Ruby master - Feature #14975

String#append without changing receiver's encoding

08/08/2018 03:15 AM - ioquatix (Samuel Williams)

Status:	Open	
Priority:	Normal	
Assignee:	ioquatix (Samuel Williams)	
Target version:		

Description

I'm not sure where this fits in, but in order to avoid garbage and superfluous function calls, is it possible that String#<<, String#concat or the (proposed) String#append can avoid changing the encoding of the receiver?

Right now it's very tricky to do this in a way that doesn't require extra allocations. Here is what I do:

```
class Buffer < String
    BINARY = Encoding::BINARY

    def initialize
        super

        force_encoding(BINARY)
    end

    def << string
        if string.encoding == BINARY
            super(string)
        else
            super(string.b) # Requires extra allocation.
        end

        return self
    end

    alias concat <<
end
```

When the receiver is binary, but contains byte sequences, appending UTF_8 can fail:

```
"Foobar".b << "Føøbar"
=> "FoobarFøøbar"

> "Føøbar".b << "Føøbar"
Encoding::CompatibilityError: incompatible character encodings: ASCII-8BIT and UTF-8
```

So, it's not possible to append data, generally, and then call force_encoding(Encoding::BINARY). One must ensure the string is binary before appending it.

It would be nice if there was a solution which didn't require additional allocations/copies/linear scans for what should basically be a memcpy.

See also: https://bugs.ruby-lang.org/issues/14033 and https://bugs.ruby-lang.org/issues/13626#note-3

There are two options to fix this:

1/ Don't change receiver encoding in any case.
2/ Apply 1, but only when receiver is using Encoding::BINARY

History

#1 - 08/08/2018 03:35 AM - ioquatix (Samuel Williams)

One other option would be to add String#append which preserves encoding.

#2 - 08/08/2018 08:23 PM - shevegen (Robert A. Heiler)

I think this should be filed under Feature simply because it may change
existing functionality.

#3 - 08/08/2018 11:36 PM - nobu (Nobuyoshi Nakada)

- Backport deleted (2.3: UNKNOWN, 2.4: UNKNOWN, 2.5: UNKNOWN)

- Tracker changed from Bug to Feature

#4 - 08/09/2018 06:17 AM - ioquatix (Samuel Williams)

nobu (Nobuyoshi Nakada) Thanks for updating.

#5 - 08/09/2018 09:03 AM - naruse (Yui NARUSE)

The motivation sounds reasonable.
Changing String#<< and String#concat is not acceptable.
Adding String#append is no idea; up to matz. (maybe byte-* name is better?)

#6 - 08/09/2018 10:20 AM - naruse (Yui NARUSE)

Just FYI, StringIO behaves what you want.

```
irb(main):002:0> require'stringio'
=> false
irb(main):004:0> sio=StringIO.new("".b)
=> #<StringIO:0x00007faf3312cca0>
irb(main):005:0> sio << "□□□"
=> #<StringIO:0x00007faf3312cca0>
irb(main):006:0> sio << "□□□".encode("eucjp")
=> #<StringIO:0x00007faf3312cca0>
irb(main):008:0> sio.string
=> "\xE3\x81\x82\xE3\x81\x84\xE3\x81\x86\xA4\xA2\xA4\xA4\xA4\xA6"
irb(main):009:0> sio.string.encoding
=> #<Encoding:ASCII-8BIT>
```

#7 - 08/09/2018 11:25 AM - ioquatix (Samuel Williams)

I didn't know about StringIO being an option. I will investigate it.

#8 - 08/09/2018 11:27 AM - ioquatix (Samuel Williams)

> Changing String#<< and String#concat is not acceptable.

Since the current behaviour is underspecified, I feel like we should at least consider it as an option.

Practically speaking, I don't expect the encoding to change on the receiver. It's surprising, unexpected behaviour.

I actually think that the arguments should be converted into the encoding of the receiver and appended, not changing the receivers encoding. In the case the conversion fails, it should throw an exception.

