Article
Benying Zou · Jul 13, 2017 19m read

use punycode in caché

Hi everybody, We have written a convert from Punycode. This is reformed from javascript. Perhaps many non-English systems need to use this feature in domain name resolution. Anyone can use and change the code as needed.

```
 Class Util.Punycode Extends %RegisteredObject
{

Parameter MaxInt = 2147483647;

/// Bootstring parameters
Parameter Base = 36;

Parameter TMin = 1;

Parameter TMax = 26;

Parameter Skew = 38;

Parameter Damp = 700;

Parameter InitialBias = 72;

Parameter InitialN = 128;

Parameter Delimiter = "-";

/// Regular expressions
Parameter RegexPunycode = "^xn--";

/// non-ASCII chars
Parameter RegexNonASCII = "[^\x00-\x7E]";

/// RFC 3490 separators
Parameter RegexSeparators = "[\x2E\u3002\uFF0E\uFF61]";

/// A generic error utility function.<br>
/// @private<br>
/// @param {String} type The error type.<br>
/// @returns {Error} Throws a `RangeError` with the applicable error message.
Method Error(type) [ Private ]
{
  If (type = "overflow")
  {
    Throw $$$ERROR("Overflow: input needs wider integers to process")
    Quit
  }

  If (type = "not-basic")
```

```
  {
    Throw $$$ERROR("Illegal input >= 0x80 (not a basic code point")
    Quit
  }

  If (type = "invalid-input")
  {
    Throw $$$ERROR("invalid-input")
    Quit
  }
  Throw $$$ERROR("Unknown error type: "_type)
}

/// A generic `Array#map` utility function.<br>
/// @private<br>
/// @param {Array} array The array to iterate over.<br>
/// @param {Function} callback The function that gets called for every array
/// item.<br>
/// @returns {Array} A new array of values returned by the callback function.
Method Map(list, fnName As %String) As %List [ Private ]
{
  //#Dim result As %ArrayOfDataTypes
  Set output = $LB()
  Set length = $LISTLENGTH(list)
  For i=length:-1:1
  {
    Set v = $LIST(list, i)
    If (fnName = "regexNonASCII")
    {
      Set matcher = ##class(%Regex.Matcher).%New(..#RegexNonASCII, v)
      If (matcher.Locate())
      {
        Set v = "xn--"_..Encode(v)
      }
    }
    Elseif (fnName = "regexPunycode")
    {
      Set matcher = ##class(%Regex.Matcher).%New(..#RegexPunycode, v)
      If (matcher.Locate())
      {
        Set v = $extract(v, 5, *)
        Set v = ..Decode($ZCONVERT(v, "l"))
      }
    }
    Set $LIST(output, i) = v
  }
  Quit output
}

///
A simple `Array#map`-like wrapper to work with domain name strings or email addresses
. <br>
/// @private<br>
/// @param {String} domain The domain name or email address.<br>
/// @param {Function} callback The function that gets called for every character.<br>
/// @returns {Array} A new string of characters returned by the callback function.
Method MapDomain(string As %String, fnName As %String) As %String [ Private ]
{
  Set result = ""
```

```
  If (string["@")
  {
    // In email addresses, only the domain name should be punycoded. Leave
    // the local part (i.e. everything up to `@`) intact.
    Set result = $PIECE(string, "@", 1)_"@"
    Set string = $PIECE(string, "@", 2)
  }

  // Avoid `split(regex)` for IE8 compatibility. See #17.
  Set matcher = ##class(%Regex.Matcher).%New(..#RegexSeparators, string)
  Set string = matcher.ReplaceAll($char($ZHEX("2e")))

  Set labels = $LISTFROMSTRING(string, ".")
  Set encoded = $LISTTOSTRING(..Map(labels, fnName), ".")

  Quit result_encoded
}

/// Creates an array containing the numeric code points of each Unicode
/// character in the string. While JavaScript uses UCS-2 internally,
/// this function will convert a pair of surrogate halves (each of which
/// UCS-2 exposes as separate characters) into a single code point,
/// matching UTF-16.<br>
/// @see `Ucs2encode`<br>
/// @param {String} string The Unicode input string (UCS-2).<br>
/// @returns {Array} The new array of code points.
