String Encoding and R

Posted on February 21, 2018

This blog post is an attempt to explore, and answer, the surprisingly difficult question:

> How do I write UTF-8 encoded content to a file?

Let's suppose that you are a package author who needs to process some text provided by the user. To keep your life simple, you want to ensure that everything you read and write is encoded in the UTF-8 encoding, since that encoding can broadly represent characters from nearly all languages. Your gut reaction might be to open a connection and write to it, like the following:

```
write_utf8 <- function(text, f = tempfile()) {

  # ensure text is UTF-8
  utf8 <- enc2utf8(text)

  # create connection with UTF-8 encoding
  con <- file(f, open = "w+", encoding = "UTF-8")
  writeLines(utf8, con = con)
  close(con)

  # read back from the file just to confirm
  # everything looks as expected
  readLines(f, encoding = "UTF-8")

}
```

And this might appear to do the right thing on your computer.

```
write_utf8("brûlée")
```

```
[1] "brûlée"
```

Unfortunately, a Windows user reports trouble when attempting to write the Japanese character '鬼':

```
> write_utf8("鬼")
[1] "<U+9B3C>"
```

But you ask your Japanese friend to test your function out, and all appears fine on his system:

```
> write_utf8("鬼")
[1] "鬼"
```

But, even weirder, on your Japanese friend's system, he gets some odd output when attempting to write "brûlée":

```
> write_utf8("brûlée")
[1] "br璢馥"
```

Something is definitely going wrong here, but *what*?

String Encoding

Before we dive into R's internals, it behooves us to discuss encodings. I will try to introduce just enough material so we can understand what encoding is and how written language is understood by a computer, but I will gloss over a bit of history. Please see What Every Programmer Absolutely, Positively Needs To Know About Encodings And Character Sets To Work With Text for an excellent introduction to encoding if you want to know more.

How does a computer store text? Language is complex. Computers, conversely, are simple. They have bytes. Sequences of zeroes and ones. Our only choice is to figure out how to translate the complex elements of written language into bytes. How do we do that?

The answer to that is encoding. An encoding is used to map a *piece of language* to a *sequence of bytes*. As an example, you might be familiar with the ASCII (American Standard Code for Information Interchange) encoding. This is a system for encoding English characters, as well as a subset of commonly-used symbols. In this system, the letter 'a' is represented by the byte sequence `01100001`. You can convince yourself of this fact with `pryr::bits()`:

```
pryr::bits('a')
```

```
[1] "01100001"
```

Rather than the sequence of bytes, we could also say that the letter ‘a’ is mapped to the same integer represented by that sequence of bytes. That number is 97:

```
pryr::bits(97L)
```

```
[1] "00000000 00000000 00000000 01100001"
```

Or, more commonly, you might see that number represented using a hex encoding:

```
pryr::bits(0x61L)
```

```
[1] "00000000 00000000 00000000 01100001"
```

Still, the essence is the same. If one is to process bytes using the ASCII encoding, that means that the byte sequence `01100001` should be *interpreted* as the English letter ‘a’, and the letter ‘a’ should be *encoded* on the system as the byte sequence `01100001`. For brevity, we can denote that byte sequence by the same integer it happens to represent; `97`. More commonly, though, these numbers are represented using hex notation, and so this byte sequence would be represented as `0x61`.

ASCII, however, is quite limited as an encoding. It was not designed to be able to represent the whole breadth of written language; rather, it was just designed to represent 128 commonly used characters and symbols from English. What if we want to represent non-English characters?

You might imagine one solution would be to define different encodings for different languages. This is precisely what is done on Windows: Japanese users can install a Japanese version of Windows to get a Japanese locale with characters encoded in a Japanese encoding; Chinese users can install a Chinese version of Windows to get a Chinese encoding, and so on.

These systems hit a road block, though – what if you want to represent *multiple* languages within the same stream of bytes? All the encodings discussed thus far are systems for representing a single written language; what if we want to represent them all?

Enter Unicode. The Unicode standard is effectively a standardized system for the encoding of *all* written languages. The unicode standard was specifically designed to be ASCII compatible – in other words, with the unicode standard, the letter 'a' is still mapped to the code point 97; or, more commonly represented under the unicode standard, `<U+0061>`. Note that this is just the hex representation of that same number. In the Unicode standard, the character 'û' is mapped to code point `<U+00FB>`; the character 'é' maps to `<U+00E9>`; the character '鬼' maps to `<U+9B3C>`, and so on. To represent the character '鬼' on a system, we need to write a byte sequence that represents the code point `<U+9B3C>`.

