Configuring Infinispan 10.0

Table of Contents

Chapter 1. Configuration

Infinispan offers both declarative and programmatic configuration.

1.1. Configuring caches declaratively

Declarative configuration comes in a form of XML document that adheres to a provided Infinispan configuration XML schema.

Every aspect of Infinispan that can be configured declaratively can also be configured programmatically. In fact, declarative configuration, behind the scenes, invokes the programmatic configuration API as the XML configuration file is being processed. One can even use a combination of these approaches. For example, you can read static XML configuration files and at runtime programmatically tune that same configuration. Or you can use a certain static configuration defined in XML as a starting point or template for defining additional configurations in runtime.

There are two main configuration abstractions in Infinispan: `global` and `cache`.

Global configuration

Global configuration defines global settings shared among all cache instances created by a single EmbeddedCacheManager. Shared resources like thread pools, serialization/marshalling settings, transport and network settings, JMX domains are all part of global configuration.

Cache configuration

Cache configuration is specific to the actual caching domain itself: it specifies eviction, locking, transaction, clustering, persistence etc. You can specify as many named cache configurations as you need. One of these caches can be indicated as the `default` cache, which is the cache returned by the `CacheManager.getCache()` API, whereas other named caches are retrieved via the `CacheManager.getCache(String name)` API.

Whenever they are specified, named caches inherit settings from the default cache while additional behavior can be specified or overridden. Infinispan also provides a very flexible inheritance mechanism, where you can define a hierarchy of configuration templates, allowing multiple caches to share the same settings, or overriding specific parameters as necessary.

Embedded and Server configuration use different schemas, but we strive to maintain them as compatible as possible so that you can easily migrate between the two.

One of the major goals of Infinispan is to aim for zero configuration. A simple XML configuration file containing nothing more than a single infinispan element is enough to get you started. The configuration file listed below provides sensible defaults and is perfectly valid.

infinispan.xml

```
<infinispan />
```

However, that would only give you the most basic, local mode, non-clustered cache manager with no caches. Non-basic configurations are very likely to use customized global and default cache elements.

Declarative configuration is the most common approach to configuring Infinispan cache instances. In order to read XML configuration files one would typically construct an instance of DefaultCacheManager by pointing to an XML file containing Infinispan configuration. Once the configuration file is read you can obtain reference to the default cache instance.

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

or any other named instance specified in `my-config-file.xml`.

```
Cache someNamedCache = manager.getCache("someNamedCache");
```

The name of the default cache is defined in the `<cache-container>` element of the XML configuration file, and additional caches can be configured using the `<local-cache>`,`<distributed-cache>` ,`<invalidation-cache>` or `<replicated-cache>` elements.

The following example shows the simplest possible configuration for each of the cache types supported by Infinispan:

```
<infinispan>
   <cache-container default-cache="local">
      <transport cluster="mycluster"/>
      <local-cache name="local"/>
      <invalidation-cache name="invalidation" mode="SYNC"/>
      <replicated-cache name="repl-sync" mode="SYNC"/>
      <distributed-cache name="dist-sync" mode="SYNC"/>
   </cache-container>
</infinispan>
```

1.1.1. Cache configuration templates

As mentioned above, Infinispan supports the notion of *configuration templates*. These are full or partial configuration declarations which can be shared among multiple caches or as the basis for more complex configurations.

The following example shows how a configuration named `local-template` is used to define a cache named `local`.

```
<infinispan>
   <cache-container default-cache="local">
      <!-- template configurations -->
      <local-cache-configuration name="local-template">
         <expiration interval="10000" lifespan="10" max-idle="10"/>
      </local-cache-configuration>

      <!-- cache definitions -->
      <local-cache name="local" configuration="local-template" />
   </cache-container>
</infinispan>
```

Templates can inherit from previously defined templates, augmenting and/or overriding some or all of the configuration elements:

```
<infinispan>
   <cache-container default-cache="local">
      <!-- template configurations -->
      <local-cache-configuration name="base-template">
         <expiration interval="10000" lifespan="10" max-idle="10"/>
      </local-cache-configuration>

      <local-cache-configuration name="extended-template" configuration="base-
template">
         <expiration lifespan="20"/>
         <memory>
            <object size="2000"/>
         </memory>
      </local-cache-configuration>

      <!-- cache definitions -->
      <local-cache name="local" configuration="base-template" />
      <local-cache name="local-bounded" configuration="extended-template" />
   </cache-container>
</infinispan>
```

In the above example, `base-template` defines a local cache with a specific *expiration* configuration. The `extended-template` configuration inherits from `base-template`, overriding just a single parameter of the *expiration* element (all other attributes are inherited) and adds a *memory* element. Finally, two caches are defined: `local` which uses the `base-template` configuration and `local-bounded` which uses the `extended-template` configuration.

