GlobalLogic®

Elastic Java Heap for Scaling and Clustering Enterprise Applications

Sourabh Goel, Engineering Manager
Anshul Rohilla, Engineering Development

Noida, India

Elastic memory attempts to address two fundamental limitations imposed by Java memory management: limited heap sizes and garbage collection (GC) overhead. These drawbacks prevent Java applications from leveraging random access memory in order to store huge amounts of data objects for high throughput. This paper will (a) expand upon the research currently being conducted in the field of off-heap management of Java objects and (b) propose an approach to augmenting the cost-effective scaling-up/out of enterprise applications.

www.globallogic.com

Table of Contents

Introduction

Before the advent of 64-bit platforms, a JVM was mostly limited to 2GB of heap size, given the inherent addressing limitations of a 32-bit architecture. With 64-bit JVMs and operating systems now becoming ubiquitous, developers have attempted to test how much heap space can be allocated to a JVM instance with measurable performance gains. The maximum heap size allocation has been estimated at around 16GB, beyond which performance gains begin to taper off. This behavior is attributed to overheads associated with longer GC cycles, as well as memory pagination issues related to the size of a single addressable unit of memory and management of these pages (aka, thrashing).

Distributed or replicated caching are common strategies for creating fault tolerant, highly available and scalable enterprise applications. In Enterprise Java, popular caching solutions like Memcached for distributed caching and Infinispan for both replicated and distributed caching use an object map as data storage for supporting put() and get() operations. This object map is managed across a cluster to provide scalability. A cluster created by using these caching solutions is also subject to limitations imposed by Java heap. In their current default implementations, they can only support the scaling-up of heap to 16GB per server/node instance on the cluster.

If one were to integrate Java's elastic memory concept with caching/clustering solutions by creating the object map in an off-heap location (at least theoretically), only the amount of physical memory present on a node would act as a deterrent to scaling. Given the fact that RAM costs have been coming down significantly over the last few years, organizations have more flexibility in creating a solution that balances both cost and throughput.

Off-Heap Object Management

The management of Java objects in a memory space above and beyond limited heap size – also referred to as off-heap – has been proposed as a potential solution to this problem. One of the key strategies proposed for the off-heap memory management of objects is driven by the use of the ByteBuffer class in Java. The ByteBuffer class was introduced in J2SE 1.4. Using this class, a program may attempt to utilize a system's memory, by-passing Java's own management of objects, although the responsibility of boundary checks and mapping of memory chunks onto an object in Java would be delegated to the program itself. Moreover, the program has to ensure that it does not attempt to go beyond the permissible address space of memory; otherwise, it may be subjected to the quirks of the underlying operating system's own memory management algorithms for individual processes.

A. MapDB: An Off-Heap Java Solution

MapDB is an open source project available under Apache License 2.0. Using MapDB, we can create an instance of either a concurrent Tree or HashMap. All the objects that are pushed onto this Tree or HashMap using a put() operation will be stored in an off-heap location. Below is a code snippet that we used for creating an instance of MapDB.

```
DB db = DBMaker.newDirectMemoryDB().make();
Map map = db.getTreeMap("MyCache");
```

We executed a test case on a Windows 7 machine with 4GB of maximum RAM available. 256MB max heap was allocated to the JVM. The below code was executed under this JVM configuration to insert a set of ten million plain vanilla string objects into a map created on a standard Java heap.

```
Map<Object, Object> c = new HashMap<Object,
Object>();
for (long ctr = 0;; ctr ++; ctr <1000000) {
c.put(String.valueOf(ctr), ctr);
if (ctr % 1000000 == 0) {
System.out.println("" + ctr);
}
}
```

The JVM ran out of heap space and threw the java.lang. OutOfMemoryError: (i.e., Java heap space exception) after inserting a little over five million objects in the Map.

The same test case was executed using MapDb. The instance of the Map used was obtained using MapDB, and

objects were inserted into it. It is to be noted that the max heap space allocated to JVM remained 256MB. Below is the code snippet used for executing this operation.

```
DB db = DBMaker.newDirectMemoryDB().make();
Map<Object, Object> c = db.getTreeMap("mapDb");
for (long ctr = 0;; ctr ++; ctr <1000000) {
        c.put(String.valueOf(ctr), ctr);
        if (ctr % 1000000 == 0) {
                System.out.println("" + ctr);
        }
}
```

It was observed that the system successfully inserted ten million records without throwing any exception. We executed multiple iterations of our test case with consistent results, which convinced us that MapDB was indeed using off-heap space for storing all those objects.

