Binary object format

The format is designed to act as an universal serialization format. It can serialize arbitrary object graphs (including reference loops between objects). It is cross-platform in a sense that Ignite clients written in different languages understand that format.

Also it worth noting that there are two matters:

- Binary serialization format.
- Binary object container format.

By default Ignite uses binary serialization for storing objects in caches. Consider a following example:

```java
IgniteCache cache = ignite.cache("cache");
// user object will be saved to a storage after binary serialization
cache.put(1, new Pair(1, 2));
// stored binary object will be converted back to a Java object
Pair val = cache.get(1);
```

Here an object of a user class will be converted to binary format to be stored in a cache. And a reverse conversion will take a place when object is read.

Types and schemes

Binary object container format basically allows to store a set of named fields with their values. A simple example with a such structure is plain Java class:

```java
class Foo {
    int bar;
    String baz;
}
```

Binary containers are context-dependent. It means that serialized byte array does not bear all information about a class name and field names. To restore Java object from bytes a type and a schema should be registered on a receiving side. It works quite naturally for an example of storing data to a cache provided earlier. But some information about type and fields must be present in binary form. Binary object container format stores type id and schema id for that purposes. Having them and a proper context it is possible to restore e.g. Java object from bytes.

Schematically such context can be represented as follows:

```java
Context::Map<Integer, Pair<String, SchemaRegistry>> // maps typeId to type name and SchemaRegistry
SchemaRegistry::Map<Integer, Schema> // maps schemaid to Schema
Schema::List<Pair<Integer, String>> // fieldIds to corresponding field names
```

Layout

At a very top level a binary serialized object consists of the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>24 bytes</td>
</tr>
<tr>
<td>Data</td>
<td>variable length</td>
</tr>
<tr>
<td>Footer</td>
<td>variable length</td>
</tr>
</tbody>
</table>

Header contains various meta information and a structure is described below. Data part contains values of binary object fields. Footer contains field offsets. A particular field offset point out a field position.

**Header**

Sizes are in bytes.

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value type</td>
<td>1</td>
<td>Used for nesting, value is 0x67 as it is binary object</td>
</tr>
<tr>
<td>Format version</td>
<td>1</td>
<td>To be able add new format versions in future</td>
</tr>
<tr>
<td>Flags</td>
<td>2</td>
<td>Various flags, see below</td>
</tr>
<tr>
<td>Type id</td>
<td>4</td>
<td>Hash function of type name</td>
</tr>
</tbody>
</table>
Hash code 4 | Hash code for a cache key (not very useful for values)
---|---
Total length 4 | Object length in bytes including Header, Data and Footer
Schema id 4 | Hash function of field ids
Footer position 4 | Offset in bytes from object beginning

**NOTE:** at a time of writing constants with offsets were located in GridBinaryMarshaller class.

**Flags**

- User type – for system types (e.g. jdk ones) optimizations can be used.
- Has schema – false is used when object has 0 fields.
- Raw data – see a dedicated paragraph.
- Use 1 byte offset – if set each field offset (in footer) takes 1 byte.
- Use 2 byte offset – if 1 byte offset flag is not set and a current flag is set each offset takes 2 bytes. If neither offset size flags are set then an offset takes 4 byte
- Compact footer – in that mode there is no field ids in a footer, see a dedicated paragraph.
- .NET custom type – this flag indicates presence of fields having .NET-specific types not present in Java. E.g. unsigned integer primitive types like `uint`. If the flag is set .NET-specific fields ids are stored in object data end.
- Others are left for future purposes.

**NOTE:** constants were located in BinaryUtils class.

**Footer**

Offsets of object fields are stored in *Footer* part. With stored offsets it takes constant time to find where a particular field resides. Header specifies a footer position. Each offset can take 1, 2 or 4 bytes (it is specified by offset flags in a header). Footer can be compact or not (more details later).

**Example**

Let's consider an example of an object with two fields:

```java
class Example {
    int foo = 123;
    String bar = "abc";
}
```

And here it is after serialization:

```
67 01 2B 00 // 67 -- type, binary object; 01 -- format version; 2B 00 -- flags, (check endianess) 0000 0000
0010 1011, user type, has schema, 1 byte offsets, compact footer
28 4E 07 E5 // typeId
C3 0F 60 A5 // hash code
27 00 00 00 // total length
D0 22 77 DD // schema id
25 00 00 00 // footer position; header ends here
03 7B 00 00 // 03 -- type int, 7B 00 00 00 -- int value 123
00 09 03 00 // 09 -- type String, 03 00 00 00 -- length, 61 62 63 -- "abc"
00 00 61 62
63 18 1D // 18 -- field foo offset, 1D -- field bar offset
```

**Value types**

Binary format support multiple types as first-class citizens. As described below each binary container starts with byte 0x67 which indicates a *binary container* type. Each field value inside binary container (except nulls) starts from one byte specifying a type. In the example provided earlier there were 0x03 and 0x09 type bytes for int and String conversely. First (and only) byte of a null value is 0x65. Among other supported types there are maps and lists. If any type is not supported directly then it can be represented as nested binary container (type 0x67).

**Footer modes**

Binary format supports two modes for writing object Footer. It is controlled by BinaryConfiguration.setCompactFooter. Initially there were no compact footer mode in the format. Instead additionally to an offset *fieldId* was stored for each field in a footer. For the previous serialization example verbose footer looks as follows:
Hash code

Calculated hash code is stored in object header. Hash code is used when object acts as key in cache, otherwise it is perhaps redundant. By default a hash code is calculated using bytes from Data part.

