Terracotta Distributed Cache

About Terracotta Documentation

This documentation is about Terracotta DSO, an advanced distributed-computing technology aimed at meeting special clustering requirements.

Terracotta products without the overhead and complexity of DSO meet the needs of almost all use cases and clustering requirements. To learn how to migrate from Terracotta DSO to standard Terracotta products, see Migrating From Terracotta DSO. To find documentation on non-DSO (standard) Terracotta products, see Terracotta Documentation. Terracotta release information, such as release notes and platform compatibility, is found in Product Information.

- **Introduction**
- How DSO Clustering Works
- Platform Concepts
- Hello Clustered World
- **Setup and Configuration**
- Planning for a Clustered App
- Configuring Terracotta DSO
- Configuration Reference
- Installation
- **APIs**
- Using Annotations
- Cluster Events
- Data Locality Methods
- Distributed Cache
- Clustered Async Data Processing
- **Tool Guides**
- Developer Console
- Operations Center
- tim-get (TIM Management Tool)
- Platform Statistics Recorder
- Eclipse Plugin
- Sessions Configurator
- Clustering Spring Webapp with Sessions Configurator
- Maven
- JMX
- **Testing, Tuning, and Deployment**
- Top 5 Tuning Tips
- Testing a Clustered App
- Tuning a Clustered App
- Deployment Guide
- Operations Guide
- **FAQs and Troubleshooting**
- General FAQ
- DSO Technical FAQ
- Troubleshooting Guide
- Gotchas
- Non-portable Classes
- **Reference**
- Migrating From DSO
- Concept and Architecture Guide
- Examinator Reference Application
- Clustered Data Structures Guide
- Integrating Terracotta DSO
- Clustering Spring Framework
- Integration Modules Manual
- AspectWerkz Pattern Language
- Glossary

Release: 3.6
Publish Date: November, 2011 **Documentation Archive »**

DEPRECATED

The Terracotta Distributed Cache based on the Terracotta Integration Module (TIM) `tim-distributed-cache` has been *deprecated* in favor of the

Error rendering macro 'html'

Notify your Confluence administrator that "Bob Swift Atlassian Apps - HTML" requires a valid license. Reason: EXPIRED

.

Terracotta Distributed Cache

- Introduction
- How to Implement and Configure
- A Simple Distributed Cache
- Terracotta Distributed Cache in a Reference Application

Introduction

The Terracotta Distributed Cache is an interface providing a simple distributed eviction solution for map elements. The Distributed Cache, implemented with the Terracotta Integration Module (TIM) `tim-distributed-cache`, provides a number of advantages over more complex solutions:

- Simple – API is easy to understand and code against.
- Distributed – Eviction is distributed along with data to maintain coherence.
- Standard – Data eviction is based on standard expiration metrics.
- Lightweight – Implementation does not hog resources.
- Efficient – Optimized for a clustered environment to minimize faulting due to low locality of reference.
- Fail-Safe – Data can be evicted even if written by a failed node or after all nodes have been restarted.
- Self-Contained – Implements a Map for optional ready-to-use distributed cache.
- Native – Designed for Terracotta to eliminate integration issues.

How to Implement and Configure

Under the appropriate conditions, the Terracotta Distributed Cache can be used in any Terracotta cluster. If your application can use the Distributed Cache's built-in Map implementation for a cache, you can avoid having to customize your own data structure. See #A Simple Distributed Cache for instructions on using the Distributed Cache with the provided Map implementation.

Requirements

To ensure that the Terracotta Distributed Cache performs well and without errors, check the following requirements.

Classpath Requirements

The TIMs `tim-distributed-cache` and `tim-concurrent-collections` must be on your application's classpath at runtime. See #Installing the TIM to learn how to install `tim-distributed-cache`.

When you install `tim-distributed-cache`, `tim-concurrent-collections` is automatically installed.

Eviction Parameters

The Terracotta Distributed Cache has the following eviction parameters:

Time to Live
The Time to Live (TTL) value determines the maximum amount of time an object can remain in the cache before becoming eligible for eviction, regardless of other conditions such as use.
Time to Idle
The Time to Idle (TTI) value determines the maximum amount of time an object can remain idle in the cache before becoming eligible for eviction. TTI is reset each time the object is used.
Target Max In-Memory Count
The in-memory count is the maximum number of elements allowed in a region in any one client (any one application server). If this target is exceeded, eviction occurs to bring the count within the allowed target. The flexibility of using a target as opposed to hard limit serves to improve concurrency. 0 means no eviction takes place (infinite size is allowed).
Target Max Total Count
The total count is the maximum total number of elements allowed for a region in all clients (all application servers). If this target is exceeded, eviction occurs to bring the count within the allowed target. The flexibility of using a target as opposed to hard limit serves to improve concurrency. 0 means no eviction takes place (infinite size is allowed).

To maximize cache performance benefits, configure and tune these parameters to optimize data retention and eviction behavior. To learn how to configure the Terracotta Distributed Cache eviction parameters, see #Usage Pattern.

JDK Version

The Terracotta Distributed Cache requires JDK 1.5 or greater.

Installing the TIM

To use the Terracotta Distributed Cache, you must both install `tim-distributed-cache` and include the evictor JAR file in your classpath.

