构成复杂的表达式。

1. 赋值运算

WMLScript 语言可以通过以下几种方式来给变量赋值，其中最简单的就是仅仅使用运算符“ = ”，当然使用其他一些运算符的赋值方式也是可以的。

运算符	运算
=	赋值
+=	加(数)/连接(字符串) 并赋值
-=	减并赋值
*=	乘并赋值
/=	除并赋值
div=	除 (整除)并赋值
%=	取余数并赋值
<<=	左移（位操作）并赋值
>>=	右移（位操作）并赋值
>>>=	右移补零（位操作）并赋值
&=	位与并赋值
^=	位异或并赋值
|=	位或并赋值

```
var a = "abc";
var b = a;
   b = "def";  // 变量a的值为 "abc"
```

2. 算术运算

WMLScript支持全部的基本二进制算术运算：

运算符	运算
+	加(数)/连接(字符串)
-	减
*	乘
/	除
div	整除

另外，WMLScript还支持一些复杂的二进制运算，如：

运算符	运算
%	取余
<<	左移（位操作）
>>	带符号的右移（位操作）
>>>	右移补零（位操作）
&	位与
|	位或
^	位异或

同时，WMLScript支持基本的单目运算，包括：

运算符	运算
+	加
-	减
—	自减
++	自增

~ 位非

例如：

```
var y = 1/3;
var x = y*3+(++b);
```

3. 逻辑运算

WMLScript 支持基本的逻辑运算：

运算符	运算
&&	逻辑与
||	逻辑或
!	逻辑非 (单目)

执行逻辑与运算时，首先检测第一个操作数是否为真，如果为假，则逻辑与运算的结果为假；如果为真，则把第二个操作数的检测结果作为逻辑与运算的结果。如果第一个操作数是无效的，则不再检测第二个操作数，逻辑与运算结果为无效。

同样，执行逻辑或运算时，首先检测第一个操作数并给出检测结果，如果结果为真，则逻辑或运算结果为真；如果测试结果为假，则用第二个操作数的检测结果作为逻辑与运算的结果。如果第一个操作数是无效的，则不必检测第二个操作数，逻辑与运算结果为无效。

```
weAgree = (iAmRight && yourAreRight) ||
          (!iAmRight && !youAreRight);
```

WMLScript 要求逻辑运算使用布尔型值，支持其他类型和布尔型之间的自动转换 (见第6.4节)。

注意 如果逻辑与或逻辑非的第一个操作数是无效的，则不必测试第二个操作数，逻辑运算结果为无效。

如：

```
var a = (1/0) || foo( ); /无效，不执行调用 foo( )的操作。
var b = true  || (1/0); 真/
var c = false || (1/0); /无效
```

4. 字符串运算

WMLScript 支持字符串合并运算。运算符“ + ”和“ += ”执行字符串合并运算，其他的字符串运算 ⊖ 由标准字符串库函数支持 (详见 [WMLSLibs])。

```
var str  = "Beginning" + "End";
var chr = String.charAt(str,10); // chr = "E"
```

5. 比较运算

WMLScript支持基本的比较运算：

运算符	运算
<	小于
<=	小于等于
==	等于
>=	大于等于
>	大于

⊖ 兼容性注解：ECMAScript支持字符串对象和每一个字符串的长度属性。 WMLScript不支持对象。但WMLScript库提供相似的功能。

!= 不等于

比较运算符的运算规则：

- 布尔型真值大于假值。
- 整型按整数的大小比较。
- 浮点型按浮点数的大小比较。
- 字符串型按字符串中字符码的在[UNICODE]中的顺序比较。
- 无效型如果进行比较的数中有一个为无效型，则比较的结果为无效。

例如：

```
var res = (myAmount > yourAmount);
var val = ((1/0) == invalid);       // val = invalid
```

6. 矩阵运算

WMLScript不支持矩阵 ⊖ 运算或类似的运算，然而在标准字符串库函数中（详见[WMLSLibs]）有支持字符串作类似矩阵运算的函数。字符串可以包含用特定分隔符分隔的元素，分隔符由应用程序指定。所以，字符串库函数中有一些函数支持字符串矩阵的生成和管理。如：

```
function dummy( ) {
  var str  = "Mary had a little lamb";
  var word = String.elementAt(str,4," ");
};
```

7. 逗号运算

WMLScript 支持“，”运算符，以此将多个运算式合成一个表达式。“，” 运算符的运算结果是第二个操作数的值。

```
for (a=1, b=100; a < 10; a++,b++) {
  ...do something...
};
```

在函数调用时使用逗号（而不是逗号运算符）分隔参数，以及在变量声明时分隔多个变量声明。在这些情况下，如果存在逗号运算符，则必须使用括号将它括起来。如：

```
var a=2;
var b=3, c=(a,3);
myFunction("Name", 3*(b*a,c)); 两个参数："Name",9
```

8. 条件运算

WMLScript支持条件运算，它使用三个操作数，该运算符按照第一个操作数的布尔值选择后两个参数中的一个参数进行计算。如果第一个操作数（条件）的布尔型值为真，运算结果是对第二个操作数的计算结果；如果第一个操作数（条件）的布尔型值为假或无效，运算结果是对第三个操作数的计算结果。

```
myResult = flag* "Off" : "On (value=" + level + ")";
```

注意 该运算符的行为类似一个if语句 (详见6.3.5节中“ If语句 ”)。如果条件为假或无

⊖ 兼容性注解：ECMAScript支持矩阵。

效，则计算第三个操作数。

9. 类型运算

尽管WMLScript 是一种弱类型语言，仍然存在内在的数据类型，包括：布尔型、浮点型、整型、字符串型、无效型等。类型运算符 (typeof)返回下面列出的整型值 ⊖，每一种值对应一种指定的类型，取值是：

类型	编码
整型	0
浮点型	1
字符串型	2
布尔型	3
无效型	4

类型运算并不将结果从一种类型转换成另一种类型，而是在表达式计算完成之后，返回这个类型。

```
var str    = "123";
var myType = typeof str; // myType = 2
```

10. 有效（isvalid） 运算

该运算符检查给定的表达式的类型是否有效，如果表达式的类型为无效，则返回布尔型假值；否则返回真值。有效运算并不将结果从一种类型转换成另一种类型，而是在表达式计算完成之后，返回这个类型。

```
var str = "123";
var ok  = isvalid str;   /真
var tst = isvalid (1/0); /假
```

11. 表达式

WMLScript 支持其他编程语言所支持的大多数表达式，最简单的表达式是常数或变量，它仅仅计算常数或变量。如：

```
567
66.77
"This is too simple"
'This works too'
true
myAccount
```

复杂的表达式可以由简单的表达式和运算符以及函数调用定义。如：

```
myAccount + 3
(a + b)/3
initialValue + nextValue(myValues);
```

12. 表达式优先级

表6-4总结了WMLScript中所有的运算符，同时包括这些运算符的优先级（运算的次序）和结合方向（从左到右或者从右到左）。

⊖ 兼容性注解：ECMAScript规定typeof 运算返回一个字符串表示的变量类型。

表6-4 WMLScript 中的运算符

优 先 级①	运算顺序	运 算 符	操作数类型	结果类型	运 算
1	R	++	数字	数字	自增运算（单目）
1	R	—	数字	数字	自减运算（单目）
1	R	+	数字	数字	单目加法运算
1	R	-	数字	数字	负号运算
1	R	~	整型	整型	位非法运算（单目）
1	R	！	布尔型	布尔型	逻辑非法运算（单目）
1	R	Typeof	任何	整型	返回数据类型法运算（单目）
1	R	isvalid	任何	布尔型	有效性检验法运算（单目）
2	L	*	数字	数字	乘法运算
2	L	/	数字	浮点型	除法运算
2	L	div	整型	整型	整除法运算
2	L	%	整型	整型	取余法运算
3	L	-	数字	数字	减法运算
3	L	+	数字或字符串型	数字或字符串型	加（数字）或合并（字符串）运算
4	L	<<	整型	整型	左移运算
4	L	>>	整型	整型	带符号右移运算
4	L	>>>	整型	整型	补零的右移运算
5	L	<,<=	数字或字符串型	布尔型	小于，小于等于
5	L	>,>=	数字或字符串型	布尔型	大于，大于等于
6	L	= =	数字或字符串型	布尔型	等于（逻辑运算）
6	L	!=	数字或字符串型	布尔型	不等于（逻辑运算）
7	L	&	整型	整型	按位与运算
8	L	^	整型	整型	按位异或运算
9	L	|	整型	整型	按位或运算
10	L	&&	布尔型	布尔型	逻辑与运算
11	L	||	布尔型	布尔型	逻辑或运算
12	L	?:	布尔型，任何类型，任何类型	任何类型	条件表达式
13	R	=	变量，任何类型	任何类型	赋值运算
13	R	*=,/=,%=,-=,div=	变量，数字，	数字，	数值运算并赋值
13	R	+=	变量，数字，字符串	数字，字符串	加法或合并（字符串）运算合并赋值
13	R	<<=,>>=,>>>=,&=,^=,| =	变量，整型	整型	按位运算并赋值
14	L	'	任何类型	任何类型	逗号运算符

① 约束：0具有最高优先级。

6.3.4 函数

WMLScript的函数是WMLScript编译单元中被命名的一部分，当它被调用时，将执行一系列特定的语句，并返回相应的值。本节将描述在WMLScript中如何声明和使用一个函数。

1. 函数声明

函数声明用来声明一个WMLScript 函数，包括其函数名及可选择的参数项（形式参数列表），同时包括函数调用时所要执行的一个语句块。所有的函数均具有以下特征：

- 函数声明不可嵌套。
- 在一个编译单元中，函数名必须唯一。
- 必须给函数中所有参数传递值。
- 在给函数传递值时，必须保证传递的参数数目和函数声明时定义的参数数目一致。
- 函数的参数类似于局部变量，不过它在函数体开始前就完成了初始化。
- 函数通常都返回一个值，默认返回值为空字符串，当然，函数中可以使用 return语句返回其他特定的值。

WMLScript中的函数不是数据类型[⊖]，是一种语言的语法特征。

语法：

```
FunctionDeclaration:
    extern opt function Identifier ( FormalParameterList opt ) Block ;opt
FormalParameterList:
    Identifier
    FormalParameterList , Identifier
```

注意 关键字extern为可选项，用它来定义从外部可访问的外部函数，外部函数可在定义函数的编译单元以外调用。标识符表示定义的函数名。形式参数（可选项）是用逗号分隔的一列参数名。块语句是函数的主体部分，当调用函数时，首先通过参数传递完成形式参数的初始化，然后就执行块语句。

例：

```
function currencyConverter(currency, exchangeRate) {
  return currency*exchangeRate;
};
extern function testIt( ) {
  var UDS = 10;
  var FIM = currencyConverter(USD, 5.3);
};
```

2. 函数调用

函数调用的方式依赖于函数声明的位置。下面一节介绍 WMLScript支持的三种函数调用：局部函数调用、外部函数调用和库函数调用。

(1) 局部函数

局部函数是指在同一个编译单元内定义的函数，调用时只需使用函数名和用逗号隔开的一列传递参数即可（当然，参数的数目必须与函数所定义的参数[⊜]相匹配）。

⊖ 兼容性注解：ECMAScript中的函数是实际的数据类型。

⊜ 兼容性注解：ECMAScript支持在一个函数调用中的大量参数。

语法：

```
LocalScriptFunctionCall
   FunctionName Arguments
FunctionName：
   Identifier
Arguments：
   ( )
   ( ArgumentList )
ArgumentList：
   AssignmentExpression
   ArgumentList , AssignmentExpression
```

在同一个编译单元内的函数可以在它被声明前调用。如：

```
function test2(param) {
  return test1(param+1);
};
function test1(val) {
  return val*val;
};
```

(2) **外部函数**

外部函数的调用只能发生在外部函数存在的时候，即该外部函数已经在其他的编译单元中声明。调用的方式和局部函数基本相同，不过需要加上相应的外部编译单元名。

语法：

```
ExternalScriptFunctionCall
   ExternalScriptName # FunctionName Arguments
ExternalScriptName:
   Identifier
```

必须用编译指示 use url (**详见6.**3.7**节)来指定外部编译单元，它指明了从一个可在函数声明中使用的名字和外部编译单元之间的对应关系，该名字和符号“ #”用来作为标准函数调用的前缀。**

```
use url OtherScript "http://www.host.com/script"

function test3(param) {
  return OtherScript#test2(param+1);
};
```

(3) **库函数**

当函数为 WMLScript **标准库函数** [WMLSLibs]**时，使用库函数调用。**

语法：

```
LibraryFunctionCall:
   LibraryName . FunctionName Arguments
LibraryName：
   Identifier
```

库函数的调用需要采用其函数库名及函数名，二者用小圆点“ .”连接(详见6.3.6 **节)。**

例如：

```
function test4(param) {
  return Float.sqrt(Lang.abs(param)+1);
};
```

3. 默认返回值

函数的默认返回值为空字符串。函数的返回值有时是被忽略的。例如，当函数调用是一条单独的语句时：

```
function test5( ) {
  test4(4);
};
```

6.3.5 语句

WMLScript **的语句由表达式和关键字按照一定的语法规则构成。一个语句可以占用多行，而一行也可以包括多条语句。**

本节介绍 WMLScript [⊖] **中的几种语句，包括空语句、表达式语句、块语句、中断语句**（break）**、继续语句**（continue）**、循环语句**（for，while）**、条件语句**（if...else）**、返回语句**（return）**、变量声明语句**（var）**。**

1. 空语句

空语句用在程序中需要语句却不需要操作的地方。

语法：

EmptyStatement：

;

例如：

```
while (!poll(device)) ; /等待，直到函数poll( )为真
```

2. 表达式语句

表达式语句可以用来给变量赋值、描述数学表达式、调用函数等。

语法：

ExpressionStatement:

Expression ;

Expression：

AssignmentExpression

Expression , *AssignmentExpression*

例如：

```
str  = "Hey "+ yourName;
val3 = prevVal + 4;
counter++;
myValue1 = counter, myValue2 = val3;
alert("Watch out!");
retVal = 16*Lang.max(val3,counter);
```

3. 块语句

⊖ **兼容性注解：**ECMAScript**支持**for..in **和**with**语句。**

块语句以{开始，结束}封装，里面包含一个语句序列。块语句可以用于使用单句的任何地方。

语法：

```
Block :
   { StatementList opt }
StatementList :
   Statement
   StatementList Statement
```

例如：

```
{
  var i = 0;
  var x = Lang.abs(b);
  popUp("Remember!");
}
```

4. 变量声明语句

该语句用来声明变量，同时可以进行变量的初始化（如果不对变量进行初始化，则默认为空字符串）。所声明的变量仅在当前函数中有效（详见 6.3.2节中）。

语法：

```
VariableStatement :
    var VariableDeclarationList ;
VariableDeclarationList :
   VariableDeclaration
   VariableDeclarationList , VariableDeclaration
VariableDeclaration :
   Identifier VariableInitializer opt
VariableInitializer :
   = ConditionalExpression
```

注意 标识符是变量的名字，它可以是任何合法的标识符。条件表达式是变量的初始化值，它可以是任何合法的表达式，在变量声明时，对这个表达式进行计算。

在一个函数中变量的名字必须唯一。

例如：

```
function count(str) {
  var result = 0;       /初始化一次
  while (str != "") {
    var ind = 0;        /初始化多次，每次进入循环都执行初始化
    // 修改字符串
  };
  return result
};
function example(param) {
  var a = 0;
  if (param > a) {
    var b = a+1;        /可以使用变量a，b
```

```
  } else {
    var c = a+2;          /可以使用变量 a，b，c
  };
  return a;               /可以访问变量a，b，c
};
```

5. If 语句

If 语句是条件语句，它包括两部分：条件部分和语句部分。语句部分包括一条或两条语句，当满足条件时，执行第一条语句，否则执行第二条语句（或者不进行操作）。

语法：

```
IfStatement:
   if ( Expression ) Statement else Statement
   if ( Expression ) Statement
```

注意 表达式(Expression)可以是WMLScript中的任何合法表达式，只要其计算结果是（或可以转换为）布尔型或无效型值。如果表达式的计算结果为真，则执行第一条语句，如果结果为假或者无效则执行第二条 else语句（可选）。 被执行的语句可以是任何WMLScript中的合法语句，同时包括嵌套的If语句。值得注意的是else语句总是与离它最近的if配套使用。

例如：

```
if (sunShines) {
  myDay = "Good";
  goodDays++;
} else
  myDay = "Oh well...";
```

6. While 语句

该语句为循环语句，当表达式为真时则执行循环中语句。

语法：

```
WhileStatement:
   while ( Expression   ) Statement
```

注意 表达式(Expression)可以是WMLScript中的任何合法表达式，只要其计算结果为（或可以转换为）布尔型或无效型值。在每次执行循环语句前都要对表达式进行计算，如果结果为真，则执行循环语句；如果结果为假或无效，则执行下一条语句。

循环中的语句在条件满足时始终执行。

例如：

```
var counter = 0;
var total   = 0;
while (counter < 3) {
  counter++;
  total += c;
};
```

7. For 语句

该语句也是一种循环语句，它包括三个条件表达式和一个语句。三个条件表达式用括号

括起来，并用分号分隔。

语法：

ForStatement：

for (Expression$_{opt}$ *; Expression*$_{opt}$ *; Expression*$_{opt}$ *) Statement*

for (var VariableDeclarationList ; Expression$_{opt}$ *Expression*$_{opt}$ *) Statement*