#9 - 08/10/2018 01:16 AM - shyouhei (Shyouhei Urabe)

ioquatix (Samuel Williams) wrote:

> > Changing String#<< and String#concat is not acceptable.
>
> Since the current behaviour is underspecified, I feel like we should at least consider it as an option.
>
> Practically speaking, I don't expect the encoding to change on the receiver. It's surprising, unexpected behaviour.
>
> I actually think that the arguments should be converted into the encoding of the receiver and appended, not changing the receivers encoding. In the case the conversion fails, it should throw an exception.

Though perhaps not documented at the String#<<'s place, this is not how our M17N is designed in principle. There is no automatic encoding conversion. Programmers should be aware of the fact that encodings do exist, and should treat them by their own. If you want to change this you need a concrete reason why, not just because it seems handy.

That said, you are not requesting to "convert" the string contents but just ignore their encodings. This is possible I think, depending on operations.

So my advice here is: don't expand your battle front. Stick to the idea to push the feature you want, that is, appending arbitrary string to a binary.

#10 - 08/10/2018 01:32 AM - ioquatix (Samuel Williams)

That makes total sense, so it seems like we are in agreement, when the receiver is BINARY, we can append anything to it without changing the encoding. What do you think? I can make PR.

#11 - 08/10/2018 03:02 AM - jeremyevans0 (Jeremy Evans)

Making String#<< not change the encoding of the receiver may cause backward compatibility issues (raising an exception where an exception is not currently raised):

```
b = 'a'.force_encoding('BINARY')
u = "\u00ff".force_encoding('UTF-8')

# This currently results in b having UTF-8 encoding, since b is 7-bit.
# With this proposal, b would have BINARY encoding.
b << u

# This doesn't raise exception now, since b is now has UTF-8 encoding.
# With this proposal, an exception would be raised, since the
# encodings are not compatible can b is no longer 7-bit.
u << b
```

I think it may be worth adding support for this feature, but it should probably be a new method (maybe #bconcat since #b already returns a copy with binary encoding) to not break backwards compatibility.

#12 - 08/10/2018 03:50 AM - ioquatix (Samuel Williams)

jeremyevans0 (Jeremy Evans) I agree with you. It's a problem.

Just for completeness, here is the error you talk about:

```
b = 'a'.force_encoding(Encoding::BINARY)
u = "\u00ff".force_encoding(Encoding::UTF_8)

b << u
b.force_encoding(Encoding::BINARY)

# Encoding::CompatibilityError: incompatible character encodings: UTF-8 and ASCII-8BIT
u << b
```

IMHO, anyone relying on this behaviour is walking on fire. But, you are right, there is the potential to break existing code. I believe the correct solution is for people to avoid using binary buffers for this use case. There already exists Encoding::ASCII which would make more sense. So if we limited to Encoding::BINARY it at least has a specific semantic model. One way to fix the above, would be to turn the Encoding::UTF_8 receiver into Encoding::BINARY. I'm not sure I like that solution, but it does work in a predictable way and avoids introducing exceptions where none existed before.

Do you think there is a way we can find a compromise? I'd rather not add yet another string concatenation function. I sort of admire Ruby for being opinionated, so I think if we can find a solution here without adding more options/arguments/methods, that would be ideal. WDYT?

#13 - 08/10/2018 04:23 AM - jeremyevans0 (Jeremy Evans)

ioquatix (Samuel Williams) wrote:

> IMHO, anyone relying on this behaviour is walking on fire. But, you are right, there is the potential to break existing code. I believe the correct solution is for people to avoid using binary buffers for this use case. There already exists Encoding::ASCII which would make more sense. So if we limited to Encoding::BINARY it at least has a specific semantic model. One way to fix the above, would be to turn the Encoding::UTF_8 receiver into Encoding::BINARY. I'm not sure I like that solution, but it does work in a predictable way and avoids introducing exceptions where none existed before.

