PDX Serialization Internals

PDX serialization is the preferred serialization format for storing objects in Geode. PDX serialization is designed to serialize data as compactly as possible, while still providing the capability to read individual fields in a serialized object for query processing. PDX is also designed for ease of use, backwards and forwards compatibility between different versions of your objects.

Bruce Schuchardt wrote up a excellent article on using PDX on the pivotal blog: Data Serialization: How to Run Multiple Big Data Apps on a Single Data Store with GemFire. In this article, we're going to dive behind the scenes and look at how PDX implements these features.

Tell Us about Yourself

A serialized object must arrive at it's destination with enough information to deserialize the object. Self describing formats such as JSON or XML are easy to convert to objects because the description is embedded in the text. However systems designed for efficiency tend to separate the description of an object from the serialized data itself. With thrift and protobuf, that description is defined with an IDL and turned into code. With Avro, object descriptions are defined in JSON schemas and two systems can exchange schema definitions in a handshake.

PDX takes the approach of schema exchange and cranks it up a notch by taking advantage of Geode's data storage and distribution capabilities. With PDX, serialized object descriptions are called "types" and types are stored within the Geode distributed system in a PDX type registry. The serialized data contains a unique type id that can be used to look up the type from the registry.

blocked URL

A typical type

PDX objects are optimized for size, but also for random access to individual fields. A serialized byte array in PDX format looks something like this

```
| HEADER | LENGTH | DSID | ID | FIELDS              | OFFSETS  |

| 1 byte | 4 bytes|  1   | 3  | variable...         | offset...|
```

- **HEADER** - This is just a magic number to tell Geode this is a PDX object as opposed to an object serialized in some other format.
- **DSID** - The distributed system id for the system that generated this type.
- **ID** - An id that uniquely identifies what PDX type is used to deserialize this data.
- **FIELDS** - The actual data. Fixed sized fields are written in the number of bytes needed for the field, for example an int takes 4 bytes. Variable length fields (eg a string) are written with a length followed by field data.
- **OFFSETS** - The offsets to each *variable* length field (except for the first one). Offsets to fixed length fields are not included here because those offsets are stored in the PDX type definition. The size of each offset is based on the size of the overall serialized data. For example a byte array < 32K in size would have 2 byte offsets.

The PDX type associated with a given type ID has

- The type id
- The name of the Java (or C#, or C++) class to create when deserializing this blob.
- A map of fieldName -> PdxField. The PdxField contains the information necessary to locate the field in the serialized data. It has an offset to location of a variable length offset in the serialized data. It also has a second offset that is added to whatever offset is read from the serialized data. The sum of those offsets points to the location of the field in the serialized data.

You may observe that the serialized data could potentially be encoded slightly more compactly, for example by using something like varints used in protobuf to encode an integer in 1-5 bytes. PDX trades off this potential space savings for the ability to read a single field quickly.

To see how this works, imagine reading a single field 'price' using PdxInstance. PDX will look up the field based on the name, read the variable length offset for that field from the serialized bytes, and then seek to the location of the the field and read the single field. This capability to read individual fields is used extensively in the querying system to avoid deserializing more than necessary while evaluating a query.

How types get around

At the most basic level, PDX types are stored in a Geode replicated region called PdxTypes. That region is available on all peers within a distributed system. When a new type is being defined, the type registry uses a distributed lock to ensure that it obtains a unique id. It then puts the new type in the region using a transaction. The type is now known to all peers within the distributed system. If the system is using persistence, the type registry region will also be persistent so that the type information can be recovered on restart.

Clients obtain types lazily when they try to deserialize an object. If a type is not known to a client, the client fetches the type from a server and caches it in it's own local type registry.

ID generation across WAN sites

Each WAN site can independently assign ids to types. To ensure different WAN sites do not assign the same ID to different types, the type ID is prefixed with a distributed system id which is unique for each WAN site. When a type is defined in one WAN site, it is added to the queue to be sent to other WAN sites to ensure they receive the type information before the data.

Forwards and Backwards Compatibility

An important thing to observe about the type registry is that all versions of an object are stored in the type registry. When a member deserializes an object, it looks up the PDX type that matches the id in the serialized data. From the point of view of the member deserializing the object, that type may have some missing or extra fields because it was serialized with a different version of the code.

The PDX system will fill in missing fields with a default value. For *extra* fields, the PDX system will save these unread fields and tack them back on to the serialized data if the same object is serialized again.

One interesting consequence of this behavior is that the actual serialized object contains the *union* of all of the fields from different versions of a class. For example if two client applications are interacting with a Person object, and one application adds a twitter handle and the other adds facebook id, the resulting Person will have both fields in the serialized data, even though *neither* application has both fields on the class. Because PDX automatically generates a type ID for any new object description it sees, it happily generates a new type id for this new hybrid Person object with both fields.

Summary

PDX is an easy to use serialization system that is fast for serialization, deserialization, and random field access. I hope this behind the scenes look helps you understand some of the design goals and capabilities of PDX.