Be aware that for multi-valued elements (such as `properties`) the inheritance is additive, i.e. the child configuration will be the result of merging the properties from the parent and its own.

1.1.2. Cache configuration wildcards

An alternative way to apply templates to caches is to use wildcards in the template name, e.g. `basecache*`. Any cache whose name matches the template wildcard will inherit that configuration.

```
<infinispan>
    <cache-container>
        <local-cache-configuration name="basecache*">
            <expiration interval="10500" lifespan="11" max-idle="11"/>
        </local-cache-configuration>
        <local-cache name="basecache-1"/>
        <local-cache name="basecache-2"/>
    </cache-container>
</infinispan>
```

Above, caches `basecache-1` and `basecache-2` will use the `basecache*` configuration. The configuration will also be applied when retrieving undefined caches programmatically.

If a cache name matches multiple wildcards, i.e. it is ambiguous, an exception will be thrown.

XInclude support

The configuration parser supports XInclude which means you can split your XML configuration across multiple files:

infinispan.xml

```
<infinispan xmlns:xi="http://www.w3.org/2001/XInclude">
    <cache-container>
        <local-cache name="cache-1"/>
        <xi:include href="included.xml" />
    </cache-container>
</infinispan>
```

included.xml

```
<local-cache name="cache-1"/>
```

the parser supports a minimal subset of the XInclude spec (no support for XPointer, fallback, text processing and content negotiation).

1.1.3. Declarative configuration reference

For more details on the declarative configuration schema, refer to the configuration reference. If you are using XML editing tools for configuration writing you can use the provided Infinispan schema to assist you.

1.2. Configuring caches programmatically

Programmatic Infinispan configuration is centered around the CacheManager and ConfigurationBuilder API. Although every single aspect of Infinispan configuration could be set programmatically, the most usual approach is to create a starting point in a form of XML configuration file and then in runtime, if needed, programmatically tune a specific configuration to suit the use case best.

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

Let's assume that a new synchronously replicated cache is to be configured programmatically. First, a fresh instance of Configuration object is created using ConfigurationBuilder helper object, and the cache mode is set to synchronous replication. Finally, the configuration is defined/registered with a manager.

```
Configuration c = new ConfigurationBuilder().clustering().cacheMode(CacheMode
.REPL_SYNC).build();

String newCacheName = "repl";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

The default cache configuration (or any other cache configuration) can be used as a starting point for creation of a new cache. For example, lets say that `infinispan-config-file.xml` specifies a replicated cache as a default and that a distributed cache is desired with a specific L1 lifespan while at the same time retaining all other aspects of a default cache. Therefore, the starting point would be to read an instance of a default Configuration object and use `ConfigurationBuilder` to construct and modify cache mode and L1 lifespan on a new `Configuration` object. As a final step the configuration is defined/registered with a manager.

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration dcc = manager.getDefaultCacheConfiguration();
Configuration c = new ConfigurationBuilder().read(dcc).clustering().cacheMode
(CacheMode.DIST_SYNC).l1().lifespan(60000L).build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

As long as the base configuration is the default named cache, the previous code works perfectly fine. However, other times the base configuration might be another named cache. So, how can new configurations be defined based on other defined caches? Take the previous example and imagine that instead of taking the default cache as base, a named cache called "replicatedCache" is used as base. The code would look something like this:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration rc = manager.getCacheConfiguration("replicatedCache");
Configuration c = new ConfigurationBuilder().read(rc).clustering().cacheMode(
CacheMode.DIST_SYNC).l1().lifespan(60000L).build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

Refer to CacheManager , ConfigurationBuilder , Configuration , and GlobalConfiguration javadocs for more details.

1.2.1. ConfigurationBuilder Programmatic Configuration API

While the above paragraph shows how to combine declarative and programmatic configuration, starting from an XML configuration is completely optional. The ConfigurationBuilder fluent interface style allows for easier to write and more readable programmatic configuration. This approach can be used for both the global and the cache level configuration. GlobalConfiguration objects are constructed using GlobalConfigurationBuilder while Configuration objects are built using ConfigurationBuilder. Let's look at some examples on configuring both global and cache level options with this API:

1.2.2. Enabling JMX MBeans and statistics

Sometimes you might also want to enable collection of global JMX statistics at cache manager level or get information about the transport. To enable global JMX statistics simply do:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics()
  .enable()
  .build();
```

Please note that by not enabling (or by explicitly disabling) global JMX statistics your are just turning off statistics collection. The corresponding MBean is still registered and can be used to manage the cache manager in general, but the statistics attributes do not return meaningful values.

