Ruby master - Feature #18949
Deprecate and remove replicate and dummy encodings
07/29/2022 05:40 PM - Eregon (Benoit Daloze)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

Description
Ruby has a lot of accidental complexity.
Sometimes it becomes clear some features bring a lot of complexity and yet provide little value or are used very rarely.
Also most Ruby users do not even know about these features.
Replicate and dummy encodings seem to clearly fall into this category, almost nobody uses them but they add a significant complexity and also add a significant performance overhead.
Notably, the existence of those means the number of encodings in a Ruby runtime is actually variable and not fixed.
That means extra synchronization, hashtable lookups, indirections, function calls, etc.

Replicate Encodings
Replicate encodings are created using Encoding#replicate(name).
It almost sounds like an alias but in fact it is more than that and creates a new Encoding object, which can be used by a String:

```ruby
e = Encoding::US_ASCII.replicate('MY-US-ASCII')
s = "abc".force_encoding(e)
p s.encoding # => e
p s.encoding.name # => 'MY-US-ASCII'
```

This seems completely useless.
There is an obvious first step here which is to change Encoding#replicate to return the receiver, and just install an alias for it.
That avoids creating more encoding instances needlessly.

I think we should also deprecate and remove this method though, it is never a good idea to have a global mutable map like this.
If someone want extra aliases for encodings, they can easily to do so by having their own Hash: { alias => encoding }.fetch(name) { Encoding.find(name) }.

Dummy Encodings
Dummy encodings are not real encodings. They are artificial encodings designed to look like encodings, but don't function as encodings in Ruby.
From the docs:
```
enc.dummy? # true or false
```

Returns true for dummy encodings. A dummy encoding is an encoding for which character handling is not properly implemented. It is used for stateful encodings.

I wonder why we have those half-implemented encodings in core, it sounds to me like unfinished work which should not have been merged.

The "codepoints" of dummy encodings are just "bytes" and so they behave the same as Encoding::BINARY, with the exception of the UTF-16 and UTF-32 dummy encodings.

UTF-16 and UTF-32 dummy encodings
These two are special dummy encodings.
What they do is they scan the first 2 or 4 bytes of the String, and if those bytes are a byte-order mark (BOM),
the true "actual" encoding is resolved to UTF-16BE/UTF-16LE or UTF-32BE/UTF-32LE.
Otherwise, Encoding::BINARY is returned.
This logic is done by get_actual_encoding().
What is weird is this check is not done on String creation, no, it is done *every time* the encoding of that String is accessed (and the result is not stored on the String).
That is a needless overhead and really unreliable semantics.
Do we really want a String which automatically changes between UTF-16LE and UTF-16BE based on mutating its bytes? I think nobody wants that:

```ruby
s = "\xFF\xFE\x00\b\x00\d\x00".force_encoding("UTF-16")
p s #=> "\uFEFF\u6100\u6200\u6300\u6400"
```

I think the path is clear, we should deprecate and then remove Encoding::UTF_16 and Encoding::UTF_32 (dummy encodings).
And then we no longer need get_actual_encoding() and the overhead it adds to every String method.

We could also keep those constants and make them refer the native-endian UTF-16/32.
But that could cause confusing errors as we would change the meaning of them.
We could add Encoding::UTF_16NE / Encoding::UTF_16_NATIVE_ENDIAN if that is useful.

Another possibility would be to resolve these encodings on String creation, like:

```ruby
"\xFF\xFE".force_encoding("UTF-16").encoding #=> UTF-16LE
String.new("\xFF\xFE", encoding: Encoding::UTF_16).encoding #=> UTF-16LE
"ab".force_encoding("UTF-16").encoding # exception, not a BOM
String.new("ab", encoding: Encoding::UTF_16).encoding # exception, not a BOM
```

I think it is unnecessary to keep such complexity though.
A class method on String or Encoding like e.g. Encoding.find_from_bom(string) is so much clearer and efficient (no need to special case those encodings in String.new, #force_encoding, etc).

FWIW JRuby seems to use getActualEncoding() only in 2 places (scanForCodeRange, inspect), which is an indication those dummy UTF encodings are barely used if ever. Similarly, TruffleRuby only has 4 usages of GetActualEncodingNode.

**Existing dummy encodings**

```ruby
> Encoding.list.select(&:dummy?)

```

So besides UTF-16/UTF-32 dummy, it's only 7 encodings.
Does anyone use one of these 7 dummy encodings?

What is interesting to note, is that these encodings are exactly the ones that are also not ASCII-compatible, with the exception of UTF-16BE/UTF-16LE/UTF-32BE/UTF-32LE (non-dummy).
As a note, UTF-{16,32}{BE,LE} are ASCII-compatible in codepoints but not in bytes, and Ruby uses the bytes definition of ASCII-compatible.
There is potential to simplify encoding compatibility rules and encoding compatibility checks based on that.
So what this means is if we removed dummy encodings, all encodings except UTF-{16,32}{BE,LE} would be ASCII-compatible, which would lead to significant simplifications for many string operations which currently need to handle dummy encodings specially.
Unicode encodings like UTF-{16,32}{BE,LE} already have special behavior for some Ruby methods, so those are already handled specially in some places (they are the only encodings with minLength > 1).

```ruby
> Encoding.list.reject(&:ascii_compatible?)

