Ruby master - Feature #2034

Consider the ICU Library for Improving and Expanding Unicode Support

09/03/2009 10:36 AM - runpaint (Run Paint Run Run)

Status:	Rejected
Priority:	Normal
Assignee:	naruse (Yui NARUSE)
Target version:	2.6

Description

=begin
Has consideration been recently given to employing the ICU library (http://site.icu-project.org/) in Ruby? The bindings are in C and the library mature. My ignorance of the Ruby source not withstanding, this would allow existing String methods, among others, to support non-ASCII characters in an incremental manner.

For a trivial example, consider String#to_i. It currently understands only ASCII characters which represent digits. ICU provides a u_charDigitValue(code_point) function which returns the integer corresponding to the given Unicode codepoint. Were String#to_i to use this, it would work with non-ASCII counting systems, thus removing at least one of the "as long as it's ASCII" caveats currently associated with String methods.

More generally, if it's desirable for String methods to properly support Unicode, and if the principle barrier is the difficulty of the implementation, then might there be at least a partial solution in marrying Ruby with ICU?

If ICU is unfeasible, I'd appreciate understanding why. There are multiple approaches to what I term the second phase of Unicode support in Ruby, and it will be easier to choose between them if I understand the constraints. :-) (Of course, if a direction has already been determined, and work on it is underway, I will gladly bow out ;-)).
=end

Related issues:

Related to CommonRuby - Feature #10084: Add Unicode String Normalization to S...	**Closed**	**07/23/2014**
Related to CommonRuby - Feature #10085: Add non-ASCII case conversion to Stri...	**Closed**	

History

#1 - 09/04/2009 01:30 AM - runpaint (Run Paint Run Run)

=begin

> If ICU is unfeasible, I'd appreciate understanding why.

> It converts everything to UTF-16 internally.

Thank you, Nikolai. I understand how that would make its conversion or transliteration APIs problematic, but for property lookup/resolution would it not be easier to use its functions that accept a codepoint than write our own? Or is it just a downside of the CSI model that String methods can't work correctly with Unicode? (As ever, I apologise for using the BTS to ask questions, but I don't know where else I can).
=end

#2 - 09/04/2009 03:35 AM - naruse (Yui NARUSE)

- Status changed from Open to Assigned

=begin
The main reason is resource.
We worked around encoding independent area so far.

Anyway, if we have ICU library, we still have some problems:

- dependency
- non-Unicode
- where to implement

First, the dependency to ICU seems a problem.
ICU is well portable library, but Ruby is also portable.
We may be in trouble in some environment.
Moreover the same version of Ruby but another version of ICU environment
will be hard to support.

So core String class using ICU seems difficult.
Bundled some library seems more acceptable.

Second, non-Unicode encodings may be a problem.
But regexp works differently between Unicode and non-Unicode now.
So methods of some ICU wrapper can't work with non-Unicode strings is acceptable.

Third, where to implement maybe the largest problem.
As I stated, implement to String class is hard to accept.
So the wrapper will be another class or module.
Naming problem around APIs will occur.
=end

#3 - 09/05/2009 05:26 AM - runpaint (Run Paint Run Run)

=begin
Thank you, naruse.

> First, the dependency to ICU seems a problem.
> ICU is well portable library, but Ruby is also portable.
> We may be in trouble in some environment.

If ICU was not available could we not fall back to the current behaviour?

> Moreover the same version of Ruby but another version of ICU environment
> will be hard to support.

If Ruby tracks the stable build of ICU is this likely to be a problem. I imagine that the main ICU updates come when a new version of Unicode is released.

> Second, non-Unicode encodings may be a problem.
> But regexp works differently between Unicode and non-Unicode now.
> So methods of some ICU wrapper can't work with non-Unicode strings is acceptable.

Indeed. The fallback to ASCII-only semantics is always available, and mirrors how we handle non-ASCII-compatible encodings now.

> Third, where to implement maybe the largest problem.
> As I stated, implement to String class is hard to accept.
> So the wrapper will be another class or module.
> Naming problem around APIs will occur.

Well, String would be ideal. We would have the current function and a Unicode function, then choose between them when the method was invoked.

If that isn't possible, has the idea of String::Unicode been considered? I assume it would be too major of a change.

Another approach would be a 'unicode' library in stdlib that when loaded monkey-patched String. I don't favour this, however, because it seems hacky; String should work transparently with Unicode, IMO.

