Ruby - Feature #18579

Concatenation of ASCII-8BIT strings shouldn't behave differently depending on string contents

02/09/2022 11:32 PM - tenderlovemaking (Aaron Patterson)

Status:	Rejected	
Priority:	Normal	
Assignee:		
Target version:		

Description

Currently strings tagged with ASCII-8BIT will behave differently when concatenating depending on the string contents.

When concatenating strings the resulting string has the encoding of the LHS. For example:

```
z = a + b
```

z will have the encoding of a (if the encodings are compatible).

However ASCII-8BIT behaves differently. If b has "ASCII-8BIT" encoding, then the encoding of z will sometimes be the encoding of a, sometimes it will be the encoding of b, and sometimes it will be an exception.

Here is an example program:

```
def concat a, b
  str = a + b
  str
end

concat "bar",                       "foo".encode("US-ASCII")
# Return value encoding is LHS, UTF-8
concat "bar".encode("US-ASCII"),  "foo".b
# Return value encoding is LHS, US-ASCII
concat "□□",                      "foo".b                    # Return value encoding is LHS, UTF-8
concat "bar",                       "bad\376\377str".b
# Return value encoding is RHS, ASCII-8BIT.  Why?
concat "□□",                      "bad\376\377str".b         # Exception
```

This behavior is too hard to understand. Usually we think LHS encoding will win, or there will be an exception. Even worse is that string concatenation can "infect" strings. For example:

```
def concat a, b
  str = a + b
  str
end

str = concat "bar", "bad\376\377str".b # this worked
p str
str = concat "□□", str               # exception
p str
```

The first concatenation succeeded, but the second one failed. As a developer it is difficult to find where the "bad string" was introduced. In the above example, the string may have been read from the network, but by the time an exception is raised it is far from where the "bad string" originated. In the above example, the bad data came from like 6, but the exception was raised on line 8.

I propose that ASCII-8BIT strings raise an exception if they cannot be converted in to the LHS encoding. So the above program would become like this:

```
def concat a, b
  str = a + b
  str
end

concat "bar",                       "foo".encode("US-ASCII")
# Return value encoding is LHS, UTF-8
```

```
concat "bar".encode("US-ASCII"),  "foo".b
# Return value encoding is LHS, US-ASCII
concat "□□",                      "foo".b                    # Return value encoding is LHS, UTF-8
concat "bar",                       "bad\376\377str".b         # Exception <--- NEW!!
concat "□□",                      "bad\376\377str".b         # Exception
```

I'm open to other solutions, but the underlying issue is that concatenating an ASCII-8BIT string with a non-ASCII-8BIT string is usually a bug and by the time an exception is raised, it is very far from the origin of the string.

History

#1 - 02/09/2022 11:37 PM - tenderlovemaking (Aaron Patterson)

Also, I think this case should raise an exception:

```
concat "□□",                    "foo".b                    # Return value encoding is LHS, UTF-8
```

But it's probably too aggressive. I think my proposal is not as aggressive but would still help to catch bugs.

#2 - 02/10/2022 12:20 AM - naruse (Yui NARUSE)

- Status changed from Open to Rejected

The encoding of the resulted string depends "ascii only or not" and "ascii compatibility".
The principle of the resulted encoding is

- if LHS is ascii only
  - if RHS is ascii only
    - LHS's encoding
  - else if RHS is ascii compatible
    - RHS's encoding
  - else (RHS is ascii incompatible)
    - exception.
- else if LHS is ascii compatible
  - if RHS is ascii only
    - LHS's encoding
  - else if RHS is ascii compatible
    - if LHS's encoding equals to RHS's encoding
      - LHS's encoding
    - else
      - exception
  - else (RHS is ascii incompatible)
    - exception.
- else if LHS is ascii incompatible
  - if LHS's encoding equals to RHS's encoding
    - LHS's encoding
  - else
    - exception

#3 - 02/10/2022 01:09 AM - tenderlovemaking (Aaron Patterson)

naruse (Yui NARUSE) wrote in #note-2:

> The encoding of the resulted string depends "ascii only or not" and "ascii compatibility".
> The principle of the resulted encoding is
>
> - if LHS is ascii only
>   - if RHS is ascii only
>   - LHS's encoding
>   - else if RHS is ascii compatible
>   - RHS's encoding
>   - else (RHS is ascii incompatible)
>   - exception.
> - else if LHS is ascii compatible
>   - if RHS is ascii only
>   - LHS's encoding
>   - else if RHS is ascii compatible
>   - if LHS's encoding equals to RHS's encoding
>   - LHS's encoding
>   - else
>   - exception
>   - else (RHS is ascii incompatible)
>   - exception.
> - else if LHS is ascii compatible

        - if LHS's encoding equals to RHS's encoding
            - LHS's encoding
        - else
            - exception

Is there anything we can do to simplify these rules? It's way too complicated to remember.