```

What can we do with such a dummy non-ASCII-compatible encoding?
Almost nothing useful:
s = "abc".encode("IBM037")
> "\x81\x82\x83"
> s.bytes
> [129, 130, 131]
> s.codepoints
> [129, 130, 131]
> s == "abc"
> false
> "éte".encode("IBM037")
> "\x51\x51\x51"

So about the only thing that works with them is String#encode.

I think we could preserve that functionality, if actually used (does anyone use one of these 7 dummy encodings?), through:

> "éte".encode("IBM037")
=> "\x51\x51\x51" (.encoding == BIN)
> "\x51\x51\x51".encode("UTF-8", "IBM037") # encode from IBM037 to UTF-8
=> "éte" (.encoding == UTF-8)

That way there is no need for those to be Encoding instances, we would only need the conversion tables.

It is even better if we can remove them, so the notion of "dummy encodings" can disappear completely and nobody needs to understand or implement them.

**rb_define_dummy_encoding(name)**

The C-API has rb_define_dummy_encoding(const char *name).
This creates a new Encoding instance with dummy?=true, and it is also non-ASCII-compatible.
There seems to be no purpose to this besides storing the metadata of an encoding which does not exist in Ruby.
This seems a really expensive/complex way to handle that from the VM point of view (because it dynamically creates an Encoding and add it to lists/maps/etc).
A simple replacement would be to mark the String as BINARY and save the encoding name as an instance variable of that String.
Since anyway Ruby can't understand anything about that String, it's just raw bytes to Ruby's eyes.

**Summary**

I suggest we deprecate replicate and dummy encodings in Ruby 3.2.
And then we remove them in the next version.

This will significantly simplify string-related methods, and the behavior exposed to Ruby users.

It will also significantly speedup encoding lookup in CRuby (and other Ruby implementations).
With a fixed number of encodings we can ensure all encoding indices fit in 7 bits, and ENCODING_GET can be simply RB_ENCODING_GET_INLINED.
get_actual_encoding() will be gone and its overhead as well.
rb_enc_from_index() would be just return global_enc_table->list[index].enc; instead of the expensive behavior currently with GLOBAL_ENC_TABLE_EVAL which takes a lock and more when there are multiple Ractors.
Many checks in these methods would be removed as well.
Yet another improvement would be to load all encodings eagerly, that is small and fast in my experience, what is slow and big is the conversion tables, that'd simplify must_encindex() further.
These changes would affect most String methods, which use

STR_ENC_GET->get_encoding which does:
  get_actual_encoding->rb_enc_from_index and possibly ->enc_from_index
  ENCODING_GET->RB_ENCODING_GET_INLINED and possibly ->rb_enc_get_index->enc_get_index_str->rb_att
r->get

Some of these details are mentioned in https://github.com/ruby/ruby/pull/6095#discussion_r915149708.
The overhead is so large that it is worth handling some hardcoded encoding indices directly in String methods.
This feels wrong, getting the encoding from a String should be simple, straightforward and fast.

Further optimizations will be unlocked as the encoding list becomes fixed and immutable.
For example, the name-to-Encoding map is then immutable and could use perfect hashing.
Inline caching those lookups also becomes easier as the the map cannot change.
Also that map would no longer need synchronization, etc.
To Decide

Each item is independent. I think 1 & 2 are very important, 3 less but would be nice.

1. Deprecate and then remove Encoding#replicate and rb_define_dummy_encoding(). With that there is a fixed number of encodings, a lot of simplifications and many optimizations become available. They are used respectively in only 1 gem and 5 gems, see https://bugs.ruby-lang.org/issues/18949#note-4

2. Deprecate and then remove the dummy UTF-16 and UTF-32 encodings. This removes the need for get_actual_encoding() which is expensive. This functionality seems rarely used in practice, and it only works when such strings have a BOM, which is very rare.

3. Deprecate and then remove other dummy encodings, so there are no more dummy "half-implemented" encodings and all encodings are ASCII-compatible in terms of codepoints.

History

#1 - 07/29/2022 05:48 PM - byroot (Jean Boussier)

Based on my recent work on String#<< performance, I concur that encoding management often takes longer than the actual string operation, and that get_actual_encoding() is a very big offender.

I'm also curious what was the intended use for dummy encodings and whether it's actually in use somewhere.

So I'm in total support of this proposal.

#2 - 07/29/2022 08:17 PM - noahgibbs (Noah Gibbs)

I've been working on YJIT string handling. At this point our string handling is so simple that I don't think this would affect us either way. For encodings, we mostly call CRuby functions that handle them. So short-term, I don't think this would cause us any implementation difficulties.