I suppose there are two distinct questions here:

1) Should String methods work correctly with Unicode by default?
2) If so, do we hand-roll a solution or use ICU?

Assuming we answer (1) in the affirmative, I'd be surprised if re-implementing parts of ICU took fewer resources than using it directly.

Perhaps the next stage is to investigate what methods in String need to be changed, and to what extent?
=end

#4 - 09/05/2009 06:39 PM - naruse (Yui NARUSE)

=begin

> If ICU was not available could we not fall back to the current behaviour?

No, if so, it lose portability of scripts.
Codes will depend both Ruby's version and whether ICU is installed or not.
Codes wrote with ICU runs different in without ICU.
This will be a trouble maker.

> Well, String would be ideal.

> We would have the current function and a Unicode function,
> then choose between them when the method was invoked.

Adding Unicode sensitive functions to String may be accepted.
For example if the library is loaded, String#unicode_to_i is available.

> If that isn't possible, has the idea of String::Unicode been considered?
> I assume it would be too major of a change.

This seems hard to accept.
Ruby uses large class: don't split classes if they are similar.

> Another approach would be a 'unicode' library in stdlib that when loaded monkey-patched String.
> I don't favour this, however, because it seems hacky; String should work transparently with Unicode, IMO.

This is a approach 1.8's jcode.rb used.
Of course, it's not good.

> I suppose there are two distinct questions here:
>
> 1) Should String methods work correctly with Unicode by default?

NO.
Ruby's core methods treat ASCII as special.
For example, variables naming rule #1853, \s \w \d of regexp, and so on.

> 2) If so, do we hand-roll a solution or use ICU?

I won't write from scratch, but if Martin does, I don't object.
How about Martin?

> Perhaps the next stage is to investigate what methods in String need to be changed, and to what extent?

Matz said Ruby's core methods are ASCII sensitive, are't Unicode.
So exist methods won't change.
Unicode sensitive methods will be added as another methods.
=end

#5 - 09/06/2009 12:05 AM - runpaint (Run Paint Run Run)

=begin

> Adding Unicode sensitive functions to String may be accepted.
> For example if the library is loaded, String#unicode_to_i is available.

Do you like that API? It feels clumsy to me. Users will have to change their code just to handle Unicode strings, when Ruby could make the determination automatically.

> Matz said Ruby's core methods are ASCII sensitive, are't Unicode.
> So exist methods won't change.

:-(

FWIW, Python uses http://hg.python.org/cpython/file/a31c1b2f4ceb/Tools/unicode/makeunicodedata.py to make the C header files, e.g. http://hg.python.org/cpython/file/a31c1b2f4ceb/Modules/unicodedata_db.h and http://hg.python.org/cpython/file/a31c1b2f4ceb/Modules/unicodename_db.h then http://hg.python.org/cpython/file/a31c1b2f4ceb/Modules/unicodedata.c as the API. IOW, a generalised approach of the Onigurma patch.

It would take me a while, but if Martin doesn't have the time, I may be able to produce something along those lines, albeit without the clever optimisations initially.
=end

#6 - 09/06/2009 12:13 AM - runpaint (Run Paint Run Run)

=begin
Another data point: Perl 6 is optionally linking against ICU (http://github.com/rakudo/rakudo/tree).
=end

#7 - 09/06/2009 12:33 AM - hramrach (Michal Suchanek)

=begin
I cannot speak for Ruby core team but I personally do not like ICU.

It's not that I am against Unicode but Unicode is only part of what
Ruby aims to support yet ICU is huge and written in C++ which would
drastically enlarge Ruby core size and reduce portability.

Better support for character classes outside of the ASCII range is
certainly desirable but it requires more design and planning than
"Hey, I saw ICU, it's cool, let's use it".

First the questions

- What exactly we want supported for what purposes?
- What the cost would be?
- Do we really want to pay that cost?

have to be answered.

There are clearly multiple options and I haven't gathered enough data
to even make an opinion what would be good option for Ruby.

Thanks

Michal

=end

#8 - 09/07/2009 01:38 AM - naruse (Yui NARUSE)

=begin

> Adding Unicode sensitive functions to String may be accepted.
> For example if the library is loaded, String#unicode_to_i is available.

Do you like that API? It feels clumsy to me.

No, I think so too.