注意 第一个表达式(Expression)或变量声明语句(var VariableDeclarationList)常用来初始化一个计数变量 。该表达式可以用var随意声明一个新的变量，该变量的生命期同其他变量一样，在当前函数的剩余部分有效（详见6.3.2节中）。

第二个表达式（条件表达式）可以是 WMLScript中的任何合法表达式，只要其计算结果为（或可以转换为）布尔型或无效型值。在每次通过该循环时前都要对条件表达式进行计算，如果结果为真，则执行循环中的语句。这一过程为可选的，在默认的情况下，认为条件始终被满足。

第三个表达式（增量表达式）通常用来更新记数变量。循环中的语句在条件满足时才开始执行。

例如：

```
for (var index = 0; index < 100; index++) {
  count += index;
  myFunc(count);
};
```

8. 中断（Break）语句

中断语句用来终止while 或 for循环，然后执行循环体外的下一条语句。在 while 或 for 循环之外使用中断语句会产生错误报警。

语法：

BreakStatement：

break ;

例如：

```
function testBreak(x) {
  var index = 0;
  while (index < 6) {
    if (index == 3) break;
    index++;
  };
  return index*x;
};
```

9. 继续（Continue）语句

该语句用来终止当前的 while 或 for循环。当循环体中遇到 Continue语句时，程序跳过Continue语句之后的、其余的循环体中的语句，开始下一次循环。 Continue语句终止的是一次循环，而不是整个循环过程：

- 在 while 循环中，它将跳到判断条件是否满足处。
- 在for 循环中，它将跳到更新表达式处。

语法：

ContinueStatement:
continue ;

例如：

```
var index = 0;
var count = 0;
while (index < 5) {
   index++;
   if (index == 3)
      continue;
   count += index;
};
```

10. **返回（Return）语句**

Return**语句用在函数体中表示函数的返回值，如果函数中没有** Return**语句或者不执行任何的函数返回语句，则函数的返回值为默认的空字符串。**

语法：

ReturnStatement:
return *Expression* $_{opt}$;

例如：

```
function square( x ) {
  if (!(Lang.isFloat(x))) return invalid;
  return x * x;
};
```

6.3.6 函数库

WMLScript **支持函数库的使用** ⊖ **。每个函数库都是一个函数的集合，它们对库函数作了逻辑上的分类。函数调用时，需使用函数库名及函数名，二者间用小圆点“ .”隔开，函数名后还应带上相应的参数。**

库函数调用的例子：

```
An example of a library function call:
function dummy(str) {
  var i = String.elementAt(str,3," ");
};
```

• 标准库

标准库的详细说明见 WMLScript **标准库** [WMLSLibs]**。**

6.3.7 编译指示

WMLScript**支持编译指示，用它来说明编译层的信息。编译指示在编译单元的开始说明，位于所有的函数声明之前。所有的编译指示都以关键字开头，然后跟着编译指示的特殊属性。**

⊖ **兼容性注解：**ECMAScript**不支持函数库，它支持具有属性的一组预定义对象。**WMLScript**使用库来支持相似的功能。**

语法：

```
CompilationUnit:
    Pragmas opt FunctionDeclarations
Pragmas :
    Pragma
    Pragmas Pragma
Pragma :
    use PragmaDeclaration    ;
PragmaDeclaration:
    ExternalCompilationUnitPragma
    AccessControlPragma
    MetaPragma
```

下面详细介绍编译指示。

1. 外部编译单元

WMLScript**编译单元可以通过一个**URL**访问， 这样每个**WMLScript **函数都可以通过给定的**WMLScript **源程序的**URL **和函数名访问。在调用外部编译单元的函数时必须使用** use url **编译指示。**

语法：

```
ExternalCompilationUnitPragma
    url Identifier StringLiteral
```

use url**编译指示指定了外部**WMLScript**源程序的位置**(URL)**，并赋予它一个本地名称。该名称可以在函数声明时使用，从而实现外部函数调用 (详见** 6.3.4**节中“ 外部函数 ”)。**

```
use url OtherScript "http://www.host.com/app/script"

function test(par1, par2) {
  return OtherScript#check(par1-par2);
};
```

上例的执行过程如下：

- **编译指示指定了一个**WMLScript**编译单元的**URL**。**
- **通过函数调用，装载**URL(http://www.host.com/app/script)**指定的**WMLScript**编译单元。**
- **验证装入的编译单元的内容并执行给定的函数 (校验)。**

编译指示use url **有它的本地名称，但是在同一个编译单元中，该名称必须唯一。它支持如下的**URL**：**

- **统一的资源定位器**[RFC1738]**，不带“** # **”和字段标识符。该方式的定义在** [WAE]**中。**
- **相对**URL[RFC1808]**，不带“** # **”和字段标识符。基本的** URL**是标识当前编译单元的** URL**。**

必须按照URL**的转义规则使用** URL**，在编译时不执行自动的转义及** URL**语法和有效性检测。**

2. 访问控制

WMLScript **编译单元通过访问控制编译指示来保护自身的内容，它在调用外部函数之前执行访问控制。编译单元中的访问控制编译指示不可多于一个。**

语法：

```
AccessControlPragma:
    access AccessControlSpecifier
AccessControlSpecifier:
    public
    domain StringLiteral
    path StringLiteral
    domain StringLiteral   path StringLiteral
```

每当一个脚本程序调用外部函数时，首先需进行访问控制校验，判断调用者是否具有访问目的编译单元的权限。访问控制标识符用来指定域和路径的属性，参照这些属性进行访问控制校验。如果一个编译单元具有一个域和路径属性，或者是具有域或路径属性其中之一的属性，则URL必须和该属性的值匹配。匹配的过程如下：访问域参照引用 URL的域名部分进行后缀匹配，访问路径参照引用 URL的路径部分进行前缀匹配，域和路径的匹配遵循 URL的基本规则。

域名的后缀匹配要使用子域名的所有元素，而且必须和所有元素匹配（如： www.wap-forum.org 与wapforum.org匹配, 但与 forum.org不匹配)。

路径的前缀匹配要使用所有的路径元素，而且必须和所有的元素匹配 (如： /X/Y 与/X匹配，但是与/XZ不匹配)。

域属性的默认值是当前编译单元的域，路径属性的默认值是“ / ”。

为了简化应用开发，可以不知道当前编译单元的绝对路径，路径属性采用相对的 URL [RFC1808]，用户代理将这些相对路径转换成绝对路径，然后参照路径属性执行前缀的匹配操作。

假设一个编译单元的访问属性如下：

```
use access domain "wapforum.org" path "/finance"
```

允许下面的引用URL调用在该编译单元中定义的外部函数。

```
http://wapforum.org/finance/money.cgi
https://www.wapforum.org/finance/markets.cgi
http://www.wapforum.org/finance/demos/packages.cgi?x=123&y=456
```

不允许下面的引用URL调用在该编译单元中定义的外部函数。

```
http://www.test.net/finance
http://www.wapforum.org/internal/foo.wml
```

编译单元可以使用公共访问控制属性将全部的外部函数定义为可访问的（即：允许从任何一个编译单元中调用外部函数）。如：

```
use access public
```

在默认的情况下，访问控制为允许值。

3. 元信息（Meta-Information）

编译指示也常常用来定义编译单元中特殊的元信息。元信息以属性名和取值定义，本规范没有定义任何属性，也没有规定用户代理该如何解释元数据。用户代理不用对元数据进行操作。

语法：

```
MetaPragma   :
```

```
    meta MetaSpecifier
MetaSpecifier :
    MetaName
    MetaHttpEquiv
    MetaUserAgent
MetaName :
    name MetaBody
MetaHttpEquiv :
    http equiv MetaBody
MetaUserAgent    :
    user agent MetaBody
MetaBody :
    MetaPropertyName MetaContent MetaScheme
```

元（Meta）编译指示有三个特征：属性名、内容（属性的取值）和选择方式（定义了一种用于解释属性值的形式或结构）三个特征的值为字符串文字。

(1) 名字

元编译指示指的是被源服务器使用的元信息，用户代理应该忽视任何被命名为这种属性的元数据。网络服务器不应发送包含元编译指示的 WMLScript内容。

```
use meta name "Created" "18-March-1998"
```

(2) HTTP Equiv

HTTP Equiv元编译指示被用来定义一种元信息，这种元信息给出了 HTTP头的解释（见[RFC2068]）。如果这种元数据在到达用户代理之前它的编译单元就被编译，则要将这些数据转换成WSP或HTTP响应头。

```
use meta http equiv "Keywords" "Script,Language"
```

(3) 用户代理

用户代理元编译指示指的是被用户代理使用的元信息，这种元数据必须传送给用户代理，并且不能被网络的任何中间媒介删除。

```
use meta user agent "Type" "Test"
```

6.4 数据类型的自动转换规则

在某些情况下WMLScript的运算符要求它们的操作数必须是给定的数据类型，这时就需要进行数据类型的转换， WMLScript支持自动的数据类型转换。本节将详细介绍不同类型的数据是如何进行转换的。

6.4.1 基本的转换规则

WMLScript是一种弱类型语言，变量在声明时，无须指定其类型。但是，实际上WMLScript可以处理五种类型的数据。

- 布尔型表示一个布尔型的量，即真值或假值。
- 整型表示一个整型值。
- 浮点型表示一个整型值。
- 字符串型表示一串字符。

- 无效型表示一个无效的值。

一个变量在任何时候都可以取上述任意一种类型的数据值， WMLScript使用typeof运算来决定变量和表达式的类型（无须进行类型转换）。

WMLScript中的每个运算符都事先定义了操作数的类型，如果给定的操作数不符合要求的类型，将进行自动的类型转换。下面给出两种数据类型间合法的自动类型转换的定义。

1. 转换成字符串

从其他的数据类型合法地转换成字符串类型的规则如下：

- 整型值被转换成一个十进制数字字符串，转换遵循十进制整数文字的数字字符串的语法规则。有关数字字符串的语法规则参见6.5.4节。
- 浮点型值转换成一个与具体实现有关的字符串表达形式，转换遵循十进制浮点文字的数字字符串语法规则（有关数字字符串的语法规则参见6.5.4节）。最终的字符串表示必须等于原来的浮点数值（如， .5 可以表示成 “ 0.5 ”、“ .5e0 ” 等)。
- 布尔型的“ 真 ”值将转换成字符串“ true ”，“ 假 ”值将转换成字符串“ false ”。
- 无效型数据不能转换成字符串。

2. 转换成整型

从其他的数据类型合法地转换成整型的规则如下：

- 如果字符串数据是一个能用十进制方式表示的整数（有关数字字符串的语法规则参见6.5.4节），那么它就转换成这个整型值，否则它不可以转换成整型值。
- 浮点型数据不能转换成整型。
- 布尔型“ 真 ”值转换成整型值1，“ 假 ”值转换成整型值 0。
- 无效型数据不能转换成整数值。

3. 转换成浮点型

从其他的数据类型合法地转换成浮点型的规则如下：

- 如果字符型数据包含一个浮点数的有效表达形式（有关数字符串的语法规则参见 5.4节），那么它就转换成这个浮点值，否则它不能转换成浮点型。
- 整数值转换成相应的浮点值。
- 布尔型“ 真 ”值转换成浮点型值1.0，“ 假 ”值转换成整型值0.0。
- 无效型数据不能转换成浮点值。

由于准确地表示一个数据类型的能力有限，字符串与浮点型之间的转换是不连续的，这种转换会导致准确型的损失。

4. 转换成布尔型

从其他的数据类型合法地转换成布尔型的规则如下：

- 空字符串 （“ ”） 转换成“ 假 ”值，其他的字符串转换成“ 真 ”值。
- 整数值0转换成 “ 假 ”值，其他的整数转换成“ 真 ”值。
- 浮点值0转换成“ 假 ”值，其他的浮点数转换成“ 真 ”值。
- 无效型数据不能转换成布尔型。

5. 转换成无效型

任何一种非无效型数据转换成无效型都是不合法的，无效值只可能是错误运算的结果或是一个文字值。在大多数情况下，如果一个运算符的操作数是无效的，运算的结果也是无效

的。(详见6.3.3节中“条件运算”“类型运算”和“有效运算”)。

6. 小结

表6-5总结了/所有的合法转换规则：

表6-5 数据类型的合法转换规则

给定的类型	布 尔 型	整 型	浮 点 型	字 符 串 型
布尔型：True	—	1	1.0	“true”
布尔型：False	—	0	0.0	“false”
整型：0	False	—	0.0	“0”
任何其他的整数	True	—	与这个数的浮点数	一个十进制整数的字符串表示形式
浮点型：0.0	False	非法	—	一个浮点值的字符串表示形式，这种表示与具体的实现有关，即“0.0”
其他的浮点数	True	非法	—	与具体实现有关的浮点值的字符串表示形式
空字符串	False	非法	非法	—
非空字符串	True	字符串表示的整数值，（如果有效，见6.5.4节关于十进制整数文字的数字字符串语法），否则，不能转换成整数值	字符串表示的浮点值，（如果有效，见6.5.4节关于十进制浮点文字的数字字符串语法），否则，不能转换成整数值	—
无效数据	非法	非法	非法	非法

6.4.2 运算符数据类型的转换规则

前面介绍了在两种数据类型之间进行合法转换的基本规则。对于 WMLScript 中的运算符而言，如何恰当地使用这些规则选择操作数类型、执行相应的运算以及得到恰当的结果类型呢？本节将给出相关的附加准则。

- 附加的转换规则是分步定义的，每一步必须按给定的顺序进行，直到操作数的数据类型、相应的运算方式和结果类型都确定为止。
- 如果操作数的类型和要求的类型匹配，则运算结果也使用这个类型。
- 如果操作数的类型和要求的类型不匹配，则要进行操作数转换。
- 合法转换：如果在操作数的类型和要求的数据类型之间存在合法的转换规则（详见 6.4.1 节），则按照这个规则进行。
- 非法转换：如果在操作数的类型和要求的数据类型之间不存在合法的转换规则（详见 6.4.1节），则不能进行转换。
- 如果对操作数而言，存在一种合法的转换规则，则按照该规则执行该转换，然后按照转换后的类型运算，得到符合该类型的结果。如果按照该规则转换后的结果是无效值，则运算结果也是无效值。
- 如果对操作数而言，不存在合法的转换规则，则完成附加规则的下一步操作。

根据给定的操作数类型，表 6-6总结了运算数据类型的转换规则。

表6-6 运算数据类型的转换规则

操作数类型	附加准则	举例
布尔型	如果操作数为布尔型或者可以转换成布尔型，则执行布尔型运算，其结果也为布尔型 否则，得到无效结果	True && 3.4 => boolean 1 && 0 => boolean “A” || “” => boolean ! 42 => boolean ! invalid => invalid 3 && invalid => invalid
整型	如果操作数为整型或者可以转换成整型，则执行整型运算，其结果也为整型 否则，得到无效结果	“7” >> 2 => integer true << 2 => integer 7.2 >> 3 => invalid 2.1 div 4 => invalid
浮点型	如果操作数为浮点型或者可以转换成浮点型，则执行浮点型运算。其结果也为浮点型 否则，得到无效结果	-
字符串型	如果操作数为字符串型或者可以转换成字符串型，则执行字符串型运算。其结果也为字符串型 否则，得到无效结果	-
整型或浮点型（单目）	如果操作数为整型或者可以转换成整型，则执行整型运算。其结果也为整型 否则，如果操作数为浮点型或者可以转换成浮点型，则执行浮点型运算。其结果也为浮点型 否则，得到无效结果	+ 10 => integer - 10.3 => float - “33” => integer + “47.3” => float +true => integer 1 - false => integer 0 = “ABC” => invalid
整型或浮点型	如果操作数中至少有一个为浮点型，那么将其余的操作数都转换成浮点型然后则执行浮点型运算，其结果为浮点型 否则，如果操作数为整型或者可以转换成整型，则执行整型运算。其结果也为整型 否则，如果操作数可以转换成浮点型①，则执行浮点型运算。其结果也为浮点型 否则，得到无效结果	= “9e9999” => invalid 100 / 10.3 => float 33*44 => integer “10”*3 => integer 3.4*“4.3” => float “10”-“2” => integer “2.3”*“3” => float 3.2*“a” => invalid 9*“9e9999” => invalid
整型、浮点型或字符串型	如果操作数中至少有一个为字符型，那么将其余的操作数都转换成字符串型然后则执行字符串型运算，其结果为字符串型 否则，如果操作数中至少有一个为浮点型1，那么将其余的操作数都转换成浮点型然后则执行浮点型运算，其结果为浮点型 否则，如果操作数为整型或者可以转换成整型①，则执行整型运算。其结果也为整型 否则，返回无效值	invalid*1=> invalid 12+3 => integer 32.4+65=> float “12”+5.4=> string 43.2<77=> float “Hey”<56=> string 2.7+“4.2” => string 9.9+true=>float 3<false=> integer “A”+invalid=> invalid
任何（Any）	任何类型都可以	a = 37.3 => float b = typeof “s” => string

