Plugging Infinispan With User Defined Externalizers

Exported from JBoss Community Documentation Editor at 2013-05-22 13:50:30 EDT

Table of Contents

1 Introduction

One of the key aspects of Infinispan is that it often needs to marshall/unmarshall objects in order to provide some of its functionality. For example, if it needs to store objects in a write-through or write-behind cache store, the stored objects need marshalling. If a cluster of Infinispan nodes is formed, objects shipped around need marshalling. Even if you enable lazy deserialization, objects need to be marshalled so that they can be lazily unmarshalled with the correct classloader.

Using standard JDK serialization is slow and produces payloads that are too big and can affect bandwidth usage. On top of that, JDK serialization does not work well with objects that are supposed to be immutable. In order to avoid these issues, Infinispan uses JBoss Marshalling for marshalling/unmarshalling objects. JBoss Marshalling is fast, produces very space efficient payloads, and on top of that during unmarshalling, it enables users to have full control over how to construct objects, hence allowing objects to carry on being immutable.

Starting with 5.0, users of Infinispan can now benefit from this marshalling framework as well, and they can provide their own externalizer implementations, but before finding out how to provide externalizers, let's look at the benefits they bring.

2 Benefits of Externalizers

The JDK provides a simple way to serialize objects which, in its simplest form, is just a matter of extending java.io.Serializable, but as it's well known, this is known to be slow and it generates payloads that are far too big. An alternative way to do serialization, still relying on JDK serialization, is for your objects to extend java.io.Externalizable. This allows for users to provide their own ways to marshall/unmarshall classes, but has some serious issues because, on top of relying on slow JDK serialization, it forces the class that you want to serialize to extend this interface, which has two side effects: The first is that you're forced to modify the source code of the class that you want to marshall/unmarshall which you might not be able to do because you either, don't own the source, or you don't even have it. Secondly, since Externalizable implementations do not control object creation, you're forced to add set methods in order to restore the state, hence potentially forcing your immutable objects to become mutable.

Instead of relying on JDK serialization, Infinispan uses JBoss Marshalling to serialize objects and requires any classes to be serialized to be associated with an Externalizer interface implementation that knows how to transform an object of a particular class into a serialized form and how to read an object of that class from a given input. Infinispan does not force the objects to be serialized to implement Externalizer. In fact, it is recommended that a separate class is used to implement the Externalizer interface because, contrary to JDK serialization, Externalizer implementations control how objects of a particular class are created when trying to read an object from a stream. This means that readObject() implementations are responsible of creating object instances of the target class, hence giving users a lot of flexibility on how to create these instances (whether direct instantiation, via factory or reflection), and more importantly, allows target classes to carry on being immutable. This type of externalizer architecture promotes good OOP designs principles, such as the principle of single responsibility.

It's quite common, and in general recommended, that Externalizer implementations are stored as inner static public classes within classes that they externalize. The advantages of doing this is that related code stays together, making it easier to maintain. In Infinispan, there are two ways in which Infinispan can be plugged with user defined externalizers:

2.1 User Friendly Externalizers

In the simplest possible form, users just need to provide an Externalizer implementation for the type that they want to marshall/unmarshall, and then annotate the marshalled type class with {@link SerializeWith} annotation indicating the externalizer class to use. For example:

```
import org.infinispan.marshall.Externalizer;
import org.infinispan.marshall.SerializeWith;

@SerializeWith(Person.PersonExternalizer.class)
public class Person {

   final String name;
   final int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public static class PersonExternalizer implements Externalizer<Person> {
      @Override
      public void writeObject(ObjectOutput output, Person person)
            throws IOException {
         output.writeObject(person.name);
         output.writeInt(person.age);
      }

      @Override
      public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
         return new Person((String) input.readObject(), input.readInt());
      }
   }
}
```

At runtime JBoss Marshalling will inspect the object and discover that's marshallable thanks to the annotation and so marshall it using the externalizer class passed. To make externalizer implementations easier to code and more typesafe, make sure you define type <T> as the type of object that's being marshalled/unmarshalled.

Even though this way of defining externalizers is very user friendly, it has some disadvantages:

- Due to several constraints of the model, such as support different versions of the same class or the need to marshall the Externalizer class, the payload sizes generated via this method are not the most efficient.
- This model requires for the marshalled class to be annotated with {@link SerializeWith} but a user might need to provide an Externalizer for a class for which source code is not available, or for any other constraints, it cannot be modified.
- The use of annotations by this model might be limiting for framework developers or service providers that try to abstract lower level details, such as the marshalling layer, away from the user.

If you're affected by any of these disadvantages, an alternative method to provide externalizers is available via more advanced externalizers:

2.2 Advanced Externalizers

AdvancedExternalizer provides an alternative way to provide externalizers for marshalling/unmarshalling user defined classes that overcome the deficiencies of the more user-friendly externalizer definition model explained in Externalizer. For example:

```
import org.infinispan.marshall.AdvancedExternalizer;

public class Person {

   final String name;
   final int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public static class PersonExternalizer implements AdvancedExternalizer<Person> {
      @Override
      public void writeObject(ObjectOutput output, Person person)
            throws IOException {
         output.writeObject(person.name);
         output.writeInt(person.age);
      }

      @Override
      public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
         return new Person((String) input.readObject(), input.readInt());
      }

      @Override
      public Set<Class<? extends Person>> getTypeClasses() {
         return Util.<Class<? extends Person>>asSet(Person.class);
      }

      @Override
      public Integer getId() {
         return 2345;
      }
   }
}
```

The first noticeable difference is that this method does not require user classes to be annotated in anyway, so it can be used with classes for which source code is not available or that cannot be modified. The bound between the externalizer and the classes that are marshalled/unmarshalled is set by providing an implementation for getTypeClasses() which should return the list of classes that this externalizer can marshall:

