Unicodeエンコード方式

UTF-8、UTF-16、およびUTF-32間の相互運用方法

Ken Lunde

2001年 9月 01日

この記事では、Unicodeの最近の成果をかいつまんで説明し、そのエンコード方式であるUTF-8、UTF-16、およびUTF-32を概観します。またこの記事では、簡単なアルゴリズムを使用することによって、これらの3つのエンコード方式をどのように相互運用できるかについても説明します。Unicode Transformation Format (UTF) 相互運用性の一例として実用的なPerl機能を示し、また完全UTF相互運用性を提供する3つのUnicode対応ライブラリーについても説明します。Unicodeでは基本多言語面 (BMP) 以外の文字も使用しますので、オペレーティング・システムやアプリケーションでは、1,112,064個の有効なUnicodeコード・ポイントの全範囲をサポートし、かつ各UTF間で相互運用が可能でなければなりません。

Unicode標準は単一の (しかし膨大な) 文字セットと考えられていますが、Unicode標準は、3つのエンコード方式UTF-8、UTF-16、およびUTF-32で表すことができます。

最新版のUnicodeであるV3.1は、2つのISO標準、つまり、ISO 10646-1:2000 (Part 1: アーキテクチャーと基本多言語面) およびISO 10646-2:2000 (Part 2: 補足プレーン) と同等です。UnicodeとISO 10646の間の緊密な関係は重要なものであり、この関係は今後も維持されるでしょう。これにより、すべてのUnicode対応ソフトウェアは、一般に受け入れられている国際標準にも確実に準拠するようになります。

文字セットとしてのUnicode

文字セットとしてのUnicodeは、それぞれが最大65,536コード・ポイントを持つ17個のプレーンで構成されています。**プレーン**とは、256 x 256のマトリックス内に納められている文字のグループで、各プレーンには最大65,536文字が含まれています。プレーンは、65,536個のコード・ポイントが連続しているものと考えることもできます。最初のプレーンは特別なもので、プレーン00または**基本多言語面** (BMP) と呼ばれ、63,488個の使用可能なコード・ポイントのみを含んでいます。残りの16プレーンは**補足プレーン**と呼ばれ、それぞれのプレーンには65,536個のコード・ポイントが含まれています。

BMPで未定義の2,048個のコード・ポイント (65,536から63,488を引いた残り) は**サロゲート** (surrogate)と呼ばれます。つまり、1,024個の上位サロゲートの後に1,024個の下位サロゲートが続いています。これらのサロゲートを併用することで、16個の補足プレーンに含まれる1,048,576個のコード・ポイントにアクセスすることができます。2,048個のサロゲートは、UTF-16エンコード

方式にしか使用できません。したがって、Unicodeでは、合計1,112,064個のコード・ポイントを使用することができます。

最新バージョンのUnicodeはV3.1で、このバージョンの場合、94,140文字という膨大な数の文字がBMPと3つの補足プレーンに割り当てられています (下図参照)。

プレーン	プレーン名	文字数
0 (0x00)	基本多言語面 (BMP)	49,196
1 (0x01)	用字と記号のための補足多言語プレーン (SMP)	1,594
2 (0x02)	補足表意文字プレーン (SIP)	43,253
14 (0x0E)	補足特殊目的プレーン (SPP)	97

Unicode V3.0では49,194文字が定義されていて、これらの文字はすべてBMPに含まれています。Unicode V 3.1では、2文字がBMPに追加され、残りの44,944文字は3つの補足プレーンに割り当てられています。

V3.1で最も重要な点は、それがBMPの外に文字を割り当てた最初のUnicodeバージョンであるということです。それより前のUnicodeバージョンでは、BMP外の文字をサポートするエンコード方式をサポートしていましたが、V3.1は、実際にBMPの外に文字を割り当てた最初のバージョンです。このことは、ソフトウェア・デベロッパーにとって大きな意味を持っています。

エンコード方式としてのUnicode

最新バージョンのUnicodeは、UTF-8、UTF-16、およびUTF-32という3つのエンコード方式をサポートします。これらの名前に使用されている数字、つまり8、16、32は、ビット数での基本単位を表しています。たとえば、UTF-8は、8ビットの単位 (それぞれ1バイトに相当) で構成されています。UTF-16は、16ビットの単位からなり、UTF-32は32ビットの単位を使用します。