① 如果一般的转换规则规定了从当前类型到请求类型的转换是合法的，那么可进行转换。

6.4.3 运算符和类型转换的总结

本节总结了在WMLScript 中，进行各种运算时，如何使用上述基本规则对操作数进行转换。

1. 单类型操作数的运算符

如果运算符仅仅使用一种特定类型的操作数，那么操作数的类型转换直接按照通用的类型转换规则进行。表6-7列出了在WMLScript中仅使用单类型操作数进行的运算。

表6-7 在WMLScript 中使用单类型操作数的运算符

运 算 符	操作数类型	结 果 类 型①	运 算
!	布尔型	布尔型	逻辑非 (单目的)
&&	布尔型	布尔型	逻辑与
||	布尔型	布尔型	逻辑或
~	整型	整型	位非 (单目)
<<	整型	整型	位左移
>>	整型	整型	带符号的位右移
>>>	整型	整型	补零的位右移
&	整型	整型	位与
^	整型	整型	位异或
|	整型	整型	位或
%	整型	整型	取余
div	整型	整型	整除
<<=, >>=, >>>=,	第一个操作数：变量	整型	位操作赋值
&=, ^=, |=%=, div=	第二个操作数：整型		
	第一个操作数：变量	整型	数值运算赋值
	第二个操作数：整型		

① 所有的操作员可有一个无效的结果类型。

2. 多类型操作数使用的运算符

表6-8总结了多种类型操作数能够使用的运算符。

表6-8 多类型操作数使用的运算符

运 算 符	操作数类型	结 果 类 型①	完成的运算
++	整型或浮点型	整型/浮点型	自增运算（单目）
—	整型或浮点型	整型/浮点型	自减运算（单目）
+	整型或浮点型	整型/浮点型	单目加
-	整型或浮点型	整型/浮点型	单目减（取负）
*	整型或浮点型	整型/浮点型	乘法运算
/	整型或浮点型	整型/浮点型	除法运算
-	整型或浮点型	整型/浮点型	减法运算
+	整型、浮点型或字符串型	整型/浮点型/字符串型	加法/连接字符串运算
<, <=	整型、浮点型或字符串型	布尔型	小于，小于等于
>, >=	整型、浮点型或字符串型	布尔型	大于，大于等于
==	整型、浮点型或字符串型	布尔型	等于（相等值）
!=	整型、浮点型或字符串型	布尔型	不等于（不相等值）

（续）

运 算 符	操作数类型	结 果 类 型①	完成的运算
*=, /=, -=	第一个操作数：变量 第二个操作数：整型或浮点型	整型/浮点型	数值运算并赋值
+=	第一个操作数：变量 第二个操作数：整型或浮点型	整型/浮点型/字符串型	加并赋值
typeof	任意（any）	整型②	返回一个内部数据类型（单目）
isvalid	任意（any）	布尔型③	有效性检验运算
?:	第一个操作数：布尔型 第二个操作数：任意（any） 第三个操作数：任意（any）	任意（any）	条件表达式
=	第一个操作数：变量 第二个操作数：任意（any）	任意（any）	赋值
,	第一个操作数：任意（any） 第二个操作数：任意（any）	任意（any）	逗号运算

① 所有的运算（除非另外声明）都可有一个无效的结果类型。

② 运算不产生无效的结果类型。

③ 运算不产生无效的结果类型。

6.5 WMLScript 文法

本规范使用的语法建立在 [ECMA262]基础之上。由于 WMLScript不是完全地顺从ECMAScript的，所以[ECMA262]只作为定义WMLScript语言的基础。

6.5.1 上下文无关文法

为了定义WMLScript 编程的词法和句法结构，本节将介绍在本规范中使用上下文无关文法。

1. 概述

上下文无关的文法由许多产品包（ production）组成，每个产品包由左半部分和右半部分构成：左半部分是一个被称做非终结符的抽象的符号，右半部分由一个或多个非终结符和终结符构成。每种文法中的终结符都是从指定的字母表中抽取的。

一个给定的与上下文无关的文法定义了一种语言。开始的产品包由一明显的非终止目标符和随后跟着的一套（可能是无限个）终止符组成，这些终止符是反复的通过替换序列中的非终止符而得到的，它用一个左侧是非终止符的产品包的右半部分来替代。

2. 词汇的文法

词汇的文法定义在6.5.2节。该文法以通用字符集ISO/IEC-10646 ([ISO10646])中的字符作为终止符，它定义了一套产品包，以目标符 Input开始，该目标符描述了下面的字符序列如何转化成一个适合句子文法的由输入元素组成序列。

按照WMLScript的句子的文法，输入元素中除了空格和注释行之外的元素形成终止符，这些终止符被称做WMLScript记号（token）。WMLScript记号可以是WMLScript语言中的保留字、标识符、文字或标点符号，在形成输入元素序列时，丢弃简单的空格和单行注释。同样，如果多行注释中不包含行结束符，多行注释也将被丢弃。但是如果多行注释中包含一个或多

个行结束符，多行注释将被一个行结束符替代，它将成为输入元素流的一部分。词汇文法的产品包以“ :: ”分隔。

3. 句子的文法

句子的文法定义在6.5.3节。该文法也将WMLScript记号作为终止符，其中WMLScript记号定义和词汇文法中的相同。它定义了一套产品包，以目标符 CompilationUnit开始，该目标符描述了下面的WMLScript记号序列如何正确地转化成WMLScript程序。

当分析出一串Unicode字符是WMLScript时，首先将使用词汇文法将它们转换成一串输入元素，然后按照句子文法的一个应用分析这些输入元素。如果在遇到目标非终结符 CompilationUnit时，还不能得到输入元素串的分析结果，即不能产生一个实例，则程序出现文法错误。

句子文法的产品包以“ : ”分隔。

4. 数字符串的文法

数字符串的文法定义了如何将字符串转成数字值，该文法类似于词汇文法中和数字文字相关的部分，它以采用Unicode字符集的字符作为终止符。关于该文法的详细定义见6.5.4节。

数字符串文法的产品包以“ :: ”分隔。

5. 文法符号

词汇文法、字符串文法和句子文法中的终止符在产品包和本规范的书写中均使用固定宽度（fixed width） 字体，这种作法和实际的程序书写方式一样。

非终止符使用斜体字。非终止符的定义由定义的非终止符的名字加一个或多个冒号引出（冒号的数目代表着该产品包属于哪种文法），下面的行中是一个或多个可选的右半部分。例如，句子的文法定义为：

```
WhileStatement :
    while ( Expression ) Statement
```

这个例子说明，非终止符WhileStatement 的定义是：以“ while ”记号开头，后跟“（”记号、一个“ Expression ”和“ ）”记号，然后是一个Statement，其中Expression 和Statement 是非终止符。文法定义为：

```
ArgumentList :
    AssignmentExpression
    ArgumentList , AssignmentExpression
```

这个例子说明，ArgumentList 可以是一个单独的参数表达式（AssignmentExpression），或者是以逗号连接的一个参数列表（ArgumentList）和一个参数表达式（AssignmentExpression）。定义ArgumentList采用了递归的方式，也就是采用了自己定义自己的方式。通过这个定义可以知道，ArgumentList 包括任意多个（非0）以逗号连接的参数，每个参数的表示形式都是参数表达式，这种采用递归方式定义非终止符的情况是很常见的。

下缀“ *opt* ”出现在终止符和非终止符后面，表示该符号是一个可选项（ optional symbol）。可选项的存在，使产品包实际上有了两个右边部分，一个是包含可选项的，一个是不包含可选项。这意味着表示形式：

```
VariableDeclaration:
    Identifier VariableInitializer
```

是下面表示形式的一个缩写：

```
VariableDeclaration:
   Identifier
   Identifier VariableInitializer
```

又如:

```
IterationStatement:
   for ( Expression opt ; Expression opt ; Expression opt ) Statement
```

如果去掉一个下缀“ *opt* **”，可以写成**:

```
IterationStatement :
   for ( ; Expression opt ; Expression opt ) Statement
   for ( Expression ; Expression opt ; Expression opt ) Statement
```

如果再去掉一个下缀“ *opt* **”，又可以依次写成**:

```
IterationStatement:
   for ( ; ; Expression opt ) Statement
   for ( ; Expression ; Expression opt ) Statement
   for ( Expression ; ; Expression opt ) Statement
   for ( Expression ; Expression ; Expression opt ) Statement
```

如果再去掉一个下缀“ *opt* **”，又可以依次写成**:

```
IterationStatement    :
   for ( ; ; ) Statement
   for ( ; ; Expression ) Statement
   for ( ; Expression   ; ) Statement
   for ( ; Expression   ; Expression   ) Statement
   for ( Expression   ; ; ) Statement
   for ( Expression   ; ; Expression   ) Statement
   for ( Expression   ; Expression ; ) Statement
   for ( Expression   ; Expression   ; Expression   ) Statement
```

所以，非终止符“ *IterationStatement* **”实际上有8种可选的右半部分。**

行终止符“ *LineTerminator* **” 可以在输入元素流中的任何两个**WMLScript**记号之间出现任意多次，不影响程序文法的正确性。**

在文法定义中，当“ one of **”后面跟有一个或多个冒号时，它表明下面几行终止符里的任何一个都可以作为定义。例如，在**WMLScript**的词汇文法中有如下定义：**

```
ZeroToThree   :: one of
   0      1      2      3
```

上面的定义方式仅仅是下面定义方式的一种简单的表达形式：

```
ZeroToThree   ::
   0
   1
   2
   3
```

在词汇文法和数字字符串文法的产品包中，出现可选择项是由多个字符组成时，表明这些字符组成这样一个记号。

产品包的右半部分可以用“ but not **”定义禁止使用的扩展部分，如：**

```
Identifier ::
   IdentifierName but not ReservedWord
```

意味着非终止符“ Identifier ”可以由任意字符序列代替，这些字符序列可以是“ IdentifierName ”，但是不能是“ ReservedWord ”。

最后要说明的是，有少数非终止符采用了正体的描述词组，因为如果不这样做，只能不切实际地列出所有的选择字符。如：

```
SourceCharacter  :
   any Unicode character
```

6. 源文本

WMLScript源文本实际上是一个字符序列，这些字符可以用 ISO/IEC-10646 ([ISO10646]) 通用字符集中的字符来描述。目前，这个字符集是 Unicode 2.0 ([UNICODE])。在本书中，ISO10646 和Unicode指同一个字符集：

```
SourceCharacter ::
   any Unicode character
```

WMLScript文档不需要使用全部的 Unicode编码（如 UCS-4）来进行编码，任何包含 Unicode字符子集的字符编码方案（“ charset ”）都可能被使用（如 US-ASCII, ISO-8859-1等等）。

每个WMLScript程序能够只用 ASCII字符（即Unicode最开始的128个字符）来表示，非ASCII的Unicode字符仅能出现在注释和字符串文字中。在字符串文字中，任何Unicode 字符可以被表示为包含6个ASCII字符（即 \u加上4个十六进制数）的Unicode的转义序列。在编译中，这样的转义序列被当作注释的一部分而被忽略。在字符串文字中，Unicode 转义序列给文字的字符串值提供了一个字符。

6.5.2 WMLScript词汇文法

下面是WMLScript中定义的词汇文法：

```
SourceCharacter  ::
  any Unicode character
WhiteSpace ::
  <TAB>
  <VT>
  <FF>
  <SP>
  <LF>
  <CR>
LineTerminator ::
  <LF>
  <CR>
  <CR><LF>
Comment ::
  MultiLineComment
  SingleLineComment
MultiLineComment ::
  /* MultiLineCommentChars */
```

MultiLineCommentChars::
  *MultiLineNotAsteriskChar MultiLineCommentChars*opt
  ** PostAsteriskCommentChars*opt
PostAsteriskCommentChars::
  *MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars*opt
  ** PostAsteriskCommentChars*opt
MultiLineNotAsteriskChar::
  *SourceCharacter but not asterisk **
MultiLineNotForwardSlashOrAsteriskChar
  *SourceCharacter but not forward-slash / or asterisk **
SingleLineComment::
  *// SingleLineCommentChars*opt
SingleLineCommentChars::
  *SingleLineCommentChar SingleLineCommentChars*opt
SingleLineCommentChar::
  SourceCharacter but not LineTerminator
Token ::
  ReservedWord
  Identifier
  Punctuator
  Literal
ReservedWord ::
  Keyword
  KeywordNotUsedByWMLScript
  FutureReservedWord
Keyword :: *one of*

access	equiv	meta	var
agent	extern	name	while
break	for	path	url
continue	function	public	
div	header	return	
div=	http	typeof	
domain	if	use	
else	isvalid	user	

KeywordNotUsedByWMLScript :: one of

delete	null		
in	this		
lib	void		
new	with		

FutureReservedWord :: one of

case	default	finally	super
catch	do	import	switch
class	enum	private	throw
const	export	sizeof	try
debugger	extends	struct	

Identifier ::
  IdentifierName but not ReservedWord
IdentifierName ::
  IdentifierLetter

```
  IdentifierName IdentifierLetter
  IdentifierName DecimalDigit
IdentifierLetter   :: one of㊀
a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
  _
DecimalDigit   :: one of
  0    1    2    3    4    5    6    7    8    9
Punctuator :: one of㊁

=    >      <      ==     <=     >=
!=   ,      !      ~      ?      :
.    &&     ||     ++     --     +
-    *      /      &      |      ^
%    <<     >>     >>>    +=     -=
*=   /=     &=     |=     ^=     %=
<<=  >>=    >>>=   (      )      {
}    ;      #
Literal :: ㊂
  InvalidLiteral
  BooleanLiteral
  NumericLiteral
  StringLiteral
InvalidLiteral :: ㊃
  invalid
BooleanLiteral :: ㊄
  true
  false
NumericLiteral ::
  DecimalIntegerLiteral
  HexIntegerLiteral
  OctalIntegerLiteral
  DecimalFloatLiteral
DecimalIntegerLiteral::
  0
  NonZeroDigit DecimalDigitsopt
NonZeroDigit :: one of
  1   2   3   4   5   6   7   8   9
HexIntegerLiteral::
  0x HexDigit
  0X HexDigit
  HexIntegerLiteral HexDigit
HexDigit :: one of
```

㊀ **兼容性注解：**ECMAScript**支持在标识符名称中使用美元符号（**$**）。**

㊁ **兼容性注解：**ECMAScript**支持矩阵和方括号（**[ ]**）的使用。**

㊂ **兼容性注解：**ECMAScript**支持字符**Null**文字。**

㊃ **兼容性注解：**ECMAScript**不支持无效型**invalid**。**

㊄ **兼容性注解：**ECMAScript**支持大小写布尔文字。**

```
  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  A  B  C  D  E  F
OctalIntegerLiteral    ::
  0 OctalDigit
  OctalIntegerLiteral OctalDigit
OctalDigit :: one of
  0  1  2  3  4  5  6  7
DecimalFloatLiteral::
  DecimalIntegerLiteral . DecimalDigits opt ExponentPart opt
  . DecimalDigits ExponentPart opt
  DecimalIntegerLiteral ExponentPart
  DecimalDigits ::
       DecimalDigit
       DecimalDigits DecimalDigit
ExponentPart ::
  ExponentIndicator SignedInteger
ExponentIndicator:: one of
  e  E
SignedInteger ::
  DecimalDigits
  + DecimalDigits
  - DecimalDigits
StringLiteral ::
  "DoubleStringCharacters    opt  "
'SingleStringCharacters    opt '
DoubleStringCharacters::
  DoubleStringCharacter DoubleStringCharacters opt
SingleStringCharacters::
  SingleStringCharacter SingleStringCharacters opt
DoubleStringCharacter::
  SourceCharacter but not double-quote or "backslash \ or LineTerminator
  EscapeSequence
SingleStringCharacter::
  SourceCharacter but not single-quote or' backslash \ or LineTerminator
  EscapeSequence
EscapeSequence ::
  CharacterEscapeSequence
  OctalEscapeSequence
  HexEscapeSequence
  UnicodeEscapeSequence
CharacterEscapeSequence::
  \ SingleEscapeCharacter
SingleEscapeCharacter    :: one of
  '    "    \     /     b     f     n     r     t
HexEscapeSequence::
  \x HexDigit HexDigit
OctalEscapeSequence::
  \ OctalDigit
```

```
  \ OctalDigit OctalDigit
  \ ZeroToThree OctalDigit OctalDigit
ZeroToThree  :: one of
  0   1   2   3
UnicodeEscapeSequence    ::
  \u HexDigit HexDigit HexDigit HexDigit
```

6.5.3 WMLScript 句子文法

下面是WMLScript **中定义的句子文法**:

```
PrimaryExpression:⊖
   Identifier
   Literal
   ( Expression )
CallExpression: ⊜
   PrimaryExpression
   LocalScriptFunctionCall
   ExternalScriptFunctionCall
   LibraryFunctionCall
LocalScriptFunctionCall
   FunctionName Arguments
ExternalScriptFunctionCall
   ExternalScriptName # FunctionName Arguments
LibraryFunctionCall:
   LibraryName . FunctionName Arguments
FunctionName:
   Identifier
ExternalScriptName:
   Identifier
LibraryName:
   Identifier
Arguments:
   ( )
   ( ArgumentList )
ArgumentList:
   AssignmentExpression
   ArgumentList , AssignmentExpression
PostfixExpression:
   CallExpression
   Identifier ++
   Identifier (
UnaryExpression:⊜
   PostfixExpression
```

⊖ **兼容性注解：**ECMAScript**也支持对象和**this。

⊜ **兼容性注解：**ECMAScript**支持矩阵（**[ ]**）和对象分配（**new**）。***MemberExpression***用来指定库函数；如：**String.length(“abc”),**不能访问一个对象的成员。**