2.2.1 Linking Externalizers with Marshaller Classes

Once the Externalizer's readObject() and writeObject() methods have been implemented, it's time to link them up together with the type classes that they externalize. To do so, the Externalizer implementation must provide a getTypeClasses() implementation. For example:

```
import org.infinispan.util.Util;
...
@Override
public Set<Class<? extends ReplicableCommand>> getTypeClasses() {
  return Util.asSet(LockControlCommand.class, RehashControlCommand.class,
      StateTransferControlCommand.class, GetKeyValueCommand.class,
      ClusteredGetCommand.class, MultipleRpcCommand.class,
      SingleRpcCommand.class, CommitCommand.class,
      PrepareCommand.class, RollbackCommand.class,
      ClearCommand.class, EvictCommand.class,
      InvalidateCommand.class, InvalidateL1Command.class,
      PutKeyValueCommand.class, PutMapCommand.class,
      RemoveCommand.class, ReplaceCommand.class);
}
```

In the code above, ReplicableCommandExternalizer indicates that it can externalize several type of commands. In fact, it marshalls all commands that extend ReplicableCommand interface, but currently the framework only supports class equality comparison and so, it's not possible to indicate that the classes to marshalled are all children of a particular class/interface.

However there might sometimes when the classes to be externalized are private and hence it's not possible to reference the actual class instance. In this situations, users can attempt to look up the class with the given fully qualified class name and pass that back. For example:

```
@Override
public Set<Class<? extends List>> getTypeClasses() {
  return Util.<Class<? extends List>>asSet(
         Util.loadClass("java.util.Collections$SingletonList"));
}
```

2.2.2 Externalizer Identifier

Secondly, in order to save the maximum amount of space possible in the payloads generated, advanced externalizers require externalizer implementations to provide a positive identified via getId() implementations or via XML/programmatic configuration that identifies the externalizer when unmarshalling a payload. In order for this to work however, advanced externalizers require externalizers to be registered on cache manager creation time via XML or programmatic configuration which will be explained in next section. On the contrary, externalizers based on Externalizer and SerializeWith require no pre-registration whatsoever. Internally, Infinispan uses this advanced externalizer mechanism in order to marshall/unmarshall internal classes.

So, getId() should return a positive integer that allows the externalizer to be identified at read time to figure out which Externalizer should read the contents of the incoming buffer, or it can return null. If getId() returns null, it is indicating that the id of this advanced externalizer will be defined via XML/programmatic configuration, which will be explained in next section.

Regardless of the source of the the id, using a positive integer allows for very efficient variable length encoding of numbers, and it's much more efficient than shipping externalizer implementation class information or class name around. Infinispan users can use any positive integer as long as it does not clash with any other identifier in the system. It's important to understand that a user defined externalizer can even use the same numbers as the externalizers in the Infinispan Core project because the internal Infinispan Core externalizers are special and they use a different number space to the user defined externalizers. On the contrary, users should avoid using numbers that are within the pre-assigned identifier ranges which can be found at the end of this article. Infinispan checks for id duplicates on startup, and if any are found, startup is halted with an error.

When it comes to maintaining which ids are in use, it's highly recommended that this is done in a centralized way. For example, getId() implementations could reference a set of statically defined identifiers in a separate class or interface. Such class/interface would give a global view of the identifiers in use and so can make it easier to assign new ids.

2.2.3 Registering Advanced Externalizers

The following example shows the type of configuration required to register an advanced externalizer implementation for Person object shown earlier stored as a static inner class within it:

XML:

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers>
        <advancedExternalizer externalizerClass="Person$PersonExternalizer"/>
      </advancedExternalizers>
    </serialization>
  </global>
  ...
</infinispan>
```

Programmatically:

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
   .addAdvancedExternalizer(new Person.PersonExternalizer());
```

As mentioned earlier, when listing these externalizer implementations, users can optionally provide the identifier of the externalizer via XML or programmatically instead of via getId() implementation. Again, this offers a centralized way to maintain the identifiers but it's important that the rules are clear: An AdvancedExternalizer implementation, either via XML/programmatic configuration or via annotation, needs to be associated with an identifier. If it isn't, Infinispan will throw an error and abort startup. If a particular AdvancedExternalizer implementation defines an id both via XML/programmatic configuration and annotation, the value defined via XML/programmatically is the one that will be used. Here's an example of an externalizer whose id is defined at registration time:

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers>
        <advancedExternalizer id="123"
                              externalizerClass="Person$PersonExternalizer"/>
      </advancedExternalizers>
    </serialization>
  </global>
  ...
</infinispan>
```

Programmatically:

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
   .addAdvancedExternalizer(123, new Person.PersonExternalizer());
```

Finally, a couple of notes about the programmatic configuration. GlobalConfiguration.addExternalizer() takes varargs, so it means that it is possible to register multiple externalizers in just one go, assuming that their ids have already been defined via @Marshalls annotation. For example:

```
builder.serialization()
   .addAdvancedExternalizer(new Person.PersonExternalizer(),
                            new Address.AddressExternalizer());
```

2.2.4 Preassigned Externalizer Id Ranges

This is the list of Externalizer identifiers that are used by Infinispan based modules or frameworks. Infinispan users should avoid using ids within these ranges.

Infinispan Tree Module:	1000 - 1099
Infinispan Server Modules:	1100 - 1199
Hibernate Infinispan Second Level Cache:	1200 - 1299
Infinispan Lucene Directory:	1300 - 1399
Hibernate OGM:	1400 - 1499
Hibernate Search:	1500 - 1599