Further options at the global JMX statistics level allows you to configure the cache manager name which comes handy when you have multiple cache managers running on the same system, or how to locate the JMX MBean Server:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics()
    .cacheManagerName("SalesCacheManager")
    .mBeanServerLookup(new JBossMBeanServerLookup())
  .build();
```

1.2.3. Configuring the thread pools

Some of the Infinispan features are powered by a group of the thread pool executors which can also be tweaked at this global level. For example:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
   .replicationQueueThreadPool()
     .threadPoolFactory(ScheduledThreadPoolExecutorFactory.create())
  .build();
```

You can not only configure global, cache manager level, options, but you can also configure cache level options such as the cluster mode:

```
Configuration config = new ConfigurationBuilder()
  .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .sync()
    .l1().lifespan(25000L)
    .hash().numOwners(3)
  .build();
```

Or you can configure eviction and expiration settings:

```
Configuration config = new ConfigurationBuilder()
           .memory()
             .size(20000)
          .expiration()
             .wakeUpInterval(5000L)
             .maxIdle(120000L)
           .build();
```

1.2.4. Configuring transactions and locking

An application might also want to interact with an Infinispan cache within the boundaries of JTA and to do that you need to configure the transaction layer and optionally tweak the locking settings. When interacting with transactional caches, you might want to enable recovery to deal with transactions that finished with an heuristic outcome and if you do that, you will often want to enable JMX management and statistics gathering too:

```
Configuration config = new ConfigurationBuilder()
  .locking()
    .concurrencyLevel(10000).isolationLevel(IsolationLevel.REPEATABLE_READ)
    .lockAcquisitionTimeout(12000L).useLockStriping(false).writeSkewCheck(true)
    .versioning().enable().scheme(VersioningScheme.SIMPLE)
  .transaction()
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .recovery()
  .jmxStatistics()
  .build();
```

1.2.5. Configuring cache stores

Configuring Infinispan with chained cache stores is simple too:

```
Configuration config = new ConfigurationBuilder()
   .persistence().passivation(false)
   .addSingleFileStore().location("/tmp").async().enable()
   .preload(false).shared(false).threadPoolSize(20).build();
```

1.2.6. Advanced programmatic configuration

The fluent configuration can also be used to configure more advanced or exotic options, such as advanced externalizers:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .serialization()
    .addAdvancedExternalizer(998, new PersonExternalizer())
    .addAdvancedExternalizer(999, new AddressExternalizer())
  .build();
```

Or, add custom interceptors:

```
Configuration config = new ConfigurationBuilder()
  .customInterceptors().addInterceptor()
    .interceptor(new FirstInterceptor()).position(InterceptorConfiguration.Position
.FIRST)
    .interceptor(new LastInterceptor()).position(InterceptorConfiguration.Position
.LAST)
    .interceptor(new FixPositionInterceptor()).index(8)
    .interceptor(new AfterInterceptor()).after(NonTransactionalLockingInterceptor
.class)
    .interceptor(new BeforeInterceptor()).before(CallInterceptor.class)
  .build();
```

For information on the individual configuration options, please check the configuration guide.

1.3. Configuration Migration Tools

The configuration format of Infinispan has changed since schema version 6.0 in order to align the embedded schema with the one used by the server. For this reason, when upgrading to schema 7.x or later, you should use the configuration converter included in the *all* distribution. Simply invoke it from the command-line passing the old configuration file as the first parameter and the name of the converted file as the second parameter.

Unix/Linux/macOS:

```
bin/config-converter.sh oldconfig.xml newconfig.xml
```

Windows:

```
bin\config-converter.bat oldconfig.xml newconfig.xml
```

If you wish to help write conversion tools from other caching systems, please contact infinispan-dev.