Method Ucs2decode(string As %String) As %ArrayOfDataTypes [ Private ]
{
  Set output = ##class(%ArrayOfDataTypes).%New()
  Set counter = 0
  Set length = $length(string)
  While (counter < length)
  {
    Set counter = counter + 1
    Set value = $ascii($extract(string, counter))
    If ((value >= $zhex("D800")) && (value <= $zhex("DBFF") && (counter < length)))
    {
      Set counter = counter + 1
      Set extra = $ascii($extract(string, counter))
      Set tmp = $zboolean(extra, $zhex("FC00"), 1)
      If (tmp = $zhex("DC00"))
      {
        Set a = $zboolean(value, $zhex("3FF"), 1) * (2 ** 10) + $zboolean(extra,
$zhex("3FF"), 1) + $zhex("10000")
        Do output.SetAt(a, (output.Count() + 1))
      } Else {
        Do output.SetAt(value, (output.Count() + 1))
        Set counter = counter - 1
      }
    } Else {
      Do output.SetAt(value, (output.Count() + 1))
    }
  }
  Quit output
}

/// Creates a string based on an array of numeric code points.<br>
/// @see `Usc2decode`<br>
/// @param {Array} codePoints The array of numeric code points.<br>
```

```
/// @returns {String} The new Unicode string (UCS-2).
Method Ucs2encode(array As %ArrayOfDataTypes) As %String [ Private ]
{
  Set output = ""
  Set length = array.Count()
  For i=1:1:length
  {
    Set output = output_$char(array.GetAt(i))
  }
  Quit output
}

/// Converts a basic code point into a digit/integer.
/// @see `DigitToBasic()`<br>
/// @private<br>
/// @param {Number} codePoint The basic numeric code point value.<br>
/// @returns {Number} The numeric value of a basic code point (for use in
/// representing integers) in the range `0` to `base - 1`, or `base` if
/// the code point does not represent a value.
Method BasicToDigit(codePoint) As %Integer [ Private ]
{
  Set res = ..#Base
  If ((codePoint - $zhex("30")) < $zhex("0A"))
  {
    Set res = codePoint - $zhex("16")
  }
  Elseif ((codePoint - $zhex("41")) < $zhex("1A"))
  {
    Set res = codePoint - $zhex("41")
  }
  Elseif ((codePoint - $zhex("61")) < $zhex("1A"))
  {
    Set res = codePoint - $zhex("61")
  }
  Quit res
}

/// Converts a digit/integer into a basic code point.
/// @see `BasicToDigit()`<br>
/// @private<br>
/// @param {Number} digit The numeric value of a basic code point.<br>
/// @returns {Number} The basic code point whose value (when used for
/// representing integers) is `digit`, which needs to be in the range
/// `0` to `base - 1`. If `flag` is non-zero, the uppercase form is
/// used; else, the lowercase form is used. The behavior is undefined
/// if `flag` is non-zero and `digit` has no uppercase form.
Method DigitToBasic(digit, flag) As %Integer [ Private ]
{
  Set result = digit + 22
  If (digit < 26)
  {
    Set result = result + 75
  }
  Set tmp = 0
  If (flag '= 0)
  {
    Set tmp = 2**5
  }
  Quit result - tmp
```

```
}

/// Bias adaptation function as per section 3.4 of RFC 3492.<br>
/// https://tools.ietf.org/html/rfc3492#section-3.4<br>
/// @private
Method Adapt(delta, numPoints, firstTime As %Boolean) As %Integer [ Private ]
{
  Set k = 0
  Set delta1 = delta / ..#Damp \ 1
  If ('firstTime)
  {
    Set delta1 = delta / 2 \ 1
  }
  Set delta2 = delta1 / numPoints \ 1
  Set delta1 = delta1 + delta2
  Set baseMinusTMin = ..#Base - ..#TMin

  While (delta1 > ((baseMinusTMin * ..#TMax) / 2 \ 1))
  {
    Set k = k + ..#Base
    Set delta1 = delta1 / baseMinusTMin \ 1
  }

  Quit (k + ((baseMinusTMin + 1) * delta1 / (delta1 + ..#Skew))) \ 1
}

/// Converts a Punycode string of ASCII-only symbols to a string of Unicode
/// symbols.<br>
/// @param {String} input The Punycode string of ASCII-only symbols.<br>
/// @returns {String} The resulting string of Unicode symbols.
Method Decode(input As %String) As %String [ Private ]
{
  // Don't use UCS-2.

