UCS vs UTF-8 as Internal String Encoding

written on Thursday, January 9, 2014

Unicode is a fascinating mess. It's fascinating in many ways, but one of the most interesting one is how well it works given the complexity. It's easily forgotten when working with Unicode that even though the Unicode Consortium actively develops the Unicode standard there is a colorful history behind it. Besides Unicode that everybody knows, there is also the Universal Character Set as defined by ISO 10646.

Nowadays Unicode and ISO's UCS are for most practical purposes the same thing with a slightly different name. This however was not always the case and in the beginning those were different undertakings. The reason this is still somewhat relevant information, is because in some ways history still shines through.

I'm not going to talk about all the history of Unicode here, many people have been doing this before. In case you *are* curious, here are some links that might be of interest:

- Unicode and ISO/IEC 10646
- Early Years of Unicode
- Universal Character Set

However there are some basics that are important to know even today to get an understanding of why things work in certain ways. When Unicode and ISO 10646 were not the same thing yet, they had different ideas of how things should work. This can still be seen today in the encoding names. UCS2 and UCS4 are 2 and 4 byte encodings for the Universal Character Set (UCS). The UTF formats came later and until today they stand both for Unicode Transformation Format as well as UCS Transformation Format.

UTFs

Unicode started out early as a 16bit encoding roughly equivalent to the now deprecated UCS2 encoding. When it became clear that Unicode needed more than ~16bit of characters the hope for having a fixed length encoding was not particularly high any more.

The ISO group already previously developed an encoding for their standard to map the many characters that was ASCII compatible. The format was called UTF-1 but had some serious problems such as the lack of self synchronization. If you were presented with a potentially truncated or destroyed steam of data you might have received garbage unless you knew what data was there in the string before.

In 1992 UTF-8 was created (so a few years before Unicode 2.0) and nowadays it's probably the most popular encoding for data exchange. UTF-8 supports an astonishing 80% of the internet. UTF-8 is very pragmatic but not just that, it's also very reliable.

When the Unicode consortium gave up on UCS-2 because it was no longer possible to support all of Unicode as a fixed length encoding with just 16 bits per unit two things had to happen: they needed to introduce a variable length encoding with 16 bit unit sizes and they had to reduce the total number of code points that can ever be addressed to find a way to make UTF-16 get the same characteristics as UTF-8. Namely it was necessary that if the stream of data was corrupted, you would not be presented with potentially misleading characters. For instance if a character did not fit into a single 16 bit unit and needed a second one, it was necessary that everything continues working. If the first one was lost in transmission the second one is actually an invalid character that can be ignored and not accidentally taken for a wrong character.

In order to achieve that they came up with the concept of surrogate pairs to encode characters if they don't fit into ~16 bit. Essentially UTF-16 is now defined as an encoding that is either 2 or 4 bytes long but never more than that. The way this works is that 65.536 characters fit into just one unit. If a character is outside of that range it gets split up into two units of 16 bits each and the data is mangled in a way that makes them uniquely distinct. If you would look at one of the two surrogates in the pair separately without the knowledge of the other, it would become apparently that this is in incomplete character. For this trick to work the range *D800* to *DBFF* are reserved for the trail surrogates and *DC00* to *DF00* for the lead ones. As an end result the total range of what can be encoded by UTF-16 is the lowest of all UTF formats.

UTF-32 is the same as the old UCS-4 encoding and remains fixed width. Why can this remain fixed width? As UTF-16 is now the format that can encode the least amount of characters it set

the limit for all formats. It was defined that 1,112,064 was the total number of code points that will ever be defined by either Unicode or ISO 10646. Since Unicode is now only defined from *0* to *10FFFF* UTF-32 sounds a bit like a pointless encoding now as it's 32 bit wide, but only ever about 21 bits are used which makes this very wasteful.

While UTF-8 is defined all the way up to 31 bit it now can not practically grow past 4 byte which means that the worst case of all formats is the same:

	UTF-8	**UTF-16**	**UTF-32**
Highest code point	10FFFF	10FFFF	10FFFF
Code unit size	8 bit	16 bit	32 bit
Byte order dependent	no	yes	yes
Fewest bytes per character	1	2	4
Most bytes per character	4	4	4

I have noticed that seems to surprise people because the assumption is that the worst case of UTF-8 would be more than 4 bytes. This would have been been theoretically correct if UTF-16 would not exist and Unicode would have received more than 1,112,064 characters.

Which One To Pick?

Out of all the choices UTF-8 immediately looks like the best one. UTF-32 is clearly out when it comes to memory consumption as you will be wasting about 40% or memory. UTF-8 is also not byte order dependent which is an immediate win, but it also works with C strings (so is backwards compatible) and worst case it only wastes as much memory as all the other formats.

Upon further introspection it however becomes clear that depending on the language of the text stored, UTF-16 will become more space efficient. For instance for Japanese text UTF-16 will be more space efficient than UTF-8 as many characters are on the basic plane and with that will only require a single UTF-16 code point whereas they are high enough up in the range that they will require 3 bytes for UTF-8.

This however is not the case when Japanese text is mixed with ASCII control structures. For instance XML or HTML documents include enough in-line control data that is in the ASCII range that UTF-8 becomes more efficient as a format compared to UTF-16 (before compression). For instance the front page of the Japanese Wikipedia is 92KB in UTF-8 and 166KB in UTF-16.

