package com.twitter.tweetypie.serverutil.logcachewrites

import com.twitter.servo.cache.Cached

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.tweetypie.TweetId

import com.twitter.tweetypie.core.Serializer

import com.twitter.tweetypie.thriftscala.CachedTweet

import com.twitter.util.Time

import java.util.Base64

/\*\*

\* A record of a tweet cache write. This is used for debugging. These log

\* messages are scribed to test\_tweetypie\_tweet\_cache\_writes.

\*/

case class TweetCacheWrite(

tweetId: TweetId,

timestamp: Time,

action: String,

value: Option[Cached[CachedTweet]]) {

/\*\*

\* Convert to a tab-separated string suitable for writing to a log message.

\*

\* Fields are:

\* - Tweet id

\* - Timestamp:

\* If the tweet id is a snowflake id, this is an offset since tweet creation.

\* If it is not a snowflake id, then this is a Unix epoch time in

\* milliseconds. (The idea is that for most tweets, this encoding will make

\* it easier to see the interval between events and whether it occured soon

\* after tweet creation.)

\* - Cache action ("set", "add", "replace", "cas", "delete")

\* - Base64-encoded Cached[CachedTweet] struct

\*/

def toLogMessage: String = {

val builder = new java.lang.StringBuilder

val timestampOffset =

if (SnowflakeId.isSnowflakeId(tweetId)) {

SnowflakeId(tweetId).unixTimeMillis.asLong

} else {

0

}

builder

.append(tweetId)

.append('\t')

.append(timestamp.inMilliseconds - timestampOffset)

.append('\t')

.append(action)

.append('\t')

value.foreach { ct =>

// When logging, we end up serializing the value twice, once for the

// cache write and once for the logging. This is suboptimal, but the

// assumption is that we only do this for a small fraction of cache

// writes, so it should be ok. The reason that this is necessary is

// because we want to do the filtering on the deserialized value, so

// the serialized value is not available at the level that we are

// doing the filtering.

val thriftBytes = Serializer.CachedTweet.CachedCompact.to(ct).get

builder.append(Base64.getEncoder.encodeToString(thriftBytes))

}

builder.toString

}

}

object TweetCacheWrite {

case class ParseException(msg: String, cause: Exception) extends RuntimeException(cause) {

override def getMessage: String = s"Failed to parse as TweetCacheWrite: $msg"

}

/\*\*

\* Parse a TweetCacheWrite object from the result of TweetCacheWrite.toLogMessage

\*/

def fromLogMessage(msg: String): TweetCacheWrite =

try {

val (tweetIdStr, timestampStr, action, cachedTweetStr) =

msg.split('\t') match {

case Array(f1, f2, f3) => (f1, f2, f3, None)

case Array(f1, f2, f3, f4) => (f1, f2, f3, Some(f4))

}

val tweetId = tweetIdStr.toLong

val timestamp = {

val offset =

if (SnowflakeId.isSnowflakeId(tweetId)) {

SnowflakeId(tweetId).unixTimeMillis.asLong

} else {

0

}

Time.fromMilliseconds(timestampStr.toLong + offset)

}

val value = cachedTweetStr.map { str =>

val thriftBytes = Base64.getDecoder.decode(str)

Serializer.CachedTweet.CachedCompact.from(thriftBytes).get

}

TweetCacheWrite(tweetIdStr.toLong, timestamp, action, value)

} catch {

case e: Exception => throw ParseException(msg, e)

}

}