⊜ **兼容性注解：**ECMAScript**不支持**delete**和**void**运算。**ParseInt **和** parseFloat**被作为库函数来支持。**ECMAScript **不支持运算**isvalid**。**

```
    typeof UnaryExpression
    isvalid UnaryExpression
    ++ Identifier
    - Identifier
    + UnaryExpression
    - UnaryExpression
    ~ UnaryExpression
    ! UnaryExpression
MultiplicativeExpression:⊖
    UnaryExpression
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression div UnaryExpression
    MultiplicativeExpression % UnaryExpression
AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression
ShiftExpression:
    AdditiveExpression
    ShiftExpression << AdditiveExpression
    ShiftExpression >> AdditiveExpression
    ShiftExpression >>> AdditiveExpression
RelationalExpression:
    ShiftExpression
    RelationalExpression < ShiftExpression
    RelationalExpression > ShiftExpression
    RelationalExpression <= ShiftExpression
    RelationalExpression >= ShiftExpression
EqualityExpression:
    RelationalExpression
    EqualityExpression == RelationalExpression
    EqualityExpression != RelationalExpression
BitwiseANDExpression:
    EqualityExpression
    BitwiseANDExpression & EqualityExpression
BitwiseXORExpression:
    BitwiseANDExpression
    BitwiseXORExpression ^ BitwiseANDExpression
BitwiseORExpression:
    BitwiseXORExpression
    BitwiseORExpression | BitwiseXORExpression
LogicalANDExpression:
    BitwiseORExpression
    LogicalANDExpression && BitwiseORExpression
LogicalORExpression:
    LogicalANDExpression
```

⊖ **兼容性注解：**ECMAScript**不支持整数除法（**div**）。**

```
   LogicalORExpression || LogicalANDExpression
ConditionalExpression:
   LogicalORExpression
   LogicalORExpression ? AssignmentExpression : AssignmentExpression
AssignmentExpression:
   ConditionalExpression
   Identifier AssignmentOperator AssignmentExpression
AssignmentOperator    :: one of
   =  *=  /=  %=  += -=  <<=  >>=  >>>=  &=  ^=  |=  div=
Expression :
   AssignmentExpression
   Expression , AssignmentExpression
Statement :⊖
   Block
   VariableStatement
   EmptyStatement
   ExpressionStatement
   IfStatement
   IterationStatement
   ContinueStatement
   BreakStatement
   ReturnStatement
Block :
   { StatementList opt }
StatementList :
   Statement
   StatementList Statement
VariableStatement:
   var VariableDeclarationList ;
VariableDeclarationList:
   VariableDeclaration
   VariableDeclarationList , VariableDeclaration
VariableDeclaration:
   Identifier VariableInitializer opt
VariableInitializer:
   = ConditionalExpression
EmptyStatement:
   ;
ExpressionStatement:
   Expression ;
IfStatement :⊜
   if ( Expression ) Statement else Statement
   if ( Expression ) Statement
IterationStatement    :⊝
```

⊖ **兼容性注解：不支持带有声明的**ECMAScript。

⊜ *else***总是和最近的***if* **匹配。**

⊝ **兼容性注解：不支持带有***for* **语句的**ECMAScript。

WhileStatement
ForStatement

WhileStatement:
while (Expression) Statement

ForStatement:
for (Expression$_{opt}$ *; Expression*$_{opt}$ *; Expression*$_{opt}$ *) Statement*
for (var VariableDeclarationList ; Expression$_{opt}$ *; Expression*$_{opt}$ *) Statement*

ContinueStatement:㊀
continue ;

BreakStatement:㊁
break ;

ReturnStatement:
return Expression$_{opt}$ *;*

FunctionDeclaration:㊂
extern$_{opt}$ function *Identifier (FormalParameterList*$_{opt}$*) Block ;*$_{opt}$

FormalParameterList:
Identifier
FormalParameterList , Identifier

CompilationUnit:
Pragmas$_{opt}$ *FunctionDeclarations*

Pragmas:㊃
Pragma
Pragmas Pragma

Pragma:
use *PragmaDeclaration ;*

PragmaDeclaration:
ExternalCompilationUnitPragma
AccessControlPragma
MetaPragma

ExternalCompilationUnitPragma
url *Identifier StringLiteral*

AccessControlPragma:㊄
access *AccessControlSpecifier*

AccessControlSpecifier:
public
domain *StringLiteral*
path *StringLiteral*
domain *StringLiteral path StringLiteral*

MetaPragma:
meta *MetaSpecifier*

MetaSpecifier:
MetaName

㊀ **连续语句只能在*while* 和*for*语句中使用。**
㊁ ***break*语句只能在*while* 和*for*语句中使用。**
㊂ **兼容性注解：**ECMAScript**不支持关键字*extern*.。**
㊃ **兼容性注解：**ECMAScript**不支持*pragmas*。**
㊄ **编辑单元只能包含一个访问控制附注。**

```
    MetaHttpEquiv
    MetaUserAgent
MetaName :
    name MetaBody
MetaHttpEquiv   :
    http equiv MetaBody
MetaUserAgent :
    user agent MetaBody
MetaBody :
    MetaPropertyName MetaContent MetaSchemeopt
MetaPropertyName :
    StringLiteral
MetaContent :
    StringLiteral
MetaScheme :
    StringLiteral
FunctionDeclarations:
    FunctionDeclaration
    FunctionDeclarations FunctionDeclaration
```

6.5.4 数字字符串的文法

下面是WMLScript中定义的数字字符串文法，数字字符串文法定义了如何将字符串转成数字值。该语法类似于词汇语法中与数字文字相关的部分，它以 Unicode字符作为终止符。

下面的语法规则定义了如何将字符串转成数字值 :

- 十进制整型文字使用以下面目标符 StringDecimalIntegerLiteral开始的产品包。
- 十进制浮点文字使用以下面目标符 StringDecimalFloatingPointLiteral开始的产品包。

```
StringDecimalIntegerLiteral:::
    StrWhiteSpaceopt StrDecimalIntegerLiteral StrWhiteSpaceopt
StringDecimalFloatingPointLiteral:
    StrWhiteSpaceopt StrDecimalIntegerLiteral StrWhiteSpaceopt
    StrWhiteSpaceopt StrDecimalFloatingPointLiteral StrWhiteSpaceopt
StrWhiteSpace :::
    StrWhiteSpaceChar StrWhiteSpaceopt
StrWhiteSpaceChar :::
    any Unicode character with character code less than or equal to 32
StrDecimalIntegerLiteral:::
    StrDecimalDigits
    + StrDecimalDigits
    - StrDecimalDigits
StrDecimalFloatingPointLiteral:::
    StrDecimalDigits . StrDecimalDigitsopt StrExponentPartopt
    . StrDecimalDigits StrExponentPartopt
    StrDecimalDigits StrExponentPart
StrDecimalDigits :::
    StrDecimalDigit
    StrDecimalDigits StrDecimalDigit
```

```
StrDecimalDigit   ::: one of
   0  1  2  3  4  5  6  7  8  9
StrExponentPart:::
   StrExponentIndicator StrSignedInteger
StrExponentIndicator    ::: one of
   e  E
StrSignedInteger:::
   StrDecimalDigits
   + StrDecimalDigits
   - StrDecimalDigits
```

6.6 WMLScript 字节码的解释器

WMLScript语言的文本格式必须被编译成二进制格式后才能被 WMLScript 字节码的解释程序识别。使用在第7章“WMLScript标准库规范”中给出的编码格式， WMLScript编译器对一个编译单元进行编码，生成字节码。一个编译单元（见 6.5.1节中“句子的文法”）包含了各种WMLScript编译指示和函数， WMLScript编译器以编译单元为输入，并生成相应的WMLScript输出字节码。

6.6.1 解释器结构

WMLScript解释器以WMLScript字节码为输入，并执行经过编码的调用函数。图 6-1给出了与WMLScript字节码有关的主要部分。

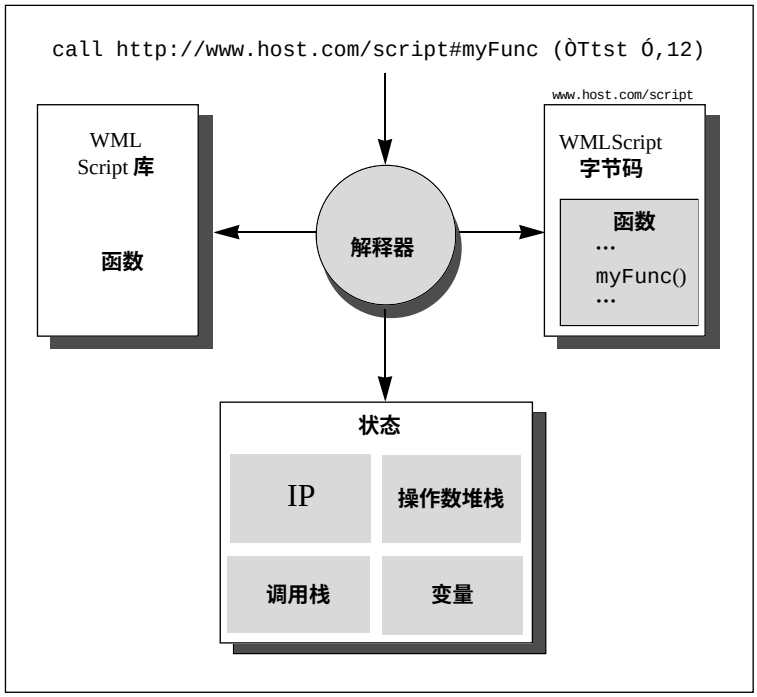

图6-1 WMLScript解释器的总体结构

在WMLScript编译单元被编码成字节码之后，WMLScript解释器就可以进行函数的调用和执行，每个函数都有确定数目的输入参数和相应的指令来实现一定的功能。因此，对WMLScript函数的调用必须指定函数名、调用参数和相关的编译单元。如果函数执行过程正常结束，解释器就会向调用者交出控制权并返回参数。

执行WMLScript函数也就意味着对其字节码指令进行解释。当一个函数在执行时，解释器保持以下状态信息：

- IP（指令指示器） 指向正在被解释的字节码指令。
- 变量 保存函数中的参数和变量。
- 操作数堆栈 用于表达式计算和在调用者与被调函数之间进行参数传递。
- 函数调用堆栈 WMLScript函数可以在当前或别的编译单元中调用其他的函数或库函数。函数调用堆栈包含了与函数名和其返回地址相关的信息。

6.6.2 WMLScript和URL

万维网是提供信息和服务的网络。下述三个方面的特色使得它具有优越的交互性：

- 统一的命名模式 命名用统一资源定位器（URL）实现，它能为任何网络资源提供标准的命名方式 （见RFC[1738]）。
- 标准的传输协议 （如：HTTP）。
- 标准的内容类型 （如：HTML、WMLScript）

假设WMLScript与HTML和万维网有相同的参考结构，WMLScript编译单元也用URL命名，并能由符合HTTP语义的标准协议（如 [WSP]）传输。[RFC1738]给出了URL的定义，并规定了URL使用的字符集。

在WMLScript中，URL 用于以下一些场合：

- 用户代理发起一个WMLScript调用时（见6.6.2节中“ URL调用和参数传递 ”）
- 需要指定外部编译单元时（见 6.3.7节中“ 外部编译单元 ”）
- 需要确定访问控制信息时（见 6.3.7节中“ 访问控制 ”）

1. URL 方案

WMLScript解释器必须实现[WAE]中规定的URL方案。

2. 字段锚

WMLScript采用了HTML公认的标准，用位置命名指定资源。一个 WMLScript字段锚是由关键字‘ URL ’打头，紧跟一个‘ # ’标志，后面是字段标识符。 WMLScript用字段锚来识别编译单元中的WMLScript函数。字段锚的语法规则将在下节介绍。

3. URL调用语法

本节介绍了URL调用的语法结构，它类似于用在函数调用和文法中的 WMLScript词法和语法部分，并以US-ASCII字符集中的字符作为终止符。

```
http://www.host.com/scr#foo(1,-3,'hello')                    // OK
http://www.host.com/scr#bar(1,-3+1      ,'good')             // Error
http://www.host.com/scr#test      ( foo(1,-3,'hello'   ))    // Error
```

这里只提到了字段锚 (#)的语法 (有关URL语法的详细信息，参见[RFC1808])。

```
URLCallFragmentAnchor:::
```

FunctionName( )

FunctionName(ArgumentList)

FunctionName ::=

FunctionNameLetter

FunctionName FunctionNameLetter

FunctionName DecimalDigit

FunctionNameLetter ::= one of

a b c d e f g h I j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

DecimalDigit ::= one of

0 1 2 3 4 5 6 7 8 9

ArgumentList :

Argument

ArgumentList , Argument

Argument ::=

WhiteSpace $_{opt}$ *Literal WhiteSpace* $_{opt}$

WhiteSpace ::=

any US-ASCII character with character code less than or equal to 32

Literal ::=

InvalidLiteral

BooleanLiteral

NumericLiteral

StringLiteral

InvalidLiteral ::=

invalid

BooleanLiteral ::=

true

false

NumericLiteral ::=

SignedDecimalIntegerLiteral

SignedDecimalFloatLiteral

SignedDecimalIntegerLiteral ::=

DecimalIntegerLiteral

+ DecimalIntegerLiteral

- DecimalIntegerLiteral

DecimalIntegerLiteral ::=

DecimalDigit DecimalDigits $_{opt}$

SignedDecimalFloatLiteral ::=

DecimalFloatLiteral

+ DecimalFloatLiteral

```
        - DecimalFloatLiteral
   DecimalFloatLiteral:::
        DecimalIntegerLiteral . DecimalDigits opt ExponentPart opt
        . DecimalDigits ExponentPart opt
        DecimalIntegerLiteral ExponentPart
   DecimalDigits :::
        DecimalDigit
        DecimalDigits DecimalDigit
   ExponentPart :::
        ExponentIndicator SignedInteger
   ExponentIndicator    ::: one of
        e  E
   SignedInteger :::
        DecimalDigits
        + DecimalDigits
        - DecimalDigits
   StringLiteral :::
 "DoubleStringCharactersopt "
   'SingleStringCharactersopt '
DoubleStringCharacters:::
        DoubleStringCharacter DoubleStringCharacters opt
   SingleStringCharacters:::
        SingleStringCharacter SingleStringCharacters opt
   DoubleStringCharacter:::
        SourceCharacter    but not double-quote"or backslash \
        EscapeSequence
   SingleStringCharacter:::
        SourceCharacter    but not single-quote'or backslash \
        EscapeSequence
   EscapeSequence :::
        CharacterEscapeSequence
        OctalEscapeSequence
        HexEscapeSequence
        UnicodeEscapeSequence
CharacterEscapeSequence:::
        \ SingleEscapeCharacter
   SingleEscapeCharacter    ::: one of
    '  "  \  /  b  f  n  r  t
   HexEscapeSequence:::
        \x HexDigit HexDigit
```

```
OctalEscapeSequence:::
        \ OctalDigit
        \ OctalDigit OctalDigit
        \ ZeroToThree OctalDigit OctalDigit
   ZeroToThree  ::: one of
        0  1  2  3
UnicodeEscapeSequence:::
   \u HexDigit HexDigit HexDigit HexDigit
```

4. URL 调用和参数传递

用户代理可以调用一个WMLScript外部函数，条件是使用URL和字段锚提供以下一些信息：

- 编译单元的URL（如，http://www.x.com/myScripts.scr）
- 以函数名和参数作为字段锚（如：testFunc(‘Test%20argument’,-8))。

用字段作结尾的URL格式如下：

```
http://www.x.com/myScripts.scr#testFunc('Test%20argument',-8)
```

如果给定的URL表示的是一个合法的WMLScript编译单元，那么：

- 进行访问控制检查（见6.3.7节中“访问控制”）。如果调用者无权访问本编译单元，调用失败。
- 字段锚中的函数名必须与编译单元中的外部函数匹配，如果不匹配，调用会失败。
- 解析字段锚中的参数表（见6.6.2节中“字段锚”），给定的变量及其类型（字符串文字表示字符串数据类型，整型文字表示整数数据类型）将传递给相应的函数。如果参数列表语法不合法，调用失败。

5. 字符转义

URL调用既可以采用URL转义[RFC1738]定义URL，也可以采用WMLScript字符串转义（见6.3.1节中“字符串文字”），作为字符串文字中的Unicode字符。传递字符串文字作函数参数时，一个URL既不能使用URL转义规则，也不能使用WMLScript字符串文字转义规则。

6. 相对URL

正如[RFC1808]所述，WMLScript也采用了相对URL，它还规定了如何在一个WMLScript编译单元上下文中解析相对URL的方法。一个WMLScript编译单元的基本URL是指标识那个单元的URL。

6.6.3 字节码语义

以下各节描述了生成WMLScript字节码时必须遵守的一般编码规则，这些规则说明了WMLScript编译器从解释器工作状态中可以假设成立的内容。

1. 函数参数的传递

在WMLScript或库函数被调用时，操作数堆栈中的变量必须按它们在函数说明中的顺序存取。因此，第一个变量最先被推入堆栈，然后是第二个变量，依次类推。

2. 自动的函数返回值

在没有返回说明时，WMLScript函数必须返回一个空字符串。每当解释器到达了没有返

回指令的函数末尾，能依靠解释器自动地返回空字符串。

3. 变量初始化

WMLScript编译器依靠解释器把所有函数的局部变量初始化成一个空字符串。因此，编译器无需对没有初始化描述的变量生成初始化代码。

6.6.4 访问控制

WMLScript提供了两种机制，控制WMLScript对编译单元中函数的访问：外部关键字和特定的访问控制语句。因此，WMLScript解释器必须支持以下一些功能：