  Set output = ##class(%ArrayOfDataTypes).%New()
  Set inputLength = $length(input)
  Set i = 0
  Set n = ..#InitialN
  Set bias = ..#InitialBias

  // Handle the basic code points: let `basic` be the number of input code
  // points before the last delimiter, or `0` if there is none, then copy
  // the first basic code points to the output.
  Set basic = ..LastIndexOf(input, ..#Delimiter) + 1

  For j=1:1:(basic-2)
  {
    Set charcode = $ascii($extract(input, j))
    If (charcode >= $zhex("80"))
    {
      Do ..Error("not-basic")
    }
    Do output.SetAt(charcode, output.Count() + 1)
  }

  // Main decoding loop: start just after the last delimiter if any basic code
  // points were copied; start at the beginning otherwise.
  Set index = 1
  If (basic > 1)
```

```
{
  Set index = basic
}
While (index <= inputLength)
{
  // `index` is the index of the next character to be consumed.
  // Decode a generalized variable-length integer into `delta`,
  // which gets added to `i`. The overflow checking is easier
  // if we increase `i` as we go, then subtract off its starting
  // value at the end to obtain `delta`.

  Set oldi = i
  Set w = 1
  Set k = ..#Base
  Set con = 1
  While (con > 0)
  {
    If (index > inputLength)
    {
      Do ..Error("invalid-input")
    }

    Set digit = ..BasicToDigit($ascii($extract(input, index)))
    Set index = index + 1

    If ((digit >= ..#Base) || (digit > ((..#MaxInt - i) / w \ 1)))
    {
      Do ..Error("overflow 1")
    }

    Set i = i + (digit * w)

    Set t = ..#TMin
    If (k > bias)
    {
      If (k >= (bias + ..#TMax))
      {
        Set t = ..#TMax
      }
      Else
      {
        Set t = k - bias
      }
    }

    If (digit < t)
    {
      Set con = 0
      quit
    }

    Set baseMinusT = ..#Base - t
    If (w > (..#MaxInt / baseMinusT \ 1))
    {
      Do ..Error("overflow 2")
    }

    Set w = w * baseMinusT
    Set k = k + ..#Base
```

```
    }
    Set out = output.Count() + 1

    Set bias = ..Adapt((i - oldi), out, (oldi=0))

    // `i` was supposed to wrap around from `out` to `0`,
    // incrementing `n` each time, so we'll fix that now:
    If ((i / out \ 1) > (..#MaxInt - n))
    {
      Do ..Error("overflow 3")
    }

    Set n = n + (i / out \ 1)
    Set i = i - (out * (i / out \ 1))

    Set outcount = output.Count()
    For id=outcount:-1:i+1
    {
      Do output.SetAt(output.GetAt(id), id+1)
    }
    Do output.SetAt(n, i+1)
    Set i = i + 1
  }

  Set res = ""
  Set outcount = output.Count()
  For i=1:1:outcount
  {
    Set c = $char(output.GetAt(i))
    Set res = res_c
  }
  Quit res
}

/// Converts a string of Unicode symbols (e.g. a domain name label) to a
/// Punycode string of ASCII-only symbols.<br>
/// @param {String} input The string of Unicode symbols.<br>
/// @returns {String} The resulting Punycode string of ASCII-only symbols.
Method Encode(input As %String) As %String [ Private ]
{
  Set output = ##class(%ArrayOfDataTypes).%New()

  // Convert the input in UCS-2 to an array of Unicode code points.
  Set inputCodes = ..Ucs2decode(input)

  // Cache the length.
  Set inputLength = inputCodes.Count()

  // Initialize the state.
  Set n = ..#InitialN
  Set delta = 0
  Set bias = ..#InitialBias

  // Handle the basic code points.
  For i=1:1:inputLength
  {
    Set currentValue = inputCodes.GetAt(i)
    If (currentValue < $zhex("80"))
    {
```

```
      Do output.SetAt($char(currentValue), output.Count() + 1)
    }
  }

  Set basicLength = output.Count()
  Set handledCPCount = basicLength

  // `handledCPCount` is the number of code points that have been handled;
  // `basicLength` is the number of basic code points.

  // Finish the basic string with a delimiter unless it's empty.
  If (basicLength > 0)
  {
    Do output.SetAt(..#Delimiter, output.Count() + 1)
  }

  // Main encoding loop
  While (handledCPCount < inputLength)
  {
    // All non-basic code points < n have been handled already. Find the next
    // larger one:
    Set m = ..#MaxInt
    For i=1:1:inputLength
    {
      Set currentValue = inputCodes.GetAt(i)

      If ((currentValue >= n) && (currentValue < m))
      {
        Set m = currentValue
      }
    }

    // Increase `delta` enough to advance the decoder's <n,i> state to <m,0>,
    // but guard against overflow.
