Allow invalid string to work with regexp

07/24/2013 02:47 PM - naruse (Yui NARUSE)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td>matz (Yukihiro Matsumoto)</td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Legacy Ruby 1.8 could regexp match with broken strings. People can find characters from binary data on the age.

After Ruby 1.9, Ruby raises Exception if it does regexp match with broken strings. So it became hard to work with character-wise regexp matching with binary data.

Following patch allows it with the constant Regexp::LOOSE_ENCODING.

commit eb0111ff7ae3f563ce201c4a5f724f121336d42d
Author: NARUSE, Yui naruse@ruby-lang.org
Date:   Mon Jul 22 05:37:44 2013 +0900

```diff
* Regexp
  * New constant:
    * Regexp::ENCODING_LOOSE: declare execute matching even if the target string
      is invalid byte sequence. [experimental]

diff --git a/NEWS b/NEWS
index f5fe388..ade0b03 100644
--- a/NEWS
+++ b/NEWS
@@ -35,6 +35,11 @@ with all sufficient information, see the ChangeLog file.
+ * Regexp
 + New constant:
   + * Regexp::ENCODING_LOOSE: declare execute matching even if the target string
     is invalid byte sequence. [experimental] +
     
     misc
   + misc
     Mutex#owned? is no longer experimental.
```

```diff
+* Regexp
+  + New constant:
+  + * Regexp::ENCODING_LOOSE: declare execute matching even if the target string
+  + is invalid byte sequence. [experimental] +
+  + String
+  + New methods:
+  +   String#scrub and String#scrub! verify and fix invalid byte sequence. diff --git a/re.c b/re.c
+  + index e5cc79d..230a2e0 100644 --- a/re.c +++ b/re.c @@ -256,6 +256,7 @@ rb_memsearch(const void *x0, long m, const void *y0, long n, rb_encoding *enc)
+  + #define REG_ENCODING_LOOSE FL_USER7
+
#define REG_LITERAL FL_USER5
#define REG_ENCODING_NONE FL_USER6
+#define REG_ENCODING_LOOSE FL_USER7

#define KCODE_FIXED FL_USER4
@@ -263,6 +264,7 @@ rb_memsearch(const void *x0, long m, const void *y0, long n, rb_encoding *enc)
(ONIG_OPTION_IGNORECASE|ONIG_OPTION_MULTILINE|ONIG_OPTION_EXTEND)
+#define ARG_ENCODING_FIXED 16
+#define ARG_ENCODING_NONE 32
+STATIC ARG_ENCODING_LOOSE 64

static int
char_to_option(int c)
@@ -1251,7 +1253,8 @@ rb_reg_prepare_enc(VALUE re, VALUE str, int warn)
{
    rb_encoding *enc = 0;
```
if (rb_enc_str_coderange(str) == ENC_CODERANGE_BROKEN) {
    if (!(RBASIC(re)->flags & REG_ENCODING_LOOSE) &&
        rb_enc_str_coderange(str) == ENC_CODERANGE_BROKEN) {
        rb_raise(rb_eArgError, "invalid byte sequence in %s",
                 rb_enc_name(rb_enc_get(str)));@@ -2433,6 +2436,9 @@
* enc, if (options & ARG_ENCODING_NONE) { re->basic.flags |= REG_ENCODING_NONE; }

 re->ptr = make_regexp(RSTRING_PTR(unescaped), RSTRING_LEN(unescaped), enc,
     options & ARG_REG_OPTION_MASK, err,
@@ -3091,6 +3097,7 @@
     options = RREGEXP(re)->ptr->options & ARG_REG_OPTION_MASK;
 if (RBASIC(re)->flags & KCODE_FIXED) options |= ARG_ENCODING_FIXED;
 if (RBASIC(re)->flags & REG_ENCODING_NONE) options |= ARG_ENCODING_NONE;

 if (RBASIC(re)->flags & REG_ENCODING_LOOSE) options |= ARG_ENCODING_LOOSE;
 return options;
}

@@ -3579,6 +3586,8 @@

 rb_define_const(rb_cRegexp, "FIXEDENCODING", INT2FIX(ARG_ENCODING_FIXED));
 /* see Regexp.options and Regexp.new */
 rb_define_const(rb_cRegexp, "NOENCODING", INT2FIX(ARG_ENCODING_NONE));

 rb_define_const(rb_cRegexp, "LOOSEENCODING", INT2FIX(ARG_ENCODING_LOOSE));

 rb_global_variable(&reg_cache);

--- a/string.c b/string.c
index 1d784e3..caf0baf 100644
--- a/string.c
+++ b/string.c
@@ -3970,7 +3970,7 @@
 cp = sp;
 str_enc = STR_ENC_GET(str);
 rb_enc_associate(dest, str_enc);

 ENC_CODERANGE_SET(dest, rb_enc_asciicompat(str_enc) ? ENC_CODERANGE_7BIT : ENC_CODERANGE_VALID);

 /ENCY_CODERANGE_SET(dest, rb_enc_asciicompat(str_enc) ? ENC_CODERANGE_7BIT : ENC_CODERANGE_VALID);/
$VERBOSE = @verbose
end
@@ -958,6 +962,17 @@ class TestRegexp < Test::Unit::TestCase
} end