> Users will have to change their code just to handle Unicode strings, when
> Ruby could make the determination automatically.

Those Unicode sensitive methods leave as is for compatibility
even if Ruby could make the determination automatically.

> Matz said Ruby's core methods are ASCII sensitive, are't Unicode.
> So exist methods won't change.

:-(

Ruby core methods are for programers.
So naming collisions are resolved for programers side.

> FWIW, Python uses .. as the API. IOW, a generalised approach of the Onigurma patch.

Thanks,

> It would take me a while, but if Martin doesn't have the time, I may be able to
> produce something along those lines, albeit without the clever optimisations initially.

If you do, I'll discuss about this and its API on next developper's meeting.

> Another data point: Perl 6 is optionally linking against ICU (http://github.com/rakudo/rakudo/tree).

Ruby hates compile options which change behaviors over Ruby Layer.
So those options are hard to be merged.

> I cannot speak for Ruby core team but I personally do not like ICU.
>
> It's not that I am against Unicode but Unicode is only part of what
> Ruby aims to support yet ICU is huge and written in C++ which would
> drastically enlarge Ruby core size and reduce portability.

Yes, so if we use ICU, the library will be a bundled or external library, not core.

> Better support for character classes outside of the ASCII range is
> certainly desirable but it requires more design and planning than
> "Hey, I saw ICU, it's cool, let's use it".

true.

> First the questions
>
> - What exactly we want supported for what purposes?

Unicode Normalization or other conversions, Unicode sensitive case convertions or matches
and so on are the base framework of current internet specs.
For exmaple, if uri library supports IRI, they need those functions.

> - What the cost would be?

Human resource and distribution size.

> - Do we really want to pay that cost?

YES, more and more specs depend on Unicode.
=end

#9 - 10/19/2009 01:03 AM - pedz (Perry Smith)

=begin
I discovered ICU and ICU4R back in 2007 and I just now moved it to
Ruby 1.9. I'm a pretty big advocate of using ICU. There is nothing
that has as many encodings as ICU to my knowledge. It is the only one
that addresses many of the EBCDIC encodings (of which there 147 some
odd of them).

The reason I came to use ICU is the application I'm working on needs
to translate EBCDIC encoded Japanese characters to something a browser
can use such as utf-8. ICU is the only portable library that I found
and it is also the only library that had the encodings that I needed.

I'm assuming a few things here. One is that this:

http://yokolet.blogspot.com/2009/07/design-and-implementation-of-ruby-m17n.html

is accurate for the most part. In particular, this paper seems to say
that there is choice between a UCS model and an CSI model and Ruby 1.9
has choosen CSI. From my perspective, a CSI model should be an
envelope around a UCS model.

My background is working inside IBM for 20+ years and I've bumped into
multi-byte language issues since 1989. I'm not an expert by any
meaans but I have seen IBM struggle with this for decades.

Perhaps only IBM and legacy IBM applications have these issues. I
simply don't know but I will say that all of the other open source
language encoding implementations are very small in the number of
encodings they do compared to what you see when dealing with legacy
international applications.

In the text below, I will use "aaa" to represent a string using an
encoding of A, "bbb" will represent a string using an encoding of B,

and so on. I will also simply put B to stand for encoding B.

I believe that the CSI model is a great choice: why translate everything all the time? If an application is going to read data and write it back out, translating it is both a waste of time and error prone.

I believe the implementors of a UCS model fall back and say that if the application is going to compare strings they must be in a common encoding -- Ruby agrees with this point. And, they also would argue that if you want to translate "aaa" into B, it is simply more practical to go to a common encoding C first. Then you have only 2N encoders instead of N^2 encoders. To me, that argument is very sound. If plausible, I would allow specific A to B translators to be plugged in.

The key place where I believe Ruby's choice of a CSI model wins is the fact that there are a lot of places that data can be used and manipulated without translation. Keeping and using the CSI model in all those places is a clear win. In all those places, the data is opaque; it is not interpreted or understood by the application.

Opaque data can be compared for equality as Ruby appears to be doing now -- the two strings must have the same encoding and byte for byte compare as equal.