#4 - 02/10/2022 02:40 AM - mame (Yusuke Endoh)

Here is my more declarative understanding of the current behavior:

- (A) if LHS is "compatible" with RHS, the more "specific" encoding is picked
- (B) otherwise, an exception is raised

where "compatible" means:

- (C) both have the same encoding, or
- (D) both have ASCII-compatible encoding and at least one string consists of only ASCII characters
    - note that the actual encoding of the string that consists of only ASCII characters does not matter

where "specific" means:

- (E) if Rule D meets in one way (i.e., if the other string does NOT consist of only ASCII characters), the encoding of the other one is picked
- (F) if Rule D meets in two way (i.e., if both strings consist of only ASCII characters), LHS's encoding is picked by default

Examples:

```
# Both strings have ASCII-compatible encoding.
# Both strings DO consist of only ASCII characters.
# According to Rule A, D, and F, LHS's encoding (UTF-8) is picked.
concat "bar", "foo".encode("US-ASCII") # UTF-8

# Both strings have ASCII-compatible encoding.
# Both strings DO consist of only ASCII characters.
# According to Rule A, D, and F, LHS's encoding (UTF-8) is picked.
concat "bar".encode("US-ASCII"), "foo".b # US-ASCII

# Both strings have ASCII-compatible encoding.
# The former does NOT consist of only ASCII characters, and the latter DOES.
# According to Rule A, D, and E, the former's encoding (UTF-8) is picked.
concat "□□", "foo".b # UTF-8

# Both strings have ASCII-compatible encoding.
# The former DOES consist of only ASCII characters, and the latter does NOT.
# According to Rule A, D, and E, the latter's encoding (ASCII-8BIT) is picked.
concat "bar", "bad\376\377str".b # ASCII-8BIT

# Both strings have ASCII-compatible encoding.
# Both do NOT consist of only ASCII characters, so they are not compatible.
# According to Rule B, it raises an exception.
concat "□□", "bad\376\377str".b # Exception
```

Note that UTF-8, US-ASCII, and ASCII-8BIT are all "ASCII-compatible". This is why the name "ASCII-8BIT" includes "ASCII", as far as I understand.

#5 - 02/10/2022 03:14 AM - nirvdrum (Kevin Menard)

naruse (Yui NARUSE) wrote in #note-2:

> The encoding of the resulted string depends "ascii only or not" and "ascii compatibility".
> The principle of the resulted encoding is
>
> - if LHS is ascii only
>   - if RHS is ascii only
>   - LHS's encoding
>   - else if RHS is ascii compatible
>   - RHS's encoding
>   - else (RHS is ascii incompatible)
>   - exception.
> - else if LHS is ascii compatible
>   - if RHS is ascii only
>   - LHS's encoding
>   - else if RHS is ascii compatible
>   - if LHS's encoding equals to RHS's encoding
>   - LHS's encoding

        - else
            - exception
    - else (RHS is ascii incompatible)
        - exception.
- else if LHS is ascii compatible
    - if LHS's encoding equals to RHS's encoding
        - LHS's encoding
    - else
        - exception

The rules are actually a bit more complicated than that because empty strings get special treatment. The decision as two whether two strings are compatible involves checking their encodings, checking their code ranges, and possibly checking their contents.

```
a = 'abc'
b = 'def'.encode('UTF-32LE')

(a + b).encoding  # incompatible character encodings: UTF-8 and UTF-32LE (Encoding::CompatibilityError)


c = ''.encode('UTF-32LE')

(a + c).encoding # <Encoding:UTF-8>
```

I'm in favor of Aaron's proposal, providing it doesn't break the world. Unfortunately, many people don't really understand Ruby's various encodings and because many operations mask that fact, strings propagate with either unexpected encodings or with an ENC_CODERANGE_BROKEN code range. From an implementation point of view, a lot could be simplified if the rules only involved comparing the two encodings. All the various encoding compatibility checks could be reduced down to a lookup table and the most common pairings could be cached. The savings could be substantial as encoding compatibility checks occur all over the place.

