UTF-16

维基百科，自由的百科全书

注意：本页面含有Unihan新版用字：「𫜹 (https://www.unicode.org/cgi-bin/GetUnihanData.pl?codepoint=2A6A5)、𤭢 (https://www.unicode.org/cgi-bin/GetUnihanData.pl?codepoint=24B62)」。有关字符可能會错误显示，詳见Unicode扩展汉字。

UTF-16是Unicode字符编码五层次模型的第三层：字符编码表（Character Encoding Form，也称为"storage format"）的一种实现方式。即把Unicode字符集的抽象码位映射为16位长的整数（即码元）的序列，用于数据存储或传递。Unicode字符的码位，需要1个或者2个16位长的码元来表示，因此这是一个变长表示。

UTF是"Unicode/UCS Transformation Format"的首字母缩写，即把Unicode字符转换为某種格式之意。UTF-16正式定義於ISO/IEC 10646-1的附錄C，而RFC2781也定義了相似的做法。

目录

UTF-16描述

Unicode的编码空间从U+0000到U+10FFFF，共有1,112,064个码位（code point）可用来映射字符。Unicode的编码空间可以划分为16个平面（plane），每个平面包含2^{16}（65,536）个码位。16个平面的码位可表示为从U+xx0000到U+xxFFFF，其中xx表示十六进制值从00_{16}到10_{16}，共计16个平面。第一个平面称为**基本多语言平面**（Basic Multilingual Plane, **BMP**），或稱第零平面（Plane 0），其他平面称为**辅助平面**（Supplementary Planes）。基本多语言平面內，從U+D800到U+DFFF之間的码位區段是永久保留不映射到Unicode字符。UTF-16就利用保留下来的0xD800-0xDFFF区段的码位來對輔助平面的字符的码位進行編碼。

从U+0000至U+D7FF以及从U+E000至U+FFFF的码位

第一个Unicode平面（码位从U+0000至U+FFFF）包含了最常用的字符。该平面被称为基本多语言平面，缩写为*BMP*（Basic Multilingual Plane，BMP）。UTF-16与UCS-2编码这个范围内的码位为16比特长的单个码元，数值等价于对应的码位。BMP中的这些码位是仅有的可以在UCS-2中表示的码位。

从U+10000到U+10FFFF的码位

辅助平面（Supplementary Planes）中的码位，在UTF-16中被编码为**一对**16比特长的码元（即32位元，4字節），称作*代理对*（Surrogate Pair），具体方法是：

1. 码位减去 `0x10000`，得到的值的范围为20比特长的 `0...0xFFFFF`。
2. 高位的10比特的值（值的范围为 `0...0x3FF`）被加上 `0xD800` 得到第一个码元或称作高位代理（high surrogate），值的范围是 `0xD800...0xDBFF`。由于高位代理比低位代理的值要小，所以为了避免混淆使用，Unicode标准现在称高位代理为**前导代理**（lead surrogates）。
3. 低位的10比特的值（值的范围也是 `0...0x3FF`）被加上 `0xDC00` 得到第二个码元或称作低位代理（low surrogate），现在值的范围是 `0xDC00...0xDFFF`。由于低位代理比高位代理的值要大，所以为了避免混淆使用，Unicode标准现在称低位代理为**后尾代理**（trail surrogates）。

UTF-16解碼

lead \ trail	DC00	DC01	…	DFFF
D800	10000	10001	…	103FF
D801	10400	10401	…	107FF
⋮	⋮	⋮	⋱	⋮
DBFF	10FC00	10FC01	…	10FFFF

上述算法可理解为：辅助平面中的码位从U+10000到U+10FFFF，共计FFFFF个，即2^{20}=1,048,576个，需要20位来表示。如果用两个16位长的整数组成的序列来表示，第一个整数（称为前导代理）要容纳上述20位的前10位，第二个整数（称为后尾代理）容纳上述20位的后10位。还要能根据16位整数的值直接判明属于前导整数代理的值的范围（2^{10}=1024），还是后尾整数代理的值的范围（也是2^{10}=1024）。因此，需要在基本多语言平面中保留不对应于Unicode字符的2048个码位，就足以容纳前导代理与后尾代理所需要的编码空间。这对于基本多语言平面总计65536个码位来说，仅占3.125%。

由于前导代理、后尾代理、BMP中的有效字符的码位，三者互不重叠，搜索是简单的：一个字符编码的一部分不可能与另一个字符编码的不同部分相重叠。这意味着UTF-16是自同步（self-synchronizing）的：可以通过仅检查一个码元来判定给定字符的下一个字符的起始码元。UTF-8也有类似优点，但许多早期的编码模式就不是这样，必须从头开始分析文本才能确定不同字符的码元的边界。