- 外部函数　只有外部函数可以被其他编译单元调用（见6.3.4节）。
- 访问控制　在一个编译单元中定义的外部函数，可以被那些与给定访问域和访问路径定义相匹配的其他编译单元访问。

6.7 WMLScript的二进形式

以下各节给出了WMLScript字节码使用规范。WMLScript字节码是经过编译的WMLScript函数的紧凑二进制表示形式，这种格式适合于在窄带信道上的传输，不会有功能信息或说成是语义信息的损失。

6.7.1 习惯用法

以下各节描述了生成WMLScript字节码时常用的编码习惯和数据类型。

1. 使用的数据类型

WMLScript字节码规范中使用了如下的数据类型（见表6-9）：

表6-9 WMLScript 字节码规范中使用的数据类型

数 据 类 型	定　义
bit	一个比特数据
byte	八比特不透明数据
int8	八比特有符号整数
u_int8	八比特无符号整数
int16	十六比特有符号整数
u_int16	十六比特无符号整数
mb_u_int16	十六比特无符号整数，用在多字节格式中。详细说明见6.7.1节中“多字节整型格式”
int32	三十二比特有符号整数
u_int32	三十二比特无符号整数
mb_u_int32	三十二比特无符号整数，用在多字节格式中。详细说明见6.7.1节中“多字节整型格式”
Float32	三十二比特有符号的浮点数，ANSI/IEEE Std 754-1985 [IEEE754] 格式

网络字节顺序采用低字节高置（big-endian）的存取次序，换句话说，网络首先传输最高字节，然后传输低字节。字节的网络比特顺序也采用低比特高置的存取次序，即最先描述的比特段存储在这个字节的最高位地址中。

2. 多字节整型格式

对于整数值，编码采用多字节表示法，一个多字节整数由一系列的八位组 (octet)构成。一个八位组中，最重要的比特位是连续标志位，其余的 7个比特是标量值，连续标志位为 1表明这个八位组不是多字节序列的尾字节。若一个整数值的编码序列由 N个八位组构成，那么前面N－1个字节的连续标志位是1，最后一个字节的连续标志位是0。

每个八位组中非连续标记位的7个比特也采用低比特高置的存取次序，即最高位首先被传输。所有的八位组也以低字节高置的存取次序排列，换句话说，最高八位组上的 7个比特首先被传输。对于取值小于7个比特的情况，未被使用的比特位必须置0。

举一个例子，整数值0xA0编成2个字节的序列0x81 0x20，整数0x60被编成1个字节的序列0x60。

3. 字符编码

WMLScript字符串使用Unicode[UNICODE]字符集。WMLScript字节码支持以下Unicode字符编码：

- UTF-8 (见 [RFC2279])。
- UCS-2 (见 [ISO10646])。
- UCS-4 (见 [ISO10646])。

编译器必须从中选择一种编码方法，对WMLScript字节码中的字符串进行编码。

整个WMLScript语言结构，如WMLScript中的函数名，只是用Unicode字符集的一个子集编写的，也就是用US-ASCII字符集的子集写成的。因此，WMLScript字节码中的函数命名必须采用固定的UTF-8编码。

4. 符号约定

WMLScript字节码就是一个字节序列，这个序列是以二进制方式来表示函数，它含有解释器执行有关函数所需的全部信息。字节码可以分解为片段和子片段，每个片段都包含一个WMLScript逻辑单元。

WMLScript字节码的结构和内容见表6-10：

表6-10 WMLScript 字节码的结构和内容

名 称	数据类型和长度	注 释
字节码中一个片段的名称	由于片段有时不能分解为更小的子片段，所以这里规定了一个为片段预留的数据类型及其长度。子片段的规范在另一个表中给出，并提供访问这个表的索引	给出本片段的概述
下一片段的名称。任意数目的片段都可用一个表格描述		

这里使用下列约定：

- 字节码的片段为表格中一行。
- 每个片段可分成几个子片段并由不同的表格描述，这时会提供一个子片段表格的索引。
- 重复片段使用片段名后加（…）表示。

6.7.2 WMLScript字节码

WMLScript编码有两个主要部分：常量文字和用于描述函数功能的结构。因此，WMLScript字节码包含以下片段（见表6-11）：

表6-11 WMLScript 字节码包含的片段

名 称	数据类型和长度	注 释
HeaderInfo	见6.7.3节	包含与字节码相关的常用信息
ConstantPool	见6.7.4节	包含所有常量的信息，这些常量是 WMLScript编译单元字节码的组成部分
PragmaPool	见6.7.5节	包含与编译指示相关的信息，这些编译指示也是 WMLScript编译单元字节码的组成部分
FunctionPool	见6.7.6节	包含所有与函数编码及其功能相关的信息

下面各节详细描述了这些片段及其子片段的编码。

6.7.3 字节码码头

WMLScript字节码码头包含表6-12所列的一些信息：

表6-12 WMLScript 字节码码头包含的信息

名 称	数据类型和长度	注 释
VersionNumber	byte	WMLScript字节码的版本号。该字节的高 4位表示主版本号减一的数字，低4位是次版本号 如果当前版本为1.0，那么版本号的编码为0x00
CodeSize	mb_u_int32	以字节为单位表示字节码长度(不包括版本号和这个变量)

6.7.4 常量池

常量池包含WMLScript函数要用到的全部常量。每个常量都有一个相对于0的索引号，该索引号由其在常量表中的位置确定。WMLScript指令使用这个索引来访问相应的变量（见表6-13）。

表6-13 常量池

名 称	数据类型和长度	注 释
NumberOfConstants	mb_u_int16	给出常量池中常量的个数
Constants...	见6.7.4节中“常量”	包含常量池中各常量的定义，常量的个数由 NumberOfConstants给出

1. 常量

在字节码中，常量按顺序存放。每个常量的编码由其类型定义开始（整型、浮点型、字符串等等），接着是对应类型的常量数据，表示实际的常量值（见表 6-14）。

表6-14 常量

名 称	数据类型和长度	注 释
ConstantType	u_int8	常量类型
ConstantValue	见6.7.4节中“整型数”、“浮点数”、“字符串”	对应类型的常量值

用到的常量类型编码如表 6-15所示。

表6-15 用到的常量类型编码

代 码	类 型	编 码
0	8比特有符号整数	见6.7.4节中“8比特有符号整数”
1	16比特有符号整数	见6.7.4节中“16比特有符号整数”
2	32比特有符号整数	见6.7.4节中“32比特有符号整数”
3	32比特有符号浮点数	见6.7.4节中“浮点数”
4	UTF－8型字符串	见6.7.4节中“UTF-8型字符串”
5	UCS－2型字符串	见6.7.4节中“UCS-2型字符串”
6	UCS－4型字符串	见6.7.4节中“UCS-4型字符串”
7	空字符串	见6.7.4节中“空字符串”
8-255	备用	

(1) 整型数

WMLScript字节码支持8比特、16比特和32比特有符号整数常量。通过选择保存在整型常量值中的最小整数常量类型，编译器可以对WMLScript字节码的大小进行优化。

1) 8比特有符号整数 8比特有符号整数常量的表达形式如表6-16所示：

表6-16 8比特有符号整数常量的表达形式

名 称	数据类型和长度	注 释
ConstantInteger8	Int8	8比特有符号整型常量值

2) 16比特有符号整数 16比特有符号整数常量的表达形式如6-17所示：

表6-17 16比特有符号整数常量的表达形式

名 称	数据类型和长度	注 释
ConstantInteger16	Int16	16比特有符号整型常量值

3) 32比特有符号整数 32比特有符号整数常量的表达形式如6-18所示：

表6-18 32比特有符号整数常量的表达形式

名 称	数据类型和长度	注 释
ConstantInteger32	Int32	32比特有符号整型常量值

(2) 浮点数

浮点型常量使用32比特的ANSI/IEEE Std 754-1985 [IEEE754]格式（见表6-19）。

表6-19 浮点型常量使用32比特的格式

名 称	数据类型和长度	注 释
ConstantFloat32	Float32	32比特浮点型常量值

(3) 字符串

WMLScript字节码支持多种把Unicode字符串常量 ⊖ 编码转化到常量池中的方法。通过选择保存浮点常量值所需的最小字符串常量类型，编译器可对字节码的大小进行优化。

⊖ 声明：字串常量能包含嵌入的空字符。

1) UTF-8 型字符串 对UTF-8型字符串编码变换成字节码时，首先要给出字符串的长度，然后是内容（见表6-20）。

表6-20 VTF-8型字符串

名 称	数据类型和长度	注 释
StringSizeUTF8	mb_u_int32	后续字符串的字节数（不包括当前变量）
ConstantStringUTF8	StringSizeUTF8 字节	采用UTF-8编码的Unicode字符常量（不以null结尾）的值。有关字符编码的详细说明见6.7.1节中“字符编码”

2) UCS-2 型字符串 对UTF-2型字符串编码变换成字节码时，首先要给出字符串的长度，然后是内容（见表6-21）。

表6-21 UCS-2型字符串

名 称	数据类型和长度	注 释
StringSizeUCS2	mb_u_int32	后续字符串的字节数（不包括当前变量）
ConstantStringUCS2	StringSizeUC2字节	采用UCS－2编码的Unicode字符常量（不以null结尾）的值。有关字符编码的详细说明见6.7.1节中“字符编码”

3) UCS-4 型字符串 对UTF-4型字符串编码变换成字节码时，首先要给出字符串的长度，然后是内容（见表6-22）。

表6-22 UCS-4型字符串

名 称	数据类型和长度	注 释
StringSizeUCS4	mb_u_int32	后续字符串的字节数（不包括当前变量）
ConstantStringUCS4	StringSizeUCS4 字节	采用UCS-4编码的Unicode字符常量（不以空白符结尾）的值。有关字符编码的详细说明见6.7.1节中“字符编码”

4) 空字符串

不需要对空字符串的值进行额外的编码。

6.7.5 编译指示池

编译指示池包含了那些在编译单元中定义的编译指示信息（见表 6-23）。

表6-23 编译指示池

名 称	数据类型和长度	注 释
NumberOfPragmas	mb_u_int16	给出编译指示的个数
Pragmas...	见6.7.5节中“编译指示”	包含编译指示池中各种编译指示的定义。编译指示的个数由 NumberOfPragmas给出

1. 编译指示

在字节码中编译指示按顺序存放。每个编译指示的编码由其类型定义开始，接着是相应的编译指示数据，用于表示实际的编译指示值（见表 6-24）。

表6-24 编译指示

名 称	数据类型和长度	注 释
PragmaType	u_int8	常量类型，后面是标识值
PragmaValue	见6.7.5节中“访问控制编译指示”、“元信息编译指示”	相应类型的标识值

用到的编译指示类型的编码如表 6-25所示。

表6-25 用到的编译指示类型的编码

代 码	类 型	编 码
0	访问控制禁止	见6.7.5节中“访问控制禁止”
1	访问域	见6.7.5节中“访问域”
2	访问路径	见6.7.5节中“访问路径”
3	用户代理性质	见6.7.5节中“用户代理性质”
4	用户代理性质和方案	见6.7.5节中“用户代理性质和方案”
5-255	备用	

(1) 访问控制编译指示

使用三种不同的编译指示类型，可以把访问控制信息编成字节码，这三种类型是：访问控制禁止、访问域和访问路径。对每一种访问控制编译指示类型，编译指示池只包含一个入口。

- 访问控制禁止 这个编译指示表示对编译单元的访问控制被禁止。如果编译指示池含有访问域和访问路径的入口，其值将被忽略，这里无需额外的编码。
- 访问域 这个编译指示给出了用于访问控制的访问域（见表 6-26）。

表6-26 访问域

名 称	数据类型和长度	注 释
AccessDomainIndex	mb_u_int16	常量池索引，指向包含访问域值的字符常量，使用的常量类型必须在4到7之间

- 访问路径 这个编译指示给出了用于访问控制的访问路径（见表 6-27）。

表6-27 访问路径

名 称	数据类型和长度	注 释
AccessPathIndex	mb_u_int16	常量池索引，指向包含访问路径值的字符常量，使用的常量类型必须在4到7之间

(2) 元信息编译指示

这个编译指示含有 WMLScript解释器需要的元信息，这些元信息包括以下一些部分：名称、内容和方案（可选）。

- 用户代理性质 对用户代理性质的编码，首先是指出性质的名称，然后是给出在常量池中的索引值（见表6-28）。

表6-28 用户代理性质

名 称	数据类型和长度	注 释
PropertyNameIndex	mb_u-int16	常量池索引，指向包含性质名称的字符常量，使用的常量类型必须在4到7之间
ContentIndex	mb_u_int16	常量池索引，指向包含资源值的字符常量，使用的常量类型必须在4到7之间

- **用户代理性质和方案** 这个编译指示的编码从其资源名称开始，然后是其数值，最后是附加的需求（见表6-29）。

表6-29 用户代理性质和方案

名 称	数据类型和长度	注 释
PropertyNameIndex	mb_u-int16	常量池索引，指向包含资源名称的字符常量，使用的常量类型必须在4到7之间
ContentIndex	mb_u_int16	常量池索引，指向包含资源值的字符常量，使用的常量类型必须在4到7之间
SchemeIndex	mb_u_int16	常量池索引，指向包含代理需求的字符常量，使用的常量类型必须在4到7之间

6.7.6 函数池

函数池包含所有WMLScript函数的定义。每个函数都有一个从0开始的索引号，由其在常量列表中的位置确定，WMLScript指令使用这个索引来访问相应的函数（见表6-30）。

表6-30 函数池

名 称	数据类型和长度	注 释
NumberOfFunctions	u_int8	给出函数池中函数的个数
FunctionNameTable	见6.7.6节中“函数名表”	函数名表含有所有字节码中出现的外部函数名
Functions...	见6.7.6节中“函数名”	包含各个函数的字节码

1. 函数名表

外部函数的名称存放在函数名表中，函数名的存放必须与其在函数池中的顺序一致。未声明为外部函数的函数名不能出现在函数名表中。函数名表的格式如表6-31所示。

表6-31 函数名表的格式

名 称	数据类型和长度	注 释
NumberOfFunctionNames	U_int8	给出函数名表中函数名的个数
FunctionNames...	见6.7.6节中“函数名”	每个函数名都按其在函数池中的顺序出现

- **函数名** 函数名仅供那些外部函数使用。每个函数名的格式如表6-32所示。

表6-32 函数名的格式

名 称	数据类型和长度	注 释
FunctionIndex	u_int8	函数名将提供给此索引对应的函数
FunctionNameSize	u_int8	函数名的长度（不包括本变量自身的长度）
FunctionName	FunctionNameSize给出的长度	对应的函数名，采用UTF-8编码。有关函数名编码的详细信息见6.7.1节中“字符编码”

2. 函数

每个函数的定义包括起始段和代码段（见表6-33）。

表6-33 函数

名 称	数据类型和长度	注 释
NumberOfArguments	u_int8	函数的输入变量的个数
NumberOfLocalVariables	u_int8	局部变量的个数（不包括输入变量）
FunctionSize	mb_u_int32	函数代码段的字节数（不包括本变量的长度）
CodeArray	见6.7.6节中“代码段”	函数的代码段

- 代码段 代码段含有实现一个WMLScript函数所需的所有指令，第6.8节将给出WMLScript指令集的详细信息（见表6-34）。

表6-34 代码段

名 称	数据类型和长度	注 释
Instructions...	见第6.8节	指令代码

6.7.7 指标限度

字节码最大长度	4294967295比特
常量池中最大常量数	65535
常量类型数的最大值	256
字符常量的最大长度	4294967295比特
URL常量的最大长度	4294967295比特
函数名的最大长度	255
标识类型数的最大值	256
标识集中最大标识数	65536
函数池中最大函数数	255
函数允许的最大输入变量数	255
函数允许的最大局部变量数	255
函数参数和局部变量个数的最大值	256
WMLScript库的最大个数	65536
WMLScript库中函数个数的最大值	256

6.8 WMLScript指令集

WMLScript指令集中的指令都是汇编语句，用于对 WMLScript语言结构和行为的编码，这些指令在许多平台上都能很容易的高效执行。

6.8.1 约定规则

表6-35给出了WMLScript解释器的转换规则：

表6-35 WMLScript 解释器的转换规则

规则—操作数类型	转 换 规 则
1—布尔型	见6.4.2节中有关布尔变量的转换规则
2—整型	见6.4.2节中有关整型变量的转换规则
3—浮点型	见6.4.2节中有关浮点变量的转换规则
4—字符型	见6.4.2节中有关字符变量的转换规则
5—整型或浮点型（单目）	见6.4.2节中有关整型或浮点（一元）变量的转换规则
6—整型或浮点型	见6.4.2节中有关整型或浮点变量的转换规则
7—整型、浮点型或字符型	见6.4.2节中有关整型，浮点或字符变量的转换规则
8—任意	见6.4.2节中有关操作变量的转换规则

