Embedding Infinispan 10.0

Table of Contents

Chapter 1. Using Infinispan as an embedded cache in Java SE

Running Infinispan in embedded mode is very easy. First, we'll set up a project, and then we'll run Infinispan, and start adding data.

embedded-cache quickstart

All the code discussed in this tutorial is available in the embedded-cache quickstart.

1.1. Creating a new Infinispan project

The only thing you need to set up Infinispan is add it's dependencies to your project.

1.1.1. Maven users

If you are using Maven (or another build system like Gradle or Ivy which can use Maven dependencies), then this is easy. Just add the following to the `<dependencies>` section of your `pom.xml`:

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-embedded</artifactId>
  <!-- Replace ${version.infinispan} with the
  version of Infinispan that you're using. -->
  <version>${version.infinispan}</version>
</dependency>
```

Which version of Infinispan should I use?

We recommend using the latest stable version of Infinispan. All releases are displayed on the downloads page.

Alternatively, you can use the POM from the quickstart that accompanies this tutorial.

1.1.2. Ant users

If you are using Ant, or another build system which doesn't provide declarative dependency management, then the Infinispan distribution zip contains a lib/ directory. Add the contents of this to the build classpath.

1.2. Running Infinispan on a single node

In order to run Infinispan, we're going to create a `main()` method in the `Quickstart` class. Infinispan comes configured to run out of the box; once you have set up your dependencies, all you need to do

to start using Infinispan is to create a new cache manager and get a handle on the default cache.

Quickstart.java

```
public class Quickstart {

   public static void main(String args[]) throws Exception {
      Cache<Object, Object> c = new DefaultCacheManager().getCache();
   }

}
```

We now need a way to run the main method! To run the Quickstart main class: If you are using Maven:

```
$ mvn compile exec:java
-Dexec.mainClass="org.infinispan.quickstart.embeddedcache.Quickstart"
```

You should see Infinispan start up, and the version in use logged to the console.

Congratulations, you now have Infinispan running as a local cache!

1.3. Use the default cache

Infinispan exposes a Map-like, JSR-107-esque interface for accessing and mutating the data stored in the cache. For example:

DefaultCacheQuickstart.java

```
// Add a entry
cache.put("key", "value");
// Validate the entry is now in the cache
assertEqual(1, cache.size());
assertTrue(cache.containsKey("key"));
// Remove the entry from the cache
Object v = cache.remove("key");
// Validate the entry is no longer in the cache
assertEqual("value", v);
```

Infinispan offers a thread-safe data-structure:

DefaultCacheQuickstart.java

```
// Add an entry with the key "key"
cache.put("key", "value");
// And replace it if missing
cache.putIfAbsent("key", "newValue");
// Validate that the new value was not added
```

By default entries are immortal but you can override this on a per-key basis and provide lifespans.

DefaultCacheQuickstart.java

```
//By default entries are immortal but we can override this on a per-key basis and
provide lifespans.
cache.put("key", "value", 5, SECONDS);
assertTrue(cache.containsKey("key"));
Thread.sleep(10000);
```

to run using maven:

```
$ mvn compile exec:java
-Dexec.mainClass="org.infinispan.quickstart.embeddedcache.DefaultCacheQuickstart"
```

1.4. Use a custom cache

Each cache in Infinispan can offer a different set of features (for example transaction support, different replication modes or support for eviction), and you may want to use different caches for different classes of data in your application. To get a custom cache, you need to register it with the manager first:

CustomCacheQuickstart.java

```
public static void main(String args[]) throws Exception {
   EmbeddedCacheManager manager = new DefaultCacheManager();
   manager.defineConfiguration("custom-cache", new ConfigurationBuilder()
             .eviction().strategy(LIRS).maxEntries(10)
             .build());
   Cache<Object, Object> c = manager.getCache("custom-cache");
}
```

The example above uses Infinispan's fluent configuration, which offers the ability to configure your cache programmatically. However, should you prefer to use XML, then you may. We can create an identical cache to the one created with a programmatic configuration:

To run using maven:

```
$ mvn compile exec:java
-Dexec.mainClass="org.infinispan.quickstart.embeddedcache.CustomCacheQuickstart"
```

infinispan.xml

```
<infinispan
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:infinispan:config:10.0
http://www.infinispan.org/schemas/infinispan-config-10.0.xsd"
        xmlns="urn:infinispan:config:10.0">

   <cache-container default-cache="default">
       <local-cache name="xml-configured-cache">
          <memory>
            <object size="100"/>
          </memory>
       </local-cache>
   </cache-container>

</infinispan>
```

We then need to load the configuration file, and use the programmatically defined cache:

XmlConfiguredCacheQuickstart.java

```
public static void main(String args[]) throws Exception {
    Cache<Object, Object> c = new DefaultCacheManager("infinispan.xml").getCache("xml-
configured-cache");
}
```

To run using maven:

```
$ mvn compile exec:java
-Dexec.mainClass="org.infinispan.quickstart.embeddedcache.XmlConfiguredCacheQuickstart
"
```

Chapter 2. Using Infinispan as an embedded data grid in Java SE

Clustering Infinispan is simple. Under the covers, Infinispan uses JGroups as a network transport, and JGroups handles all the hard work of forming a cluster.

clustered-cache quickstart

All the code discussed in this tutorial is available in the clustered-cache quickstart.

2.1. Sharing JGroups channels

By default all caches created from a single CacheManager share the same JGroups channel and multiplex RPC messages over it. In this example caches 1, 2 and 3 all use the same JGroups channel.

