Gemfire Tips and Tricks

Tim Dalsing

# Introduction

The purpose of this document is to describe tips, tricks, and best practices for use of Gemfire. This document is intended as a cookbook and reference to guide the developer in making optimal design and coding choices, and to provide samples that are beyond the simple examples provided in the Gemfire distribution.

Each section describes a particular problem and the technique that is used to solve the problem.

# IDs and ID Generation

Selecting a data type for representing unique IDs for a distributed system is special challenge. A common strategy is to use values generated by the UUID class in the JDK. However, the resulting IDs have several problems:

The hashCode tends to be very non-uniform for values generated by UUID. This results in non-uniform distribution of data across partition regions.

Since the String value is fairly long, the equals method is somewhat slow. Since the key uses the hashCode and equals to determine identity, the equals method is called many times. This can affect performance in some situations.

A numeric value, such as int or long, is a better choice. If the ID is generated by simply incrementing a value, the distribution is perfectly uniform. Also, hashCode and equals are very fast and efficient. If the smaller range of values afforded by an int are sufficient, it is the best choice since the hashCode is simply the value and there won’t be any hashtable collisions. A long is required for most situations if the range of IDs cannot be determined in advance or controlled in some way.

Creating unique IDs for the keys to a region is a common problem. On a single server node this problem is relatively easy to solve. However, on a distributed system, where the IDs must be unique across the entire cluster, the problem is more difficult to solve.

There are a number of ways to generate unique IDs:

1. The next ID is stored in a partition region that is persistent. The key to the ID region is the domain of the ID (e.g., the name of the region where the ID is actually used). A Function is used to get the next ID and update the current ID. Since the Function can be configured to write only to the primary the server node the call is routed to be predictable. The update of the next ID must be synchronized, so this technique is not appropriate for high volume applications.
2. Each server node has a prefix that is the high byte for the next ID. Each server gets its own prefix. The first value for the ID is the prefix shifted to the highest byte. Then the ID is incremented from the initial value. A local region with persistence is used to store the next ID. A Function that randomly picks a server node to get the next ID is used. This option is very fast and efficient since the generation of IDs is distributed across cluster. Making sure the prefix is different on each server node is the most difficult aspect of this technique. This technique also works on applications that use the Gemfire client.

If the source system for the data supplies its own ID, it can be mapped to an internal ID using an xref region.

The Gemfire Utilities include a key framework with ID generation classes.

# Key Classes

A common design strategy in Gemfire applications is to use standard Java wrappers, such as Integer or Long, or String as the key class. A better solution is to use a different key class for each domain class. For example, an Account class should have an accompanying AccountKey class. This has several advantages:

1. The key-constraint attribute of the region ensures the right value is used when putting data into a region. If a String or Long is used it can easily be the wrong value.
2. If ID is changed (such as from String to Long) all of the method signatures that have the key class do not have to be changed.
3. Compound key structures that are used with co-location can be created.

The Gemfire Utilities key framework includes a base class for key classes.

# Xrefs

There are a number of situations where a query may not be the most efficient way to find objects in the cache. One important use case is the need to perform operations on a subset of the data in a region based on some criteria. A query will require returning a result set, then operating on the results. A more efficient option is to use xrefs that allow direct access to the keys and data without the need to build, parse, and execute a query.

An xref region has a key that represents some value that is “indexed,” that is, the lookup value. This could be an account number, social security number, etc. The value in the xref region is a list of keys for objects that match the indexed value. The xref region is normally a local region without persistence (since the data is in other regions it can be easily rebuilt). A CacheListener is used to maintain the xrefs.

If the regions involved in the xref are partition, a PartitionListener is used to rebuild the xrefs when a bucket is moved or a secondary becomes primary. This is due to the fact that the CacheListener only fires on the primary, so the xrefs won’t exist on the secondaries.

# Partition vs. Replicate Regions

The simplistic analysis that drives the decision to use partition or replicate regions mostly involves whether the data will fit in a single node. If the data comfortable fits in a single node a replicate region should be used. If the data does not fit in a single node use a partition region. However, there are a few subtleties that are missing from this analysis:

1. Replicate regions do not perform as well for write-heavy use cases. Partition regions are better suited for this even if the amount of data is very small. This is particularly true for large clusters with many nodes.
2. Writing the same key in a replicate region on different servers simultaneously can result in locking problems and poor performance. A Function that is called via onServers should never be used to write into a replicate region for this reason.
3. A partition region can be used to distribute logic execution on many servers, and the partition region doesn’t even need to have any data.
4. Writes to a partition region always go the primary if a Function is used with the optimizeForWrite option. This is useful for counters and accumulators that would normally require a distributed lock. Since every call for the same key goes to the same node a local lock can be used instead. The region may have very little data.
5. A smaller heap size is generally desirable since it lessens the workload on the garbage collector. This is especially true if the write volume is high, or there is significant amount of data subject to expiration. In some situations it may be better to use a partition region for the reference data, even at the expense of some addition network traffic.