However, unicode is *not* an encoding. It's a system of uniquely mapping language constructs to numbers. We still need a way of representing the unicode standard in an actual encoding – that is, mapping characters defined in the unicode standard to a sequence of bytes. Somewhat confusingly, there are *multiple* ways of encoding unicode characters. UTF-8 is one such encoding, but there also exist variants such at UTF-16 and UTF-32. Discussing the differences is a bit out of scope for this blog post, but effectively they're just different ways of mapping the Unicode-specified code points to byte sequences. There just happen to be different tradeoffs between different ways of representing the different sequences. Nowadays, UTF-8 dominates because its pros tend to outweigh its cons, but it is not entirely ubiquitous.

The heart of what I want to illustrate, though, is that:

- Text is represented on computers by sequences of bytes;
- An encoding is used to map sequences of bytes to the written language it represents;
- The unicode standard is an effort to map written language to a single, standardized encoding;
- UTF-8 is the most common way of encoding unicode characters, but it is not the only way.

We're now armed with enough knowledge to speak about encodings in R.

String Encoding in R

In R, character vectors have two pieces of information: a sequence of bytes, and an encoding in which those bytes should be interpreted. The encoding of a particular string can be queried with the `Encoding()` function:

```
Encoding('a')
```

```
[1] "unknown"
```

There are four encodings that are commonly reported in this case:

- `latin1` : Also known as ISO/IEC 8859-1, this is a specific kind of 'extended ASCII' encoding;
- `UTF-8` : An encoding system used for representing text encoded using the Unicode standard;
- `bytes` : The associated text has no 'real' encoding; that is, the text may not be interpretable as a written language;
- `unknown` : The associated encoding is 'unknown', but assumed to be the same as the encoding associated with the active locale – that is, the locale as reported by `Sys.getlocale()` .

R uses the iconv library to translate text between the various encodings. For example:

```
utf8 <- "\u00fb"  # 'û'
latin1 <- iconv(utf8, to = "latin1")
paste(latin1, "(latin1):", pryr::bits(latin1))
```

```
[1] "û (latin1): 11111011"
```

```
paste(utf8,   "(UTF-8) :", pryr::bits(utf8))
```

```
[1] "û (UTF-8) : 11000011 10111011"
```

Hopefully this drives home the point that we can represent the same character with a different sequence of bytes under different encodings.

So then, what goes wrong in our aforementioned example? We tried to write content to a connection opened with UTF-8 encoding, but that didn't seem to work.

Wait, what does that actually *mean*:

> write content to a connection opened with UTF-8 encoding

Our content is *already* UTF-8 encoded, isn't it? so what we *really* want to do is just write our text byte-for-byte to the associated connection. So why are we trying to specify an encoding on the connection?

The documentation from `?writeLines` makes this more clear:

> Normally, character strings with marked encodings are converted to the current encoding before being passed to the connection (which might do further re-encoding)

Let's unpack this a bit. This line states that translation between encodings is potentially happening at two places:

- First, the text you provided to `writeLines()` is translated to the *current*, or *native*, encoding;
- Next, that text is sent to the connection, which attempts to translate it back to the *requested* (in our case, `UTF-8`) encoding.

In other words, in our attempt to write UTF-8 content to a file, we unknowingly did a round trip through the system encoding! And, as you might imagine, this will fail if you happen to attempt to write characters that aren't representable in the user's system encoding – for example, Japanese characters in an English locale.

In other words, what we *really* want to do is:

1. Ensure that content is UTF-8 encoded;
2. Open a connection that skips any possible translation as well;
3. Ask `writeLines()` to skip any possible translation.

Opening a connection with `encoding = "native.enc"` makes step 2 possible, and invoking `writeLines(..., useBytes = TRUE)` makes step 3 possible. Let's re-write our `write_utf8()` routine once more with this in mind:

```
write_utf8 <- function(text, f = tempfile()) {

  # step 1: ensure our text is utf8 encoded
  utf8 <- enc2utf8(text)

  # step 2: create a connection with 'native' encoding
  # this signals to R that translation before writing
  # to the connection should be skipped
  con <- file(f, open = "w+", encoding = "native.enc")
```

```
    # step 3: write to the connection with 'useBytes = TRU
    # telling R to skip translation to the native encoding
    writeLines(utf8, con = con, useBytes = TRUE)

    # close our connection
    close(con)

    # read back from the file just to confirm
    # everything looks as expected
    readLines(f, encoding = "UTF-8")

}
```

How does this work, exactly? The `useBytes` argument of `writeLines()` effectively means, "pretend this text is in the native encoding, and perform no translation". Then, by opening a connection using `encoding = "native.enc"`, the connection receives data that it *assumes* is in the native encoding, and we have requested to translate to the native encoding. Since we're sending native encoding in, and requesting native encoding out, no translation is performed. R hints at this behavior in the **Encoding** section of `?file`:

> Additionally, "" and "native.enc" both mean the 'native' encoding, that is the internal encoding of the current locale and hence no translation is done.

