Sequential-byte serializer

1. Contents

2. Introduction

This is a set of classes for serializing and deserializing data to and from a file.

It is most common to serialize .Net objects to JSON, XML or using the provided mechanism in the .Net framework which writes the in-memory bytes representing a given object to a file.

This, however, is a system for sequentially writing bytes to a file organized in entries and records, and thus representing a file format of its own.

3. Background

I got this code by reversing an app and thought it might be interesting and useful.

Also, I don't know where the original source came from, so if you do, I would be grateful if you let me know. (I know what I've reversed, but it was probably copy-pasted from another place, and I would very much like to find the original source code)

Credits: this is reversed from the Gold Parser system, though I believe this particular code was rewritten from somewhere else. Some might wonder why I'm reversing the Gold parser and writing articles on the matter, and the reason is that I'm developing my own parser system for a few projects of mine and wanted to see how it is done, and the "Gold Parser" is a wonderful open-source example with grammars freely available for almost any major programming language, and I advise anyone to check their website should one need a free and powerful parser system :
http://codeprompt.github.io/sequential-byte-serializer

Also, you can read and download a .html or .pdf of this and other articles on my website:
http://codeprompt.github.io/sequential-byte-serializer

4. Using the code

In few words, the system contains a reader and a writer, along with its own exception class.

A reader is created for a file and used to read data, while the writer is created for a given file path and used to write data, just like regular .Net file streams on top of which those are actually created.

The information is organized in records, each containing entries, so each object that needs to be serialized will be represented by a record and each field - represented by an entry - which represents a single boxed primitive value in the form of an object and a Type expressed by an EntryType enumeration.

A thing to note here is how this limits the type of objects that can be serialized to ones containing only primitive value fields, so, if an object containing fields of non-primitive types must be serialized, those fields need to be substituted with fields of type "int" pointing to object indexes and of-course "int index" fields need to be added on the other hand, to the referenced objects when serializing those.

5. Overview

IOException is A simple exception class, no different than the regular .Net exception, which it inherits, existing only for the sake of the name.

An entry represents a single storable value.

It has a Value - an object that boxes the primitive which the entry represents, and a Type - indicating the primitive type that the entry represents (Empty/UInt16/String/Boolean/Byte/Error). Working with those entry objects is very convenient when one does not know the type of the value that will be retrieved, for example.

The output file consists of records and each record consists of a different number of entries. For example,

one might start a new record for each object that needs to be serialized and write each field of the object to the file.

The writer is used to write data to a file.

After creating an instance of the writer, one must open it providing a desired storage file path and file header string which is stored at the beginning of the file, indicating the file type - something like the MZ magic word in windows PE(.exe) files. The writer also can be closed, in order to be disposed of, instead of waiting to be automatically garbage collected.

After the writer has been opened one can use "StoreEmpty", "StoreBoolean", "StoreInt16", "StoreByte" and "StoreString" to write data to the file and "NewRecord" to start next record.

The reader is used to read data from a file, previously created with the writer.

After creating an instance of the reader, one must open it providing a desired storage file path or a BinaryReader from that file. The reader also can be closed, in order to be disposed of, instead of waiting to be automatically garbage collected.

After the reader has been opened one can use "RetrieveEntry", "RetrieveString", "RetrieveInt16", "RetrieveBoolean" and "RetrieveByte" to get data, or "GetNextRecord" to jump to the next record.

6. Point Map

```
IOException : Exception


EntryType : byte
  .Empty
  .UInt16
  .String
  .Boolean
  .Byte
  .Error


Entry
  .EntryType                                    Type
  .object                                       Value

  .()
  .(EntryType, object)                          type, value


EntryList
  .[int]                -> Entry
  .Add(Entry)               -> int              value


Reader
  .bool                                         RecordComplete
  .string                                       Header
  .int                                          EntryCount
  .bool                                         EndOfFile

  _BinaryReader                                 _Reader
  _int                                          _EntryCount
  _int                                          _EntriesRead

  .Open(string)                                 path
  .Open(BinaryReader)                           reader
  .Close()

  .GetNextRecord()          -> bool
  .RetrieveEntry()          -> Entry
  .RetrieveString()         -> string
  .RetrieveInt16()          -> int
  .RetrieveBoolean()            -> bool
  .RetrieveByte()           -> byte


Writer
  _FileStream                                   _File
  _BinaryWriter                                 _Writer
  _EntryList                                    _CurrentRecord

  .Open(string, string)                         path, header
  .Close()

  .NewRecord()
  .StoreEmpty()
```

.StoreBoolean(bool)	***value***
.StoreInt16(int)	***value***
.StoreByte(byte)	***value***
.StoreString(string)	***value***

---

7. Member Map - Reader

RecordComplete
Whether the current record has been completely red, or in other words whether _EntriesRead equals _EntryCount.

Header
The opened file header.

EntryCount
The count of all the entries in the current record.

EndOfFile
Whether the end of file is reached. We get that from the position and length of the "_Reader"'s underlying stream.

~()
The destructor calls "Close()".
->Close()

.Open(string path)
Try
->BinaryReader(path)
->Open(BinaryReader)
Catch

.Open(BinaryReader reader)
Set the "_Reader", reset "_EntryCount" and "_EntriesRead" to "0",
retrieve the file header by "RawReadCString()" and set it - "_FileHeader".
->RawReadCString()

.Close()
Close the reader.