Nested objects and object graphs

As was already mentioned each field (not raw) value in binary format starts with 1 byte indicating a type. If a serialized object class does not have special serialization in binary format (like other simple types like int, long, String have) it can be serialized as a nested binary object. In that case it's value starts from 0x67 byte.

Things become even more interesting when there is a need to store an object links from which form a graph with cycles. For example a following tree representation will produce such object graph:

```java
class TreeNode {
    TreeNode parent; // parent, null for root
    TreeNode left; // left child
    TreeNode right; // right child
}
```

Let's examine a serialization of tree with just 3 nodes, 1 root and 2 children. 3 nodes, root, a, b.

```java
root.parent = null, root.left = a, root.right = b
a.parent = root, a.left = a.right = null
b.parent = root, b.left = b.right = null
```

```
67 01 2B 00 // root header
A2 7D 10 9B // type id
3C FE A8 6D // hash
60 00 00 00 // length
FE DE C9 12 // schema id
5D 00 00 00 // footer offset
65 // root.parent == null
67 01 2B 00 // root.left == a header
A2 7D 10 9B // type id
D4 4B 3A CF // hash
22 00 00 00 // length
FE DE C9 12 // schema id
1F 00 00 00 // footer offset
66 // type handle
31 00 00 00 // a.parent == root handle offset
65 65 // a.left == null, a.right == null
18 1D 1E // a footer
67 01 2B 00 // root.right == b header
A2 7D 10 9B // type id
F2 10 3F 09 // hash
22 00 00 00 // length
FE DE C9 12 // schema id
1F 00 00 00 // footer offset
66 // type handle
53 00 00 00 // b.parent == root handle offset
65 65 // b.left == null, b.right == null
18 1D 1E // b footer
18 39 3B // root footer
```

Here links a parent object which comes before children in binary stream are marked with a type byte 0x66 which is Handle type (link, reference). And a value is 4-byte integer of a back offset from this handle field to a original object. It is a back offset because an original object is located before a handle. Reminder null is encoded as single byte 0x65. Note that each object (root, a, b) has the same type id and schema id here.
Binding with Java classes

Let's consider an read/write example with a cache:

```java
IgniteCache cache = ignite.cache("cache");
// user object will be saved to a storage after binary serialization
cache.put(1, new Pair(1, 2));
// stored binary object will be converted back to a Java object
Pair val = cache.get(1);
```

As we know inside cache storage Pair will be stored in binary format. But how forward and backward conversion is performed?

For a forward conversion it is roughly as follows:

1. typeId is calculated from class name (some kind of hash function).
2. For each field name fieldId is calculated (hash function).
3. Based on fieldIds schemald is calculated (fields are ordered).
4. typeId and schemald are written in Header.
5. Field values are serialized and written one by one.
6. Footer with field offsets is written.

Backward conversion requires BinarySchema and a class name accessible for a given typeId and schemald, let's assume that we have them. Here is how it goes:

1. BinarySchema and a class name are obtained for typeId and schemald from object header.
2. New instance of Java object is instantiated using the class name.
3. Fields are read one by one and a field name is determined from BinarySchema. Field value is assigned to a corresponding field of the Java object.

In fact it is possible that BinarySchema and a class name is not accessible for a particular binary object. E.g. when your first operation is reading a value from cache. There is a special machinery for such cases – BinaryMetadata registration. Generally it allows to request remotely needed metadata for typeId /schemald including class names and schemas. Detailed description is outside of this document scope.

Raw data

Additionally for storing a number of field values binary object format is capable for storing raw bytes which are supposed for custom interpretation. One often example is serialization for classes implementing `Binarylizable`. `Binarylizable` object has user defined serialization/deserialization methods. And after binary serialization such object will contain no fields (and no schema).

Consider an example:

```java
public class Custom implements Binarylizable {
    private int val = 0x77;

    @Override public void writeBinary(BinaryWriter writer) throws BinaryObjectException {
        writer.rawWriter().writeInt(val);
    }

    @Override public void readBinary(BinaryReader reader) throws BinaryObjectException {
        val = reader.rawReader().readInt();
    }
}
```

And here are bytes after serialization:

```plaintext
67 01 25 00 // flags 25 00 -- 0000 0000 0010 0101 -- user type, raw data, compact footer
F3 BE 3A 90 // type id
22 A3 00 00 // hash code
1C 00 00 00 // length
00 00 00 00 // schema id (no schema)
18 00 00 00 // raw data offset
77 00 00 00 // int value 0x77
```

Also internally format allows to store raw data additionally to fields serialized using ordinary binary serialization technique. In such case serialized object structure will be different. Details are not described here as the case seems unusual.

Format problems
• Very large header – to wasteful for short tuples like pair of ints.
• Embedded hash – has meaning only for cache keys, useless for value.
• Byte array based hash calculation – inconsistent with objects allowing different representation for equal values (e.g. BigDecimal).
• Encoding null with 1 byte – wasteful.
• Variable-length field length is encoded as int (4 bytes) – wasteful and redundant as length can be determined from offsets.
• 1 type byte for each field – might be wasteful, if an external schema is maintained (like in SQL) type can be omitted.