#3 - 07/29/2022 11:53 PM - Eregon (Benoit Daloze)

noahgibbs (Noah Gibbs) wrote in #note-2:

I've been working on YJIT string handling. At this point our string handling is so simple that I don't think this would affect us either way. For encodings, we mostly call CRuby functions that handle them. So short-term, I don't think this would cause us any implementation difficulties.

Right, so this should also benefit YJIT as much as it benefits the CRuby interpreter performance-wise since it will make string/encoding-related functions faster.

I assume "would affect us either way/would cause us any implementation difficulties" means this change would be easy to adopt for YJIT, i.e., no or low effort but yet the performance gains would be valuable.

Since it simplify String encodings it benefits all Ruby implementations and all Ruby JITs.

#4 - 07/30/2022 12:57 PM - Eregon (Benoit Daloze)

I have looked at usage of Encoding#replicate and rb_define_dummy_encoding().

Encoding#replicate seems only used in 1 CRuby test, 1 ruby/spec file and exactly 1 gem (!), using the command below:

```ruby
$ gem-codesearch \'.replicate\b'
# I removed irrelevant matches
/srv/gems/ruby_encoding_iroha-0.1.0/lib/ruby_encoding_iroha.rb:Encoding::ASCII_8BIT.replicate("IROHA")
```

https://github.com/ima1zumi/ruby_encoding_iroha seems a demo gem (it's a self-made encoding, not a real one), from what I understand not something serious.

There was a talk related to it at RubyKaigi 2021: https://speakerdeck.com/ima1zumi/dive-into-encoding

That related encoding_iroha gem uses rb_encdb_replicate(), rb_declare_transcoder() and rb_register_transcoder() which are private APIs. rb_encdb_replicate() is only used by this gem.

To support new encodings properly, they should be added to Ruby, example. This has been done multiple times already (CESU-8, IBM720).

rb_define_dummy_encoding() is used in just a few gems:

```ruby
$ gem-codesearch rb_define_dummy_encoding
# I removed redundant matches and matches from CRuby */.c and header files
/srv/gems/jberkel-mysql-ruby-2.8.1/ext/mysql.c: enc_index = rb_define_dummy_encoding(aliases[0]);
/srv/gems/nkf-0.1.1/ext/nkf/nkf.c: idx = rb_define_dummy_encoding(name);
/srv/gems/pg-1.4.2/ext/pg.c: enc_index = rb_define_dummy_encoding(aliases[0]);
/srv/gems/tk-0.4.0/ext/tk/tcltklib.c: if (RTEST(rb_define_dummy_encoding(RSTRING_PTR(name)))) {
/srv/gems/win32ole-1.8.8/ext/win32ole/win32ole.c: idx = rb_define_dummy_encoding(enc_cstr);
```

So that is just 5 unique usages of rb_define_dummy_encoding() among all public gems.

08/14/2022
**jberkel-mysql-ruby** seems to be an old version of **ruby-mysql**, which no longer uses `rb_define_dummy_encoding()`.

*nkf* calls it in `rb_nkf_enc_get()`. `rb_nkf_enc_get()` seems only called with encodings that Ruby knows, so the `rb_define_dummy_encoding()` call seems dead/unreachable code.

I note it does refer to the existing dummy CP50220/CP50221 encodings: [code](https://github.com/ged/ruby-pg/pull/472).

*pg* uses it to *create the JOHAB encoding*. I will take a look at how to replace that usage: [https://github.com/ged/ruby-pg/pull/472](https://github.com/ged/ruby-pg/pull/472).

*tk* gem seems to *already have a fallback* for when `rb_define_dummy_encoding()` is not available. It might also be never reached but it is hard to tell from the code. Otherwise it could likely be replaced by the BINARY encoding.

*win32ole* uses it [here](https://github.com/ged/ruby-pg/pull/472). It could be dead code given this path is only taken for CP* encodings Ruby does not already know. Otherwise it could likely be replaced by the BINARY encoding.

---

**#5 - 07/31/2022 01:56 PM - Eregon (Benoit Daloze)**

- Description updated

**#6 - 07/31/2022 01:57 PM - Eregon (Benoit Daloze)**

I have updated the issue description with a To Decide section to summarize for the dev meeting: [https://bugs.ruby-lang.org/issues/18949#To-Decide](https://bugs.ruby-lang.org/issues/18949#To-Decide)

**#7 - 07/31/2022 02:00 PM - Eregon (Benoit Daloze)**

- Description updated

**#8 - 08/01/2022 07:01 AM - duerst (Martin Dürst)**

For `Encoding.list.filter { |e| e.dummy? }`, I get the following list:

```ruby
#<Encoding:UTF-16 (dummy)>,
#<Encoding:UTF-32 (dummy)>,
#<Encoding:IBM037 (dummy)>,
#<Encoding:ISO-2022-JP (dummy)>,
#<Encoding:ISO-2022-JP-2 (dummy)>,
#<Encoding:CP50220 (dummy)>,
#<Encoding:CP50221 (dummy)>,
#<Encoding:UTF-7 (dummy)>,
```

This is quite a diverse collection. I have no clue for example whether ISO-2022-JP-2 is, or ever was, in use. But on the other hand, ISO-2022-JP, UTF-16, and UTF-32 definitely have their uses. They are not so much used directly when processing strings (because indeed for these encodings, most string operations don't work, or don't work correctly), but they are used when transcoding to another encoding.