これらの3つのエンコード方式は、共通な1つの特徴を持っています。16の補足プレーンにある1,048,576個のコード・ポイントは、4バイトすなわち32ビットで表されているということです。UTF-8は4バイトを使用し、UTF-16は2つの16ビット単位を使用し (上位サロゲートおよび下位サロゲート)、UTF-32は1つの32ビット単位を使用します。

UTF-8エンコード方式

UTF-8エンコード方式は可変長で、文字は1、2、3、または4バイトでエンコードされます。Unicode (BMP) の最初の128文字、U+0000～U+007Fは、1バイトでエンコードされ、ASCIIコードと同じになります。U+0080～U+07FF (BMP) は2バイトでエンコードされ、U+0800～U+FFFF (これもBMP) は3バイトでエンコードされます。16補足プレーンの1,048,576文字は、4バイトでエンコードされます。

UTF-16エンコード方式

UTF-16エンコード方式は可変長の16ビット単位で表現されます。それぞれの文字は、1つまたは2つの16ビット単位で構成されています。バイトの単位で言えば、それぞれの文字は2バイトまたは4バイトで構成されています。このエンコード方式では、単独の16ビット単位がBMP全体をエン

コードします。ただし、"サロゲート" と呼ばれる2,048コード・ポイントを除きます。サロゲートは、16補足プレーンの1,048,576文字をエンコードするためにペアで使用されます。

U+D800～U+DBFFは1,024の上位サロゲートであり、U+DC00～U+DFFFは1,024の下位サロゲートです。上位サロゲートと下位サロゲート (つまり、2つの16ビット単位) を組み合せて、16補足プレーン内の単一文字を表します。

UTF-32エンコード方式

UTF-32エンコード方式は、固定長の32ビット (4バイト) 表現です。UCS-4エンコード方式に精通している人は、UTF-32エンコード方式がUCS-4エンコード方式のサブセットに過ぎず、Unicodeの17プレーンしかカバーしないということに注意してください。言い換えれば、UTF-32のエンコード範囲は0x00000000～0x0010FFFFなのです。

UTF-16とUTF-32のバイト順序に注意

UTF-8エンコード方式はバイトで構成されています。各文字は、1、2、3、または4バイトで表されます。UTF-16およびUTF-32エンコード方式は、それぞれ16ビットと32ビット単位で構成されています。つまり、バイト順序が意味を持ちます。幸いにも、デベロッパーは、UTF-16またはUTF-32テスト・データの最初の文字としてByte Order Mark (バイト・オーダー・マーク、BOM) を使用することを勧められています。BOMにより、解釈ソフトウェアが使用するバイト順序が決まります。この2つのバイト順序は、リトル・エンディアンおよびビッグ・エンディアンと呼ばれます。Intelプロセッサー (通常、Windowsを実行するコンピューターを作動させている) は、リトル・エンディアン・バイト順序を使用します。Mac OSや大部分のUnix 系OSを稼動するコンピューターは、ほとんどビッグ・エンディアン・バイト順序を使用します。UTF-16エンコード方式の場合、BOMは、ビッグ・エンディアン・バイト順序では0xFEFFとして表され、リトル・エンディアン・バイト順序では0xFFFEとして表されます。UTF-32エンコード方式の場合、これらは0x0000FEFFおよび0xFFFE0000となります。

たとえば、0x4Eと0x00の2つのバイトを考えてみます。16ビット単位では、これらのバイトは、バイト順序に応じて0x4E00または0x004Eになります。0x4E00 (ビッグ・エンディアン) は、0x004E (リトル・エンディアン) と同様、「1」を意味する漢字です。0x004E (ビッグ・エンディアン) は、0x4E00 (リトル・エンディアン) と同様、ローマ字 "N" を表します。これでお分かりのように、バイト順序を正しく解釈しないと、大変な結果を招くことになります。