Integrating Distributed Cache Solutions with Off-Heap Memory Management of Java Objects

Memcahced and Infinispan provide open source, distributed object caching solutions that enable enterprise applications to achieve high throughput by reducing the database load. Both of these solutions utilize an in-memory key-value store for small chunks of arbitrary data (e.g., strings, objects) resulting from database calls, API calls, or page rendering.

Memcached offers only a distributed mode of caching; it does not support replication. Distribution is just one of the cache mode options in Infinispan apart from replication and invalidation. In a replicated cache, all nodes in a cluster hold all objects (i.e., if an object exists on one node, it will also exist on all other nodes). Since Memcached does not support replication, it cannot be used in solutions where high availability is desired.

Infinispan supports a distributed cache mode with high fault tolerance and availability so that a certain user-defined number of copies are replicated. A distributed cache provides a far greater degree of scalability than a replicated cache.

A. Extending Infinispan to Use an Off-Heap Map

Infinispan is an open source in-memory data grid platform. It exposes a JSR-107 compatible cache interface (which in turn extends java.util.Map) in which you can store objects.

The value offered by Infinispan is primarily due to the distributed mode that it provides, where caches cluster together and expose a massive heap and high availability. For example, assume you have twenty nodes in your cluster, with each node running on a 64-bit platform and a 64-bit JVM. Also assume that each node has 16GB of space to dedicate to a distributed grid. If you want three copies per data item, you would get a 140GB memory backed virtual heap that is efficiently accessible from anywhere in the grid. If a server fails, the grid simply creates new copies of the lost data and puts them on other servers. Applications looking for massive throughput are no longer forced to delegate the majority of their data lookups to a large single database server (i.e., the bottleneck that exists in most enterprise applications).

To create an instance of the Map that will be used for storing an object in Infinispan, we used the below code fragment from the DefaultCacheManager class.

```
ConcurrentMap<String, CacheWrapper>
= ConcurrentMapFactory.makeConcurrentMap();
```

The below code snippet demonstrates what would happen if the Map created in this class was replaced by an off-heap map created and backed by MapDB.

```
ConcurrentMap<String, CacheWrapper> caches ;
 DB db = DBMaker.newMemoryDB().make();
 caches = db.getHashMap("mapDB");
```

One can theoretically harness an infinite amount of highly available heap space per server instance. Let's extrapolate on our earlier example of twenty nodes in a cluster, but assume that each node can now be potentially assigned to a not-so-expensive 64GB of space for infinispan. If you wanted three copies per data item, you would get a 560GB memory backed virtual heap!

B. Extending Memcached to Use an Off-Heap Map

Similar to the use case leveraged for Infinispan, one can build a clustered array of 560GB of virtual, highly available heap using Memcached. Memcache uses its class ConcurrentLinkedHashMap to create an instance of the Map to store Java objects.

```
ConcurrentMap<K, Node<K, V>> data =new
ConcurrentHashMap<K, Node<K,
V>>(maximumCapacity, 0.75f, concurrencyLevel);
```

This can be replaced by the below code snippet to leverage the off-heap Map created by MapDB.

```
ConcurrentMap<K, Node<K, V>> data;
DB db = DBMaker.newMemoryDB().make();
data = db.getHashMap("mapDB");
```

Conclusion

The off-heap management of Java objects for applications running on relatively low-end server nodes can be integrated with distributed caching solutions. This strategy can offer significant scaling options to organizations looking for highly scalable yet cost-effective options for delivering the expected throughput to their expanding user base while managing increasingly vast amounts of data.

References

We would like to acknowledge the excellent work done by Jan Kotek on his MaDB project, which allowed us to experiment with off-heap Java memory management. We would also like to acknowledge the great work done by the Infinispan and Memchaced teams, who have created these state-of-the-art open source caching solutions for the community.

1. Jboss Infinispan
http://www.jboss.org/infinispan

2. Jan Kotek –MapDb
http://www.mapdb.org

3. Memcache
http://memcached.org

4. Oracle Java
http://www.oracle.com/us/technologies/java/standard-edition/overview/index.html

5. Microsoft Windows
http://windows.microsoft.com/en-IN/windows7/products/home

About GlobalLogic Inc.

GlobalLogic is a full-lifecycle product development services leader that combines deep domain expertise and cross-industry experience to connect makers with markets worldwide.Using insight gained from working on innovative products and disruptive technologies, we collaborate with customers to show them how strategic research and development can become a tool for managing their future. We build partnerships with market-defining business and technology leaders who want to make amazing products, discover new revenue opportunities, and accelerate time to market.

For more information, visit www.globallogic.com

GlobalLogic®

Contact

Emily Younger
+1.512.394.7745
emily.younger@globallogic.com