Technically, opaque data can be concatenated and spliced as well. This is one place that Ruby's 1.9 implementation surprised me a bit. It could be that "aaa" + "bbb" yields String that is a list of SubStrings. I'll write as x = ["aaa", "bbb"]. That would have many useful concepts: length would be the sum of the length of all the SubStrings. x[1] would be "a". x[4] would be "b". x[2,2] would yield a String with two SubStrings (again, this is just how I'm representing it) ["a", "b"]. x.encoding would return Mixed in these cases. Encoding would be a concept attached to a SubString rather than String. x.each would return a sequence of "a", "a", "a", "b", "b", "b" each with a encoding of A for the "a"s and B for the "b"s. String would still be what most applications use. Rarely would they need to know about the SubStrings.

Many text manipulations can be done with opaque data because the characters themselves are still not being interpreted by the application. To the application are just "dodads" that those human guys know about. I believe that if Ruby wants to hold strongly to the CSI model that encoding agnostic string manipulations should be implemented.

The places where the actual characters are "understood" by an application is for sorting (collation) and if, for some external reason, they need to be translated to a particular encoding.

Sorting not only depends upon the encoding but also the language. Sorting could be done with routines specific to an encoding plus language but I believe that is impractical to implement. Utopia would be the ability to plug (and grow) sort routines that would be specific to the encoding and language with a fall back going to a sort routine tailored for the language and a common encoding such as UTF-16 and if the language was not known (or implemented), fall back to sorting based upon just the encoding, and if that was not available, fall back to a sort based upon a common encoding.

As has been pointed out already, the String#to_i routine needs to be encoding savvy. There are probably a few more methods that need to be encoding savvy.

The translations, collations, and other places that characters must be understood by the application is where I believe using ICU is a hugh win. ICU should not be used all the time because most of the time, no undetanding of the characters are needed by the application. But if translation or collation are needed, ICU is a hugh repository that is already implemented and available.

I have not seen arguments aginst ICU that I believe hold much weight. It is more portable than any iconv implementation (because iconv has been stuff into the libc implemntation and pulling it back apart

looked really hard to me). The fact that it is hugh is just a
reflection of the size of the problem.

=end

#10 - 10/19/2009 04:42 AM - naruse (Yui NARUSE)

=begin

> needs to translate EBCDIC encoded Japanese characters

What is the encoding and do you the converter for the encoding should be included?
I guess, the converter can convert the encoding to EUC-JP by an algorithm.

> I'm assuming a few things here. One is that this:
> http://yokolet.blogspot.com/2009/07/design-and-implementation-of-ruby-m17n.html
> is accurate for the most part.

I wrote it.

> It could be that "aaa" + "bbb" yields String that is a list of
> SubStrings. I'll write as x = ["aaa", "bbb"]. That would have many
> useful concepts: length would be the sum of the length of all the
> SubStrings. x[1] would be "a". x[4] would be "b". x[2,2] would yield
> a String with two SubStrings (again, this is just how I'm representing
> it) ["a", "b"]. x.encoding would return Mixed in these cases.
> Encoding would be a concept attached to a SubString rather than
> String. x.each would return a sequence of "a", "a", "a", "b", "b",
> "b" each with a encoding of A for the "a"s and B for the "b"s. String
> would still be what most applications use. Rarely would they need to
> know about the SubStrings.

That consept is sometimes introduced as rope.

http://jp.rubyist.net/magazine/?0022-RubyConf2007Report#l13
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.9450
http://www.kmonos.net/wlog/39.php#_1841040529 in Japanese
http://d.hatena.ne.jp/ku-ma-me/20070730/p1 in Japanese

Rope is:

- fast string concatnation
- fast substring get
- can't change substring
- slow index access to a character

But Ruby's string is mutable.
This seems a critical issue for rope.
Moreover Ruby users often use regexp match to strings.
I don't think rope has enough merit to implement despite such tough environment.

> I believe that if Ruby wants to hold strongly to the
> CSI model that encoding agnostic string manipulations should be
> implemented.

Ruby is practical languate, although Ruby use CSI model :-)
In current situation, such consept is hard to realize in Ruby
because of performance, difficulty of implementation, and needs.

> Sorting not only depends upon the encoding but also the language.
> Sorting could be done with routines specific to an encoding plus
> language but I believe that is impractical to implement.

Yes, String needs language.
This is an open problem.
We may have to implement rope for languages.

> It is more portable than any iconv implementation (because iconv has
> been stuff into the libc implemntation and pulling it back apart
> looked really hard to me).

For String, core of Ruby, iconv is out of the question.
Core library and its dependency must as portable as Ruby.