The second best option from an implementation standpoint would involve looking at the encodings and code ranges. I think this covers almost all of the intended use cases. The downside is we'd potentially need to do a code range scan just to determine whether two strings are compatible. But, if the code range is already known, it'd be a fairly cheap check. The existing rules already consider code ranges, but the rules are rather difficult to follow. I think from an end-user point of view, it's next to impossible to keep them straight. As for the implementation, the checks aren't mutually exclusive so we have to run through each one in a waterfall pattern. Many of the early checks never amount to anything in the common case of an application using Ruby's default encodings, but need to be checked to maintain compatibility for more esoteric cases. I spent a lot of time trying to optimize these checks in TruffleRuby, but they can't be easily re-ordered.

Having to then also consider the string contents is another level of checking that I don't think really helps much. I can't tell if the empty string check is just an optimization to avoid code range scans in some situations, although a code range scan of an empty string should be rather quick. Otherwise, I don't know what the use case is in allowing two strings with otherwise incompatible encodings to appear as compatible if one of them is empty.

Assuming we retain code range checks, I'd really like to see ENC_CODERANGE_BROKEN result in failed compatibility checks. As far as I can tell, the encoding/string compatibility checks exist to help the user avoid confusing situations where concatenating two strings would result in a broken string. To do that, the encoding and the string contents are considered (both code range and byte size). If we know the encodings are incompatible, we don't allow the concatenation to proceed (ignoring the empty string case), presumably because the resulting string would be ENC_CODERANGE_BROKEN. But, if we concatenate two strings with the same encoding where one of them is already ENC_CODERANGE_BROKEN, the concatenation proceeds and the result is ENC_CODERANGE_BROKEN. To the extent that these checks are intended to help the user from doing something that will be difficult to debug later on, I don't see why one case is allowed and the other is not.*

* - One odd case is concatenating two strings that are both ENC_CODERANGE_BROKEN could result in a string that is ENC_CODERANGE_VALID. I think the only way that happens in practice is if someone is reading binary data from I/O and setting the bytes directly into a UTF-8 string or maybe prematurely calling String#force_encoding. While that currently works today, it looks to me like broken code. Preventing such concatenations from proceeding would make code range scans considerably cheaper and I suspect would result in more robust user code.

#6 - 02/10/2022 03:27 AM - naruse (Yui NARUSE)

I think Aaron can implement what provides such encoding principle.

I think it will break the world while the implementation still provides M17N. If it gives up to support legacy encodings and only support UTF-8, it will become something like what Python 3 provides after you fix network related code.

#7 - 02/10/2022 07:25 AM - duerst (Martin Dürst)

tenderlovemaking (Aaron Patterson) wrote in #note-3:

> naruse (Yui NARUSE) wrote in #note-2:
>
> > The encoding of the resulted string depends "ascii only or not" and "ascii compatibility".
> > The principle of the resulted encoding is
> >
> > - if LHS is ascii only
> >   - if RHS is ascii only
> >   - LHS's encoding
> >   - else if RHS is ascii compatible

        - RHS's encoding
      - else (RHS is ascii incompatible)
        - exception.
  - else if LHS is ascii compatible
    - if RHS is ascii only
      - LHS's encoding
    - else if RHS is ascii compatible
      - if LHS's encoding equals to RHS's encoding
        - LHS's encoding
      - else
        - exception
    - else (RHS is ascii incompatible)
      - exception.
  - else if LHS is ascii compatible
    - if LHS's encoding equals to RHS's encoding
      - LHS's encoding
    - else
      - exception

> Is there anything we can do to simplify these rules? It's way too complicated to remember.

First, it should be "* else if LHS is ascii incompatible" in the third bullet on the top level.

Second, it's not really that difficult to understand what happens. An encoding defines a mapping from a sequence of bytes to a sequence of characters. What concatenation does is that it concatenates the sequence of bytes and the sequence of characters at the same time, if it can easily know that that's possible (*). If not, then an error is produced.

(*) The reason for writing "if it can easily know" is because there are cases such as the following:

```
"résumé".encode('iso-8859-1') + "résumé".encode('iso-8859-2')
```

Because "é" is encoded with the same byte in both iso-8859-1 and iso-8859-2, this concatenation would in theory work with the above rules, but it's too much to teach the implementation about which 8-bit bytes,... code for the same character in which encoding.