But, again for emphasis, it's necessary to understand that translation can occur at two places:

1. First, most of R's routines for writing (e.g. `writeLines()`) will translate text to the native encoding unless you request otherwise;
2. Next, most R connections will *assume* they receive text in the native encoding, and so will translate to the encoding associated with the connection when possible.

What we're doing is effectively 'tricking' R into believing it's writing text in the native encoding, and through that trick, R skips any translation of the text we're sending to the connection.

You test this out on your Windows machine, and to your relief:

```
> write_utf8("brûlée")
[1] "brûlée"
```

```
> write_utf8("鬼")
[1] "鬼"
```

Unfortunately, your Japanese friend still reports an issue:

```
> write_utf8("brûlée")
[1] "br璢馥"
```

You decide to dive in and take a look. Armed with the knowledge of encodings, you decide to look at what R believes the encoding for "brûlée" is. Here's what you see in RStudio:

```
> Sys.setlocale(locale = "English")
[1] "LC_COLLATE=English_United States.1252;LC_CTYPE=English_United
> Encoding("brûlée")
[1] "latin1"
> charToRaw("brûlée")
[1] 62 72 fb 6c e9 65
> Sys.setlocale(locale = "Japanese")
[1] "LC_COLLATE=Japanese_Japan.932;LC_CTYPE=Japanese_Japan.932;LC_M
> Encoding("brûlée")
[1] "unknown"
> charToRaw("brûlée")
[1] 62 72 fb 6c e9 65
```

Interesting – for some reason, R is selecting the wrong encoding for the "brûlée" string literal in a Japanese locale. Unfortunately, I don't fully understand the mechanism by which R guesses the encoding of a string. But at least now we know "here be dragons" when it comes to the encoding of R string literals.

Fortunately, we can avoid this altogether by using unicode code points in the string literal explicitly:

```
> text <- "br\u00Fbl\u00e9e"
> text
[1] "brûlée"
> Encoding(text)
[1] "UTF-8"
```

There's one more thing we need to watch out for. It's common to write to files by name, rather than by explicitly opening and closing a connection. For example:

```
writeLines(text, file = "results/output.txt")
```

Behind the scenes, R will create a file connection for you, using the `file()` function. Let's look at its signature:

```
file
```

```
function (description = "", open = "", blocking = TRUE,
    raw = FALSE, method = getOption("url.method", "defau
{
    .Internal(file(description, open, blocking, encoding
        raw))
}
<bytecode: 0x7ffbd4067c98>
<environment: namespace:base>
```

That file connection gets opened with an *encoding*. That encoding is supplied by the `encoding` option:

```
getOption("encoding")
```

```
[1] "native.enc"
```

Now, the default value of that encoding is `native.enc`, but it's quite possible that users will have modified that option. However, when writing content you want to ensure that you're writing to a connection opened with `encoding = "native.enc"`. To that end you should *avoid writing to files by name, and instead explicitly open and close the connection yourself.*

Summary

1. Encodings are a specification for the translation of written language into sequences of bytes that can be processed and understood by a computer.

2. By default, when attempting to write to a connection, R will translate your strings through the system encoding. This translation can fail if you attempt to write UTF-8 content that is not representable in the system encoding.

3. This translation can be suppressed, through the use of the `useBytes = TRUE` argument to `writeLines()`, and by ensuring connections are opened using `encoding = "native.enc"`.

4. Since the encoding used by connections is configurable through the R `encoding` option, it is important to explicitly open and close connections with the `native.enc` encoding.

5. Portable R scripts should use unicode code points, to avoid accidental mis-encoding of string literals.

Learning More

- "What Every Programmer Absolutely, Positively Needs To Know About Encodings And Character Sets To Work With Text ". http://kunststube.net/encoding/

- "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)". https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/

Share and Follow

Tweet

Share

Comments

ALSO ON MUSINGS ON R AND C++

5 years ago • 1 comment
Parent Frames and Backwards ...

5 years ago • 2 comments
Pitfalls in Writing Portable

6 years ago
Debu Valgr and C

11 Comments Musings on R and C++ Login

Recommend 5 Tweet Share Sort by Best

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Aletta Smits • 2 years ago

Thank you so much!

• Reply • Share ›

Related Posts

The RStudio macOS Rendering Bug 09 Mar 2019

Parent Frames and Backwards Compatibility 04 Jan 2017

Pitfalls in Writing Portable C++98 Code 14 Sep 2016