.GetNextRecord() ->bool
While "_EntryCount" and "_EntriesRead" read an entry.
Retrieve ushort. The ushort must be 77, indicating a record.
If so, retrieve another ushort which indicates the number of entries.
Set "_EntryCount" from that and "_EntriesRead" to 0.
Return true.
Else, if the flag does not match return false.

.RetrieveEntry() ->Entry
If the record has been completed throw "IOException".
Increment "_EntriesRead"
Create new entry.
->Entry()

Read a byte from the file, and cast it to EntryType. Set entry.Type
If Empty (indicated by 69) set the "entry.Value" to empty string.

If Byte (98) read a byte and set the "entry.Value"
->_Reader.ReadByte()

If Boolean (66) read a byte and set the "entry.Value" to "true" if it is "1", else set it to false.
->_Reader.ReadByte()

If UInt16 (73) read ushort and set the "entry.Value"
->RawReadUInt16()
read 2 bytes (a and b) and shift b, then add a. (b << 8) + a

If String (83) read string and set the "entry.Value"
->RawReadCString()
Continuously read ushorts, convert them to char using "Utf32" and accumulate those in string variable,
until reading "0" - the null terminator. Return the resulting string.

->RawReadUInt16()
->ConvertFromUtf32(x)

Else set the entry.Type to Error and the value to an empty string.
Return the entry.

.RetrieveString() ->string
Call "RetrieveEntry()" and cast the value if the entry type matches, else throw the "IOException".

.RetrieveInt16() ->int
Call "RetrieveEntry()" and cast the value if the entry type matches, else throw the "IOException".

.RetrieveBoolean() ->bool
Call "RetrieveEntry()" and cast the value if the entry type matches, else throw the "IOException".

.RetrieveByte() ->byte
Call "RetrieveEntry()" and cast the value if the entry type matches, else throw the "IOException".

---

8. Member Map - Writer

.()
Initiate "_CurrentRecord"
->EntryList()

~()
The destructor calls "Close()".
->Close()

.Open(string path, string header)
Try
Open FileStream and store it in "_File".
Open BinaryWriter and store it in "_Writer".
Write the file header to the file.
->FileStream(path, FileMode.Create)
->BinaryWriter(_File)
->RawWriteCString(header)

Catch
Throw "IOException"

.Close()
Write all the accumulated entries to the file in the form of a record.
In order to avoid too much IO operations (or at least this is the logical reason),
when storing a value it is written to the "EntryList _CurrentRecord"
and when "NewRecord()" is called those are written all at once.
->WriteRecord()
->_File.Close()

.NewRecord()
In order to avoid too much IO operations (or at least this is the logical reason),
when storing a value it is written to the "EntryList _CurrentRecord"
and when "NewRecord()" is called those are written all at once.
->_WriteRecord()

_WriteRecord()
If there are no entries in "_CurrentRecord" return.
Else, for each entry, act according to its type by writing the byte code representing that type,
cast its value and write it.Here is how this works:

If the "entry.Type" is "Boolean"
->RawWriteByte(66)
->RawWriteByte(1) or "0" if false

Else if the "entry.Type" is "Byte"
->RawWriteByte(98)
->Convert.ToByte(x)
->RawWriteByte(x)

Else if the "entry.Type" is "String"
->RawWriteByte(83)
->Convert.ToString(x)
->RawWriteCString(x)

For each char of the string - get its charcode and write it as ushort
->RawWriteInt16(charcode)

Else if the "entry.Type" is "UInt16"
->RawWriteByte(73)
->Convert.ToInt32(x)
->RawWriteInt16(x)

Get 2 bytes from the ushort "x".
This is done by right-shifting the value by 8 for each byte other than the least-significant one,
because when casting to byte only the least significant byte is taken and the rest is cut off.
The logical and with 255, on the other hand, don't change the value at all but is used to implicitly cast to byte.
It is redundant in our case where we use an explicit cast to byte and can be omitted altogether.

byte a = (byte)(value & 255);
(int)value: 1101 0010 0011 1000
(int) 255: 0000 0000 1111 1111
bitwise &: 0000 0000 0011 1000
to byte: 0011 1000

byte b = (byte)(value >> 8 & 255);
(int)value: 1101 0010 0011 1000
>> 8: 0000 0000 1101 0010
(int) 255: 0000 0000 1111 1111
bitwise &: 0000 0000 1101 0010
to byte: 1101 0010

Else, write "69" for empty.
->RawWriteByte(69)

.StoreEmpty()
Create new entry accordingly and store it in "_CurrentRecord"
->Entry(EntryType.Empty, "")

.StoreBoolean(bool value)
Create new entry accordingly and store it in "_CurrentRecord"
->Entry(EntryType.Boolean, value)

.StoreInt16(int value)
Create new entry accordingly and store it in "_CurrentRecord"
->Entry(EntryType.UInt16, value)

.StoreByte(byte value)
Create new entry accordingly and store it in "_CurrentRecord"
->Entry(EntryType.Byte, value)

.StoreString(string value)
Create new entry accordingly and store it in "_CurrentRecord"
->Entry(EntryType.String, value)

---

9. Credits

Special credits to users Sentry and Rotem on stackoverflow.com for explaining the bitwise operations and casting to byte.

Viktor Chernev (vchernev91@abv.bg)
https://codeprompt.github.io/Sequential-byte-serializer.htm
24 Jun 2018