    Set handledCPCountPlusOne = handledCPCount + 1
    If ((m-n) > ((..#MaxInt - delta) / handledCPCountPlusOne \ 1))
    {
    Do ..Error("overflow 4")
    }

    Set delta = delta + ((m - n) * handledCPCountPlusOne)
    Set n = m

    For i=1:1:inputLength
    {
      Set currentValue = inputCodes.GetAt(i)

      If (currentValue < n)
      {
        Set delta = delta + 1
        If (delta > ..#MaxInt)
        {
          Do ..Error("overflow 5")
        }
      }

      If (currentValue = n)
      {
        // Represent delta as a generalized variable-length integer.
```

```
        Set q = delta
        Set k = ..#Base
        Set con = 1
        While (con > 0)
        {
          Set t = ..#TMin
          If (k > bias)
          {
            If (k >= (bias + ..#TMax))
            {
              Set t = ..#TMax
            }
            Else
            {
              Set t = k - bias
            }
          }

          If (q < t)
          {
            Set con = 0
            quit
          }
          Set qMinusT = q - t
          Set baseMinusT = ..#Base - t

          Set qMinusTModeBaseMinusT = qMinusT - ((qMinusT / baseMinusT \ 1) *
baseMinusT)

          Do output.SetAt($char(..DigitToBasic(t +
qMinusTModeBaseMinusT, 0)), output.Count() + 1)

          Set q = qMinusT / baseMinusT \ 1

          Set k = k + ..#Base
        }
        Do output.SetAt($char(..DigitToBasic(q, 0)), output.Count() + 1)
        Set bias = ..Adapt(delta, handledCPCountPlusOne,
handledCPCount = basicLength)
        Set delta = 0
        Set handledCPCount = handledCPCount + 1
      }
    }
    Set delta = delta + 1
    Set n = n + 1
  }
  Set res = ""
  For i=1:1:output.Count()
  {
    Set res = res_output.GetAt(i)
  }
  Quit res
}

/// Find the position of the last occurrence of a substring in a string.<br>
/// @param {String} str Specifies the string to search<br>
/// @param {String} substr Specifies the string to find<br>
/// @returns {Ingeger} the last position
Method LastIndexOf(str As %String, substr As %String) As %Integer [ Private ]
```

```
{
  Set pos = 0
  Set tmpstr = str
  Set lengthSub = $length(substr)
  Do {
    Set index = $find(tmpstr, substr)
    If (index > 0)
    {
      Set pos = pos + index - lengthSub
      Set tmpstr = $extract(tmpstr, index, *)
    }
  } While (index > 0)

  If (pos < 0)
  {
    Set pos = 0
  }
  Quit pos
}

/// Converts a Punycode string representing a domain name or an email address
/// to Unicode. Only the Punycoded parts of the input will be converted, i.e.
/// it doesn't matter if you call it on a string that has already been
/// converted to Unicode.<br>
/// @param {String} input The Punycoded domain name or email address to
/// convert to Unicode.<br>
/// @returns {String} The Unicode representation of the given Punycode
/// string.
ClassMethod ToUnicode(input As %String = "") As %String
{
  Set ReturnValue=""
  If ($L(input)>0)
  {
    Set Punycode=##class(Util.Punycode).%New()
    Set ReturnValue=Punycode.MapDomain(input, "regexPunycode")
  }
  Quit ReturnValue
}

/// Converts a Unicode string representing a domain name or an email address to
/// Punycode. Only the non-ASCII parts of the domain name will be converted,
/// i.e. it doesn't matter if you call it with a domain that's already in
/// ASCII.<br>
/// @param {String} input The domain name or email address to convert, as a
/// Unicode string.<br>
/// @returns {String} The Punycode representation of the given domain name or
/// email address.
ClassMethod ToASCII(input As %String = "") As %String
{
  Set ReturnValue = ""
  If ($L(input)>0)
  {
    Set Punycode=##class(Util.Punycode).%New()
    Set ReturnValue = Punycode.MapDomain(input, "regexNonASCII")
  }
  Quit ReturnValue
}

}
```

#Code Snippet #Object Data Model #Caché