package com.twitter.tweetypie.tweettext

import com.twitter.tweetypie.tweettext.TweetText.\_

import com.twitter.twittertext.Extractor

import java.lang.Character

import scala.annotation.tailrec

import scala.collection.JavaConverters.\_

object Truncator {

val Ellipsis = "\u2026"

/\*\*

\* Truncate tweet text for a retweet. If the text is longer than

\* either of the length limits, code points are cut off from the end

\* of the text and replaced with an ellipsis. We keep as much of the

\* leading text as possible, subject to these constraints:

\*

\* - There are no more than `MaxDisplayLength` characters.

\*

\* - When converted to UTF-8, the result does not exceed `MaxByteLength`.

\*

\* - We do not break within a single grapheme cluster.

\*

\* The input is assumed to be partial HTML-encoded and may or may

\* not be NFC normalized. The result will be partial HTML-encoded

\* and will be NFC normalized.

\*/

def truncateForRetweet(input: String): String = truncateWithEllipsis(input, Ellipsis)

/\*\*

\* Truncate to [[com.twitter.tweetypie.tweettext.TweetText#OrginalMaxDisplayLength]] display

\* units, using "..." as an ellipsis. The resulting text is guaranteed to pass our tweet length

\* check, but it is not guaranteed to fit in a SMS message.

\*/

def truncateForSms(input: String): String = truncateWithEllipsis(input, "...")

/\*\*

\* Check the length of the given text, and truncate it if it is longer

\* than the allowed length for a Tweet. The result of this method will

\* always have:

\*

\* - Display length <= OriginalMaxDisplayLength.

\* - Length when encoded as UTF-8 <= OriginalMaxUtf8Length.

\*

\* If the input would violate this, then the text will be

\* truncated. When the text is truncated, it will be truncated such

\* that:

\*

\* - Grapheme clusters will not be split.

\* - The last character before the ellipsis will not be a whitespace

\* character.

\* - The ellipsis text will be appended to the end.

\*/

private[this] def truncateWithEllipsis(input: String, ellipsis: String): String = {

val text = nfcNormalize(input)

val truncateAt =

truncationPoint(text, OriginalMaxDisplayLength, OriginalMaxUtf8Length, Some(ellipsis))

if (truncateAt.codeUnitOffset.toInt == text.length) text

else text.take(truncateAt.codeUnitOffset.toInt) + ellipsis

}

/\*\*

\* Indicates a potential TruncationPoint in piece of text.

\*

\* @param charOffset the utf-16 character offset of the truncation point

\* @param codePointOffset the offset in code points

\*/

case class TruncationPoint(codeUnitOffset: Offset.CodeUnit, codePointOffset: Offset.CodePoint)

/\*\*

\* Computes a TruncationPoint for the given text and length constraints. If `truncated` on

\* the result is `false`, it means the text will fit within the given constraints without

\* truncation. Otherwise, the result indicates both the character and code-point offsets

\* at which to perform the truncation, and the resulting display length and byte length of

\* the truncated string.

\*

\* Text should be NFC normalized first for best results.

\*

\* @param withEllipsis if true, then the truncation point will be computed so that there is space

\* to append an ellipsis and to still remain within the limits. The ellipsis is not counted

\* in the returned display and byte lengths.

\*

\* @param atomicUnits may contain a list of ranges that should be treated as atomic unit and

\* not split. each tuple is half-open range in code points.

\*/

def truncationPoint(

text: String,

maxDisplayLength: Int = OriginalMaxDisplayLength,

maxByteLength: Int = OriginalMaxUtf8Length,

withEllipsis: Option[String] = None,

atomicUnits: Offset.Ranges[Offset.CodePoint] = Offset.Ranges.Empty

): TruncationPoint = {

val breakPoints =

GraphemeIndexIterator

.ends(text)

.filterNot(Offset.Ranges.htmlEntities(text).contains)

val ellipsisDisplayUnits =

withEllipsis.map(Offset.DisplayUnit.length).getOrElse(Offset.DisplayUnit(0))

val maxTruncatedDisplayLength = Offset.DisplayUnit(maxDisplayLength) - ellipsisDisplayUnits

val ellipsisByteLength = withEllipsis.map(Offset.Utf8.length).getOrElse(Offset.Utf8(0))

val maxTruncatedByteLength = Offset.Utf8(maxByteLength) - ellipsisByteLength

var codeUnit = Offset.CodeUnit(0)

var codePoint = Offset.CodePoint(0)

var displayLength = Offset.DisplayUnit(0)

var byteLength = Offset.Utf8(0)

var truncateCodeUnit = codeUnit

var truncateCodePoint = codePoint

@tailrec def go(): TruncationPoint =

if (displayLength.toInt > maxDisplayLength || byteLength.toInt > maxByteLength) {

TruncationPoint(truncateCodeUnit, truncateCodePoint)

} else if (codeUnit != truncateCodeUnit &&

displayLength <= maxTruncatedDisplayLength &&

byteLength <= maxTruncatedByteLength &&

(codeUnit.toInt == 0 || !Character.isWhitespace(text.codePointBefore(codeUnit.toInt))) &&

!atomicUnits.contains(codePoint)) {

// we can advance the truncation point

truncateCodeUnit = codeUnit

truncateCodePoint = codePoint

go()

} else if (breakPoints.hasNext) {

// there are further truncation points to consider

val nextCodeUnit = breakPoints.next

codePoint += Offset.CodePoint.count(text, codeUnit, nextCodeUnit)

displayLength += Offset.DisplayUnit.count(text, codeUnit, nextCodeUnit)

byteLength += Offset.Utf8.count(text, codeUnit, nextCodeUnit)

codeUnit = nextCodeUnit

go()

} else {

TruncationPoint(codeUnit, codePoint)

}

go()

}

/\*\*

\* Truncate the given text, avoiding chopping HTML entities and tweet

\* entities. This should only be used for testing because it performs

\* entity extraction, and so is very inefficient.

\*/

def truncateForTests(

input: String,

maxDisplayLength: Int = OriginalMaxDisplayLength,

maxByteLength: Int = OriginalMaxUtf8Length

): String = {

val text = nfcNormalize(input)

val extractor = new Extractor

val entities = extractor.extractEntitiesWithIndices(text)

extractor.modifyIndicesFromUTF16ToUnicode(text, entities)

val avoid = Offset.Ranges.fromCodePointPairs(

entities.asScala.map(e => (e.getStart().intValue, e.getEnd().intValue))

)

val truncateAt = truncationPoint(text, maxDisplayLength, maxByteLength, None, avoid)

text.take(truncateAt.codeUnitOffset.toInt)

}

}