> The fact that it is hugh is just a reflection of the size of the problem.

I think the problem is too heavy to treat by current Ruby.
And Ruby 1.9 uses CSI model; it is beyond ICU, which uses UCS model.
=end

#11 - 10/20/2009 02:27 PM - duerst (Martin Dürst)

=begin
Hello Perry,

On 2009/10/19 1:03, Perry Smith wrote:

> Issue #2034 has been updated by Perry Smith.
>
> I discovered ICU and ICU4R back in 2007 and I just now moved it to
> Ruby 1.9. I'm a pretty big advocate of using ICU. There is nothing
> that has as many encodings as ICU to my knowledge. It is the only one
> that addresses many of the EBCDIC encodings (of which there 147 some
> odd of them).

It's no surprise that ICU is strong on EBCDIC. ICU started at IBM, and
IBM still contributes a lot :-). [If IBM contributed on Ruby, Ruby may
also be stronger on EBCDIC.]

> The reason I came to use ICU is the application I'm working on needs
> to translate EBCDIC encoded Japanese characters to something a browser
> can use such as utf-8. ICU is the only portable library that I found
> and it is also the only library that had the encodings that I needed.

Can you tell me what encodings exactly you need? And which of them are
table based? (see also Yui's message) We can definitely have a look at them.

One big problem with ICU is that it is UTF-16-based, whereas Ruby
(mainly) uses UTF-8 for Unicode. But fortunately, there are exceptions.
I learned just last week at the Internationalization and Unicode
conference that there is now a purely UTF-8 based sorting routine in
ICU. I think it may make sense for Ruby to try and extract it.

> I'm assuming a few things here. One is that this:
>
> http://yokolet.blogspot.com/2009/07/design-and-implementation-of-ruby-m17n.html
>
> is accurate for the most part. In particular, this paper seems to say
> that there is choice between a UCS model and an CSI model and Ruby 1.9
> has choosen CSI. From my perspective, a CSI model should be an
> envelope around a UCS model.

Can you explain what you mean by 'envelope around UCS model'?

The way I understand your "envelope around UCS model" is that it's easy
to use an UCS model inside Ruby CSI; the main thing you have to do is to
use the -U option. But maybe you meant something different?

> I believe the implementors of a UCS model fall back and say that if
> the application is going to compare strings they must be in a common
> encoding -- Ruby agrees with this point. And, they also would argue
> that if you want to translate "aaa" into B, it is simply more
> practical to go to a common encoding C first. Then you have only 2N
> encoders instead of N^2 encoders. To me, that argument is very
> sound. If plausible, I would allow specific A to B translators to be
> plugged in.

Ruby allows this. It's actually used e.g. for Shift_JIS <-> EUC-JP
translation. The reason to use it is that it allows to transcode
"gaiji", at least to a certain extent.

> The key place where I believe Ruby's choice of a CSI model wins is the fact that there are a lot of places that data can be used and manipulated without translation. Keeping and using the CSI model in all those places is a clear win. In all those places, the data is opaque; it is not interpreted or understood by the application.
>
> Opaque data can be compared for equality as Ruby appears to be doing now -- the two strings must have the same encoding and byte for byte compare as equal.
>
> Technically, opaque data can be concatenated and spliced as well. This is one place that Ruby's 1.9 implementation surprised me a bit.

Yes, you can take the CSI model further and further. But you will always bump into problems where encodings do not match sooner or later. (btw, in Ruby, you can concatenate as long as the data is in an ASCII-compatible encoding and is ASCII-only.)

> It could be that "aaa" + "bbb" yields String that is a list of SubStrings. I'll write as x = ["aaa", "bbb"].

On the file level, this would be similar to having a file with internal change of character encoding. At the very, very early stages of Web internationalization, some people proposed such a model, but the Web went a different way. And so went most if not all text editors, you can't have a file with many different encodings at the same time. Sure file encodings and internal encodings work a bit differently, but it's not a disadvantage if those two models match.

> The places where the actual characters are "understood" by an application is for sorting (collation) and if, for some external reason, they need to be translated to a particular encoding.

There's lots more cases. In particular regular expressions. Even with Ruby's current model, it took a long time to smooth the edges.

> Sorting not only depends upon the encoding but also the language.

Yes, but please note that sorting depends on encoding in completely different ways than on language. For language, what counts is not the language of the text being sorted, but the language of the user.