```
EmbeddedCacheManager cm = new DefaultCacheManager("infinispan.xml");
Cache<Object, Object> replSyncCache = cm.getCache("replSyncCache");
Cache<Object, Object> replAsyncCache = cm.getCache("replAsyncCache");
Cache<Object, Object> invalidationSyncCache = cm.getCache("invalidationSyncCache");
```

2.2. Running Infinispan in a cluster

It is easy set up a clustered cache. This tutorial will show you how to create two nodes in different processes on the same local machine. The quickstart follows the same structure as the embedded-cache quickstart, using Maven to compile the project, and a main method to launch the node.

If you are following along with the quickstarts, you can try the examples out.

The quickstart defines two clustered caches, one in *replication mode* and one *distribution mode*.

2.2.1. Replicated mode

To run the example in replication mode, we need to launch two nodes from different consoles. For the first node:

```
$ mvn exec:java -Djava.net.preferIPv4Stack=true -Djgroups.bind_addr=127.0.0.1
-Dexec.mainClass="org.infinispan.quickstart.clusteredcache.Node" -Dexec.args="A"
```

And for the second node:

```
$ mvn exec:java -Djava.net.preferIPv4Stack=true -Djgroups.bind_addr=127.0.0.1
-Dexec.mainClass="org.infinispan.quickstart.clusteredcache.Node" -Dexec.args="B"
```

Note: You need to set `-Djava.net.preferIPv4Stack=true` because the JGroups configuration uses IPv4

multicast address. Normally you should not need `-Djgroups.bind_addr=127.0.0.1`, but many wireless routers do not relay IP multicast by default.

Each node will insert or update an entry every second, and it will log any changes.

2.2.2. Distributed mode

To run the example in distribution mode and see how entries are replicated to only two nodes, we need to launch three nodes from different consoles. For the first node:

```
$ mvn compile exec:java -Djava.net.preferIPv4Stack=true
-Dexec.mainClass="org.infinispan.quickstart.clusteredcache.Node" -Dexec.args="-d A"
```

For the second node:

```
$ mvn compile exec:java -Djava.net.preferIPv4Stack=true
-Dexec.mainClass="org.infinispan.quickstart.clusteredcache.Node" -Dexec.args="-d B"
```

And for the third node:

```
$ mvn compile exec:java -Djava.net.preferIPv4Stack=true
-Dexec.mainClass="org.infinispan.quickstart.clusteredcache.Node" -Dexec.args="-d C"
```

The same as in replication mode, each node will insert or update an entry every second, and it will log any changes. But unlike in replication mode, not every node will see every modification.

You can also see that each node holds a different set of entries by pressing Enter.

2.3. clustered-cache quickstart architecture

2.3.1. Logging changes to the cache

An easy way to see what is going on with your cache is to log mutated entries. An Infinispan listener is notified of any mutations:

```
import org.infinispan.notifications.Listener;
import org.infinispan.notifications.cachelistener.annotation.*;
import org.infinispan.notifications.cachelistener.event.*;
import org.jboss.logging.Logger;

@Listener
public class LoggingListener {

   private BasicLogger log = BasicLogFactory.getLog(LoggingListener.class);

   @CacheEntryCreated
   public void observeAdd(CacheEntryCreatedEvent<String, String> event) {
      if (event.isPre())
         return;
      log.infof("Cache entry %s = %s added in cache %s", event.getKey(), event
.getValue(), event.getCache());
   }

   @CacheEntryModified
   public void observeUpdate(CacheEntryModifiedEvent<String, String> event) {
      if (event.isPre())
         return;
      log.infof("Cache entry %s = %s modified in cache %s", event.getKey(), event
.getValue(), event.getCache());
   }

   @CacheEntryRemoved
   public void observeRemove(CacheEntryRemovedEvent<String, String> event) {
      if (event.isPre())
         return;
      log.infof("Cache entry %s removed in cache %s", event.getKey(), event.getCache(
));
   }
}
```

Listeners methods are declared using annotations, and receive a payload which contains metadata about the notification. Listeners are notified of any changes. Here, the listeners simply log any entries added, modified, or removed.

2.3.2. What's going on?

The example allows you to start two or more nodes, each of which are started in a separate process. The node code is very simple, each node starts up, prints the local cache contents, registers a listener that logs any changes, and starts storing entries of the form `key-<counter> = <local address>-counter`.

State transfer

Infinispan automatically replicates the cache contents from the existing members to joining members. This can be controlled in two ways:

- If you don't want the `getCache()` call to block until the entire cache is transferred, you can configure `clustering.stateTransfer.awaitInitialTransfer = false`. Note that `cache.get(key)` will still return the correct value, even before the state transfer is finished.
- If it's fast enough to re-create the cache entries from another source, you can disable state transfer completely, by configuring `clustering.stateTransfer.fetchInMemoryState = false`.

2.4. Configuring the cluster

First, we need to ensure that the cache manager is cluster aware. Infinispan provides a default configuration for a clustered cache manager:

```
GlobalConfigurationBuilder.getClusteredDefault().build()
```

2.4.1. Tweaking the cluster configuration for your network

Depending on your network setup, you may need to tweak your JGroups set up. JGroups is configured via an XML file; the file to use can be specified via the GlobalConfiguration:

```
DefaultCacheManager cacheManager = new DefaultCacheManager(
      GlobalConfigurationBuilder.defaultClusteredBuilder()
            .transport().nodeName(nodeName).addProperty("configurationFile",
"jgroups.xml")
            .build()
);
```

The JGroups documentation provides extensive advice on getting JGroups working on your network. If you are new to configuring JGroups, you may get a little lost, so you might want to try tweaking these configuration parameters:

- Using the system property `-Djgroups.bind_addr=127.0.0.1` causes JGroups to bind only to your loopback interface, meaning any firewall you may have configured won't get in the way. Very useful for testing a cluster where all nodes are on one machine.

You can also configure the JGroups configuration to use in Infinispan's XML configuration:

```
<infinispan>
  <jgroups>
     <!-- Add custom JGroups stacks in external files. -->
     <stack-file name="prod-tcp" path="prod-jgroups-tcp.xml"/>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Add custom JGroups stacks to clustered caches. -->
    <transport stack="prod-tcp" />
    <replicated-cache name="replicatedCache"/>
  </cache-container>
  ...
</infinispan>
```

2.5. Configuring a replicated data-grid

In replicated mode, Infinispan will store every entry on every node in the grid. This offers high durability and availability of data, but means the storage capacity is limited by the available heap space on the node with least memory. The cache should be configured to work in replication mode (either synchronous or asynchronous), and can otherwise be configured as normal. For example, if you want to configure the cache programmatically:

```
cacheManager.defineConfiguration("repl", new ConfigurationBuilder()
      .clustering()
      .cacheMode(CacheMode.REPL_SYNC)
      .build()
);
```

You can configure an identical cache using XML:

infinispan-replication.xml

```
<infinispan>
  <jgroups/>
  <cache-container default-cache="repl">
     <transport/>
     <replicated-cache name="repl" mode="SYNC" />
  </cache-container>
</infinispan>
```

along with

```
private static EmbeddedCacheManager createCacheManagerFromXml() throws IOException {
   return new DefaultCacheManager("infinispan-replication.xml");
}
```

2.6. Configuring a distributed data-grid

In distributed mode, Infinispan will store every entry on a subset of the nodes in the grid (the parameter numOwners controls how many owners each entry will have). Compared to replication, distribution offers increased storage capacity, but with increased latency to access data from non-owner nodes, and durability (data may be lost if all the owners are stopped in a short time interval). Adjusting the number of owners allows you to obtain the trade off between space, durability, and latency.

Infinispan also offers a *topology aware consistent hash* which will ensure that the owners of entries are located in different data centers, racks, or physical machines, to offer improved durability in case of node crashes or network outages.

The cache should be configured to work in distributed mode (either synchronous or asynchronous), and can otherwise be configured as normal. For example, if you want to configure the cache programmatically:

```
cacheManager.defineConfiguration("dist", new ConfigurationBuilder()
      .clustering()
      .cacheMode(CacheMode.DIST_SYNC)
      .hash().numOwners(2)
      .build()
);
```

You can configure an identical cache using XML:

infinispan-distribution.xml:

```
<infinispan>
  <jgroups/>
  <cache-container default-cache="repl">
    <transport/>
    <distributed-cache owners="2" mode="SYNC" />
  </cache-container>
</infinispan>
```

along with

```
private static EmbeddedCacheManager createCacheManagerFromXml() throws IOException {
   return new DefaultCacheManager("infinispan-distribution.xml");
}
```

Chapter 3. Executing code in the Grid

The main benefit of a Cache is the ability to very quickly lookup a value by its key, even across machines. In fact this use alone is probably the reason many users use Infinispan. However Infinispan can provide many more benefits that aren't immediately apparent. Since Infinispan is usually used in a cluster of machines we also have features available that can help utilize the entire cluster for performing the user's desired workload.

This section covers only executing code in the grid using an embedded cache, if you are using a remote cache you should review details about executing code in the remote grid.

3.1. Cluster Executor

Since you have a group of machines, it makes sense to leverage their combined computing power for executing code on all of them them. The cache manager comes with a nice utility that allows you to execute arbitrary code in the cluster. Note this feature requires no Cache to be used. This Cluster Executor can be retrieved by calling executor() on the `EmbeddedCacheManager`. This executor is retrievable in both clustered and non clustered configurations.

The ClusterExecutor is specifically designed for executing code where the code is not reliant upon the data in a cache and is used instead as a way to help users to execute code easily in the cluster.

This manager was built specifically using Java 8 and such has functional APIs in mind, thus all methods take a functional inteface as an argument. Also since these arguments will be sent to other nodes they need to be serializable. We even used a nice trick to ensure our lambdas are immediately Serializable. That is by having the arguments implement both Serializable and the real argument type (ie. Runnable or Function). The JRE will pick the most specific class when determining which method to invoke, so in that case your lambdas will always be serializable. It is also possible to use an Externalizer to possibly reduce message size further.

The manager by default will submit a given command to all nodes in the cluster including the node where it was submitted from. You can control on which nodes the task is executed on by using the `filterTargets` methods as is explained in the section.

3.1.1. Filtering execution nodes

It is possible to limit on which nodes the command will be ran. For example you may want to only run a computation on machines in the same rack. Or you may want to perform an operation once in the local site and again on a different site. A cluster executor can limit what nodes it sends requests to at the scope of same or different machine, rack or site level.

SameRack.java