**#9 - 08/01/2022 05:33 PM - nirvdrum (Kevin Menard)**

But on the other hand, ISO-2022-JP, UTF-16, and UTF-32 definitely have their uses. They are not so much used directly when processing strings (because indeed for these encodings, most string operations don't work, or don't work correctly), but they are used when transcoding to another encoding.

I think UTF-16 and UTF-32 can be handled without general support for dummy encodings. They already receive special consideration in `get_actual_encoding`. What Benoit proposed was to allow those encodings, but resolve them to the correct endianness at string creation time. That way you don't have any dummy encodings floating through the runtime.

Can you please provide additional context on when ISO-2022-JP is used? As far as I can tell, you can't perform most string operations on a string with that encoding. I'm curious what the use cases are.

**#10 - 08/02/2022 01:04 AM - znz (Kazuhiro NISHIYAMA)**

I think ISO-2022-JP is used for mail archives, and IRCnet even now.

**#11 - 08/02/2022 01:48 AM - naruse (Yui NARUSE)**

- Status changed from Open to Rejected

String is a container and an encoding is a label of it. While data whose encoding is an encoding categorized in dummy encodings in Ruby, we cannot avoid such encodings.

**#12 - 08/06/2022 12:28 PM - Eregon (Benoit Daloze)**

- Status changed from Rejected to Open

This is a disappointing answer, it sounds like there was not enough time at the dev meeting to consider this issue properly maybe? (that's fine, but then don't reject please)

I reopen because this issue is still vastly unanswered.
Also it feels not so respectful to me to answer with a single short unclear sentence to a carefully-crafted issue. I estimate I have a lot of experience in this area since I reimplemented dummy/replicate encodings in TruffleRuby recently.


naruse (Yui NARUSE) wrote in #note-11:

String is a container and an encoding is a label of it. While data whose encoding is an encoding categorized in dummy encodings in Ruby, we cannot avoid such encodings.

So, there are 3 things we can decide, which I pointed out in https://bugs.ruby-lang.org/issues/18949#note-6 It seems you are against "3. Deprecate and then remove other dummy encodings". What about the other points, 1. and 2.?

Could you explain your point of view better?

Who uses dummy encodings and what for? Since no string operations really works on them, they are indeed nothing more than a label + the transcoding table.

I agree 3. is more difficult, and is less important than 1./2..

1. seems straightforward, there seems to be extremely few usages of it, and they can be replaced by Encoding::BINARY easily.

In fact Encoding#replicate is used in zero gems, so I think it's fair enough to deprecate and then remove it without further discussion.

2. I think we need to evaluate usages of the dummy UTF-16 and UTF-32 encodings, e.g., with gem-codesearch.

For better compatibility between Rubies we need to do 2. as well, as mentioned above the overhead and complexity for dummy UTF-16/UTF-32 is too much, and JRuby & TruffleRuby already do not deal with it for many String operations.

It is also clearly a significant performance cost for CRuby.

#13 - 08/06/2022 12:42 PM - Eregon (Benoit Daloze)

Usages of Encoding::UTF_32:

$ gem-codesearch 'Encoding::UTF_32\b'
/srv/gems/chupa-text-decomposer-html-1.0.3/test/test-html.rb: assert_equal([Encoding::UTF_32], decompose(@data))

Finally replaced by LB/BE: https://github.com/ranguba/chupa-text-decomposer-html/blob/4ede8e7cbb6bf7d6e6bf30b5b2e0d6b50338/test/test-html.rb#L249-L256
/srv/gems/flydata-0.8.10.2/flydata-core/lib/flydata-core/table_def/mysql_table_def.rb: "utf32" => Encoding::UTF_32, no repository available, seems just an incorrect mapping
/srv/gems/rbs-2.6.0/core/encoding.rbs:Encoding::UTF_32: Encoding
Just a mapping of Ruby Encoding constants, safe to remove
/srv/gems/rsense-core-0.6.6/stubs/1.8/_builtin.rb: Encoding::UTF_32, Encoding.new
Just a mapping of Ruby Encoding constants, safe to remove
/srv/gems/ruby-lint-2.6.6/stubs/1.8/_builtin.rb: Encoding::UTF_32 => Encoding::UTF_32
Just a mapping of Ruby Encoding constants, safe to remove
/srv/gems/ruby_encoding_iroha-0.1.0/lib/ruby_encoding_iroha.rb: Encoding::UTF_32, Test gem

Usages of Encoding::UTF_16:

$ gem-codesearch 'Encoding::UTF_16\b' | grep -v vendor

/srv/gems/code_owners-2.0.1/lib/code_owners.rb: input.encode!(Encoding::UTF_8, Encoding::UTF_16)
/srv/gems/dependabot-common-0.205.0/lib/dependabot/clients/azure.rb: pr_description = pr_description.dup.force_encoding(Encoding::UTF_16)
/srv/gems/dependabot-common-0.205.0/lib/dependabot/clients/azure.rb: truncated_msg = "...\n\n\n\nDescription has been truncated\n\n\n\n" + pr_description
/srv/gems/eightball-0.2.2/lib/eightball.rb: string = string.encode(Encoding::UTF_16, :invalid => :replace, :undef => :replace, :replace => :replace_string)
/srv/gems/encodable_validator-0.2.0/spec/encodable_validator_spec.rb: TestRecord.validate(:target_text, :encodable => {:encodings => [Encoding::UTF_16]})
/srv/gems/fastlane-2.208.0/fastlane_core/lib/fastlane_core/ui/implementations/shell.rb: return message.encode(Encoding::UTF_8, Encoding::UTF_16) if test_message.force_encoding(Encoding::UTF_16).valid_encoding?
/srv/gems/fastlane_hotfix-2.187.0/fastlane_core/lib/fastlane_core/ui/implementations/shell.rb: return message.encode(Encoding::UTF_8, Encoding::UTF_16) if test_message.force_encoding(Encoding::UTF_16).valid_encoding?
/srv/gems/flydata-0.8.10.2/flydata-core/lib/flydata-core/redshift/string.rb: string.encode(Encoding::UTF_16, options).encode(Encoding::UTF_8)
I think ISO-2022-JP is used for mail archives, and IRCnet even now.

AFAIK Encoding::ISO_2022_JP (in Ruby) is only useful to say "this String is encoded in ISO-2022-JP", but Ruby can't do anything with that except String#encode.

Is the use case then to String#encode to another encoding before any String operation?
Because all character/codepoints String operations are broken on ISO-2022-JP (and other dummy encodings). Could it be replaced by the BINARY encoding in programs using it?
Or just an instance variable on the String to remember its original encoding?
Or eagerly converting to a non-dummy encoding as soon as receiving ISO-2022-JP bytes from an external source?


Could it be replaced by the BINARY encoding in programs using it?
No, that would lose encoding information.

Or just an instance variable on the String to remember its original encoding?

Theoretically yes, but that would make dealing with encodings more complex.

Or eagerly converting to a non-dummy encoding as soon as receiving ISO-2022-JP bytes from an external source?

Theoretically yes, but that would again complicate code somewhere. We don’t just want to move around complexity, because it stays roughly the same, and moving it around is work and may introduce additional errors.

Eregon (Benoit Daloze) wrote in #note-13:

Usages of Encoding::UTF_32:
(details omitted)
Usages of Encoding::UTF_16:
(details omitted)

More usages, but all these usages seem incorrect, and should use either UTF_16LE or UTF_16BE.

What we probably need here is a custom deprecation messages for these two constants, to tell to use the LE/BE variants instead.

You are looking for Encoding::UTF_32 in code, but what’s more relevant is UTF-32 and UTF-16 in data. These are officially registered ‘charset’ labels, see https://www.iana.org/assignments/charset-reg/charset-reg.xhtml.

Eregon (Benoit Daloze) wrote in #note-12:

This is a disappointing answer, it sounds like there was not enough time at the dev meeting to consider this issue properly maybe? (that’s fine, but then don’t reject please)

I wasn’t at the meeting, so I can’t say.

I reopen because this issue is still vastly unanswered.
Also it feels not so respectful to me to answer with a single short unclear sentence to a carefully-crafted issue.

Naruse-san usually writes very short answeres, but that doesn’t mean he doesn’t understand the matter at hand.

I estimate I have a lot of experience in this area since I reimplemented dummy/replicate encodings in TruffleRuby recently.

(Naruse-san and others have a lot of experience because they implemented these encodings in CRuby.)

The dev meeting log has no notes about a discussion:

naruse (Yui NARUSE) wrote in #note-11:

String is a container and an encoding is a label of it. While data whose encoding is an encoding categorized in dummy encodings in Ruby, we cannot avoid such encodings.

So, there are 3 things we can decide, which I pointed out in https://bugs.ruby-lang.org/issues/18949#note-6
It seems you are against "3. Deprecate and then remove other dummy encodings".

What about the other points, 1. and 2.? 

For point 2, please see above. For 1, these functions are internal, and I guess the same thing could be done with some different functions, but as far as I understand, the functionality is needed for Ruby itself. And I don’t see the point of a fixed number of encodings. That it is possible to support new encodings may no longer be as important as it was 10 or 20 years ago, but may come in handy anyway.

Could you explain your point of view better?
Who uses dummy encodings and what for? Since no string operations really works on them, they are indeed nothing more than a label + the transcoding table.

Yes. That’s exactly what they are. And I have difficulties understanding why there would be a problem with this. Supporting a real encoding is real work. But supporting dummy encodings can’t be that much work, because there isn’t much functionality.