6.8.2 致命的错误

表6-36中给出了WMLScript解释器的一些致命错误：

表6-36 WMLScript 解释器的致命错误

错 误 代 码	致 命 错 误
1 （Verification Failed）	详细说明见6.10.3节中“效验失败”
2 (Fatal Library Function Error)	详细说明见6.10.3节中“致命库函数错”
3 (Invalid Function Arguments)	详细说明见6.10.3节中“函数参数错”
4 (External Function Not Found)	详细说明见6.10.3节中“未找到外部函数”
5 (Unable to Load Compilation Unit)	详细说明见6.10.3节中“无法载入编译单元”
6 (Access Violation)	详细说明见6.10.3节中“访问错”
7 (Stack Underflow)	详细说明见6.10.3节中“堆栈下溢”
8 (Programmed Abort)	详细说明见6.10.3节中“程序异常中止”
9 (Stack Overflow)	详细说明见6.10.3节中“堆栈上溢”
10 (Out of Memory)	详细说明见6.10.3节中“存储器溢出”
11 (User Initiated)	详细说明见6.10.3节中“用户中断”
12 (System Initiated)	详细说明见6.10.3节中“系统中断”

6.8.3 优化

WMLScript指令集是实现WMLScript语言功能所需的最小指令集。由于从网关到客户端的字节码传输是窄带的，因此所用指令都经过优化使得编译器能生成长度最小的代码。有时，这意味着一些带不同参数的指令被用来执行同一个操作，编译器则需要选择最优的那个代码。

内联参数用来优化包信息，使其字节数尽可能小。表 6-37所列是一些内联优化参数：

表6-37 内联优化参数

参数符号	可用指令数	使用场合
1xxppppp	4	JUMP_FW_S,JUMP_BW_S,TJUMP_FW_S,LOAD_VAR_S
010xpppp	2	STORE_VAR_S,LOAD_CONST_S
011xpppp	4	CALL_S,CALL_LIB_S,INCR_VAR_S
00xxxxxx	63	其他指令

6.8.4 符号约定

后面的一节将具体介绍WMLScript指令，对每一指令，需给出以下信息：

- 指令 指令及其参数的符号名。
- 操作 码8比特的指令代码。
- 参数 参数的范围和含义。有些指令可以把操作码的一部分当作隐含参数，也就是说，某些比特将用来存放参数值 。
- 操作 描述指令的操作、参数及其它们对程序运行和操作数堆栈的影响。
- 操作数 描述指令所需的操作数个数及允许的操作数类型。
- 转换 给出已使用的转换规则（见6.8.1节）。
- 结果 给出结果及其类型。
- 操作数堆栈 描述指令对操作数堆栈的影响。所使用的定位符号为：‘ => ’前面的部分是指令执行前的堆栈，‘ => ’后面则代表指令执行后的堆栈。
- 错误 给出指令执行中可能会出现的致命错误（见6.8.2节）。

除了控制流转跳指令之外，所有指令执行结束后都顺序执行下一条指令。跳转指令则会给出下一条待执行指令的地址。

那些随时都可能发生的致命错可以由任何指令引起（见6.10.3节“ 外部异常 ”和“ 内存耗尽错误 ”）。

指令的结果可能是无效型值，但并非每一指令都明确指出这一点，而把它当成使用转换规则的结果，这可能是由于引入了非法或不支持的浮点常量，或者是无效型操作数的执行结果。

6.8.5 指令

以下各小节将具体描述每个指令。

1. 控制流指令

指令 JUMP_FW_S

操作码 100iiiii(*iiiii*是隐含的无符号偏移量offset)。

参数 偏移量offset是一个5比特无符号整数，取值范围为0到31。

操作 向前跳到一个偏移量 *offset*，从给定的偏移量 *offset*继续执行，偏移量从这一条指令的第一个字节算起。

操作数 —

转换类型 —

结果 —

操作数堆栈 不改变
错误 1（效验失败）

指令 JUMP_FW *offset*
操作码 00000001
参数 *offset*是一个8比特无符号整数，取值范围为0－255。
操作 向前跳到一个偏移量 *offset*，从给定的偏移量 *offset*继续执行，偏移量从这一条指令的第一个字节算起。
操作数 —
转换类型 —
操作数堆栈 不改变。
错误 1（效验失败）。

指令 JUMP_FW_W <*offset1*,*offset2*>
操作码 00000010
参数 *offset*是一个16比特无符号整数<*offset1*,*offset2*>，取值范围为0到65535。
操作 向前跳到一个偏移量 *offset*，从给定的偏移量 *offset*继续执行，偏移量从这一条指令的第一个字节算起。
操作数 —
转换类型 —
结果 —
操作数堆栈 不改变。
错误 1（效验失败）。

指令 JUMP_BW_S
操作码 101iiiii(*iiiii*是隐含的无符号偏移量 *offset*)。
参数 *offset*是一个5比特无符号整数，取值范围为0到31。
操作 向后跳到偏移量 *offset*，从给定的偏移量 *offset*继续执行。偏移量从这个指令的地址算起
操作数 —
转换类型 —
结果 —
操作数堆栈 不改变。
错误 1（效验失败）。

指令 JUMP_BW *offset*
操作码 00000011
参数 *offset*是一个8比特无符号整数，取值范围为0到255。
操作 向后跳到偏移量 *offset*，从给定的偏移量 *offset*继续执行。偏移量从这个指令的地址算起
操作数 —
转换类型 —

结果	**—**
操作数堆栈	**不改变。**
错误	1**（效验失败）。**
指令	JUMP_BW_W <*offset1*,*offset2*>
操作码	00000100
参数	*offset***是一个**16**比特无符号整数**<*offset1*,*offset2*>**，取值范围为**0**到**65535**。**
操作	**向后跳到偏移量** *offset***，从给定的偏移量** *offset***继续执行。偏移量从这个指令的地址算起**
操作数	**—**
转换类型	**—**
结果	**—**
操作数堆栈	**不改变。**
错误	1**（效验失败）。**
指令	TJUMP_FW_S
操作码	110iiiii(*iiiii***是隐含的无符号偏移量** *offset*)。
参数	*offset***是一个**5**比特无符号整数，取值范围为** 0**到**31**。**
操作	**从操作数堆栈中弹出一个值。如果弹出值错误或无效，则向前跳到偏移量** *offset***，从给定的偏移量** *offset***继续执行。偏移量从这个指令的第一个字节算起。否则，顺序执行下一条指令。**
操作数	**布尔型**
转换类型	1**—布尔型**
结果	**—**
操作数堆栈	...,**变量值**=>...
错误	1**（效验失败）**,7**（堆栈下溢）。**
指令	TJUMP_FW *offset*
操作码	00000101
参数	*offset***是一个**8**比特无符号整数，取值范围为** 0**到**255**。**
操作	**从操作数堆栈中弹出一个值。如果弹出值错误或无效，则向前跳到偏移量** *offset***，从给定的偏移量** *offset***继续执行。偏移量从这个指令的第一个字节算起，否则，顺序执行下一条指令。**
操作数	**布尔型**
转换类型	1**—布尔型**
结果	**—**
操作数堆栈	..., value =>...
错误	1**（效验失败），**7**（堆栈下溢）。**
指令	TJUMP_FW_W <*offset1*,*offset2*>
操作码	00000110
参数	*offset***是一个**16**比特无符号整数**<*offset1*,*offset2*>**，取值范围为**0**到**65535**。**

操作	**从操作数堆栈中弹出一个值。如果弹出值错误或无效，则向前跳到偏移量 *offset*，从给定的偏移量 *offset*继续执行。偏移量从这个指令的第一个字节算起，否则，顺序执行下一条指令。**
操作数	**布尔型**
转换类型	1-**布尔型**
结果	**—**
操作数堆栈	..., value =>...
错误	1**（效验失败）**，7**（堆栈下溢）。**

指令	TJUMP_BW *offset*
操作码	00000111
参数	*offset***是一个**8**比特无符号整数，取值范围为** 0**到**255**。**
操作	**从操作数堆栈中弹出一个值。如果弹出值错误或无效，则向前跳到偏移量 *offset*，从给定的偏移量 *offset*继续执行。偏移量从这个指令的第一个字节算起，否则，顺序执行下一条指令。**
操作数	**布尔型**
转换类型	1-**布尔型**
结果	**—**
操作数堆栈	..., value =>...
错误	1**（效验失败）**，7**（堆栈下溢）。**

指令	TJUMP_BW_W <*offset1*,*offset2*>
操作码	00001000
参数	*offset***是一个**16**比特无符号整数**<*offset1*,*offset2*>**，取值范围为**0**到**65535**。**
操作	**从操作数堆栈中弹出一个值。如果弹出值错误或无效，则向前跳到偏移量 *offset*，从给定的偏移量 *offset*继续执行。偏移量从这个指令的第一个字节算起，否则，顺序执行下一条指令。**
操作数	**布尔型**
转换类型	1-**布尔型**
结果	**—**
操作数堆栈	..., value =>...
错误	1**（效验失败）**，7**（堆栈下溢）。**

2. 函数调用指令

指令	CALL_S
操作码	01100iii(*iii***是一个隐含的待调函数** *findex*)。
参数	*findex***是一个**3**比特无符号整数，取值范围为** 0**到**7**。**
操作	**调用在同一函数池中定义的局部函数，从被调函数 *findex*的第一条指令继续执行。**
操作数	**任意类型的变量。**
转换类型	**—**
结果	**任意（被调函数的返回值）。**

操作数堆栈	...[agr1,[agr2...]]=>...,**返回值。**
错误	1（**效验失败**），7（**堆栈下溢**）。
指令	CALL *findex*
操作码	00001001
参数	*findex***是一个**8**比特无符号整数，取值范围为**0**到**255。
操作	**调用在同一函数池中定义的局部函数，从被调函数** *findex***的第一条指令继续执行。**
操作数	**任意类型的变量。**
转换类型	—
结果	**任意（被调函数的返回值）。**
操作数堆栈	...[arg1,[arg 2...]]=>...,**返回值。**
错误	1（**效验失败**），7（**堆栈下溢**）。
指令	CALL_LIB_S *lindex*
操作码	01101iii(*iii***是一个隐含的待调函数** *findex*)。
参数	*findex***是一个**3**比特无符号整数，取值范围为**0**到**7。 *lindex***是一个**8**比特无符号整数，取值范围为**0**到**255
操作	**调用在** *lindex***库中定义的库函数** *findex*。
操作数	**任意类型的变量（由被调库函数确定）。**
转换类型	—
结果	**无限制（被调函数的返回值）。**
操作数堆栈	...[arg1,[arg 2...]]=>...,**返回值。**
错误	1（**效验失败**），2（**致命库函数错**），7（**堆栈下溢**），8（**程序异常中止**）。
指令	CALL_LIB *findex lindex*
操作码	00001010
参数	*findex***是一个**8**比特无符号整数，取值范围为**0**到**255。 *lindex***是一个**8**比特无符号整数，取值范围为**0**到**255。
操作	**调用在**lindex**库中定义的库函数**findex。
操作数	**任意类型的变量（由被调库函数确定）。**
转换类型	—
结果	**无限制（被调函数的返回值）。**
操作数堆栈	...[arg1,[arg 2...]]=>...,**返回值。**
错误	1（**效验失败**），2（**致命库函数错**），7（**堆栈下溢**），8（**程序异常中止**）。
指令	CALL_LIB_W *findex*<*lindex*1,*lindex*2>
操作码	00001011
参数	*findex***是一个**8**比特无符号整数，取值范围为**0**到**255。 *findex***是一个**16**比特无符号整数**<*lindex*1,*lindex*2>**，取值范围为**0**到**65535。
操作	**调用在** *lindex***库中定义的库函数** *findex*。
操作数	**任意类型的变量（由被调库函数确定）。**
转换类型	—

结果　无限制（被调函数的返回值）。
操作数堆栈　...[arg1,[arg 2...]]=>...,返回值。
错误　1（效验失败），2（致命库函数错），7（堆栈下溢），8（程序异常中止）。

指令　CALL_URL *urlindex findex args*
操作码　00001100
参数　*Urlindex*是一个8比特无符号整数，取值范围为 0到255，必须指向含有一个有效URL的常量池，所用的常量类型必须在 4到7之间。
*Findex*是一个8比特无符号整数，取值范围为 0到255，必须指向含有一个有效函数名的常量池，所用的常量类型必须为 4。
Args 是一个8比特无符号整数，取值范围为 0到255，必须含有被压入操作数堆栈的函数参数的个数。
操作　调用定义在URL地址urlindex中由fndex规定的函数。
操作数　任意类型的变量（由args确定）。
转换类型　—
结果　无限制（被调函数的返回值）。
操作数堆栈　...[arg1,[arg 2...]]=>...,返回值。
错误　1（效验失败），3（非法函数参数），4（未找到外部函数），5（无法载入编译单元），6（访问无效），7（堆栈上溢）。

指令　CALL_URL_W < *urlindex*1,*urindex*2>< *findex*1,*findex*2> *args*
操作码　00001101
参数　*Urlindex*是一个16比特无符号整数<*urlindex*1,*urlindex*2>，取值范围为0到65535,必须指向含有一个有效URL的常量池，所用的常量类型必须在4到7之间。
*Findex*是一个16比特无符号整数<*findex*1,*findex*2>，取值范围为0到65535,必须指向含有一个有效函数名的常量池，所用的常量类型必须为 4。
Args 是一个无符号整数，取值范围为 0到255，必须含有被压入操作数堆栈的函数参数的个数。
操作　调用定义在URL地址*urlindex*中由*findex*规定的函数。
操作数　任意类型的变量（由args确定）。
转换类型　—
结果　无限制（被调函数的返回值）。
操作数堆栈　...[arg1,[arg 2...]]=>...,返回值。
错误　1（效验失败），3（非法函数参数），4（未找到外部函数），5（无法载入编译单元），6（访问无效），7（堆栈上溢）。

3. 变量的访问和操作

指令　LOAD_VAR_S
操作码　111iiiii(*iiiii*是隐含变量，表示被操作变量 *vindex*)
参数　*Vindex*是一个5比特无符号整数，取值范围为 0到31。
操作　把*vindex*的值压入操作数堆栈
操作数　—

转换类型	—
结果	无限制（变量的值）。
操作数堆栈	...=>...,变量值。
错误	1（效验失败）

指令	LOAD_VAR *vindex*
操作码	00001110
参数	*Vindex*是一8比特无符号整数，取值范围为0到255。
操作	把*vindex*的值压入操作数堆栈。
操作数	—
转换类型	—
结果	无限制（变量的值）
操作数堆栈	...=>...,变量值。
可能的错误	1（效验失败）

指令	STORE_VAR_S
操作码	0100iiii(*iiii*是隐含变量，表示被操作变量 *vindex*)。
参数	Vindex是一个4比特无符号整数，取值范围为0到15。
操作	从操作数堆栈中弹出一个值并存在 vindex中。
操作数	任意
转换类型	8－任意
结果	—
操作数堆栈	...,变量值=>...
可能的错误	1（效验失败）

指令	STORE_VAR *vindex*
操作码	00001111
参数	*Vindex*是一个8比特无符号整数，取值范围为0到255
操作	从操作数堆栈中弹出一个值并存在 *vindex*中。
操作数	任意
转换类型	8 － 任意
结果	—
操作数堆栈	...,变量值=>...
可能的错误	1（效验失败）

指令	INCR_VAR_S
操作码	01110iii(*iii*是隐含变量，表示被操作变量 *vindex*)。
参数	*Vindex*是一3比特无符号整数，取值范围为0到7。
操作	*vindex*加1。
操作数	—
转换类型	5－整型或浮点型（单目）

结果 —
操作数堆栈 不变
可能的错误 1（效验失败）

指令 INCR_VAR *vindex*
操作码 00010000
参数 *Vindex*是一个8比特无符号整数，取值范围为0到255。
操作 *vindex*加1。
操作数 —
转换类型 5—整型或浮点型（单目）
结果 —
操作数堆栈 不变
可能的错误 1（效验失败）

指令 DECR_VAR vindex
操作码 00010001
操作 *vindex*减1。
参数 *Vindex*是一个8比特无符号整数，取值范围为0到255。
操作数 —
转换类型 5—整型或浮点型（单目）
结果 —
操作数堆栈 不变
可能的错误 1（效验失败）

4. 常量访问

指令 LOAD_CONST_S
操作码 01010iii(*iii*是隐含变量，表示被操作常量的索引 *cindex*)。
参数 *Cindex*是一4比特无符号整数，取值范围为0到15,指向含有非空常量的常量池，使用的常量类型必须在0到7之间。
操作 把cindex对应的常量值压入操作数堆栈
操作数 —
转换类型 —
结果 任意（常量的内容）
操作数堆栈 ...=>...,常量值
可能的错误 1（效验失败）

指令 LOAD_CONST cindex
操作码 00010010
参数 *Cindex*是一个8比特无符号整数，取值范围为0到255，指向含有非空常量的常量池，使用的常量类型必须在0到7之间。

操作	把cindex对应的常量值压入操作数堆栈。
操作数	—
转换类型	—
结果	任意（常量的内容）
操作数堆栈	...=>...,常量值
可能的错误	1（效验失败）

指令	LOAD_CONST_W <*cindex*1,*cindex*2>
操作码	00010011
参数	*Cindex*是一个16比特无符号整数 <*cindex*1,*cindex*2>，取值范围为0到65535，指向含有非空常量的常量池，使用的常量类型必须在0到7之间。
操作	把*cindex*对应的常量值压入操作数堆栈。
操作数	—
转换类型	—
结果	任意（常量的内容）
操作数堆栈	...=>...,常量值
可能的错误	1（效验失败）