```
EmbeddedCacheManager manager = ...;
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_RACK).submit(...)
```

To use this topology base filtering you must enable topology aware consistent hashing through Server Hinting.

You can also filter using a predicate based on the `Address` of the node. This can also be optionally combined with topology based filtering in the previous code snippet.

We also allow the target node to be chosen by any means using a `Predicate` that will filter out which nodes can be considered for execution. Note this can also be combined with Topology filtering at the same time to allow even more fine control of where you code is executed within the cluster.

Predicate.java

```
EmbeddedCacheManager manager = ...;
// Just filter
manager.executor().filterTargets(a -> a.equals(..)).submit(...)
// Filter only those in the desired topology
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_SITE, a -> a.equals(..))
.submit(...)
```

3.1.2. Timeout

Cluster Executor allows for a timeout to be set per invocation. This defaults to the distributed sync timeout as configured on the Transport Configuration. This timeout works in both a clustered and non clustered cache manager. The executor may or may not interrupt the threads executing a task when the timeout expires. However when the timeout occurs any `Consumer` or `Future` will be completed passing back a `TimeoutException`. This value can be overridden by ivoking the timeout method and supplying the desired duration.

3.1.3. Single Node Submission

Cluster Executor can also run in single node submission mode instead of submitting the command to all nodes it will instead pick one of the nodes that would have normally received the command and instead submit it it to only one. Each submission will possibly use a different node to execute the task on. This can be very useful to use the ClusterExecutor as a `java.util.concurrent.Executor` which you may have noticed that ClusterExecutor implements.

SingleNode.java

```
EmbeddedCacheManager manager = ...;
manager.executor().singleNodeSubmission().submit(...)
```

Failover

When running in single node submission it may be desirable to also allow the Cluster Executor handle cases where an exception occurred during the processing of a given command by retrying the command again. When this occurs the Cluster Executor will choose a single node again to resubmit the command to up to the desired number of failover attempts. Note the chosen node could be any node that passes the topology or predicate check. Failover is enabled by invoking the overridden singleNodeSubmission method. The given command will be resubmitted again to a single node until either the command completes without exception or the total submission amount is equal to the provided failover count.

3.1.4. Example: PI Approximation

This example shows how you can use the ClusterExecutor to estimate the value of PI.

Pi approximation can greatly benefit from parallel distributed execution via Cluster Executor. Recall that area of the square is Sa = 4r2 and area of the circle is Ca=pi*r2. Substituting r2 from the second equation into the first one it turns out that pi = 4 * Ca/Sa. Now, image that we can shoot very large number of darts into a square; if we take ratio of darts that land inside a circle over a total number of darts shot we will approximate Ca/Sa value. Since we know that pi = 4 * Ca/Sa we can easily derive approximate value of pi. The more darts we shoot the better approximation we get. In the example below we shoot 1 billion darts but instead of "shooting" them serially we parallelize work of dart shooting across the entire Infinispan cluster. Note this will work in a cluster of 1 was well, but will be slower.

```
public class PiAppx {

   public static void main (String [] arg){
      EmbeddedCacheManager cacheManager = ..
      boolean isCluster = ..

      int numPoints = 1_000_000_000;
      int numServers = isCluster ? cacheManager.getMembers().size() : 1;
      int numberPerWorker = numPoints / numServers;

      ClusterExecutor clusterExecutor = cacheManager.executor();
      long start = System.currentTimeMillis();
      // We receive results concurrently - need to handle that
      AtomicLong countCircle = new AtomicLong();
      CompletableFuture<Void> fut = clusterExecutor.submitConsumer(m -> {
         int insideCircleCount = 0;
         for (int i = 0; i < numberPerWorker; i++) {
            double x = Math.random();
            double y = Math.random();
            if (insideCircle(x, y))
               insideCircleCount++;
         }
         return insideCircleCount;
      }, (address, count, throwable) -> {
         if (throwable != null) {
```

```
            throwable.printStackTrace();
            System.out.println("Address: " + address + " encountered an error: " +
throwable);
         } else {
            countCircle.getAndAdd(count);
         }
      });
      fut.whenComplete((v, t) -> {
         // This is invoked after all nodes have responded with a value or exception
         if (t != null) {
            t.printStackTrace();
            System.out.println("Exception encountered while waiting:" + t);
         } else {
            double appxPi = 4.0 * countCircle.get() / numPoints;