各Unicodeエンコード方式間の相互運用性

上記3つのUnicodeエンコード方式間の相互運用は、まったくアルゴリズム上の問題です。筆者は、4つの基本コード変換アルゴリズムがあれば十分だと理解しています。しかし、ソフトウェアもバイト順序を正しく処理する必要があり、またBOMを認識し、正しく処理する必要があるということも忘れてはなりません。

下表は、UTF-32エンコード方式とUTF-16エンコード方式の関係を示す例として、16個の補足プレーンがこれらのエンコード方式とどのように対応しているかを示したものです。

プレーン	UTF-32エンコード方式	UTF-16エンコード方式
1	0x00010000～0x0001FFFF	0xD800DC00～0xD83FDFFF

2	0x00020000 ~ 0x0002FFFF	0xD840DC00 ~ 0xD87FDFFF
3	0x00030000 ~ 0x0003FFFF	0xD880DC00 ~ 0xD8BFDFFF
4	0x00040000 ~ 0x0004FFFF	0xD8C0DC00 ~ 0xD8FFDFFF
5	0x00050000 ~ 0x0005FFFF	0xD900DC00 ~ 0xD93FDFFF
6	0x00060000 ~ 0x0006FFFF	0xD940DC00 ~ 0xD97FDFFF
7	0x00070000 ~ 0x0007FFFF	0xD980DC00 ~ 0xD9BFDFFF
8	0x00080000 ~ 0x0008FFFF	0xD9C0DC00 ~ 0xD9FFDFFF
9	0x00090000 ~ 0x0009FFFF	0xDA00DC00 ~ 0xDA3FDFFF
10	0x000A0000 ~ 0x000AFFFF	0xDA40DC00 ~ 0xDA7FDFFF
11	0x000B0000 ~ 0x000BFFFF	0xDA80DC00 ~ 0xDABFDFFF
12	0x000C0000 ~ 0x000CFFFF	0xDAC0DC00 ~ 0xDAFFDFFF
13	0x000D0000 ~ 0x000DFFFF	0xDB00DC00 ~ 0xDB3FDFFF
14	0x000E0000 ~ 0x000EFFFF	0xDB40DC00 ~ 0xDB7FDFFF
15	0x000F0000 ~ 0x000FFFFF	0xDB80DC00 ~ 0xDBBFDFFF
16	0x00100000 ~ 0x0010FFFF	0xDBC0DC00 ~ 0xDBFFDFFF

これらの3つのUTF間の変換方法を例示するために、いくつかの簡単なPerl関数を以下に紹介します。(注: これらの関数はあまり効率的にはできていません。しかし後述のように、効率性を向上させた商用ライブラリーが用意されています。)リスト1 のPerl関数は、単一のUTF-16文字をUTF-32エンコード方式に変換するもので、ビッグ・エンディアン・バイト順序を仮定しています。

リスト1: UTF-16からUTF-32への変換

```
sub UTF16toUTF32 ($) {
   my ($bytes) = @_;
   if ($bytes =~ /^([\x00-\xD7\xE0-\xFF][\x00-\xFF])$/) {
     pack("N",unpack("n",$bytes));
   } elsif ($bytes =~ /^([\xD8-\xDB][\x00-\xFF])([\xDC-\xDF][\x00-\xFF])$/) {
     pack("N",((unpack("n",$1) - 55296) * 1024) + (unpack("n",$2) - 56320) +
65536);
   } else {
     die "Whoah! Bad UTF-16 data!\n";
   }
}
```

リスト2 は、単一のUTF-8文字をUTF-32エンコード方式に変換するもので、これもビッグ・エンディアン・バイト順序を仮定しています。

リスト2: UTF-8からUTF-32への変換

```
sub UTF8toUTF32 ($) {
   my ($bytes) = @_;
   if ($bytes =~ /^([\x00-\x7F])$/) {
     pack("N",ord($1));
   } elsif ($bytes =~ /^([\xC0-\xDF])([\x80-\xBF])$/) {
     pack("N",((ord($1) & 31) < 6) | (ord($2) & 63));
   } elsif ($bytes =~ /^([\xE0-\xEF])([\x80-\xBF])([\x80-\xBF])/) {
     pack("N",((ord($1) & 15) < 12) | ((ord($2) & 63) <  6) | (ord($3) & 63));
   } elsif ($bytes =~ /^([\xF0-\xF7])([\x80-\xBF])([\x80-\xBF])([\x80-\xBF])/) {
     pack("N",((ord($1) & 7) > 18) | ((ord($2) & 63) < 12) | ((ord($3) &
63) <  6) | (ord($4) & 63));
   } else {
     die "Whoah! Bad UTF-8 data! Perhaps outside of Unicode (5- or 6-byte).\n";
   }
}
```