Let's say you have two words, a Swedish one (översätter, to translate), and a German one (öffnen, to open). Swedish sorts 'ö' after 'z', German sorts 'ö' with 'o', taking the difference between the two just as a secondary difference (i.e. to order words with 'o' and 'ö', first look at the rest of the word, and only if the rest of the word is identical, then order the word with 'ö' after the word with 'o').

So some people argue that in an alphabetical list, the two words above should be ordered (with some others thrown in) as follows:

abstract
nominal
öffnen (German, so goes into the 'o' section)
often
substring
xylophone
zebra
översätter (Swedish, so goes after 'z')

But this is wrong. There should be (at least) two sort orders for the above data, one for Swedish and the other for German:

Swedish sort order:

abstract
nominal
often
substring
xylophone

zebra
öffnen (all 'ö's go after 'z')
översätter

German sort order:

abstract
nominal
öffnen (all 'ö's go with 'o')
often
översätter
substring
xylophone
zebra

So there is no need for sorting to know the language of the data.

> Sorting could be done with routines specific to an encoding plus language but I believe that is impractical to implement. Utopia would be the ability to plug (and grow) sort routines that would be specific to the encoding and language with a fall back going to a sort routine tailored for the language and a common encoding such as UTF-16 and if the language was not known (or implemented), fall back to sorting based upon just the encoding, and if that was not available, fall back to a sort based upon a common encoding.

If you think this is necessary, please start implementing. In my opinion, it will take you a lot of time, with very little advantage over a single-encoding sorting implementation.

> As has been pointed out already, the String#to_i routine needs to be encoding savvy. There are probably a few more methods that need to be encoding savvy.

Lots of places can be made more encoding-savy. But overall, I think concentrating on getting more functionality for UTF-8 strings, and transcoding to UTF-8 for heavy functionality, is the way to go.

Regards, Martin.

--
#-# Martin J. Dürst, Professor, Aoyama Gakuin University
#-# http://www.sw.it.aoyama.ac.jp mailto:duerst@it.aoyama.ac.jp

=end

#12 - 10/21/2009 10:09 AM - pedz (Perry Smith)

=begin
I will try and answer both of the posts above.

Mostly, you both asked about which encodings. As absured as this may sound, I don't know.

When I fetch text from the legacy system, it has a two byte CCSID in front of it. I have a table that translates the CCSID to the name of the encoding. It is much like:

http://www-01.ibm.com/software/globalization/ccsid/ccsid_registered.jsp

I then translate the text to ICU's internal format which is UTF-16. Later, I translate it to UTF-8 because that seems to work with browsers.

I have no idea if ICU does this using tables or what. My belief is that it does not go to EUC-JP. I also do not know but I assume that many of these are not single byte

I do know that for Japanese text, usually the page text is encoded using IBM-939. Most English is in IBM-037. But the system I'm interfacing to is used world wide and I would assume that other code pages are used.

> Can you explain what you mean by 'envelope around UCS model'?

I think, based upon the reply, that you understand but to restate it:
use ICU everywhere you can but when you are forced to translate, do it
to and from some common (probably third) encoding. It just seems like
it would be much easier to do that.

> On the file level, this would be similar to having a file with internal
> change of character encoding. At the very, very early stages of Web
> internationalization, some people proposed such a model, but the Web
> went a different way. And so went most if not all text editors, you
> can't have a file with many different encodings at the same time. Sure
> file encodings and internal encodings work a bit differently, but it's
> not a disadvantage if those two models match.

I was imagining doing this only for the internal encodings. I later
mentioned that translations must be done for external reasons. I
meant that the translation would be done when going to a file or to
any external data stream.

> rope

I see this as an incorrect name which may be why it has attributes
that we do not want. To me, rope is made up of many strings. But I
wanted String made up of many -- e.g. SubStrings.

Strings would still be mutable in my scheme. It seems plausible that
a data structure could be devised that would yield the Nth character
is the same time as the current implementation. It may require more
space.

> Regexp

Yes. I totally forgot about Regexp's.

There is one thing that confused me at the end of Martin's post. To
me, data never has a language. Perhaps I'm mistaken. The data only
have a language when viewed by a user. As he points out, a sort can
only be properly done when the language of the user is taken into
account. At least, that is how I would rephrase what he said.

Am I missing a subtlety there?

