UTF-8

By Naveen PS

MAKERDEMY
LEARN EXPLORE MAKE

UTF-8

A UTF-8 encoded string is one where the octets in the string are the same as in the UTF-8 representation. A Unicode-based encoding such as UTF-8 can support many languages and can accommodate pages and forms in any mixture of those languages. Its use also eliminates the need for server-side logic to individually determine the character encoding for each page served or each incoming form submission. This significantly reduces the complexity of dealing with a multilingual site or application. A Unicode encoding also allows many more languages to be mixed on a single page than any other choice of encoding. Support for a given encoding, even a Unicode encoding, does not necessarily imply that a user agent will correctly display the text. Numerous scripts, such as Arabic and Indic, require additional rules to transform the character sequence in memory to an appropriate sequence of font glyphs for display.

Any barriers to using Unicode are very low these days. In fact, in January 2012 Google reported that over 60% of the Web in their sample of several billion pages was now using UTF-8. Add to that the figure for ASCII-only web pages (since ASCII is a subset of UTF-8), and the figure rises to around 80%.

There are three different Unicode character encodings: UTF-8, UTF-16 and UTF-32. Of these three, only UTF-8 should be used for Web content. The HTML5 specification says "Authors are encouraged to use UTF-8. Conformance checkers may advise authors against using legacy encodings. Authoring tools should default to using UTF-8 for newly-created documents."

Note, in particular, that all ASCII characters in UTF-8 use exactly the same bytes as an ASCII encoding, which often helps with interoperability and backwards compatibility.

For most of your programming, you shouldn’t have to care about encoding. You want to have character data with no representation and operate on abstract characters. You don’t care at all about the encoding and how many bytes a character turn into. That’s merely a storage issue. Virtually no one can tell you, off the top of their heads, what the UTF-8 representation of a string is because

no one thinks in UTF-8. No one wants to do that during string manipulation, either.

The problem is that some interfaces want the encoded data instead of the abstract character string. These modules usually expect that you're giving it data directly from another source without turning it into a Perl string. If you need to review these concepts, check out the "Unicode" chapter in Effective Perl Programming.

Consider the JSON module's decode function expects a UTF-8 encoded string, thinking you're going to take it directly from an HTTP response. This item is not about using this module correctly, but it's a convenient example for the general idea.

Why do we need Unicode?

In the (not too) early days, all that existed was ASCII. This was okay, as all that would ever be needed were a few control characters, punctuation, numbers and letters like the ones in this sentence. Unfortunately, today's strange world of global intercommunication and social media was not foreseen, and it is not too unusual to see English, العربية, 汉语, עִבְרִית, ελληνικά, and ភាសាខ្មែរ in the same document (I hope I didn't break any old browsers).

But for argument's sake, let's say Joe Average is a software developer. He insists that he will only ever need English, and as such only wants to use ASCII. This might be fine for Joe the user, but this is not fine for Joe the software developer. Approximately half the world uses non-Latin characters and using ASCII is arguably inconsiderate to these people, and on top of that, he is closing off his software to a large and growing economy.

Therefore, an encompassing character set including all languages is needed. Thus, came Unicode. It assigns every character a unique number called a code point. One advantage of Unicode over other possible sets is that the first 256 code points are identical to ISO-8859-1, and hence also ASCII. In addition, the vast majority of commonly used characters are representable by only two bytes, in a region called the Basic Multilingual Plane (BMP). Now a character encoding is needed to access this character set, and as the question asks, I will concentrate on UTF-8 and UTF-16.

Memory considerations

So how many bytes give access to what characters in these encodings?

UTF-8:

1 byte: Standard ASCII

2 bytes: Arabic, Hebrew, most European scripts (most notably excluding Georgian)

3 bytes: BMP

4 bytes: All Unicode characters

UTF-16:

2 bytes: BMP

4 bytes: All Unicode characters

It's worth mentioning now that characters not in the BMP include ancient scripts, mathematical symbols, musical symbols, and rarer Chinese/Japanese/Korean (CJK) characters.

If you'll be working mostly with ASCII characters, then UTF-8 is certainly more memory efficient. However, if you're working mostly with non-European scripts, using UTF-8 could be up to 1.5 times less memory efficient than UTF-16. When dealing with large amounts of text, such as large web-pages or lengthy word documents, this could impact performance.

Encoding basics

UTF-8: For the standard ASCII (0-127) characters, the UTF-8 codes are identical. This makes UTF-8 ideal if backwards compatibility is required with existing ASCII text. Other characters require anywhere from 2-4 bytes. This is done by reserving some bits in each of these bytes to indicate that it is part of a multi-byte character. In particular, the first bit of each byte is 1 to avoid clashing with the ASCII characters.

UTF-16: For valid BMP characters, the UTF-16 representation is simply its code point. However, for non-BMP characters UTF-16 introduces surrogate pairs. In this case a combination of two two-byte portions map to a non-BMP character. These two-byte portions come from the BMP numeric range, but are guaranteed by the Unicode standard to be invalid as BMP characters. In addition, since UTF-16 has two bytes as its basic unit, it is affected by endianness. To compensate, a reserved byte order mark can be placed at the beginning of a data stream which indicates endianness. Thus, if you are reading UTF-16 input, and no endianness is specified, you must check for this.

As can be seen, UTF-8 and UTF-16 are nowhere near compatible with each other. So if you're doing I/O, make sure you know which encoding you are using

Practical programming considerations

Character and String data types: How are they encoded in the programming language? If they are raw bytes, the minute you try to output non-ASCII characters, you may run into a few problems. Also, even if the character type is based on a UTF, that doesn't mean the strings are proper UTF. They may allow byte sequences that are illegal. Generally, you'll have to use a library that supports UTF, such as ICU for C, C++ and Java. In any case, if you want to input/output something other than the default encoding, you will have to convert it first.

Recommended/default/dominant encodings: When given a choice of which UTF to use, it is usually best to follow recommended standards for the environment you are working in. For example, UTF-8 is dominant on the web, and since HTML5, it has been the recommended encoding. Conversely, both .NET and Java environments are founded on a UTF-16 character type. Confusingly (and incorrectly), references are often made to the "Unicode encoding", which usually refers to the dominant UTF encoding in a given environment.

Library support: The libraries you are using support some kind of encoding. Which one? Do they support the corner cases? Since necessity is the mother of invention, UTF-8 libraries will generally support 4-byte characters properly, since 1, 2, and even 3 byte characters can occur frequently. However, not all purported UTF-16 libraries support surrogate pairs properly since they occur very rarely.

Counting characters: There exist combining characters in Unicode. For example, the code point U+006E (n), and U+0303 (a combining tilde) forms ñ, but the code point U+00F1 forms ñ. They should look identical, but a simple counting algorithm will return 2 for the first example, 1 for the latter. This isn't necessarily wrong, but may not be the desired outcome either.

Comparing for equality: A, A, and A look the same, but they're Latin, Cyrillic, and Greek respectively. You also have cases like C and Ⅽ, one is a letter, the other a Roman numeral. In addition, we have the combining characters to consider as well. For more info see Duplicate characters in Unicode.

THE END