            System.out.println("Distributed PI appx is " + appxPi +
                  " using " + numServers + " node(s), completed in " + (System
.currentTimeMillis() - start) + " ms");
         }
      });

      // May have to sleep here to keep alive if no user threads left
   }

   private static boolean insideCircle(double x, double y) {
      return (Math.pow(x - 0.5, 2) + Math.pow(y - 0.5, 2))
            <= Math.pow(0.5, 2);
   }
}
```

Chapter 4. Streams

You may want to process a subset or all data in the cache to produce a result. This may bring thoughts of Map Reduce. Infinispan allows the user to do something very similar but utilizes the standard JRE APIs to do so. Java 8 introduced the concept of a Stream which allows functional-style operations on collections rather than having to procedurally iterate over the data yourself. Stream operations can be implemented in a fashion very similar to MapReduce. Streams, just like MapReduce allow you to perform processing upon the entirety of your cache, possibly a very large data set, but in an efficient way.

Streams are the preferred method when dealing with data that exists in the cache because streams automatically adjust to cluster topology changes.

Also since we can control how the entries are iterated upon we can more efficiently perform the operations in a cache that is distributed if you want it to perform all of the operations across the cluster concurrently.

A stream is retrieved from the entrySet, keySet or values collections returned from the Cache by invoking the stream or parallelStream methods.

4.1. Common stream operations

This section highlights various options that are present irrespective of what type of underlying cache you are using.

4.2. Key filtering

It is possible to filter the stream so that it only operates upon a given subset of keys. This can be done by invoking the filterKeys method on the `CacheStream`. This should always be used over a Predicate filter and will be faster if the predicate was holding all keys.

If you are familiar with the `AdvancedCache` interface you may be wondering why you even use getAll over this keyFilter. There are some small benefits (mostly smaller payloads) to using getAll if you need the entries as is and need them all in memory in the local node. However if you need to do processing on these elements a stream is recommended since you will get both distributed and threaded parallelism for free.

4.3. Segment based filtering

This is an advanced feature and should only be used with deep knowledge of Infinispan segment and hashing techniques. These segments based filtering can be useful if you need to segment data into separate invocations. This can be useful when integrating with other tools such as Apache Spark.

This option is only supported for replicated and distributed caches. This allows the user to operate upon a subset of data at a time as determined by the KeyPartitioner. The segments can be filtered

by invoking filterKeySegments method on the `CacheStream`. This is applied after the key filter but before any intermediate operations are performed.

4.4. Local/Invalidation

A stream used with a local or invalidation cache can be used just the same way you would use a stream on a regular collection. Infinispan handles all of the translations if necessary behind the scenes and works with all of the more interesting options (ie. storeAsBinary and a cache loader). Only data local to the node where the stream operation is performed will be used, for example invalidation only uses local entries.

4.5. Example

The code below takes a cache and returns a map with all the cache entries whose values contain the string "JBoss"

```
Map<Object, String> jbossValues = cache.entrySet().stream()
              .filter(e -> e.getValue().contains("JBoss"))
              .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

4.6. Distribution/Replication/Scattered

This is where streams come into their stride. When a stream operation is performed it will send the various intermediate and terminal operations to each node that has pertinent data. This allows processing the intermediate values on the nodes owning the data, and only sending the final results back to the originating nodes, improving performance.

4.6.1. Rehash Aware

Internally the data is segmented and each node only performs the operations upon the data it owns as a primary owner. This allows for data to be processed evenly, assuming segments are granular enough to provide for equal amounts of data on each node.

When you are utilizing a distributed cache, the data can be reshuffled between nodes when a new node joins or leaves. Distributed Streams handle this reshuffling of data automatically so you don't have to worry about monitoring when nodes leave or join the cluster. Reshuffled entries may be processed a second time, and we keep track of the processed entries at the key level or at the segment level (depending on the terminal operation) to limit the amount of duplicate processing.

It is possible but highly discouraged to disable rehash awareness on the stream. This should only be considered if your request can handle only seeing a subset of data if a rehash occurs. This can be done by invoking CacheStream.disableRehashAware() The performance gain for most operations when a rehash doesn't occur is completely negligible. The only exceptions are for iterator and forEach, which will use less memory, since they do not have to keep track of processed keys.

Please rethink disabling rehash awareness unless you really know what you are doing.

4.6.2. Serialization

Since the operations are sent across to other nodes they must be serializable by Infinispan marshalling. This allows the operations to be sent to the other nodes.

The simplest way is to use a CacheStream instance and use a lambda just as you would normally. Infinispan overrides all of the various Stream intermediate and terminal methods to take Serializable versions of the arguments (ie. SerializableFunction, SerializablePredicate...) You can find these methods at CacheStream. This relies on the spec to pick the most specific method as defined here.