I agree 3. is more difficult, and is less important than 1./2..
1. seems straightforward, there seems to be extremely few usages of it, and they can be replaced by Encoding::BINARY easily. In fact Encoding#replicate is used in zero gems, so I think it's fair enough to deprecate and then remove it without further discussion.

enc_replicate is used internally as far as I understand. Maybe we can deprecate the Ruby method, but not the C function.

1. I think we need to evaluate usages of the dummy UTF-16 and UTF-32 encodings, e.g., with gem-codesearch.

See above.

For better compatibility between Rubies we need to do 2. as well, as mentioned above the overhead and complexity for dummy UTF-16:UTF-32 is too much, and JRuby & TruffleRuby already do not deal with it for many String operations. It is also clearly a significant performance cost for CRuby.

#16 - 08/08/2022 02:28 AM - nobu (Nobuyoshi Nakada)
Eregon (Benoit Daloze) wrote in #note-14:

Could it be replaced by the BINARY encoding in programs using it?

Yes, it could, and was exactly the way in 1.8 or earlier.
I don't want it again.

#17 - 08/08/2022 02:35 AM - nobu (Nobuyoshi Nakada)
Eregon (Benoit Daloze) wrote:

Replicate and dummy encodings seem to clearly fall into this category, almost nobody uses them but they add a significant complexity and also add a significant performance overhead.
Notably, the existence of those means the number of encodings in a Ruby runtime is actually variable and not fixed.
That means extra synchronization, hashtable lookups, indirections, function calls, etc.

These are not related to replicate/dummy encodings.

#18 - 08/08/2022 10:09 AM - Eregon (Benoit Daloze)
nobu (Nobuyoshi Nakada) wrote in #note-17:

Eregon (Benoit Daloze) wrote:

Replicate and dummy encodings seem to clearly fall into this category, almost nobody uses them but they add a significant complexity and also add a significant performance overhead.
Notably, the existence of those means the number of encodings in a Ruby runtime is actually variable and not fixed.
That means extra synchronization, hashtable lookups, indirections, function calls, etc.

These are not related to replicate/dummy encodings.

It very much is, as explained in the description (https://bugs.ruby-lang.org/issues/18949#Summary).
Replicate (Encoding#replicate) and extra dummy encodings (rb_define_dummy_encoding) cause a variable number of encodings, which means extra synchronization, hashtable lookups, indirections, function calls, etc.
A fixed number of encodings would simplify that a lot in complexity and in performance for getting the encoding of a String.
The dummy UTF-16:UTF-32 encodings cause the overhead of get_actual_encoding().
Other dummy encodings do not cause performance overheads (item 3. above).

#19 - 08/08/2022 10:38 AM - Eregon (Benoit Daloze)
duerst (Martin Dürst) wrote in #note-15:


Right, I wasn't sure this was what you meant there, thank you for confirming.

Or just an instance variable on the String to remember its original encoding?

Theoretically yes, but that would make dealing with encodings more complex.
Only for extremely-rarely used encodings that don't even exist in the list of 100+ encodings in Ruby by default, so I think that is reasonable.

Or eagerly converting to a non-dummy encoding as soon as receiving ISO-2022-JP bytes from an external source?

Theoretically yes, but that would again complicate code somewhere. We don't just want to move around complexity, because it stays roughly the same, and moving it around is work and may introduce additional errors.

AFAIK we know the encoding of an external string at the point it enters Ruby (e.g. reading from an external source). So associating the encoding (e.g. with force_encoding) must be done at that point. Eager conversion could also be done at that point with very little changes, and it would have the significant advantage that all String operations are actually implemented/correct and not some of them are broken (for Strings with a dummy encoding). 

In code, it'd means instead of io.gets.force_encoding("some-dummy-encoding") we'd use io.gets.encoder("UTF-8", "some-dummy-encoding").

You are looking for Encoding::UTF_32 in code, but what's more relevant is UTF-32 and UTF-16 in data. These are officially registered 'charset' labels, see https://www.iana.org/assignments/charset-reg/charset-reg.xhtml.

Do you know what this "UTF-16" charset does or mean? https://www.iana.org/assignments/charset-reg/UTF-16 does not say much. Is it always the native-endian UTF-16, does it use the BOM, what if there is no BOM? The reality AFAIK is there no such thing as an UTF-16 encoding without an endian, it's either UTF-16LE or UTF-16BE and the user should know. For the rare cases where it can be guessed from the BOM, we can add a convenience method to pick the right encoding between these two based on the BOM (as proposed in the description).

This is what Wikipedia says on the matter: https://en.wikipedia.org/wiki/UTF-16#Byte-order_encoding_schemes

Basically, we need the resolve the endian, before that we can't do anything and "UTF-16" is meaningless without a BOM or specifying LE/BE. Re-reading the BOM on every string operation is a needless overhead, it should be done once, and only paid if using UTF-16/UTF-32 strings with BOM.

Giving the dummy UTF-16/32 encoding to a String is almost always a mistake, for instance taking a substring of it with a non-zero offset will result as treating it like single-byte BINARY (because the BOM is gone then), which is clearly never what anyone wants. This resolution of the endian needs to be happen before the encoding is assigned to the String to be meaningful, so a UTF-* string always knows its endian.