The problem with that is that single BINARY encoding string can then cause havok in your application, since any string it touches would then become binary. It would defeat the purpose of having encodings at all.

> Do you think there is a way we can find a compromise? I'd rather not add yet another string concatenation function. I sort of admire Ruby for being opinionated, so I think if we can find a solution here without adding more options/arguments/methods, that would be ideal. WDYT?

I don't think modifying String#<< should be considered, it's too likely to break existing code.

I'm fairly sure you could add a non-allocating binary append method using a C extension, assuming the receiving string has enough capacity, so this would not even necessarily require a core method. Actually, if you could even get non-allocating code in pure ruby, but it wouldn't be great for performance (for small strings, for large strings it's probably fine):

```
    def << string
        enc = encoding
        string_enc = string.encoding
        if string_enc != enc
            force_encoding(string_enc)
            super(string)
            force_encoding(enc)
        else
            super(string)
        end
    end
```

That being said, I'd be in favor of having this as a separate String method. I can see it being used quite a lot in network programming where encoded text is mixed with binary protocol logic.

#14 - 08/10/2018 04:33 AM - ioquatix (Samuel Williams)

The logic you've given above is too idealistic unfortunately:

```
[11] pry(main)> x = "Foobar".freeze

=> "Foobar"
[12] pry(main)> x.force_encoding(Encoding::BINARY)

FrozenError: can't modify frozen String
from (pry):10:in `force_encoding'
```

The best you can hope for is something like https://github.com/socketry/async-io/blob/master/lib/async/io/buffer.rb#L32-L40

```
def << string
    if string.encoding == BINARY
        super(string)
    else
        super(string.b)
    end

    return self
end
```

It's even possible that string.b doesn't allocate memory... but I don't actually know.

Maybe a better option is for

```
# Encoding::CompatibilityError: incompatible character encodings: UTF-8 and ASCII-8BIT
u << b
```

to convert the RHS to the LHS encoding - if it can't then it fails. WDYT?

#15 - 08/10/2018 04:39 AM - ioquatix (Samuel Williams)

> The problem with that is that single BINARY encoding string can then cause havok in your application, since any string it touches would then become binary. It would defeat the purpose of having encodings at all.

In what situations can you have a binary string without explicitly choosing it? I'm not aware of any way to make it unless you explicitly choose it.

A binary buffer is a very special use case. It's not served very well by current String methods. And you are right, it makes networking a pain.

> I don't think modifying String#<< should be considered, it's too likely to break existing code.

If we find a change which doesn't break existing code, could it be considered?

As you say, perhaps a new method. String#append which doesn't change receiver encoding and efficiently appends binary data. One of these options would be awesome.

#16 - 08/10/2018 05:51 AM - ioquatix (Samuel Williams)

So, as a preface: without introducing (yet another) method, we are left with changing the behaviour of existing methods. I will leave that decision up to someone who is closer to Ruby. However, in an effort to move this forward, I've made a PR.

https://github.com/ruby/ruby/pull/1926

I know it's controversial. I think it's a good solution. All Ruby tests passed. matz (Yukihiro Matsumoto) what do you think? In the past it seemed like you thought it was a good idea.

[UPDATE] for some reason make test on my local system, all tests based, but on CI, it failed. I will check.

#17 - 08/10/2018 05:52 AM - ioquatix (Samuel Williams)

Just to make it crystal clear, changing the behaviour of rb_str_append affects all other functions which call it. That includes concat, << and so on.

An alternative would be to make rb_str_append2 and use that instead and only expose this behaviour through String#append. That being said, I'm okay with either option. But, I like how Ruby is opinionated. I like to say "If string is binary, all operations to append to it will result in binary" rather than "If string is binary, this one specific function has this slightly different behaviour". I think it's easier to understand.

#18 - 08/10/2018 06:19 AM - ioquatix (Samuel Williams)

So, I think what happened was I ran make test on older trunk. I rebased and changed implementation to be rb_str_append2 as mentioned above. So, this logic only affects String#append. I still wonder if we can make it work in general case, but at least this achieves what I want with minimal intrusive changes.