由于最常有的字符都在基本多文种平面中，许多软件处理代理对的部分往往得不到充分的测试。这导致了一些长期的bug与潜在安全漏洞，它们甚至存在于广为流行且评价颇高的应用软件中[1]。

从U+D800到U+DFFF的码位

Unicode标准规定U+D800...U+DFFF的值不对应于任何字符。

但是在使用UCS-2的时代，U+D800...U+DFFF内的值被占用，用于某些字符的映射。但只要不构成代理对，许多UTF-16编码解码还是能把这些不符合Unicode标准的字符映射正确的辨识、转换成合规的码元[2]。按照Unicode标准，这种码元序列本来应算作编码错误。

範例：

以U+10437编码（𐐷）为例:

1. `0x10437` 减去 `0x10000`，结果为`0x00437`，二进制为 `0000 0000 0100 0011 0111`
2. 分割它的上10位值和下10位值（使用二进制）：`0000 0000 01` 和 `00 0011 0111`

3. 添加 0xD800 到上值，以形成高位：0xD800 + 0x0001 = 0xD801
4. 添加 0xDC00 到下值，以形成低位：0xDC00 + 0x0037 = 0xDC37

- 下表总结了一起示例的转换过程，颜色指示码点位如何分布在所述的UTF-16中。由UTF-16编码过程中加入附加位的以黑色显示。

字符		普通二进制	UTF-16二进制	UTF-16 十六进制字符代码	UTF-16BE 十六进制字节	UTF-16LE 十六进制字节
$	U+0024	0000 0000 0010 0100	0000 0000 0010 0100	0024	00 24	24 00
€	U+20AC	0010 0000 1010 1100	0010 0000 1010 1100	20AC	20 AC	AC 20
𐐷	U+10437	0001 0000 0100 0011 0111	1101 1000 0000 0001 1101 1100 0011 0111	D801 DC37	D8 01 DC 37	01 D8 37 DC
𤭢	U+24B62	0010 0100 1011 0110 0010	1101 1000 0101 0010 1101 1111 0110 0010	D852 DF62	D8 52 DF 62	52 D8 62 DF

範例：UTF-16編碼程序

假設要將U+64321（16進位）轉成UTF-16編碼。因為它超過U+FFFF，所以他必須編譯成32位元（4個byte）的格式，如下所示：

```
V = 0x64321
Vx = V - 0x10000
= 0x54321
= 0101 0100 0011 0010 0001

Vh = 01 0101 0000 // Vx的高位部份的10 bits
Vl = 11 0010 0001 // Vx的低位部份的10 bits
w1 = 0xD800 //結果的前16位元初始值
w2 = 0xDC00 //結果的後16位元初始值

w1 = w1 | Vh
= 1101 1000 0000 0000
 |       01 0101 0000
= 1101 1001 0101 0000
= 0xD950

w2 = w2 | Vl
= 1101 1100 0000 0000
 |       11 0010 0001
= 1101 1111 0010 0001
= 0xDF21
```

所以這個字U+64321最後正確的UTF-16編碼應該是：

```
0xD950 0xDF21
```

而在小尾序中最后的编码应该是：

```
0x50D9 0x21DF
```

因為這個字超過U+FFFF所以無法用UCS-2的格式編碼。

16進制編碼範圍	UTF-16表示方法（二進制）	10進制碼範圍	字節數量
U+0000 - U+FFFF	xxxx xxxx xxxx xxxx - yyyy yyyy yyyy yyyy	0-65535	2
U+10000 - U+10FFFF	1101 10yy yyyy yyyy - 1101 11xx xxxx xxxx	65536-1114111	4

UTF-16比起UTF-8，好處在於大部分字符都以固定長度的字節（2字節）儲存，但UTF-16卻無法相容於ASCII編碼。

UTF-16的編碼模式

UTF-16的大尾序和小尾序儲存形式都在用。一般來說，以Macintosh製作或儲存的文字使用大尾序格式，以Microsoft或Linux製作或儲存的文字使用小尾序格式。

為了弄清楚UTF-16文件的大小尾序，在UTF-16文件的開首，都會放置一個U+FEFF字符作為Byte Order Mark（UTF-16 LE以 FF FE 代表，UTF-16 BE以 FE FF 代表），以顯示這個文字檔案是以UTF-16編碼，其中U+FEFF字符在UNICODE中代表的意義是 ZERO WIDTH NO-BREAK SPACE，顧名思義，它是個沒有寬度也沒有斷字的空白。