In our previous example we used a `Collector` to collect all the results into a `Map`. Unfortunately the Collectors class doesn't produce Serializable instances. Thus if you need to use these, there are two ways to do so:

One option would be to use the CacheCollectors class which allows for a `Supplier<Collector>` to be provided. This instance could then use the Collectors to supply a `Collector` which is not serialized.

```
Map<Object, String> jbossValues = cache.entrySet().stream()
              .filter(e -> e.getValue().contains("Jboss"))
              .collect(CacheCollectors.serializableCollector(() -> Collectors.toMap
(Map.Entry::getKey, Map.Entry::getValue)));
```

Alternatively, you can avoid the use of CacheCollectors and instead use the overloaded `collect` methods that take `Supplier<Collector>`. These overloaded `collect` methods are only available via `CacheStream` interface.

```
Map<Object, String> jbossValues = cache.entrySet().stream()
              .filter(e -> e.getValue().contains("Jboss"))
              .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue)
);
```

If however you are not able to use the `Cache` and `CacheStream` interfaces you cannot utilize `Serializable` arguments and you must instead cast the lambdas to be `Serializable` manually by casting the lambda to multiple interfaces. It is not a pretty sight but it gets the job done.

```
Map<Object, String> jbossValues = map.entrySet().stream()
              .filter((Serializable & Predicate<Map.Entry<Object, String>>) e -> e
.getValue().contains("Jboss"))
              .collect(CacheCollectors.serializableCollector(() -> Collectors.toMap
(Map.Entry::getKey, Map.Entry::getValue)));
```

The recommended and most performant way is to use an `AdvancedExternalizer` as this provides the

smallest payload. Unfortunately this means you cannot use lamdbas as advanced externalizers require defining the class before hand.

You can use an advanced externalizer as shown below:

```
   Map<Object, String> jbossValues = cache.entrySet().stream()
               .filter(new ContainsFilter("Jboss"))
               .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue)
);

   class ContainsFilter implements Predicate<Map.Entry<Object, String>> {
      private final String target;

      ContainsFilter(String target) {
         this.target = target;
      }

      @Override
      public boolean test(Map.Entry<Object, String> e) {
         return e.getValue().contains(target);
      }
   }

   class JbossFilterExternalizer implements AdvancedExternalizer<ContainsFilter> {

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

      @Override
      public Integer getId() {
         return CUSTOM_ID;
      }

      @Override
      public void writeObject(ObjectOutput output, ContainsFilter object) throws
IOException {
         output.writeUTF(object.target);
      }

      @Override
      public ContainsFilter readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
         return new ContainsFilter(input.readUTF());
      }
   }
```

You could also use an advanced externalizer for the collector supplier to reduce the payload size even further.

```
   Map<Object, String> jbossValues = cache.entrySet().stream()
               .filter(new ContainsFilter("Jboss"))
               .collect(ToMapCollectorSupplier.INSTANCE);

 class ToMapCollectorSupplier<K, U> implements Supplier<Collector<Map.Entry<K, U>, ?,
Map<K, U>>> {
      static final ToMapCollectorSupplier INSTANCE = new ToMapCollectorSupplier();

      private ToMapCollectorSupplier() { }

      @Override
      public Collector<Map.Entry<K, U>, ?, Map<K, U>> get() {
         return Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue);
      }
   }

   class ToMapCollectorSupplierExternalizer implements AdvancedExternalizer
<ToMapCollectorSupplier> {

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

      @Override
      public Integer getId() {
         return CUSTOM_ID;
      }

      @Override
      public void writeObject(ObjectOutput output, ToMapCollectorSupplier object)
throws IOException {
      }

      @Override
      public ToMapCollectorSupplier readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
         return ToMapCollectorSupplier.INSTANCE;
      }
   }
```

4.7. Parallel Computation

Distributed streams by default try to parallelize as much as possible. It is possible for the end user to control this and actually they always have to control one of the options. There are 2 ways these streams are parallelized.

Local to each node When a stream is created from the cache collection the end user can choose between invoking stream or parallelStream method. Depending on if the parallel stream was

picked will enable multiple threading for each node locally. Note that some operations like a rehash aware iterator and forEach operations will always use a sequential stream locally. This could be enhanced at some point to allow for parallel streams locally.

Users should be careful when using local parallelism as it requires having a large number of entries or operations that are computationally expensive to be faster. Also it should be noted that if a user uses a parallel stream with `forEach` that the action should not block as this would be executed on the common pool, which is normally reserved for computation operations.

Remote requests When there are multiple nodes it may be desirable to control whether the remote requests are all processed at the same time concurrently or one at a time. By default all terminal operations except the iterator perform concurrent requests. The iterator, method to reduce overall memory pressure on the local node, only performs sequential requests which actually performs slightly better.

If a user wishes to change this default however they can do so by invoking the sequentialDistribution or parallelDistribution methods on the `CacheStream`.

4.8. Task timeout

It is possible to set a timeout value for the operation requests. This timeout is used only for remote requests timing out and it is on a per request basis. The former means the local execution will not timeout and the latter means if you have a failover scenario as described above the subsequent requests each have a new timeout. If no timeout is specified it uses the replication timeout as a default timeout. You can set the timeout in your task by doing the following:

```
CacheStream<Object, String> stream = cache.entrySet().stream();
stream.timeout(1, TimeUnit.MINUTES);
```

For more information about this, please check the java doc in timeout javadoc.