リスト3 は、単一のUTF-32文字をUTF-8エンコード方式に変換するもので、これもビッグ・エンディアン・バイト順序を仮定しています。

リスト3: UTF-32からUTF-8への変換

```
sub UTF32toUTF8 ($) {
  my ($ch) = unpack("N",$_[0]);
  if ($ch <= 127) {
    chr($ch);
  } elsif ($ch <= 2047) {
    pack("C*", 192 | ($ch > 6), 128 | ($ch & 63));
  } elsif ($ch <= 65535) {
    pack("C*", 224 | ($ch > 12), 128 | (($ch > 6) & 63),
   128 | ($ch & 63));
  } elsif ($ch <= 1114111) {
    pack("C*", 240 | ($ch> 18), 128 | (($ch > 12) & 63),
   128 | (($ch> 6) & 63), 128 | ($ch & 63));
  } else {
    die "Whoah! Bad UTF-32 data! Perhaps outside of Unicode (UCS-4).";
  }
}
```

最後に、リスト4 は、単一のUTF-32文字をUTF-16エンコード方式に変換するもので、これもビッグ・エンディアン・バイト順序を仮定しています。

リスト4: UTF-32からUTF-16への変換

```
sub UTF32toUTF16 ($) {
  my ($ch) = unpack("N",$_[0]);
  if ($ch <= 65535) {
    pack("n", $ch);
  } elsif ($ch <= 1114111) {
    pack("n*", ((($ch - 65536) / 1024) + 55296),(($ch % 1024) + 56320));
  } else {
    die "Whoah! Bad UTF-32 data! Perhaps outside of Unicode (UCS-4).";
  }
}
```

これらのPerl関数は、ビッグ・エンディアン・バイト順序のみを処理するように作成してありますが、それは、わたしの開発環境では、リトル・エンディアン・データを処理する必要がないからです。わたしの作業で最後に出来上がるものは、ビッグ・エンディアン・バイト順序を使用するPostScriptを扱うファイルです。

バイナリー順序に注意

Unicodeエンコード方式でデータを表す場合、データベース・デベロッパーは、他のバイナリー順序についても知っている必要があります。UTF-8エンコード方式とUTF-32エンコード方式は、同じバイナリー順序を使用します。つまり、文字コードをそのバイト値に従って順序付けする場合は、同じ順序になります。UTF-16エンコード方式では、バイナリー順序が異なります。それは、16補足プレーンの1,048,576コード・ポイントを表すために、2,048個の上位および下位サロゲートを使用するからです。

UTF相互運用性のインプリメンテーション

完全なUTF相互運用性を提供するUnicode対応ライブラリーが、少なくとも3つあります。つまり、1,112,064個の有効なUnicodeコード・ポイントが与えられた場合、これらのライブラリはAPIによって3つのUTF間を変換可能にします。これらをインプリメントしたものとは、IBMのInternational Components for Unicode (ICU)、IBMのICUとのインターフェースを持ったX.NetのxIUA (Internationalization & Unicode Adaptor)、およびBasis TechnologyのRosetteがあります。ICUはJavaとC/C++ で使用でき、xIUAはC/C++ で使用でき、RosetteはC/C++ で使用できます (「参考文献」を参照)。ユーザーの開発ニーズを最もよく満たしてくれるのがどれであるかを判別するために、これらの3つの製品をすべて検討されることをお勧めします。