So, both options are on the table, I leave it up to someone else to make a final decision on what's best for Ruby.

#19 - 08/10/2018 06:40 AM - ioquatix (Samuel Williams)

One more thing to consider. If we add rb_str_append2, should we be more strict? Rather than doing implicit conversion, perhaps we should throw error if encodings are not the same.

#20 - 08/10/2018 01:39 PM - jeremyevans0 (Jeremy Evans)

ioquatix (Samuel Williams) wrote:

> The logic you've given above is too idealistic unfortunately:
>
> ```
> [11] pry(main)> x = "Foobar".freeze
>
> => "Foobar"
> [12] pry(main)> x.force_encoding(Encoding::BINARY)
>
> FrozenError: can't modify frozen String
> from (pry):10:in `force_encoding'
> ```

Considering:

```
x = "Foobar".freeze
x << "1"
# FrozenError (can't modify frozen String)
```

Can you explain how the logic is too idealistic?

#21 - 08/10/2018 02:01 PM - ioquatix (Samuel Williams)

It's not the receiver that's frozen, it's the argument. If the argument is frozen, force_encoding will fail. Not only that, it could introduce race conditions in true multi-threaded interpreters.

#22 - 08/10/2018 02:03 PM - ioquatix (Samuel Williams)

Oh, I see, you are changing the encoding of the buffer to match the argument, then appending it. Interesting, I didn't think of that. I only tried it the other way around so I misread your code. Anyway, I don't think it should be required, but yeah, it's an interesting idea to explore.

#23 - 08/27/2018 09:46 AM - Eregon (Benoit Daloze)

I agree the current behavior for appending with binary strings is surprising.
Especially since the result of the operation depends on whether the receiver and argument are 7-bit or not.

However, it makes sense to me to change the receiver encoding in a case such as

```
us_ascii_string + utf8_string
```

Maybe this new method be named "byteappend" since it doesn't care about encodings if the receiver is binary?

Is there a use-case for using this new method with non-binary strings?
If not, it might be better to have the method raise on non-binary receivers.

#24 - 08/27/2018 10:54 AM - ioquatix (Samuel Williams)

> Is there a use-case for using this new method with non-binary strings?

Yes, predictably appending strings without changing receiver encoding. If I make a buffer with a specific encoding, I'd prefer not to have it change encoding without me realising it.

> If not, it might be better to have the method raise on non-binary receivers.

As long as receiver and argument are same encoding, it's acceptable to append them.

My current feeling is that if we add a new #append method, it should be very strict, as is currently implemented. By being strict, it can be more efficiently implemented, and have more predictable behaviour.

#25 - 08/29/2018 10:16 AM - duerst (Martin Dürst)

As mentioned, the general idea of Ruby m17n is that strings that only contain ASCII bytes (8th bits are all 0) are treated as US-ASCII and can be combined with any ASCII-compatible encoding (taking on that encoding as a result).

The problem with this is that the encoding of the first truly non-ASCII string wins, and the second such string (assuming it's in a different encoding) produces an error. I'm not exactly sure this is consistent with the higher-level policy of failing early in case of encoding mixes, but I think it may be difficult to change.

In any way, when adding stuff to a (BINARY) buffer, the right thing conceptually is to change all the pieces to BINARY, not to rely on some of the pieces (be it the first or another) to be BINARY.

The problem with that is that it either changes the appended string's encoding (with .force_encoding 'BINARY') or needs another copy (with .b). What I always wished we had is a method that forces the encoding of a string only locally, without leaking.