4.9. Injection

The Stream has a terminal operation called forEach which allows for running some sort of side effect operation on the data. In this case it may be desirable to get a reference to the `Cache` that is backing this Stream. If your `Consumer` implements the CacheAware interface the `injectCache` method be invoked before the accept method from the `Consumer` interface.

4.10. Distributed Stream execution

Distributed streams execution works in a fashion very similiar to map reduce. Except in this case we are sending zero to many intermediate operations (map, filter etc.) and a single terminal operation to the various nodes. The operation basically comes down to the following:

1. The desired segments are grouped by which node is the primary owner of the given segment
2. A request is generated to send to each remote node that contains the intermediate and terminal operations including which segments it should process

a. The terminal operation will be performed locally if necessary
b. Each remote node will receive this request and run the operations and subsequently send the response back

3. The local node will then gather the local response and remote responses together performing any kind of reduction required by the operations themselves.
4. Final reduced response is then returned to the user

In most cases all operations are fully distributed, as in the operations are all fully applied on each remote node and usually only the last operation or something related may be reapplied to reduce the results from multiple nodes. One important note is that intermediate values do not actually have to be serializable, it is the last value sent back that is the part desired (exceptions for various operations will be highlighted below).

Terminal operator distributed result reductions The following paragraphs describe how the distributed reductions work for the various terminal operators. Some of these are special in that an intermediate value may be required to be serializable instead of the final result.

allMatch noneMatch anyMatch

The allMatch operation is ran on each node and then all the results are logically anded together locally to get the appropriate value. The noneMatch and anyMatch operations use a logical or instead. These methods also have early termination support, stopping remote and local operations once the final result is known.

collect

The collect method is interesting in that it can do a few extra steps. The remote node performs everything as normal except it doesn't perform the final finisher upon the result and instead sends back the fully combined results. The local thread then combines the remote and local result into a value which is then finally finished. The key here to remember is that the final value doesn't have to be serializable but rather the values produced from the supplier and combiner methods.

count

The count method just adds the numbers together from each node.

findAny findFirst

The findAny operation returns just the first value they find, whether it was from a remote node or locally. Note this supports early termination in that once a value is found it will not process others. Note the findFirst method is special since it requires a sorted intermediate operation, which is detailed in the exceptions section.

max min

The max and min methods find the respective min or max value on each node then a final reduction is performed locally to ensure only the min or max across all nodes is returned.

reduce

The various reduce methods 1 , 2 , 3 will end up serializing the result as much as the accumulator can do. Then it will accumulate the local and remote results together locally, before

combining if you have provided that. Note this means a value coming from the combiner doesn't have to be Serializable.

4.11. Key based rehash aware operators

The iterator, spliterator and forEach are unlike the other terminal operators in that the rehash awareness has to keep track of what keys per segment have been processed instead of just segments. This is to guarantee an exactly once (iterator & spliterator) or at least once behavior (forEach) even under cluster membership changes.

The `iterator` and `spliterator` operators when invoked on a remote node will return back batches of entries, where the next batch is only sent back after the last has been fully consumed. This batching is done to limit how many entries are in memory at a given time. The user node will hold onto which keys it has processed and when a given segment is completed it will release those keys from memory. This is why sequential processing is preferred for the iterator method, so only a subset of segment keys are held in memory at once, instead of from all nodes.

The `forEach()` method also returns batches, but it returns a batch of keys after it has finished processing at least a batch worth of keys. This way the originating node can know what keys have been processed already to reduce chances of processing the same entry again. Unfortunately this means it is possible to have an at least once behavior when a node goes down unexpectedly. In this case that node could have been processing a batch and not yet completed one and those entries that were processed but not in a completed batch will be ran again when the rehash failure operation occurs. Note that adding a node will not cause this issue as the rehash failover doesn't occur until all responses are received.

These operations batch sizes are both controlled by the same value which can be configured by invoking distributedBatchSize method on the `CacheStream`. This value will default to the `chunkSize` configured in state transfer. Unfortunately this value is a tradeoff with memory usage vs performance vs at least once and your mileage may vary.

Using `iterator` with replicated and distributed caches

When a node is the primary or backup owner of all requested segments for a distributed stream, Infinispan performs the `iterator` or `spliterator` terminal operations locally, which optimizes performance as remote iterations are more resource intensive.

This optimization applies to both replicated and distributed caches. However, Infinispan performs iterations remotely when using cache stores that are both `shared` and have `write-behind` enabled. In this case performing the iterations remotely ensures consistency.

4.12. Intermediate operation exceptions

There are some intermediate operations that have special exceptions, these are skip, peek, sorted 1 2. & distinct. All of these methods have some sort of artificial iterator implanted in the stream processing to guarantee correctness, they are documented as below. Note this means these operations may cause possibly severe performance degradation.

Skip

An artificial iterator is implanted up to the intermediate skip operation. Then results are brought locally so it can skip the appropriate amount of elements.

Sorted

WARNING: This operation requires having all entries in memory on the local node. An artificial iterator is implanted up to the intermediate sorted operation. All results are sorted locally. There are possible plans to have a distributed sort which returns batches of elements, but this is not yet implemented.

