import os

import struct

import logging

from primitives import \*

from constants import \*

from astypes import MalformedFLV

from astypes import get\_script\_data\_variable, make\_script\_data\_variable

log = logging.getLogger('flvlib.tags')

STRICT\_PARSING = False

def strict\_parser():

return globals()['STRICT\_PARSING']

class EndOfTags(Exception):

pass

def ensure(value, expected, error\_msg):

if value == expected:

return

if strict\_parser():

raise MalformedFLV(error\_msg)

else:

log.warning('Skipping non-conformant value in FLV file')

class Tag(object):

def \_\_init\_\_(self, parent\_flv, f):

self.f = f

self.parent\_flv = parent\_flv

self.offset = None

self.size = None

self.timestamp = None

def parse(self):

f = self.f

self.offset = f.tell() - 1

# DataSize

self.size = get\_ui24(f)

# Timestamp + TimestampExtended

self.timestamp = get\_si32\_extended(f)

if self.timestamp < 0:

log.warning("The tag at offset 0x%08X has negative timestamp: %d",

self.offset, self.timestamp)

# StreamID

stream\_id = get\_ui24(f)

ensure(stream\_id, 0, "StreamID non zero: 0x%06X" % stream\_id)

# The rest gets parsed in the subclass, it should move f to the

# correct position to read PreviousTagSize

self.parse\_tag\_content()

previous\_tag\_size = get\_ui32(f)

ensure(previous\_tag\_size, self.size + 11,

"PreviousTagSize of %d (0x%08X) "

"not equal to actual tag size of %d (0x%08X)" %

(previous\_tag\_size, previous\_tag\_size,

self.size + 11, self.size + 11))

def parse\_tag\_content(self):

# By default just seek past the tag content

self.f.seek(self.size, os.SEEK\_CUR)

class AudioTag(Tag):

def \_\_init\_\_(self, parent\_flv, f):

Tag.\_\_init\_\_(self, parent\_flv, f)

self.sound\_format = None

self.sound\_rate = None

self.sound\_size = None

self.sound\_type = None

self.aac\_packet\_type = None # always None for non-AAC tags

def parse\_tag\_content(self):

f = self.f

sound\_flags = get\_ui8(f)

read\_bytes = 1

self.sound\_format = (sound\_flags & 0xF0) >> 4

self.sound\_rate = (sound\_flags & 0xC) >> 2

self.sound\_size = (sound\_flags & 0x2) >> 1

self.sound\_type = sound\_flags & 0x1

if self.sound\_format == SOUND\_FORMAT\_AAC:

# AAC packets can be sequence headers or raw data.

# The former contain codec information needed by the decoder to be

# able to interpret the rest of the data.

self.aac\_packet\_type = get\_ui8(f)

read\_bytes += 1

# AAC always has sampling rate of 44 kHz

ensure(self.sound\_rate, SOUND\_RATE\_44\_KHZ,

"AAC sound format with incorrect sound rate: %d" %

self.sound\_rate)

# AAC is always stereo

ensure(self.sound\_type, SOUND\_TYPE\_STEREO,

"AAC sound format with incorrect sound type: %d" %

self.sound\_type)

if strict\_parser():

try:

sound\_format\_to\_string[self.sound\_format]

except KeyError:

raise MalformedFLV("Invalid sound format: %d",

self.sound\_format)

try:

(self.aac\_packet\_type and

aac\_packet\_type\_to\_string[self.aac\_packet\_type])

except KeyError:

raise MalformedFLV("Invalid AAC packet type: %d",

self.aac\_packet\_type)

f.seek(self.size - read\_bytes, os.SEEK\_CUR)

def \_\_repr\_\_(self):

if self.offset is None:

return "<AudioTag unparsed>"

elif self.aac\_packet\_type is None:

return ("<AudioTag at offset 0x%08X, time %d, size %d, %s>" %

(self.offset, self.timestamp, self.size,

sound\_format\_to\_string.get(self.sound\_format, '?')))

else:

return ("<AudioTag at offset 0x%08X, time %d, size %d, %s, %s>" %

(self.offset, self.timestamp, self.size,

sound\_format\_to\_string.get(self.sound\_format, '?'),

aac\_packet\_type\_to\_string.get(self.aac\_packet\_type, '?')))

class VideoTag(Tag):

def \_\_init\_\_(self, parent\_flv, f):

Tag.\_\_init\_\_(self, parent\_flv, f)

self.frame\_type = None

self.codec\_id = None

self.h264\_packet\_type = None # Always None for non-H.264 tags

def parse\_tag\_content(self):

f = self.f

video\_flags = get\_ui8(f)

read\_bytes = 1

self.frame\_type = (video\_flags & 0xF0) >> 4

self.codec\_id = video\_flags & 0xF

if self.codec\_id == CODEC\_ID\_H264:

# H.264 packets can be sequence headers, NAL units or sequence

# ends.

self.h264\_packet\_type = get\_ui8(f)

read\_bytes += 1

if strict\_parser():

try:

frame\_type\_to\_string[self.frame\_type]

except KeyError:

raise MalformedFLV("Invalid frame type: %d", self.frame\_type)

try:

codec\_id\_to\_string[self.codec\_id]

except KeyError:

raise MalformedFLV("Invalid codec ID: %d", self.codec\_id)

try:

(self.h264\_packet\_type and

h264\_packet\_type\_to\_string[self.h264\_packet\_type])

except KeyError:

raise MalformedFLV("Invalid H.264 packet type: %d",

self.h264\_packet\_type)

f.seek(self.size - read\_bytes, os.SEEK\_CUR)

def \_\_repr\_\_(self):

if self.offset is None:

return "<VideoTag unparsed>"

elif self.h264\_packet\_type is None:

return ("<VideoTag at offset 0x%08X, time %d, size %d, %s (%s)>" %

(self.offset, self.timestamp, self.size,

codec\_id\_to\_string.get(self.codec\_id, '?'),

frame\_type\_to\_string.get(self.frame\_type, '?')))

else:

return ("<VideoTag at offset 0x%08X, "

"time %d, size %d, %s (%s), %s>" %

(self.offset, self.timestamp, self.size,

codec\_id\_to\_string.get(self.codec\_id, '?'),

frame\_type\_to\_string.get(self.frame\_type, '?'),

h264\_packet\_type\_to\_string.get(

self.h264\_packet\_type, '?')))

class ScriptTag(Tag):

def \_\_init\_\_(self, parent\_flv, f):

Tag.\_\_init\_\_(self, parent\_flv, f)

self.name = None

self.variable = None

def parse\_tag\_content(self):

f = self.f

# Here there's always a byte with the value of 0x02,

# which means "string", although the spec says NOTHING

# about it..

value\_type = get\_ui8(f)

ensure(value\_type, 2, "The name of a script tag is not a string")

# Need to pass the tag end offset, because apparently YouTube

# doesn't give a \*shit\* about the FLV spec and just happily

# ends the onMetaData tag after self.size bytes, instead of

# ending it with the \*required\* 0x09 marker. Bastards!

if strict\_parser():

# If we're strict, just don't pass this info

tag\_end = None

else:

# 11 = tag type (1) + data size (3) + timestamp (4) + stream id (3)

tag\_end = self.offset + 11 + self.size

log.debug("max offset is 0x%08X", tag\_end)

self.name, self.variable = \

get\_script\_data\_variable(f, max\_offset=tag\_end)

log.debug("A script tag with a name of %s and value of %r",

self.name, self.variable)

def \_\_repr\_\_(self):

if self.offset is None:

return "<ScriptTag unparsed>"

else:

return ("<ScriptTag %s at offset 0x%08X, time %d, size %d>" %

(self.name, self.offset, self.timestamp, self.size))

tag\_to\_class = {

TAG\_TYPE\_AUDIO: AudioTag,

TAG\_TYPE\_VIDEO: VideoTag,

TAG\_TYPE\_SCRIPT: ScriptTag

}

class FLV(object):

def \_\_init\_\_(self, f):

self.f = f

self.version = None

self.has\_audio = None

self.has\_video = None

self.tags = []

def parse\_header(self):

f = self.f

f.seek(0)

# FLV header

header = f.read(3)

if len(header) < 3:

raise MalformedFLV("The file is shorter than 3 bytes")

# Do this irrelevant of STRICT\_PARSING, to catch bogus files

if header != "FLV":

raise MalformedFLV("File signature is incorrect: 0x%X 0x%X 0x%X" %

struct.unpack("3B", header))

# File version

self.version = get\_ui8(f)

log.debug("File version is %d", self.version)

# TypeFlags

flags = get\_ui8(f)

ensure(flags & 0xF8, 0,

"First TypeFlagsReserved field non zero: 0x%X" % (flags & 0xF8))

ensure(flags & 0x2, 0,

"Second TypeFlagsReserved field non zero: 0x%X" % (flags & 0x2))

self.has\_audio = False

self.has\_video = False

if flags & 0x4:

self.has\_audio = True

if flags & 0x1:

self.has\_video = True

log.debug("File %s audio",

(self.has\_audio and "has") or "does not have")

log.debug("File %s video",

(self.has\_video and "has") or "does not have")

header\_size = get\_ui32(f)

log.debug("Header size is %d bytes", header\_size)

f.seek(header\_size)

tag\_0\_size = get\_ui32(f)

ensure(tag\_0\_size, 0, "PreviousTagSize0 non zero: 0x%08X" % tag\_0\_size)

def iter\_tags(self):

self.parse\_header()

try:

while True:

tag = self.get\_next\_tag()

yield tag

except EndOfTags:

pass

def read\_tags(self):

self.tags = list(self.iter\_tags())

def get\_next\_tag(self):

f = self.f

try:

tag\_type = get\_ui8(f)

except EndOfFile:

raise EndOfTags

tag\_klass = self.tag\_type\_to\_class(tag\_type)

tag = tag\_klass(self, f)

tag.parse()

return tag

def tag\_type\_to\_class(self, tag\_type):

try:

return tag\_to\_class[tag\_type]

except KeyError:

raise MalformedFLV("Invalid tag type: %d", tag\_type)

def create\_flv\_tag(type, data, timestamp=0):

tag\_type = struct.pack("B", type)

timestamp = make\_si32\_extended(timestamp)

stream\_id = make\_ui24(0)

data\_size = len(data)

tag\_size = data\_size + 11

return ''.join([tag\_type, make\_ui24(data\_size), timestamp, stream\_id,

data, make\_ui32(tag\_size)])

def create\_script\_tag(name, data, timestamp=0):

payload = make\_ui8(2) + make\_script\_data\_variable(name, data)

return create\_flv\_tag(TAG\_TYPE\_SCRIPT, payload, timestamp)

def create\_flv\_header(has\_audio=True, has\_video=True):

type\_flags = 0

if has\_video:

type\_flags = type\_flags | 0x1

if has\_audio:

type\_flags = type\_flags | 0x4

return ''.join(['FLV', make\_ui8(1), make\_ui8(type\_flags), make\_ui32(9),

make\_ui32(0)])