For text UTF-8 has clearly won. The only times I see a UTF-16 document fly by is when I get something a Windows user accidentally created with Notepad which apparently sometimes stores as UTF-16.

Internal Encodings

However the question is not so easy when working with Unicode internally and there have been different opinions on this issue. The most prominent approaches for this have been UTF-8 and UTF-16. UTF-16 is the encoding of choice for Java, C# and Objective C (as well as the Windows API). The nice property of UTF-16 is that it allows you to be sloppy as the vast majority of data you will be presented with is probably in the basic plane. This means that operations like `strlen()` will both return the number of code units as well as the number of characters.

For a really long time there did not seem to be much of a contest to using UTF-16 as internal encoding. For a long time the only programming language (besides lots of C code) that used UTF-8 as internal encoding seemed to be Perl. Now however Ruby, Go as well as Rust have decided on using UTF-8. While Ruby can work with lots of internal string encodings, UTF-8 is the one you find most commonly.

The Value of Constant Access

So why was UTF-16 so popular?

UTF-16's biggest selling point was usually that it's possible to address characters directly. That would actually be fine, if programming languages would provide a data type with at least 21 bit of precision to hold a whole Unicode character though. C# and Java unfortunately do not support that. That Java does not provide it makes sense to some degree considering the age of the language and how the string is exposed. That C# does not support it is unfortunate however.

Rust and Go for instance have this better sorted out. While they do use UTF-8 as internal string encoding and expose this to the user, they provide 32 bit data types (called rune in Go and char in rust). In both programming languages you can iterate in actual Unicode characters over the whole string. In many cases this is plenty because parsing for instance usually only needs to look at one or two characters at the time.

In many ways the question is how valuable constant time

addressing of a single character in strings is. I think this is something that is almost impossible to answer because depending on if that's possible or not, the typical algorithms look different.

What Rust and Go gain from having UTF-8 strings is that they are very efficient when they need to juggle with bytes next to textual content. For instance many wire protocols like HTTP are based on ASCII metadata. While HTTP is technically latin1, it's very rare that you will actually encounter a genuine latin1 header. It's in fact, much more common, that people will not be aware of the latin1 part of the specification and put UTF-8 data in a header.

To take Rust as an example, parsing protocols is very efficient because in many cases a parsing step becomes a simple memcpy. The reason for this is that so much data out there is UTF-8. After copying of the data you just need to do a basic check afterwards if the UTF-8 is not invalid, which can be nicely optimized. In contrast to that UTF-16 is a trickier because you need to figure out the length of the buffer through an initial scan and then a second one to decode the data. Or you do it in one go and overallocate.

Go even gets away with using completely unchecked UTF-8 strings. In Rust it's impossible to construct a string in safe code with invalid UTF-8 characters. Go on the other hand lets you happily mix random bytes into your string, but all IO operations are required to ensure that the data is valid. While this sounds pretty terrible it's actually not too bad. I do prefer Rust's approach though which still gives you the nice handling of bytes and strings that Go provides, but errors stay somewhat contained as you can expect a string to be valid.

Rethinking Internal Formats?

For a really long time it looked like nobody would challenge the idea of using UTF-16 as internal string format but that seems to change now. On one hand some languages are exploring using UTF-8 internally, on the other hand we have Python 3 which explores the idea of switching between latin1, UCS-2 and UTF-32 on a string-by-string basis.

The Python 3 trick sounded quite good on the paper but I noticed that there are some practical downsides. For instance Emojis are outside of the basic plane which means that Python needs to represent them as UTF-32 internally. With how lots of template engines are currently structured that can cause some interesting characteristics. Jinja2 for instance currently renders in Unicode and then has a separate encoding step. If you would build a github comment page and an Emoji would be in the comments then whole your response upgrades to UTF-32 just because of a single character. In corner cases like this it might be interesting to use the streaming interface of Jinja2 to encode chunk by chunk to UTF-8 to avoid the extra cost of a more expensive internal string.

As someone who works a lot at the byte <-> Unicode boundary the idea of having strings with an internal UTF-8 encoding is very interesting. Having worked with Rust for a while now I am getting more and more convinced that the approach is a good idea. While it forces you to give up on the idea of being able to address characters individually, that is actually not a huge loss. For a start Unicode would pretty much require you to normalize your strings anyways before you do text processing due to the many ways in which you can format the strings. For instance umlauts come in combined characters but they can also be manually created by placing the regular letter followed by the combining diaeresis character.

So for quite a few operations (like validating length, font rendering etc.) the basic operations a string type provides are already non sufficient anyways. Something as simple as “is this string long enough for a tweet” already requires quite a bit of special casing.

So far at least I have not missed direct character access for anything but peeking at known ASCII characters in Rust and I don't really expect that the string would become a problem. Especially if UTF-8 stays the dominant format then keeping it internally as well makes a lot of sense and requires lots of unnecessary encoding and decoding steps and means the language does not need to provide support for ASCII strings separately.

I'm definitely expecting more languages to take the UTF-8 route in the future and just provide more tools to deal with Unicode as part of the standard library.

This entry was tagged python, rust, thoughts and unicode