指令	CONST_0
操作码	00010100
参数	—
操作	把0压入操作数堆栈
操作数	—
转换类型	—
结果	整数
操作数堆栈	...=>...,常量值0
错误	—

指令	CONST_1
操作码	00010101
参数	—
操作	把整数值1压入操作数堆栈。
操作数	—
转换类型	—
结果	整数
操作数堆栈	...=>...,常量值1
错误	—

指令	CONST_M1
操作码	00010110
参数	—

操作 **把－1压入操作数堆栈。**
操作数 **—**
转换类型 **—**
结果 **整数**
操作数堆栈 ...=>...,**常量值－1**
错误 **—**

指令 CONST_ES
操作码 00010111
参数 **—**
操作 **把空字符串压入操作数堆栈。**
操作数 **—**
转换类型 **—**
结果 **字符串**
操作数堆栈 ...=>...,**常量值_""**
错误 **—**

指令 CONST_INVALID
操作码 00011000
参数 **—**
操作 **把一无效值压入操作数堆栈。**
操作数 **—**
转换类型 **—**
结果 **无效**
操作数堆栈 ...=>...,**无效**
错误 **—**
指令 CONST_TRUE
操作码 00011001
参数 **—**
操作 **把布尔真压入操作数堆栈。**
操作数 **—**
转换类型 **—**
结果 **布尔值**
操作数堆栈 ...=>...,**布尔真**
错误 **—**

指令 CONST_FLASE
操作码 00011010
参数 **—**
操作 **把布尔假压入操作数堆栈。**
操作数 **—**

转换类型 —
结果 布尔值
操作数堆栈 ...=>...,布尔假
错误 —

5. 算术指令

指令 INCR
操作码 00011011
参数 —
操作 栈顶数值加一
操作数 整型或浮点型
转换类型 5－整型或浮点型（单目）
结果 整型或浮点型（值加1）
操作数堆栈 ...，变量值=>...,变量值＋1
错误 7（堆栈下溢）

指令 DECR
操作码 0011100
参数 —
操作 栈顶数值减一
操作数 整型或浮点型
转换类型 5－整型或浮点型（单目）
结果 整型或浮点型（值减1）
操作数堆栈 ...，变量值=>...,变量值－1
错误 7（堆栈下溢）

指令 ADD_ASG *vindex*
操作码 00011101
参数 *Vindex*是一个8比特无符号整数，取值范围为0到255
操作 弹出栈顶值并加到*vindex*中。
操作数 整型，浮点型，或字符型
转换类型 7－整型、浮点型或字符型
结果 对整型或浮点型：相加后的*vindex*。
对字符型：两串首尾相接，存于*vindex*中。
操作数堆栈 ...，变量值=>...,
错误 1（效验失败），7（堆栈下溢）

指令 SUB_ASG *vindex*
操作码 00011110
参数 *Vindex*是一个8比特无符号整数，取值范围为0到255。
操作 从*vindex*中减去操作数栈顶弹出的值。

操作数	整型或浮点型
转换类型	6－整型或浮点型
结果	相减后的变量
操作数堆栈	...，变量值=>...,
错误	1（效验失败），7（堆栈下溢）

指令	UMINUS
操作码	00011111
参数	—
操作	弹出操作数栈顶值，将其取反后再压回操作数栈。
操作数	整型或浮点型
转换类型	5－整型或浮点型（单目）
结果	整型或浮点型（取反）
操作数堆栈	...，变量值=>...,变量值取反。
错误	7（堆栈下溢）

指令	ADD
操作码	00100000
参数	—
操作	从操作数栈弹出两个值，相加后再压回操作数栈。
操作数	整型，浮点型，或字符型。
转换类型	7－整型，浮点型或字符型。
结果	对整型或浮点型：相加后的和。 对字符型：两串首尾相接生成的串。
操作数堆栈	...，变量1，变量2=>...,变量1＋变量2。
错误	7（堆栈下溢）

指令	SUB
操作码	00100001
参数	—
操作	从操作数栈弹出两个值，相减后再压回操作数栈。
操作数	整型或浮点型。
转换类型	6－整型或浮点型。
结果	整型或浮点型。
操作数堆栈	...，变量1，变量2=>...,变量1－变量2。
错误	7（堆栈下溢）

指令	MUL
操作码	00100010
参数	—
操作	从操作数栈弹出两个值，相乘后再压回操作数栈。

操作数	**整型或浮点型**
转换类型	6－**整型或浮点型**
结果	**整型或浮点型**
操作数堆栈	...，**变量**1，**变量**2=>...,**变量**1×**变量**2。
错误	7（**堆栈下溢**）

指令	DIV
操作码	00100011
参数	**—**
操作	**从操作数栈弹出两个值，相除后再压回操作数栈。**
操作数	**整型或浮点型**
转换类型	6－**整型或浮点型**
结果	**整型或浮点型**
操作数堆栈	...，**变量**1，**变量**2=>...,**变量**1÷**变量**2。
错误	7（**堆栈下溢**）

指令	IDIV
操作码	00100100
参数	**—**
操作	**从操作数栈弹出两个值，相除取整后再压回操作数栈。**
操作数	**整型**
转换类型	2－**整型**
结果	**整型**
操作数堆栈	...，**变量**1，**变量**2=>...,**变量**1÷**变量**2**后取整。**
错误	7（**堆栈下溢**）

指令	REM
操作码	00100101
参数	**—**
操作	**从操作数栈弹出两个值，相除取余后再压回操作数栈（结果的符号与被除数一致）。**
操作数	**整型**
转换类型	6－**整型或浮点型**
结果	**整型**
操作数堆栈	...，**变量**1，**变量**2=>...,**变量**1%**变量**2。
错误	7（**堆栈下溢**）

6. 位操作指令

指令	B_AND
操作码	00100110

参数	**—**
操作	**从操作数栈弹出两个值，按位相与后再压回操作数栈。**
操作数	**整型**
转换类型	**2—整型**
结果	**整型**
操作数堆栈	**...，变量1，变量2=>...,变量1&变量2。**
错误	**7（堆栈下溢）**

指令	B_OR
操作码	00100111
参数	**—**
操作	**从操作数栈弹出两个值，按位相或后再压回操作数栈。**
操作数	**整型**
转换类型	**2—整型**
结果	**整型**
操作数堆栈	**...，变量1，变量2=>...,变量1 | 变量2。**
错误	**7（堆栈下溢）**

指令	B_XOR
操作码	00101000
参数	**—**
操作	**从操作数栈弹出两个值，按位异或后再压回操作数栈。**
操作数	**整型**
转换类型	**2—整型**
结果	**整型**
操作数堆栈	**...，变量1，变量2=>...,变量1^变量2。**
错误	**7（堆栈下溢）**

指令	B_NOT
操作码	00101001
参数	**—**
操作	**弹出操作数栈顶值，按位取反后再压回操作数栈。**
操作数	**整型**
转换类型	**2—整型**
结果	**整型**
操作数堆栈	**...，变量值=>...,～变量值**
错误	**7（堆栈下溢）**

指令	B_LSHIFT

操作码	00101010
参数	**—**
操作	**从操作数栈弹出两个值，将第一变量按位左移第二变量表示的位数后再压回操作数栈。**
操作数	**整型**
转换类型	2—**整型**
结果	**整型**
操作数堆栈	…，**变量**1，**位移量**=>…,**变量**1<<**位移量。**
错误	7（**堆栈下溢**）

指令	B_RSSHIFT
操作码	00101011
参数	**—**
操作	**从操作数栈弹出两个值，将第一变量（首位为符号位）按位右移第二变量表示的位数后再压回操作数栈。**
操作数	**整型**
转换类型	2—**整型**
结果	**整型**
操作数堆栈	…，**变量**1，**位移量**=>…,**变量**1>>**位移量。**
错误	7（**堆栈下溢**）

指令	B_RSZSHIFT
操作码	00101100
参数	**—**
操作	**从操作数栈弹出两个值，将第一变量带** 0**按位右移第二变量表示的位数后再压回操作数栈。**
操作数	**整型**
转换类型	2—**整型**
结果	**整型**
操作数堆栈	…，**变量**1，**位移量**=>…,**变量**1>>>**位移量。**
错误	7（**堆栈下溢**）

7. 比较指令

指令	EQ
操作码	00101101
参数	**—**
操作	**从操作数栈弹出两个值，看变量** 1**是否等于变量**2，**比较结果压回堆栈。**
操作数	**整型，浮点型或字符型**
转换类型	7—**整型，浮点型或字符型**
结果	**布尔值**

操作数堆栈	...，变量1，变量2=>...,变量1 EQ变量2。
错误	7（堆栈下溢）

指令	LE
操作码	00101110
参数	—
操作	从操作数栈弹出两个值，看变量1是否不小于变量2，比较结果压回堆栈。
操作数	整型，浮点型或字符型
转换类型	7—整型，浮点型或字符型
结果	布尔值
操作数堆栈	...，变量1，变量2=>...,变量1 LE变量2
错误	7（堆栈下溢）

指令	LT
操作码	00101111
参数	—
操作	从操作数栈弹出两个值看，变量1是否大于变量2，比较结果压回堆栈。
操作数	整型，浮点型或字符型
转换类型	7—整型，浮点型或字符型
结果	布尔值
操作数堆栈	...，变量1，变量2=>...,变量1LT变量2
错误	7（堆栈下溢）

指令	GE
操作码	00110000
参数	—
操作	从操作数栈弹出两个值，看变量1是否转义变量2，比较结果压回堆栈。
操作数	整型，浮点型或字符型
转换类型	7—整型，浮点型或字符型
结果	布尔值
操作数堆栈	...，变量1，变量2=>...,变量1 GE变量2
错误	7（堆栈下溢）

指令	GT
操作码	00110001
参数	—
操作	从操作数栈弹出两个值，看变量1是否转义变量2，比较结果压回堆栈。
操作数	整型，浮点型或字符型
转换类型	7—整型，浮点型或字符型

结果	**布尔值**
操作数堆栈	...，变量1，变量2=>...,变量1 GT变量2
错误	7（堆栈下溢）

指令	NE
操作码	00110010
参数	**—**
操作	**从操作数栈弹出两个值，看变量1是否不等于变量2，比较结果压回堆栈。**
操作数	**整型，浮点型或字符型**
转换类型	7—整型，浮点型或字符型
结果	**布尔值**
操作数堆栈	...，变量1，变量2=>...,变量1 NE变量2
错误	7（堆栈下溢）

8. 逻辑指令

指令	NOT
操作码	00110011
参数	**—**
操作	**弹出操作数栈顶值，将其逻辑取反后压回堆栈。**
操作数	**布尔型**
转换类型	1—布尔型
结果	**布尔型**
操作数堆栈	...，变量值=>...,！变量值
错误	7（堆栈下溢）

指令	SCAND
操作码	00110100
参数	**—**
操作	**弹出操作数栈顶值，将其转换成布尔值，如果结果为假或无效，则把转换值压入操作数栈，再把布尔假压栈。反之，则只把转换值压栈。**
操作数	**无限制**
转换类型	1—布尔型
结果	**布尔型**
操作数堆栈	...，变量值=>...，false，false (如果转换值为假) ...，变量值=>...，true (如果转换值为真) ...，变量值=>...，invalid，false (如果转换值为无效)
错误	7（堆栈下溢）

指令	SCOR
操作码	00110101

参数 —
操作 弹出操作数栈顶值，将其转换成布尔值，如果结果为无效，则把布尔真压入操作数栈。反之，则把转换值压栈，再把布尔假压栈。
操作数 无限制
转换类型 1—布尔型
结果 布尔型
操作数堆栈 ...，变量值=>...，true (如果转换值为假)
...，变量值=>...，true，false(如果转换值为真)
...，变量值=>...，invalid，false (如果转换值为无效)
错误 7（堆栈下溢）

指令 TOBOOL
操作码 00110110
参数 —
操作 弹出操作数栈顶值，将其转换成布尔值，并压回操作数栈，如果弹出值为无效，则把‘无效’压栈。
操作数 无限制
转换类型 1—布尔型
结果 布尔型
操作数堆栈 ...，变量值=>...,
错误 7（堆栈下溢）

9. 堆栈指令

指令 POP
操作码 00110111
参数 —
操作 从操作数栈顶弹出一个值
操作数 无限制
转换类型 —
结果 —
操作数堆栈 ...，变量值=>...
错误 7（堆栈下溢）

10. 操作数类型访问

指令 TYPEOF
操作码 00111000
参数 —
操作 弹出操作数栈顶值，并检查其类型，将结果以整数压栈。可能的结果为：0=整型，1=浮点型，2=字符型，3=布尔型，4=无效。
操作数 无限制
转换类型 —

结果 整型
操作数堆栈 ...，变量值=>...,变量类型
错误 7（堆栈下溢）

指令 ISVALID
操作码 00111001
参数 —
操作 弹出操作数栈顶值，并检查其类型，如果为无效，则将布尔假压栈。反之，则把布尔真压栈。
操作数 无限制
转换类型 —
结果 布尔型
操作数堆栈 ...，变量值=>...,是否有效？
错误 7（堆栈下溢）

11. 函数返回指令

指令 RETURN
操作码 00111010
参数 —
操作 弹出操作数栈顶的返回值，将其返回给调用者，栈中其他值将被丢弃，然后从调用函数的下一指令继续执行。
操作数 无限制
转换类型 —
结果 —
操作数堆栈 ...，返回值=>...
错误 7（堆栈下溢）

指令 RETURN_ES
操作码 00111011
参数 —
操作 返回一个空字符串给调用者，栈中其他值将被丢弃，然后从调用函数的下一指令继续执行。
操作数 —
转换类型 —
结果 —
操作数堆栈 不变
错误 —

12. 混合指令

指令 DEBUG
操作码 00111100

参数	—
操作	无操作，保留作调试和描述用
操作数	—
转换类型	—
结果	—
操作数堆栈	不变
错误	—

6.9 字节码校验

字节码校验是在执行前或执行中进行的，其目的是验证程序代码是否符合 WMLScript字节码规范。在校验失败时，这个字节码不被执行，或是执行立即中止并向解释器发送失败信息。

以下所述就是在WMLScript解释器中需进行的效验。

6.9.1 完整性检查

在字节码执行前必须完成下列项目，以检查代码的完整性：

1) 检查版本号是否正确：必须把字节码的版本号与解释器支持的版本号相比较，其中主版本号必须匹配，次版本号则不能超过解释器所支持的次版本号。

2) 检查代码长度是否正确：字节码给出的代码长度必须与实际代码长度完全一致。

3) 检查常量池：

- 检查常量个数是否正确：常量池给出的个数必须与实际存储的常量个数一致。
- 检查常量类型是否有效：常量池给出的常量类型数必须与可支持的类型数匹配，如果使用了保留类型（8～255）将会导致校验失败。
- 检查常量长度是否正确：每个常量只能占用 WMLScript字节码规范规定的字节数（如有固定长度的整型常量），或是常量本身所带长度参数给出的字节数（如长度可变的字符常量）。

4) 检查编译指示池：

- 检查编译指示个数是否正确：编译指示池给出的个数必须与实际存储的标识个数一致。
- 检查编译指示类型是否有效：编译指示池给出的标识类型数必须与可支持的类型数匹配，如果使用了保留类型（5～255）将会导致校验失败。
- 检查编译指示池索引是否有效：
- 访问控制域和访问路径必须指向字符常量。
- 元信息标识中的常量池索引必须指向字符常量。

5) 检查函数池：

- 检查函数个数是否正确：函数池给出的个数必须与实际存储的函数个数一致。
- 检查函数名表是否正确：
- 检查函数名个数是否正确：函数名表给出的个数必须与实际存储的函数名个数一致。
- 检查函数名索引是否正确：此索引必须指向函数池中存在的函数。

- 检查函数名是否仅含有有效字符：函数名必须符合 WMLScript函数命名规则。
- 检查函数起始段是否正确：
- 检查参数和局部变量的个数是否正确：参数和局部变量的总个数不能超过 256。
- 检查函数长度是否正确：函数起始段给出的函数长度必须与实际的函数字节数完全一致。

6.9.2 运行有效性检查

在代码执行中需要检查下面的项目，用来验证所用指令及他们的参数值是否有效。

1) 检查字节码是否仅含有效指令：只有第 8章定义的指令是有效指令。

2) 检查局部变量引用是否有效：这种引用不能超过函数起始段给出的局部变量数上限。

3) 检查常量引用是否有效：

- 常量引用不能超过常量池中常量的个数 。
- 必须是每个指令允许的常量类型。
- 在URL引用中，被引用的字符常量必须含有一个有效的 URL（见[RFC1808]）。
- 在函数名引用中，被引用的字符常量必须含有一个有效的 WMLScript函数名。

4) 检查标准库索引和库函数索引是否有效：此索引不能超过 WMLScript标准库规范[WMLSLibs]规定的上限。

5) 检查局部函数调用索引是否有效：函数索引必须与函数池中规定的函数个数匹配。

6) 检查跳转操作是否在函数边界内：所有跳转都必须在定义它们的函数中有一个目的地。

7) 检查跳转目的地是否有效：所有跳转的目的地必须是一个指令的开始。

8) 检查函数结尾是否有效：函数不能于一条指令中结束。

6.10 运行错误检测和处理

WMLScript函数是用来给那些希望在任何情况下终端（这里指移动电话）都能正常工作的用户提供服务的，因此错误处理就显得异常重要。这意味着如果 WMLScript语言不提供异常处理机制，就应该提供一种工具来防止错误发生或给出错误提示并采取适当的措施。程序异常中止应只在没有其他办法时使用。