For 1, these functions are internal, and I guess the same thing could be done with some different functions, but as far as I understand, the functionality is needed for Ruby itself. And I don't see the point of a fixed number of encodings. That it is possible to support new encodings may no longer be as important as it was 10 or 20 years ago, but may come in handy anyway.

Yes, the fact that e.g. enc_replicate is used internally is an implementation detail (and it could easily be changed to something else, the encoding indices could be determined statically, no need for dynamically growing the list of encodings, etc).

The point of a fixed number of encodings:

- We know the number of encodings, they can all fit in 7 bits, and so ENCODING_GET can be simply RB_ENCODING_GET_INLINED (no need for the complicated and slow fallback logic).
- We can store all encodings in a read-only global array, there is no need to handle resizing or anything
- We can optimize the lookup of encodings by name better, by having the map read-only and potentially use perfect-hashing
- No need to synchronize changes to these array and map.
- A significant amount of complexity can be removed in all Ruby implementations currently implementing replicate/extra dummy encodings.
- Other points mentioned in https://bugs.ruby-lang.org/issues/18949#Summary

Yes. That's exactly what they are. And I have difficulties understanding why there would be a problem with this. Supporting a real encoding is real work. But supporting dummy encodings can't be that much work, because there isn't much functionality.

Those dummy encodings expose broken behavior for String operations, so IMHO they hurt more than help.
But I can already see some people disagree on that, so let's postpone 3. for now and leave existing/predefined dummy encoding untouched.

#20 - 08/09/2022 06:29 AM - duerst (Martin Dürst)

Eregon (Benoit Dalouze) wrote in #note-18:

Replicate (Encoding#replicate) and extra dummy encodings (rb_define_dummy_encoding) cause a variable number of encodings, which means extra synchronization, hashtable lookups, indirections, function calls, etc.
A fixed number of encodings would simplify that a lot in complexity and in performance for getting the encoding of a String.

I think it should be okay to in your implementation assume a maximum number of encodings (currently, the number of actual encodings in CRuby is a bit more than 100, so a max of 127 or whatever might be a good choice). Then you don't need a hashtable, you can just use an array.

Regards, Martin.
Eregon (Benoit Daloze) wrote in #note-19:

duerst (Martin Dürst) wrote in #note-15:

Or just an instance variable on the String to remember its original encoding?

Theoretically yes, but that would make dealing with encodings more complex.

Only for extremely-rarely used encodings that don't even exist in the list of 100+ encodings in Ruby by default, so I think that is reasonable.

Having a special case that is used extremely rarely is prone to hidden bugs, and is therefore bad engineering.

It's unclear whether you want to have Ruby internally deal with such rare encodings in a special way, or whether you want the user to do that, but both are bad choices.

You are looking for Encoding::UTF_32 in code, but what's more relevant is UTF-32 and UTF-16 in data. These are officially registered 'charset' labels, see https://www.iana.org/assignments/charset-reg/charset-reg.xhtml.

Do you know what this "UTF-16" charset does or mean? https://www.iana.org/assignments/charset-reg/UTF-16 does not say much.

It references RFC 2781 (see https://www.rfc-editor.org/rfc/rfc2781). That provides quite a bit of details.

Is it always the native-endian UTF-16, does it use the BOM, what if there is no BOM? The reality AFAIK is there is no such thing as an UTF-16 encoding without an endian, it's either UTF-16LE or UTF-16BE and the user should know.

Basically, we need the resolve the endian, before that we can't do anything and "UTF-16" is meaningless without a BOM or specifying LE/BE.

[A string may be big-endian or little-endian, but it's a string'sendianness, not a strings 'endian'.]

Eregon (Benoit Daloze) wrote in #note-20:

duerst (Martin Dürst) wrote in #note-20:

I think it should be okay to in your implementation assume a maximum number of encodings (currently, the number of actual encodings in CRuby is a bit more than 100, so a max of 127 or whatever might be a good choice). Then you don't need a hashtable, you can just use an array.

That would get only a small part of the benefits, would introduce a weird arbitrary limit for Encoding#replicate/rb_define_dummy_encoding and would not remove the need for synchronization.

So I don't think that's a good choice in practice for any Ruby implementation.

Eregon (Benoit Daloze) wrote in #note-21:

duerst (Martin Dürst) wrote in #note-21:

It references RFC 2781 (see https://www.rfc-editor.org/rfc/rfc2781). That provides quite a bit of details.

Thanks for the link.

So that says strings in "UTF-16" (no explicit endian) SHOULD have a BOM, and if not SHOULD be interpreted as UTF-16BE (and the Wikipedia article links to some disagreement there, which means no BOM => undefined or would need to scan bytes which seems unreasonable and unreliable).

CRuby treats it as BINARY if there is no BOM, which seems to ignore this, but does not matter if we remove Encoding::UTF_16. The RFC more or less say "UTF-16 bytes have no meaning unless the endian is specified or there is a BOM". So Ruby should probably raise on dummy UTF-16 without BOM and force resolution at String creation time (described in resolve these encodings on String creation in the description).