=end

#13 - 10/25/2009 02:58 AM - naruse (Yui NARUSE)

- Target version set to 3.0

=begin

> If you think this is necessary, please start implementing. In my
> opinion, it will take you a lot of time, with very little advantage over
> a single-encoding sorting implementation.

Unicode strings need language to decide glyph.
This is implied problem of Unicode Han Unification.
For example U+9AA8's difference between China and Japanese glyph.
http://www.atmarkit.co.jp/fxml/rensai/xmlwomanabou11/learning-xml11.html in Japanese but images are showed in

> When I fetch text from the legacy system, it has a two byte CCSID in
> front of it. I have a table that translates the CCSID to the name of
> the encoding. It is much like:
>
> http://www-01.ibm.com/software/globalization/ccsid/ccsid_registered.jsp

CCSID is a part of IBM's encoding framework: CDRA.
IBM's "Code Page" is a CCS (Coded Character Set).
And CCSID ties up Code Pages with encoding schems.

So IBM's CCSID is the same as an encoding, a charset and Microsoft's Code Page.

So a CCSID can be as an encoding if needed.

> There is one thing that confused me at the end of Martin's post. To me, data never has a language. Perhaps I'm mistaken. The data only have a language when viewed by a user. As he points out, a sort can only be properly done when the language of the user is taken into account. At least, that is how I would rephrase what he said.

Unicode unifies characters between some languages, for example above U+9AA8.
Another critical example is capital letter of i, it is not I in Turkish.
http://unicode.org/Public/UNIDATA/SpecialCasing.txt
(this is one reason why String#upcase is not Unicode sensitive)

More example is following:

- http://unicode.org/reports/tr10/ UNICODE COLLATION ALGORITHM; effects sort
- http://unicode.org/reports/tr11/ East Asian Width; effects String#center
- http://unicode.org/reports/tr18/ UNICODE REGULAR EXPRESSIONS; Tailored Support: Level 3 effects String#upcase and /i/i =end

#14 - 05/16/2011 12:55 PM - mfriedma (Michael Friedman)

Hi. I'm a newcomer to Ruby - studying it right now - but I've been writing multi-lingual systems for 15 years. I think I can shed some light on internationalization issues.

First, I have to say that I was pretty amazed when I discovered that Ruby is not either a multi-character set system or native Unicode. I just assumed that since it comes from Japan and is a relatively new language multi-byte and Unicode support would have been automatic for its developers. Well, that's life and you can't go back in time and change things.

Today, for the vast majority of serious applications, Unicode support is just required. It really doesn't matter what kind of system you are doing. Think about almost anything. Would it be reasonable if a Web mail system didn't allow Chinese e-mails? If a bulletin board system didn't allow Japanese posts? If a task management system didn't allow users to at least create their own tasks in Thai? If a blogging platform didn't support Arabic?

I understand the concerns about backward compatibility if you convert String to a native Unicode type but I think the pain would be worth it. Legacy applications could stay on old versions of Ruby if necessary. New applications would run in native Unicode.

If you do go to native Unicode you have three realistic choices - UTF-8, UTF-16, and UTF-32.

o UTF-8 - Variable width encoding - ASCII is 1 byte, many characters are 2 byte, and some characters are 3 byte - significant performance impact but nice that it preserves ASCII semantics. Biggest disadvantage is that it encourages lazy / ignorant programmers working in English to assume that CHAR = BYTE.

o UTF-16 - 2 bytes for most characters, but requires 4 bytes for characters in the Supplementary Planes. (See http://en.wikipedia.org/wiki/Plane_(Unicode)#Supplementary_Multilingual_Plane). You can choose to ignore the Supplementary Planes (which was the initial design choice for Java) but that has significant impacts on your product's suitability for East Asian languages, especially name handling. Java has been modified now into what I consider to be a bastard hybrid that supports Supplementary Planes but with great pain. See http://java.sun.com/developer/technicalArticles/Intl/Supplementary/. I strongly recommend against this approach. Sun had legacy issues since they started with Unicode support that don't apply to Ruby. Unless developers understand Unicode well enough to test with Supplementary Plane characters (and most don't) they're going to have all sorts of fun bugs when working with this approach.

o UTF-32 - 4 byte fixed width representation. Definitely the fastest and simplest implementation but a very painful memory penalty - "Hello" will now be 20 bytes instead of 5.