#8 - 02/10/2022 07:33 AM - duerst (Martin Dürst)

tenderlovemaking (Aaron Patterson) wrote:

> I propose that ASCII-8BIT strings raise an exception if they cannot be converted in to the LHS encoding. So the above program would become like this:
>
> ```
> def concat a, b
>   str = a + b
>   str
> end
>
> concat "bar",                       "foo".encode("US-ASCII")    # Return value encoding is LHS, UTF-8
> concat "bar".encode("US-ASCII"),  "foo".b                       # Return value encoding is LHS, US-ASCII
> concat "□□",                      "foo".b                     # Return value encoding is LHS, UTF-8
> concat "bar",                       "bad\376\377str".b          # Exception <--- NEW!!
> concat "□□",                      "bad\376\377str".b          # Exception
> ```
>
> I'm open to other solutions, but the underlying issue is that concatenating an ASCII-8BIT string with a non-ASCII-8BIT string is usually a bug and by the time an exception is raised, it is very far from the origin of the string.

I agree that concatenating an ASCII-8BIT string with a non-ASCII-8BIT string is usually a bug. That's because ASCII-8BIT usually stands for BINARY. But if it stands for BINARY, then also

```
concat "bar".encode("US-ASCII"),  "foo".b                       # Return value encoding is LHS, US-ASCII
concat "□□",                      "foo".b                     # Return value encoding is LHS, UTF-8
```

should produce an error, because "foo" in that case is just a way to show some byte values, and doesn't actually represent the three letters "f", "o", and "o".

So I think we either have to keep the idea that the first 128 values of ASCII-8BIT/BINARY mean ASCII characters, or we have to abandon that idea, but we have to be consistent.

#9 - 02/10/2022 02:07 PM - Eregon (Benoit Daloze)

In short, it's not specific to the binary encoding at all.
It's the same behavior with e.g. UTF-8 + ISO-8859-1.

Another way to express the rules is "if the String only has 7-bit ASCII characters", treat it as if it was US-ASCII.
Appending US-ASCII characters to any ASCII-compatible string is of course fine and vice versa.
So basically concatenation works whenever it can currently, and that's the current model for users.
It only raises if the encodings are not the same and (both sides have non-7-bit characters OR one side is ascii-incompatible).
The most specific encoding is picked if there is a choice (US-ASCII + UTF-8 -> UTF-8 is natural, isn't it?), and if all other things equal it'd be the LHS encoding.

There is a weird edge case which @nirvdrum (Kevin Menard) mentions, that empty strings are considered CR_7BIT even if their encoding is not ascii-compat, I think we should try to remove that special case.

The bigger question is whether Encoding::BINARY should be ascii-compatible.
It always was, and I think changing that would break the world, but might still be worth an experiment.
Maybe a command-line flag to debug bad BINARY usages would be helpful, and that could e.g. raise on concatenating a string of any other encoding with a BINARY String.

I think part of the issue is that:

```
"abc".b + "été" # => UTF-8 and not BINARY
# same for
"abc".b << "été"
```

which relates to #14975.

Maybe anything + BINARY -> BINARY and BINARY + anything -> BINARY would be more intuitive/safer, but it would also just make the rules more complicated, isn't it?
Also that would accept anything on the other side, while currently it raises which seems more helpful ("abé".b + "été" # => Encoding::CompatibilityError).

Avoiding binary Strings in core, for example for exception messages (using US-ASCII or UTF-8 instead) is I think something worth doing independently.
TruffleRuby already uses UTF-8 String for exception messages.

#10 - 02/10/2022 04:34 PM - tenderlovemaking (Aaron Patterson)

duerst (Martin Dürst) wrote in #note-8:

> I agree that concatenating an ASCII-8BIT string with a non-ASCII-8BIT string is usually a bug. That's because ASCII-8BIT usually stands for BINARY. But if it stands for BINARY, then also
>
> ```
> concat "bar".encode("US-ASCII"),  "foo".b                    # Return value encoding is LHS, US-ASCII
> concat "□□",                      "foo".b                    # Return value encoding is LHS, UTF-8
> ```
>
> should produce an error, because "foo" in that case is just a way to show some byte values, and doesn't actually represent the three letters "f", "o", and "o".
>
> So I think we either have to keep the idea that the first 128 values of ASCII-8BIT/BINARY mean ASCII characters, or we have to abandon that idea, but we have to be consistent.

I agree those two cases should raise an exception, but I thought a less aggressive proposal would have a greater than 0 chance of being accepted □□

> Maybe a command-line flag to debug bad BINARY usages would be helpful, and that could e.g. raise on concatenating a string of any other encoding with a BINARY String.

This could probably satisfy my needs. As I said, the underlying problem is that a binary string can enter the application, and due to the "infectious nature" of the encoding system, an exception can get raised but it is *far* from the origin of the binary string.

The binary string could have come from reading a file, reading from the network, decoding a URI or an HTML form (arbitrary binary data can be percent encoded), etc. In a large system it's hard to tell what the origin of the binary string is, and the correct fix is to patch the code at the origin of the string.