下节给出了在下载字节码和执行时会出现的错误，它不包括编程错（如：死循环等），因此，这里需要一个可由用户控制的终止机制。

6.10.1 错误检测

错误检测的目的是给编程者提供一个工具检测那些会导致不正常行为的错误（如果可能）。由于WMLScript是一种弱类型语言，所以它提供了一种特殊机制来检测由无效数据类型引起的错误。

- 检查给定的变量是否含有正确的数值： WMLScript支持类型确认库函数 [WMLSLibs]，如: Lang.isInt( )、Lang.isFloat( )、Lang.parseInt( )和Lang.parseFloat( )。
- 检查给定变量是否含有一个正确的类型值。 WMLScript支持typeof和isvalid操作符，可用于这种检验。

6.10.2 错误处理

错误处理是在错误已发生后进行，它的出现是由于这个错误无法用错误检测来避免（如存储器限制、扩展信号错误等），或是由处理过程太复杂（上溢、下溢等），这种情况可分为以下两类：

致命错 是会导致程序异常终止的错误。由于WMLScript函数总是被其他一些用户代理调用，程序终止应通知用户代理调用者。因此，有必要采取恰当的措施将错误通知用户。

非致命错 是可作为特殊返回值通知给程序的错误，并由调用程序自己决定采取那些措施。

按照不同的严重程度，下面分节对错误进行了描述。

6.10.3 致命错误

1. 字节码错

这些错误与字节码及正在被WMLScript字节码解释器执行的指令有关。它们之所以出现，有下列原因：常量池元素错、无效指令、无效参数或无法完成的指令。

(1) **校验失败**

描述 报告被调用编译单元的某段字节码未通过校验。

产生条件 当一个程序试图调用一个外部函数时。

举例 var a=3*OtherScript#doThis(param)。

严重程度 致命

可预测性 在字节码校验时被检测到。

解决方案 终止程序并向WMLScript解释器报错。

(2) **致命库函数错**

描述 报告一个库函数调用出现致命错。

产生条件 当一个库函数调用（CALL_LIB）被执行时。通常，在库函数执行时，这种错误是无法预测的。

举例 var a=3*String.format(param)

严重程度 致命

可预测性 不能预测。

解决方案 终止程序并向WMLScript解释器报错。

(3) **无效的函数参数**

描述 报告函数调用中使用的参数个数与被调函数的参数个数不匹配。

产生条件 当一个外部函数调用被执行时（CALL_URL）。

举例 编译器对某一指令产生了无效参数，或是被调函数的参数个数发生改变。

严重程度 致命

可预测性 无法预测。

解决方案 终止程序并向WMLScript解释器报错。

(4) **外部函数未找到**

描述 报告调用的外部函数无法在指定的编译单元中找到。

产生条件 当程序试图调用一个外部函数时（CALL_URL）。

举例 Var a=3*OtherScript#doThis(param)
严重程度 致命。
可预测性 无法预测。
解决方案 终止程序并向WMLScript解释器报错。

(5) **无法载入编译单元**

描述 报告在访问网络服务器中的编译单元时，由于不可恢复的错误，指定的编译单元无法载入或是不存在。
产生条件 当程序试图调用一个外部函数调时(CALL_URL)
举例 Var a=3*OtherScript#doThis(param)
严重程度 致命。
可预测性 无法预测。
解决方案 终止程序并向WMLScript解释器报错。

(6) **访问非法**

描述 报告一个访问非法错误，原因是被调外部函数在一个保护编译单元中。
产生条件 当程序试图调用一个外部函数时。
举例 Var a=3*OtherScript#doThis(param)
严重程度 致命。
可预测性 无法预测。
解决方案 终止程序并向WMLScript解释器报错。

(7) **堆栈下溢**

描述 报告一个由编程错引起（编译器产生了错误代码）的堆栈下溢。
产生条件 当程序试图对一个空栈进行出栈操作时。
举例 仅当编译器产生了错误代码时发生。
严重程度 致命。
可预测性 无法预测。
解决方案 终止程序并向WMLScript解释器报错。

2. 放弃程序执行

这种错误在WMLScript函数调用库函数Lang.abort( )来终止执行时产生。

• **程控终止**

描述 报告因Lang.abort( )函数调用造成的代码执行终止。
产生条件 当程序调用Lang.abort( )函数时。
举例 Lang.abort(“Unrecoverable error”);
严重程度 致命。
可预测性 无法预测。
解决方案 终止程序并向WMLScript解释器报错。

3. 内存耗尽错误

这些错误与WMLScript解释器的动态行为及其存储器的使用有关（详细说明见6.6.1节）。

(1) **堆栈上溢**

描述 报告一个堆栈上溢错误。

产生条件 在程序资源嵌套过深或向操作数堆栈中压入过多的变量时。
举例 Function f(x){f(x+1);}
严重程度 致命。
可预测性 无法预测。
解决方案 终止程序并向WMLScript解释器报错误。
(2) **存储器溢出**
描述 报告已无可用的存储器资源。
产生条件 当操作系统无法分配更多的空间给解释器时。
举例 Function f(x) (x=x+ ’abcdefghijklmnopqrstuvwxyz ’;f(x););
严重程度 致命。
可预测性 无法预测。
解决方案 终止程序并向WMLScript解释器报错。
4. **外部异常**
下列异常是在WMLScript字节码解释器外生成的。
(1) **用户生成终止**
描述 报告用户试图终止程序运行。
产生条件 任何时候。
举例 在应用程序运行时，用户按下了重置（reset）键。
严重程度 致命。
可预测性 无法预测。
解决方案 终止程序并向WMLScript解释器报错。
(2) **系统生成终止**
描述 报告程序运行时出现致命的外部异常，且必须立即终止。异常可能是由电量不足、关闭电源等原因引起。
产生条件 任何时候。
举例 由于电量不足系统自动关闭。
严重程度 致命。
可预测性 无法预测。
解决方案 终止程序并向WMLScript解释器报错。

6.10.4 非致命错误

1. **算术错误**
这些错误与WMLScript的算术操作有关。
(1) **除零**
描述 报告出现除0
产生条件 当程序试图除以0时（整型或浮点型除法，或取余）。
举例
```
Var a=10;
Var b=0;
Var x=a/b;
```

```
Var y=a div b;
Var z=a%b;
a/=b;
```

严重程度 非致命。

可预测性 可预测。

解决方案 结果是无效值。

(2) **整数上溢**

描述 报告出现整数上溢。

产生条件 当程序试图进行整数操作时。

举例

```
Var a=Lang.maxInt( );
Var b=Lang.maxInt( );
Var c=a+b;
```

严重程度 非致命。

可预测性 可预测（但有时很困难）。

解决方案 结果是无效值。

(3) **浮点数上溢**

描述 报告出现浮点数上溢。

产生条件 当程序试图进行浮点操作时。

举例

```
Var a=1.6e308;
Var b=1.6e308;
Var c=a*b;
```

严重程度 非致命。

可预测性 可预测（但有时很难）

解决方案 结果是无效值。

(4) **浮点数下溢**

描述 报告出现浮点数喜下溢。

产生条件 当浮点操作的结果小于可表示的下限时。

举例

```
Var a=Float.precision( );
Var b=Float.precision( );
Var c=a*b;
```

严重程度 非致命。

可预测性 可预测（但有时很困难）。

解决方案 结果为浮点值0.0。

2. 常数引用错

这些错误与运行时对常量池中的常量引用有关。

(1) **不是一个数字浮点型常量**

描述 报告引用了一个常量池中浮点文字，而不是一个数字 [IEEE754]。

产生条件 当程序试图访问一个浮点文字时，但编译器产生了一个不是数字的浮点常量。

举例 对浮点文字的引用。

严重程度 非致命。
可预测性 可预测。
解决方案 结果是无效值。
(2) **浮点常量无穷大**
描述 报告对浮点常量的引用为正的或负的无穷大。
产生条件 当程序试图引用常量池中的一个浮点文字，并且编译器产生了一个值为正的或负的无穷大的浮点常量。
举例 对浮点文字的引用。
严重程度 非致命。
可预测性 可预测。
解决方案 结果是无效值。
(3) **非法浮点引用**
描述 报告一个错误浮点常量的引用。
产生条件 当程序试图使用一个浮点值，而系统仅支持整型值时。
举例 Var a=3.14；
严重程度 非致命。
可预测性 可在运行时预测。
解决方案 结果是无效值。
3. 转换错误
这些错误与WMLScript的自动转换有关。
(1) **整数过大**
描述 报告进行整数转换时，整数值过大（正的或负的）。
产生条件 当应用程序试图进行一个自动的整数转换时。
举例 var a = -"99999999999999999999999999999999999999999999";
严重程度 非致命。
可预测性 可预测。
解决方案 结果是无效值。
(2) **浮点值过大**
描述 报告进行浮点数转换时，浮点值过大（正的或负的）。
产生条件 当应用程序试图进行一个自动的浮点数转换时 。
举例 var a = -"9999999.9999999999e99999";
严重程度 非致命。
可预测性 可预测。
解决方案 结果是无效值。
(3) **浮点值过小**
描述 报告进行浮点数转换时，浮点数值过小（正的或负的）。
产生条件 当应用程序试图进行一个自动的浮点数转换时。
举例 var a = -"0.01e-99";
严重程度 非致命

可预测性 可预测

解决方案 结果为浮点数0.0。

6.10.5 库调用及相关错误

由于WMLScript支持库的使用，在库函数中出现错误是可能的。库函数的设计的流程不属于WMLScript语言规范的内容，不过，在库设计时应遵循下面的原则：

- 应提供一种库的使用机制，使错误在发生前能够被检测到。
- 当需要将错误返回给调用者时，应将相同的错误处理机制作为 WMLScript的一个操作符。
- 尽量减少所有库函数可能出现的致命错误。

6.11 仅支持整型数的设备

除了可运行于支持浮点操作的设备， WMLScript也可运行于不支持浮点操作的设备，此时下列规则将生效：

- 变量只能含有下列内部数据类型：
  布尔型。
  整型。
  字符型。
  无效型。
- 任何引用了浮点常量的字节码 LOAD_CONST，都用一个无效值代替这个浮点常量值压入操作数堆栈。
- 除法(/)操作永远返回无效值。
- 所有与浮点型有关的转换规则都将被忽略。
- 以浮点值为参数的URL调用会因无效的URL语法而失败。

编程人员可以使用Lang.float( )[WMLSLibs]检测（在运行中）是否支持浮点操作。

6.12 内容类型

WMLScript编译单元所用的内容类型及其文本和二进编码如下：

- 文本格式：text/x-wap.wmlscript
- 二进码格式：application/x-wap.wmlscriptc

这些类型还没有IANA注册，因此只是试用的WMLScript内容类型。

6.13 术语定义

本规范中使用的术语和习惯用法如下：

字节码(Bytecode) 一种内容的编码。在这里，内容是指一系列典型的轻量级操作码（也就是指令），这些操作码用在目标硬件（或虚拟机器）之中。

客户端（Client） 客户端是向服务器发出连接请求的设备或应用程序。

内容（Content） 内容是源服务器生成或存储的数据（或事件）。典型的是，在响应用户

请求时，内容由用户代理显示或解释。

内容编码（Content Encoding） 当被用作动词时，内容编码指的是把数据对象从一种格式转换为另外一种格式的行为。通常，目标格式需要的物理空间比原格式要少，更易于处理或存储，和/或被加密。当被用作名词时，内容编码指的是一种特殊格式或编码的标准或处理。

内容格式（Content Format） 内容的实际表示。

设备（Device） 一个网络实体，能够发送和接收信息包的，并且有一个唯一的地址。在一个给定的上下文或跨越多重上下文，一个设备既可作为客户端，也可作为服务器。例如，一个设备作为其他服务器的客户端时可充当其他客户端的服务器。

Java脚本（JavaScript） Java脚本是一种实际的标准语言，它用于向HTML文档添加动态行为。Java脚本是ECMA脚本（ECMAScript）的起源技术之一。

源服务器（Origin server） 它作为一种服务器，是给定资源（或称内容）存储或将被生成的地方，通常被看作是Web服务器或HTTP服务器。

资源（Resource） 它是一个可以被URL识别的网络数据对象或服务，可以用多种表述格式所表达（例如，多种语言、数据格式、数据块尺寸和分辨率）或以其他方式进行变化。

服务器（Server） 它是一种被动地等待一个或多个客户端连接请求的设备（或应用程序），可以接收或拒绝来自客户端的连接请求。

用户（User） 用户是一个通过用户代理观看、聆听或使用资源的人。

用户代理（User Agent） 用户代理是对无线标记语言 WML、无线标记语言脚本（WMLScript）、无线电话应用接口（WTAI）或其他资源进行解释的软件或设备。它可能是文本浏览器、语音浏览器、搜索引擎等。

Web服务器（Web Server） 用作HTTP服务器的网络主机。

无线标记语言（WML） 无线标记语言是一种超文本标记语言，用于表示在窄带设备（如，电话）中传输的信息。

无线标记语言脚本（WMLScript） 它是一种脚本语言，用于移动设备编程，是JavaScript 脚本语言的一个扩展子集。

6.14 缩略语

下列缩略语用于本规范：

API	Application Programming Interface	应用编程接口
BNF	Backus-Naur Form	Backus-Naur窗体格式
ECMA	European Computer Manufacturer Association	欧洲计算机制造商协会
HTML	HyperText Markup Language [HTML4]	超文本标记语言
HTTP	HyperText Transfer Protocol [RFC2068]	超文本传输协议
IANA	Internet Assigned Number Authority	因特网域名分配权威机构
LSB	Least Significant Bits	最低有效位
MSB	Most Significant Bits	最高有效位
RFC	Request For Comments	请求注释
UI	User Interface	请求注释

URL	Uniform Resource Locator [RFC1738]	统一资源定位器
UTF	UCS Transformation Format	UCS传输格式
UCS	Universal Multiple-Octet Coded Character Set	通用多字节编码字符集
W3C	World Wide Web Consortium	万维网联盟
WWW	World Wide Web	万维网
WSP	Wireless Session Protocol	无线会话协议
WTP	Wireless Transport Protocol	无线事务处理协议
WAP	Wireless Application Protocol	无线应用协议
WAE	Wireless Application Environment	无线应用环境
WTA	Wireless Telephony Applications	无线电话应用
WTAI	Wireless Telephony Applications Interface	无线电话应用接口
WBMP	Wireless BitMaP	无线位图

6.15 参考标准

[ECMA262] Standard ECMA-262: "ECMAScript Language Specification，"ECMA, June 1997

[IEEE754] ANSI/IEEE Std 754-1985: "IEEE Standard for Binary Floating-Point Arithmetic".Institute of Electrical and Electronics Engineers, New York (1985)

[ISO10646] "Information Technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane，"ISO/IEC 10646-1:1993

[RFC1738] "Uniform Resource Locators (URL)，"T. Berners-Lee, et al., December 1994
URL: ftp://ftp.isi.edu/in-notes/rfc1738.txt

[RFC1808] "Relative Uniform Resource Locators，"R. Fielding, June 1995
URL: ftp://ftp.isi.edu/in-notes/rfc1808.txt

[RFC2279] "UTF-8, a transformation format of Unicode and ISO 10646，"F. Yergeau, January 1998
URL: ftp://ftp.isi.edu/in-notes/rfc2279.txt

[RFC2068] "Hypertext Transfer Protocol - HTTP/1.1，"R. Fielding, et al., January 1997
URL: ftp://ftp.isi.edu/in-notes/rfc2068.txt

[RFC2119] "Key Words for Use in RFCs to Indicate Requirement Levels". S. Bradner, March 1997
URL: ftp://ftp.isi.edu/in-notes/rfc2119.txt

[UNICODE] "The Unicode Standard: Version 2.0,"The Unicode Consortium, Addison-Wesley Developers Press, 1996
URL: http://www.unicode.org/

[WAP] "Wireless Application Protocol Architecture Specification,"WAP Forum, 30-April-1998

URL: http://www.wapforum.org/

[WML] "Wireless Markup Language Specification，"WAP Forum, 30-April-1998
URL: http://www.wapforum.org/

[WMLSLibs] "WMLScript Standard Libraries Specification，"WAP Forum, 30-April-1998
URL: http://www.wapforum.org/

[WSP] "Wireless Session Protocol，"WAP Forum, 30-April-1998
URL: http://www.wapforum.org/

[XML] "Extensible Markup Language (XML), W3C Proposed Recommendation 10-February-1998, REC-xml-19980210，"T. Bray, et al., February 10, 1998
URL: http://www.w3.org/TR/REC-xml

6.16 参考资料

[HTML4] "HTML 4.0 Specification, W3C Recommendation 18-December-1997, REC-HTML40-971218，"D. Raggett, et al., September 17, 1997
URL: http://www.w3.org/TR/REC-html40

[JavaScript] "JavaScript: The Definitive Guide，"David Flanagan. O'Reilly & Associates, Inc. 1997

[WAE] "Wireless Application Environment Specification，"WAP Forum, 30-April-1998
URL: http://www.wapforum.org/