Distinct

WARNING: This operation requires having all or nearly all entries in memory on the local node. Distinct is performed on each remote node and then an artificial iterator returns those distinct values. Then finally all of those results have a distinct operation performed upon them.

The rest of the intermediate operations are fully distributed as one would expect.

4.13. Examples

Word Count

Word count is a classic, if overused, example of map/reduce paradigm. Assume we have a mapping of key → sentence stored on Infinispan nodes. Key is a String, each sentence is also a String, and we have to count occurrence of all words in all sentences available. The implementation of such a distributed task could be defined as follows:

```
public class WordCountExample {

   /**
    * In this example replace c1 and c2 with
    * real Cache references
    *
    * @param args
    */
   public static void main(String[] args) {
      Cache<String, String> c1 = ...;
      Cache<String, String> c2 = ...;

      c1.put("1", "Hello world here I am");
      c2.put("2", "Infinispan rules the world");
      c1.put("3", "JUDCon is in Boston");
      c2.put("4", "JBoss World is in Boston as well");
      c1.put("12","JBoss Application Server");
      c2.put("15", "Hello world");
      c1.put("14", "Infinispan community");
      c2.put("15", "Hello world");

      c1.put("111", "Infinispan open source");
      c2.put("112", "Boston is close to Toronto");
      c1.put("113", "Toronto is a capital of Ontario");
      c2.put("114", "JUDCon is cool");
      c1.put("211", "JBoss World is awesome");
      c2.put("212", "JBoss rules");
      c1.put("213", "JBoss division of RedHat ");
      c2.put("214", "RedHat community");

      Map<String, Long> wordCountMap = c1.entrySet().parallelStream()
         .map(e -> e.getValue().split("\\s"))
         .flatMap(Arrays::stream)
         .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.
counting()));
   }
}
```

In this case it is pretty simple to do the word count from the previous example.

However what if we want to find the most frequent word in the example? If you take a second to think about this case you will realize you need to have all words counted and available locally first. Thus we actually have a few options.

We could use a finisher on the collector, which is invoked on the user thread after all the results have been collected. Some redundant lines have been removed from the previous example.

```
public class WordCountExample {
   public static void main(String[] args) {
      // Lines removed

      String mostFrequentWord = c1.entrySet().parallelStream()
         .map(e -> e.getValue().split("\\s"))
         .flatMap(Arrays::stream)
         .collect(() -> Collectors.collectingAndThen(
            Collectors.groupingBy(Function.identity(), Collectors.counting()),
               wordCountMap -> {
                  String mostFrequent = null;
                  long maxCount = 0;
                     for (Map.Entry<String, Long> e : wordCountMap.entrySet()) {
                        int count = e.getValue().intValue();
                        if (count > maxCount) {
                           maxCount = count;
                           mostFrequent = e.getKey();
                        }
                     }
                     return mostFrequent;
               }));

}
```

Unfortunately the last step is only going to be ran in a single thread, which if we have a lot of words could be quite slow. Maybe there is another way to parallelize this with Streams.

We mentioned before we are in the local node after processing, so we could actually use a stream on the map results. We can therefore use a parallel stream on the results.

```
public class WordFrequencyExample {
   public static void main(String[] args) {
      // Lines removed

      Map<String, Long> wordCount = c1.entrySet().parallelStream()
              .map(e -> e.getValue().split("\\s"))
              .flatMap(Arrays::stream)
              .collect(() -> Collectors.groupingBy(Function.identity(), Collectors
.counting()));
      Optional<Map.Entry<String, Long>> mostFrequent = wordCount.entrySet()
.parallelStream().reduce(
              (e1, e2) -> e1.getValue() > e2.getValue() ? e1 : e2);
```

This way you can still utilize all of the cores locally when calculating the most frequent element.

Remove specific entries

Distributed streams can also be used as a way to modify data where it lives. For example you may want to remove all entries in your cache that contain a specific word.

```
public class RemoveBadWords {
   public static void main(String[] args) {
      // Lines removed
      String word = ..

      c1.entrySet().parallelStream()
         .filter(e -> e.getValue().contains(word))
         .forEach((c, e) -> c.remove(e.getKey());
```

If we carefully note what is serialized and what is not, we notice that only the word along with the operations are serialized across to other nods as it is captured by the lambda. However the real saving piece is that the cache operation is performed on the primary owner thus reducing the amount of network traffic required to remove these values from the cache. The cache is not captured by the lambda as we provide a special BiConsumer method override that when invoked on each node passes the cache to the BiConsumer

One thing to keep in mind using the `forEach` command in this manner is that the underlying stream obtains no locks. The cache remove operation will still obtain locks naturally, but the value could have changed from what the stream saw. That means that the entry could have been changed after the stream read it but the remove actually removed it.

We have specifically added a new variant which is called `LockedStream`.

Plenty of other examples

The `Streams` API is a JRE tool and there are lots of examples for using it. Just remember that your operations need to be Serializable in some way.

Unresolved directive in stories.adoc - include::../../topics/cloud.adoc[leveloffset=+1]