# Local Regions

To some degree local regions in Gemfire don’t seem particularly important, but they have many uses. Some examples include:

1. Store xrefs as described above.
2. Store generated objects that cannot be serialized and distributed. For example, a local region can contain a generated ML algorithm. A CacheListener on the region that is the source of the data is used to maintain the generated objects in the local region. An example of this technique is included in the Gemfire Examples.
3. Hold temporary workflow or work-in-progress objects. A local region would hold the temporary data in conjunction with a partition region that holds the non-temporary data. Since writes to partition regions always go to the primary the temporary data will reside in the same node as the primary bucket in the partition region.
4. The next value in an ID sequence can be stored in a local region that is persistent, as described above.

# Server-side Code Deployment

## Traditional Strategies

The traditional techniques for deploying code in a Gemfire server node are:

1. Deploy JARs via GFSH.
2. Include code in JARs on the Gemfire classpath and reference the classes in cache.xml.
3. Embed the Gemfire server inside a WAR using Spring and deploy the WAR to Tomcat.
4. Embed the Gemfire server inside a Spring application, especially Spring Boot.

All of these techniques have disadvantages.

### Deploy via GFSH

This technique allows code to be deployed to a running server without restarting it, which is especially important if the node is managing a large amount of data. The disadvantages of this technique are:

1. Deploying code is coarse-grained. An entire JAR must be deployed each time.
2. The object lifecycle provided by Gemfire is relatively simplistic and doesn’t support dependency injection or other advanced lifecycle features.
3. The dependencies between code and region definitions are a bit tricky to manage. For example, a new CacheListener must be uploaded via a JAR before the region definition can be changed. If a CacheListener is removed the region definition must be changed before the JAR without the CacheListener is uploaded.

### Code on Gemfire Classpath

Managing dependencies between code and region definitions is easy, but the server must be restarted each time the code is updated. Rolling restarts can be carefully used in some situations, but any data managed by Gemfire must either be reloaded from disk or replicated from other nodes. Rebalance must be used for partition regions.

### Embedded in WAR

This technique allows deployments to be uploaded while Tomcat is still running, but since a new ClassLoader is created for each new deployment a new instance of Gemfire Cache is created each time. The data must be reloaded as if the server was restarted, and existing data must be garbage collected.

### Embedded in Spring

The considerable power of Spring is available to the application code, especially dependency injection. All of the Spring modules can be utilized. The disadvantage is that the servers must be restarted with all the problems of data reloading, replication, and rebalancing.

## Other Strategies

Several other options are available for deploying server-side code. These options provide an alternative to the traditional options and have a number of benefits.

### Rules Engine

Since the code running in the server often is implementing business rules, a rules engine is a natural choice for executing business logic. Most rule engines allow rules to be dynamically modified, and some even provide repositories to store the rules. They also can have Java code embedded inside the rule, so actual code can be deployed to server. Rule engines commonly compile the rule to Java so the execution is almost as fast as plain code.

Gemfire Examples include a component that demonstrates the use of Drools (http://drools.org/) inside of a Gemfire server node.

### Scripting

Scripting engines, such as Groovy (http://www.groovy-lang.org), JRuby, and Jython, allow for dynamic modification of scripts. Groovy is a Java dialect, so the learning curve for Java developers is relatively short. Groovy provides an API, GroovyScriptEngine (http://www.groovy-lang.org/integrating.html), which is specifically designed for embedded scripting.

Gemfire Examples includes a component that demonstrates the use of Groovy scripts inside Gemfire.

# Serialization

Java serialization should not be used on any class that is stored in a Gemfire region, even if PDX is configured. If the PDX configuration has a typo in it, or if a package or class is missing from the configuration, Java serialization will silently be used. If the data is stored on disk it can be orphaned if the PDX configuration is fixed or if the class is modified.

# Client-side Caching

Client-side caching in Gemfire seems like a very useful feature, since it reduces the trips to the server, but it has a number of disadvantages. It is useful but it must be utilized carefully. Some of the issues are:

1. Garbage collection and heap issues that are common for the server nodes must also be considered for the client nodes.
2. If the client is deployed in a cloud app, such as in CloudFoundry, the amount of heap is fairly limited.
3. Data can get very stale if subscriptions are not used.
4. If subscriptions are used the connection from the client to the server can get very chatty, especially if there are a large number of clients.