以下的例子有四個字符：「朱」（U+6731）、半角逗號（U+002C）、「聿」（U+807F）、「𪩥」（U+2A6A5）。

使用UTF-16編碼的例子						
編碼名稱	編碼次序	編碼				
		BOM	朱	,	聿	𪩥
UTF-16 LE	小尾序，不含BOM		31 67	2C 00	7F 80	69 D8 A5 DE
UTF-16 BE	大尾序，不含BOM		67 31	00 2C	80 7F	D8 69 DE A5
UTF-16 LE	小尾序，包含BOM	FF FE	31 67	2C 00	7F 80	69 D8 A5 DE
UTF-16 BE	大尾序，包含BOM	FE FF	67 31	00 2C	80 7F	D8 69 DE A5

UTF-16與UCS-2的關係

UTF-16可看成是UCS-2的父集。在沒有輔助平面字符（surrogate code points）前，UTF-16與UCS-2所指的是同一的意思。但當引入輔助平面字符後，就稱為UTF-16了。現在若有軟件聲稱自己支援UCS-2編碼，那其實是暗指它不能支援在UTF-16中超過2位元組的字集。對於小於0x10000的UCS碼，UTF-16編碼就等於UCS碼。

Microsoft Windows操作系统内核对Unicode的支持

Windows操作系统内核中的字符表示为UTF-16小尾序，可以正确处理、显示以4字节存储的字符。但是Windows API实际上仅能正确处理UCS-2字符，即仅以2字节存储的，码位小于U+FFFF的Unicode字符。其根源是Microsoft C++语言把 wchar_t 数据类型定义为16比特的unsigned short，这就与一个 wchar_t 型变量对应一个宽字符、可以存储一个Unicode字符的规定相矛盾。相反，Linux平台的GCC编译器规定一个 wchar_t 是4字节长度，可以存储一个UTF-32字符，宁可浪费了很大的存储空间。下例运行于Windows平台的C++程序可说明此点：

```
// 此源文件在Windows平台上必须保存为Unicode格式（即UTF-16小尾）
// 因为包含的汉字“”，不能在简体中文版Windows默认的代码页936（即GBK）中表示
// 该汉字在UTF-16小尾序中用4个字节表示
// Windows操作系统能正确显示这样的在UTF-16需用4字节表示的字符
// 但是Windows API不能正确处理这样的在UTF-16需用4字节表示的字符，把它判定为2个UCS-2字符

#include <windows.h>
int main()
{
    const wchar_t lwc[] = L"";

    MessageBoxW(NULL, lwc, lwc, MB_OK);

    int i = wcslen(lwc);
    printf("%d\n", i);
    int j = lstrlenW(lwc);
    printf("%d\n", j);

    return 0;
}
```

Windows 9x系统的API仅支持ANSI字符集，只支持部分的UCS-2转换。1996年发布的Windows NT 4.0的API支持UCS-2。Windows 2000开始，Windows系统API开始支持UTF-16，并支持Surrogate Pair；但许多系统控件比如文本框和label等还不支持surrogate pair表示的字符，会显示成两个字符。Windows 7及更新的系统已经良好地支持了UTF-16，包括Surrogate Pair。

Windows API支持在UTF-16LE（`wchar_t`类型）与UTF-8（代码页CP_UTF8）之间的转码。例如：

```
#include <windows.h>
int main() {
    char a1[128], a2[128] = { "Hello" };
    wchar_t w = L'页';
    int n1, n2= 5;
    wchar_t w1[128];
    int m1 = 0;

    n1 = WideCharToMultiByte(CP_UTF8, 0, &w, 1, a1, 128, NULL, NULL);
    m1 = MultiByteToWideChar(CP_UTF8, 0, a2, n2, w1, 128);
}
```

参考文献

1. Code in Apache Xalan 2.7.0 which can fail on surrogate pairs. Apache Foundation. “The code wrongly assumes it is safe to use `substring` on the input”
2. Python 2.6 decode of UTF16 does this on Linux, and it correctly handles surrogate pairs. All "CESU" decoders do it too, though they also mistranslate correct surrogate pairs into 2 characters

外部連結

- Unicode Technical Note #12: UTF-16 for Processing (http://www.unicode.org/notes/tn12/)
- A very short algorithm for determining the surrogate pair for any codepoint (http://www.unicode.org/faq/utf_bom.html#utf16-4)
- Unicode FAQ: What is the difference between UCS-2 and UTF-16? (http://www.unicode.org/faq/basic_q.html#14)
- Unicode Character Name Index (http://www.unicode.org/charts/charindex.html)
- RFC 2781: UTF-16, an encoding of ISO 10646
- java.lang.String documentation, discussing surrogate handling (http://docs.oracle.com/javase/6/docs/api/java/lang/String.html#charAt(int))

参见

- RFC 2781，UTF-16標準