One way to realize this might be the following (Ruby as pseudocode):

```
class String
  alias :old_force_encoding :force_encoding

  def force_encoding (encoding)
    if block_given?
      enc = encoding         # remember current encoding
      old_force_encoding encoding
      yield
      old_force_encoding enc # set encoding back
    else
      old_force_encoding encoding
    end
  end
end
```

This then would be used e.g. as follows:

```
b = 'a'.force_encoding('BINARY')
u = "\u00ff".force_encoding('UTF-8')    # aside: force_encoding here is a no-op,
                                        # because \u automatically produces UTF-8
u.force_encoding('BINARY') do
  b << u
end
```

That in my opinion would be the conceptually right way to do things.

What remains is that << for buffers is in many cases not very efficient; it can be way more efficient to collect the Strings to be appended in an array and then do a join. So we should not only think about this issue for <</concat/append, but also for more wholesale methods.

#26 - 08/29/2018 11:08 AM - ioquatix (Samuel Williams)

```
b = 'a'.force_encoding('BINARY')
u = "\u00ff".force_encoding('UTF-8')    # aside: force_encoding here is a no-op,
                                        # because \u automatically produces UTF-8
u.force_encoding('BINARY') do
  b << u
end
```

duerst (Martin Dürst) How would that work if u was frozen?

#27 - 08/29/2018 11:22 AM - duerst (Martin Dürst)

ioquatix (Samuel Williams) wrote:

> ```
> u.force_encoding('BINARY') do
>   b << u
> end
> ```
>
> duerst (Martin Dürst) How would that work if u was frozen?

Didn't think about it, sorry. But it's only pseudocode. The non-block #force_encoding will definitely not work on frozen strings, but the version with a block could be made to work because it's only temporarily pretending that the string has another encoding. In the end, I think frozen strings could be handled. The real problem is concurrency; the pseudocode would definitely not work if multiple threads had access to u.

#28 - 12/28/2018 01:22 PM - ioquatix (Samuel Williams)

- Target version set to 2.7

- Assignee set to ioquatix (Samuel Williams)

#29 - 12/28/2018 05:11 PM - Eregon (Benoit Daloze)

duerst (Martin Dürst) wrote:

> In any way, when adding stuff to a (BINARY) buffer, the right thing conceptually is to change all the pieces to BINARY, not to rely on some of the pieces (be it the first or another) to be BINARY.

If the LHS is not BINARY then it can't reasonably be called a "binary buffer", so let's assume LHS is binary.

I think we all agree binary_string << whatever should leave binary_string.encoding as BINARY (even if binary_string only contains all <128 bytes initially).
I wonder what would break if we changed this specific behavior.
Maybe somebody could try and report what failures we get from the test suites?
Things like binary << utf8 changing binary's encoding to UTF-8 seems nonsense and never what anyone wants (UTF-8 is not a superset of BINARY, some byte sequences are invalid in UTF-8).

OTOH usascii << utf8 changing the LHS's encoding to UTF-8 seems much more reasonable (it's still "text", not binary data, and UTF-8 is a clear superset of US-ASCII).

Maybe negotiating a compatible encoding by finding a superset (although I'd raise on ISO-8859-1 << UTF-8 rather than use BINARY, because it cannot work well if both have non-US-ASCII characters),
and only considering the String#encoding and not the coderange (the range of the bytes) would be a way to have clear semantics for appends.

> The problem with that is that it either changes the appended string's encoding (with .force_encoding 'BINARY') or needs another copy (with .b).

If #append is changed or a new method added, there is no need for any extra copying of course, the bytes are just copied in the receiver buffer.
But the allocation should be pretty cheap (and it can be escape-analyzed), and no copy or scan should be needed for super(string.b).

#30 - 12/28/2018 05:16 PM - Eregon (Benoit Daloze)

As a note, it seems semantically a << b is the same as a.replace(a + b).
I think this is good to keep, and therefore the discussion also applies to the resulting encoding of String#+.

#31 - 01/14/2019 07:33 AM - naruse (Yui NARUSE)

- Target version deleted (2.7)

#32 - 01/26/2019 11:02 AM - ioquatix (Samuel Williams)

naruse (Yui NARUSE) do you think we can make some progress on this issue for 2.7 or not?