It is much simpler and cleaner to not have Encoding::UTF_16 at all.

As we see in the list above, the dummy UTF-16 encoding seems extremely rarely used (https://bugs.ruby-lang.org/issues/18949#note-13), in practice users seem to already know to use UTF_16BE or UTF_16LE.

Even gem-codesearch "UTF-16" | grep `.rb' and gem-codesearch "UTF-16" | grep `.rb' are fairly short. If the encoding comes dynamically from an external source then it really ought to specify the endianess.

But to truly know, I think we need to deprecate dummy UTF-16/32 and see if someone really has a use-case where they cannot do without these. That seems clearly impossible if we provide a method to look at the BOM and return the encoding-with-endian, unless they actually want the BINARY
behavior which makes no sense.

#24 - 08/10/2022 04:23 AM - duerst (Martin Dürst)

@Eregon (Benoit Daloze): If you want, you can add this issue to the developers' meeting next week. I will be in Switzerland, but I should be able to participate. Maybe you can also participate?

Eregon (Benoit Daloze) wrote in #note-22:

duerst (Martin Dürst) wrote in #note-20:

I think it should be okay to in your implementation assume a maximum number of encodings (currently, the number of actual encodings in CRuby is a bit more than 100, so a max of 127 or whatever might be a good choice). Then you don't need a hashtable, you can just use an array.

That would get only a small part of the benefits, would introduce a weird arbitrary limit for Encoding#replicate/rb_define_dummy_encoding and would not remove the need for synchronization.

So I don't think that's a good choice in practice for any Ruby implementation.

Maybe you have different information, but the number of additional encodings people may create is very, very low. I guess I should know, because otherwise, people would ask me for additional transcodings.

#25 - 08/10/2022 10:20 AM - Eregon (Benoit Daloze)

duerst (Martin Dürst) wrote in #note-24:

@Eregon (Benoit Daloze):

If you want, you can add this issue to the developers' meeting next week. I will be in Switzerland, but I should be able to participate. Maybe you can also participate?

OK, I will add it.

BTW if you pass by Zurich I'd be happy to meet.

I can try to attend, however the meeting is 6am-10am local time, that is quite early for me.

Maybe we can discuss towards the end of the meeting i.e. 9am CEST/16 JST? I'll mention that on the dev meeting issue.

May have you different information, but the number of additional encodings people may create is very, very low. I guess I should know, because otherwise, people would ask me for additional transcodings.

#26 - 08/10/2022 04:12 PM - Dan0042 (Daniel DeLorme)

Eregon (Benoit Daloze) wrote in #note-25:

Can you clarify this please? With a max limit you need synchronization only when adding encodings. I hope you'll agree such a small level of complexity is not an issue. Apart from that, the two approaches should have the same complexity/overhead. So I don't understand why it would be "not nearly as helpful".

#27 - 08/10/2022 04:40 PM - Eregon (Benoit Daloze)

Dan0042 (Daniel DeLorme) wrote in #note-26:

Can you clarify this please? With a max limit you need synchronization only when adding encodings. I hope you'll agree such a small level of complexity is not an issue. Apart from that, the two approaches should have the same complexity/overhead. So I don't understand why it would be "not nearly as helpful".
In general there is no such thing as "synchronization only on writes, nothing on read", one would need something (some form of synchronization/checks) on reads too.

E.g. the array which holds the encodings by index could have volatile accesses but that's a big overhead as it prevents optimizations (if not volatile, stale reads are possible).

Or it could check if the read value is not NULL in a append-only array, but that's an extra cost.

Note the GVL does NOT help for these issues, and there is still locking needed in CRuby, since Encodings are shared across Ractors.

Perfect hashing as mentioned in [https://bugs.ruby-lang.org/issues/18949#Summary](https://bugs.ruby-lang.org/issues/18949#Summary) is also not possible with a dynamic number of encodings.

The map/Hash holding encoding names to encodings (used by Encoding.find) would need synchronization with a max limit, but would not with a fixed number of encodings.

But also I feel it's very bad semantics to specify "usages of Encoding#replicate and rb_define_dummy_encoding() are limited to 127-103=24 times, after that they raise an error".

Do you know any language or core method which says you can call me N times total (with N a small number) and after that it raises an exception?

Also I don't see the point to preserve functionality basically nobody uses (3 gems in the world, rb_define_dummy_encoding() only used in weird edge cases, and they can be replaced by BINARY easily: [https://bugs.ruby-lang.org/issues/18949#note-4](https://bugs.ruby-lang.org/issues/18949#note-4)).

I will go through these gems and make PRs to replace those usages.

#28 - 08/10/2022 09:37 PM - Dan0042 (Daniel DeLorme)

Thank you for the response.

It's true synchronization is required for a number of read operations, but I think the most important for performance, get_encoding, could become very fast with a max limit. But I digress.

More importantly I agree there's not much point to preserve functionality nobody uses. The flexibility is nice to have but doesn't seem worth the cost.

2¢