- def test_encoding_loose
  - str = u("\x80\xE3\x81\x82\x81")
  - assert_equal(0, Regexp.new("\.", Regexp::LOOSEENCODING) =~ str)
  - assert_equal(1, Regexp.new(u("\p{Any}"), Regexp::LOOSEENCODING) =~ str)
  - assert_equal(1, Regexp.new("u3042", Regexp::LOOSEENCODING) =~ str)
  - assert_equal(1, Regexp.new(u("\p{Hiragana}"), Regexp::LOOSEENCODING) =~ str)
  - assert_equal(0, Regexp.new(u("\A..\p{Hiragana}.\z"), Regexp::LOOSEENCODING) =~ str)
  - str = u("\xf1\x80\xE3\x81\x82\x81")
  - assert_equal(0, Regexp.new(u("\A..\p{Hiragana}.\z"), Regexp::LOOSEENCODING) =~ str)
  - end

+ This assertion is for porting x2() tests in testpy.py of Onigmo.

  def assert_match_at(re, str, positions, msg = nil)
  re = Regexp.new(re) unless re.is_a?(Regexp)

History

#1 - 07/26/2013 08:41 AM - matz (Yukihiro Matsumoto)
I am positive. I'd rather want to make this default (if possible).
Matz.

#2 - 07/26/2013 06:17 PM - duerst (Martin Dürst)
Sorry to be late with my comment.

naruse (Yui NARUSE) wrote:

Legacy Ruby 1.8 could regexp match with broken strings.

Well, in Ruby 1.8, strings were binary, so this isn't much of a surprise.

  People can find characters from binary data on the age.

Sorry, I don't understand "on the age"? Can you explain (Japanese is fine).

  After Ruby 1.9, Ruby raises Exception if it does regexp match with broken strings.

My understanding is that in Ruby 1.9, we don't test for valid encoding at each corner, because otherwise Ruby would be too slow, but we don't promote or allow operations on invalid data if the check happens anyway.

Creating functionality that is targeted at invalid data starts a slippery slope. We may get more and more requests for places where invalid encoding should produce some "sensible" result, and it will be more and more complex to remember all the rules. "Invalid data doesn't match." is much simpler to work with.

  So it became hard to work with character-wise regex matching with binary data.

03/10/2022
Don't we have BINARY encoding for binary data?

How would matching character-wise regexps in binary data actually work? For single-byte encodings, it's very easy, because in many cases, there is no invalid data. For other encodings, in particular UTF-8 and GB-18030 (and also Shift_JIS,...), it may be difficult to define what exactly happens, i.e. what bytes exactly are treated as binary data.

Next, what are the security implications (in particular if this is on by default, as Matz proposes)?

Also, what exactly happens to bytes or byte sequences that are invalid? Are they matched by any part of a regular expression (e.g. a simple /\/)? Are they counted for positions? Can they be matched literally with \x? Are they non-word characters? ... Is there a way to match them directly (e.g. by putting invalid data into the regexp)? My guess is that all these questions may have different preferred answers depending on the exact use case.

So the next question is what is the actual use case? Finding sequences that match characters in binary data seems to be the use case, but this can be done by converting the characters being searched for to binary encoding and using a binary regexp on binary data. So I don't see how this provides new functionality. Another way to address it is to clean up the data first, because this should anyway happen better sooner than later.

#3 - 07/27/2013 01:53 PM - naruse (Yui NARUSE)

duerst (Martin Dürst) wrote:

Sorry to be late with my comment.

naruse (Yui NARUSE) wrote:

People can find characters from binary data on the age.

Sorry, I don't understad "on the age"? Can you explain (Japanese is fine).

I mean "During the Ruby 1.8 era people can find characters from binary data by regexp matching"

After Ruby 1.9, Ruby raises Exception if it does regexp match with broken strings.

My understanding is that in Ruby 1.9, we don’t test for valid encoding at each corner, because otherwise Ruby would be too slow, but we don’t promote or allow operations on invalid data if the check happens anyway.

This affect only for regexp related one.

Other methods are note affected.

Creating functionality that is targetted at invalid data starts a slippery slope. We may get more and more requests for places where invalid encoding should produce some "sensible" result, and it will be more and more complex to remember all the rules. "Invalid data doesn't match." is much simpler to work with.

It is OK if the request is good one.

So it became hard to work with character-wise regexp matching with binary data.

Don't we have BINARY encoding for binary data?

How would matching character-wise regexps in binary data actually work? For single-byte encodings, it's very easy, because in many cases, there is no invalid data. For other encodings, in particular UTF-8 and GB-18030 (and also Shift_JIS,...), it may be difficult to define what exactly happens, i.e. what bytes exactly are treated as binary data.

As far as I tested, the behavior of my patch is likely to what people will expect.

Next, what are the security implications (in particular if this is on by default, as Matz proposes)?

It is what I mainly worry about.

Also, what exactly happens to bytes or byte sequences that are invalid? Are they matched by any part of a regular expression (e.g. a simple /\/)? Are they counted for positions? Can they be matched literally with \x? Are they non-word characters? ... Is there a way to match them directly (e.g. by putting invalid data into the regexp)? My guess is that all these questions may have different preferred answers depending on the exact use case.

See tests in my patch.

So the next question is what is the actual use case? Finding sequences that match characters in binary data seems to be the use case, but this
can be done by converting the characters being searched for to binary encoding and using a binary regexp on binary data. So I don't see how this provides new functionality. Another way to address it is to clean up the data first, because this should anyway happen better sooner than later.

#4 - 01/30/2014 06:17 AM - hsbt (Hiroshi SHIBATA)
- Target version changed from 2.1.0 to 2.2.0

#5 - 01/05/2018 09:00 PM - naruse (Yui NARUSE)
- Target version deleted (2.2.0)