いくつかの実践的な例

この数年間、私はUCS-2 (つまり、サロゲートなしのUTF-16エンコード方式。したがって、補足プレーンにアクセスする必要がない)、およびAdobe Systems CJKV文字集合のCID-Keyedフォント用のUTF-8 CMapファイルを保守しています。CMapファイルは、TrueTypeフォントやOpenTypeフォントの "cmap" テーブルと類似したもので、エンコード方式をCID (文字ID) にマップしてCIDフォントのグリフ(glyph)を識別します。UCS-2とUTF-8の間にはアルゴリズム上の関連性があるため、私はUCS-2 CMapファイルのみを保守し、次にツールを使用してほぼ自動的にUCS-2ファイルからUTF-8 CMapファイルを生成しました。こうすることで、UTF-8 CMapファイルをオリジナルのUCS-2ファイルと同期させました。私は、この目的のために簡単なPerlスクリプトを使用しました。このスクリプトは、16ビットのUCS-2表現からUTF-8エンコード方式の1、2、および3バイト表現への変換をサポートしました。

私は最近、一組の新しいUnicode CMapファイルを開発しました。これは、3つのUnicodeエンコード方式UTF-8、UTF-16、およびUTF-32の補足プレーンをサポートするものです。私は 自分のPerlツールを拡張してこれらの3つのUnicodeエンコード方式間の相互運用を可能にし、1,112,064個の有効なコード・ポイントをすべて正しく処理できるようにしました。

私は最初、3つのUnicodeエンコード方式間の変換を可能にするツールを作成しましたが、UTF-8からUTF-32へ、UTF-32からUTF-8へ、UTF-16からUTF-32へ、UTF-32からUTF-16への変換アルゴリズムだけで十分であることが分かりました。UTF-8とUTF-16間の変換は、UTF-32を中間表現として使用することで処理できます (ただし、直接コード変換アルゴリズムを使用しても簡単にインプリメントできますが)。私の関心事は速度ではなく正確性でしたので、この解決策は私のニーズを完全に満たしてくれました。他の人たちのニーズは異なるかもしれません。

次に私は、UTF-32をCMapファイルの最初の表現として使用し、そこからUTF-8およびUTF-16 CMapファイルを生成することにしました。Perlを (再度) 使用することで、このプロセスが完全に

自動化されるようになりました。私はUTF-32 CMapファイルのみを保守しますが、同等なUTF-8およびUTF-16ファイルは、単一のツールを使用して自動的に生成されます。これにより、CMapファイルの開発に使用する時間が削減され、同時に、各Unicode CMapファイル間の矛盾が生じる可能性も大幅に削減されます。

要約

この記事では、文字セットとしてのUnicodeを簡単に説明し、表現方法に3種類があることを示し、しかもそれらの間の相互運用が簡単であることも示しました。この情報を利用すれば、デベロッパーは、アプリケーションをより容易に拡張して補足プレーン (Unicode V3.1では、文字が割り当て済み) を処理できるようになります。つい最近まではデベロッパーが補足プレーンを回避し、したがって4バイト表現を回避することができました。しかし明らかに状況は変わったのです。

関連トピック

- Unicodeコンソーシアム のWebサイトは、Unicode標準に関する最新情報と、独自のサンプルUTFコード変換ルーチン (Cで書かれている) を提供しています。
- Unicode (V2.1まで) とそのエンコード方式に関する詳細情報は、私の著書CJKV Information Processing (O'Reilly社、1999年) の第3章と第4章にも記載されています。具体的には、120～130ページ (第3章) および186～196ページ (第4章) です。ただし、私の著書が発行されたときはUTF-32エンコード方式はまだ存在していなかったということに注意してください。しかし、そのエンコード方式はUCS-4エンコード方式のサブセット (つまり、0x00000000～0x0010FFFF) ですので (このことについては、私の著書の第4章に記載)、UCS-4の記述をUTF-32に適用することができます。
- X.NetのxIUAに関する詳細情報が、xIUAホーム・ページ に示されています。
- IBMのICU Unicode対応ライブラリーは、ICUホーム・ページ に含まれています。
- IBM alphaWorks に関する多くのテクノロジーが使用できますが、それには、Unicode Normalizer とUnicode Compressor が含まれます。