If I was creating Ruby from scratch I would use two types - string_utf8 and string_utf32. utf8 is optimized for storage and utf32 is optimized for speed. "string" would be aliased to string_utf32 since most applications care more about speed of string operations than memory. Documentation would strongly encourage storing files in UTF-8.

Next issue, of course, is sorting. In the discussion above, no one has mentioned Locale and Collations. A straight byte order sort in Unicode is usually useless. In fact, multi-lingual sorts are different in every language. For example, in English the correct sort is leap, llama, lover. In Spanish, however, it would be leap, lover, llama - the "ll" is sorted after all other "l" words. String sorts need to have an extra argument - locale or collation. See how database systems like Oracle and SQL Server handle this to get a better understanding. Note that multi-lingual sorts usually make no sense - how do you sort "□□", "□□□□□□□□", "English"? [Those are the strings "Chinese", "Arabic", "English" in native scripts - hopefully they will work on this forum... if not, well, here's an example of why Unicode support is necessary everywhere.]

String comparisons also require complex semantics. Many Unicode characters can be encoded as a single code point or as multiple code points (for a base character and accents and similar items). So you need to normalize before comparing - a pure bytewise comparison will give false negatives. The suggestion that you can do bytewise comparisons on opaque strings above is just incorrect.

This leads into the next issue - handling other character sets.

If the world was full of smart foresightful multi-lingual people who did everything right from the beginning then we would have had Unicode from day one of computing and ASCII and other legacy character sets would not ever have existed. Well, tough... they do and amazingly enough people are still creating web pages, files, etc. in ISO-8859-1, Big5-HKSCS, GB18030, etc. YEEARGH.

For example, if I am writing a web spider, I need to pull in web pages in any character set, parse them for links, and then follow those links. I must support any and all character sets.

So, you either need character set typed strings or the ability to convert any character set to your native types for processing. My mind quails at the complexities involved in working with character set typed strings. For example, what happens when you concatenate two strings in different character sets? I just wouldn't go there. If programmers need to do character wise processing they should convert to native character sets. That is what Java does.

See here for a list of character sets supported for conversion in Java: http://download.oracle.com/javase/1.5.0/docs/guide/intl/encoding.doc.html. Also, check out http://download.oracle.com/javase/6/docs/api/java/lang/String.html and the String constructors that take Charset arguments to properly convert byte arrays to native character set. See also http://download.oracle.com/javase/1.5.0/docs/api/java/io/InputStreamReader.html - "An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted."

I hope this is helpful. Proper support for i18n is a big job but it is necessary if Ruby is really going to be a serious platform for building global systems.

#15 - 05/17/2011 04:03 PM - naruse (Yui NARUSE)

Michael Friedman wrote:

> Hi. I'm a newcomer to Ruby - studying it right now - but I've been writing multi-lingual systems for 15 years. I think I can shed some light on internationalization issues.
>
> First, I have to say that I was pretty amazed when I discovered that Ruby is not either a multi-character set system or native Unicode. I just assumed that since it comes from Japan and is a relatively new language multi-byte and Unicode support would have been automatic for its developers. Well, that's life and you can't go back in time and change things.

Thank you for interesting to Ruby.
But you seems use Ruby 1.8 (it has limited support to multi byte characters).

If you want m17n supports, use 1.9 and read following documents.

- http://yokolet.blogspot.com/2009/07/design-and-implementation-of-ruby-m17n.html
- https://github.com/candlerb/string19
- http://blog.grayproductions.net/categories/character_encodings

#16 - 10/18/2011 09:16 AM - naruse (Yui NARUSE)

- Project changed from Ruby master to CommonRuby

- Category deleted (M17N)

- Target version deleted (3.0)

#17 - 10/23/2011 05:21 PM - naruse (Yui NARUSE)

- Project changed from CommonRuby to Ruby master

#18 - 11/20/2012 08:50 PM - mame (Yusuke Endoh)

- Target version set to 2.6

#19 - 07/23/2014 10:10 AM - duerst (Martin Dürst)

- Related to Feature #10084: Add Unicode String Normalization to String class added

#20 - 07/23/2014 11:06 AM - duerst (Martin Dürst)

- Related to Feature #10085: Add non-ASCII case conversion to String#upcase/downcase/swapcase/capitalize added

#21 - 10/21/2017 10:36 AM - naruse (Yui NARUSE)

- Status changed from Assigned to Rejected