To install the TIM, run the following command from ${TERRACOTTA_HOME}:

UNIX/Linux

```
[PROMPT] bin/tim-get.sh install tim-distributed-cache
```

Microsoft Windows

```
[PROMPT] bin\tim-get.bat install tim-distributed-cache
```

You should see output that appears similar to the following:

```
Installing tim-distributed-cache 1.3.0-SNAPSHOT and dependencies...
   INSTALLED: tim-distributed-cache 1.3.0-SNAPSHOT - Ok
   INSTALLED: tim-concurrent-collections 1.3.0-SNAPSHOT - Ok
```

Run the following command from ${TERRACOTTA_HOME} to update the Terracotta configuration file (`tc-config.xml` by default):

UNIX/Linux

```
[PROMPT] bin/tim-get.sh upgrade <path/to/Terracotta/configuration/file>
```

Microsoft Windows

```
[PROMPT] bin\tim-get.bat upgrade <path\to\Terracotta\configuration\file>
```

For more information about installing and updating TIMs, see the TIM Update Center.

Locking Requirements

Terracotta automatically provides locking for read (get) and write (put) operations on the distributed map. These locks last for the duration of the get or put operation.

Mutating an object obtained from the distributed map requires a read/write lock to avoid race conditions and potential corruption to data.

For example, assume a distributed map has an element <k1, v1> in it. The following operation does not require explicit locking:

```
myObject = getFromMyDistributedMap(k1); // Terracotta provides a lock for the duration of
getFromMyDistributedMap().
```

Adding a new element to the map also does not require explicit locking:

```
putIntoMyDistributedMap(k2, v2); // Terracotta provides a lock for the duration of putIntoMyDistributedMap().
```

However, the following operation *requires* a read/write lock:

```
myNewObject = myMutator(myObject); // myObject should be locked until it is put back into the map.
```

Note the following:

- To be shared across the cluster, the `myObject` field must be declared a Terracotta root or its class (its type) must be instrumented.
- Cluster-wide locking (Terracotta locking) must be present when `myMutator()` changes `myObject`. There are several ways to implement Terracotta locking.
- To gain visibility into the cluster and better understand how the map and object graphs are being clustered, use the Terracotta Developer Console's Object Browser.
- Use the Terracotta Developer Console's Lock Profiler to see what locks are being used and their usage patterns.

For more information on locks, Terracotta roots, and instrumenting classes, see the following resources:

- The section Resolving UnlockedSharedObjectException - Configuring Locking in Configuring Terracotta
- Sections on locks, Terracotta roots, and instrumenting classes in the Concept and Architecture Guide and in the Configuration Guide and Reference.

For more information on the Terracotta Developer Console, see the console guide.

A Simple Distributed Cache

Clustered applications with a system of record (SOR) on the backend can benefit from a distributed cache that manages certain data in memory while reducing costly application-SOR interactions. However, using a cache can introduce increased complexity to software development, integration, operation, and maintenance.

The Terracotta Distributed Cache includes a distributed Map that can be used as a simple distributed cache. This cache uses the Terracotta Distributed Cache, incorporating all of its benefits. It also takes both established and innovative approaches to the caching model, solving performance and complexity issues by:

- obviating SOR commits for data with a limited lifetime;
- making cached application data available in-memory across a cluster of application servers;
- offering standard methods for working with cache elements and performing cache-wide operations;
- incorporating concurrency for readers and writers;
- utilizing a flexible map implementation to adapt to more applications;
- minimizing inter-node faulting to speed data operations.

Structure and Characteristics

The Terracotta Distributed Cache is an interface incorporating a distributed map (an extension of ConcurrentMap in the JDK) with standard map operations. For more information about the Terracotta Distributed Cache, see:

- Terracotta Distributed Cache Javadoc
- DistributedCache Interface

getValues() is not provided, but an iterator can be obtained for Set<K> to obtain values.

Usage Pattern

A typical usage pattern for the Terracotta Distributed Cache is shown in the MyStuff class below. The next section contains a full list of configuration parameters available to `CacheConfigFactory`.

```
import org.terracotta.cache.CacheConfigFactory;
import org.terracotta.cache.DistributedCache;

public class MyStuff {

   // Mark as Terracotta root
   private DistributedCache<String, Stuff> sharedCache;

   public MyStuff() {
     if(sharedCache == null) {
DistributedCache<String, Stuff> newCache =
CacheConfigFactory.newConfig()
        .setMaxTTLSeconds(6*60 * 60) // Regardless of use, remove after 6 hours
        .setMaxTTISeconds(30*60) // Remove after 30 minutes of none-use.
        .newCache();

       // Set root - if this doesn't succeed, shutdown the newCache as it has a worthless background evictor
thread.
       sharedCache = newCache;
       if(sharedCache != newCache) {
         newCache.shutdown();
       }
   }

   public void putStuff(String key, Stuff stuff) {
       sharedCache.put(key, stuff);
   }

   public Stuff getStuff(String key) {
       return sharedCache.get(key);
   }
}
```

Cache Configuration Parameters

The configuration parameters that can be set through `CacheConfigFactory` are summarized in the following table.

Config property	Default value	Description
name	"Distributed Map"	A descriptive string used in log messages and evictor thread names.
maxTTISeconds	0	Time To Idle - the maximum amount of time (in seconds) an item can be in the map unused before expiration; 0 means never expire due to TTI.
maxTTLSeconds	0	Time To Live - the maximum amount of time (in seconds) an item may be in the map regardless of use before expiration; 0 means never expire due to TTL.
orphanEvictionEnabled	true	Determines whether "orphaned" values (values no longer local to any node) are evicted.
orphanEvictionPeriod	4	Number of times to run local eviction between doing orphan eviction.
loggingEnabled	false	Basic distributed-map logging messages saved to the Terracotta logs.
targetMaxInMemoryCount	0	Target maximum number of values stored in memory for a region on any Terracotta client (application server). If target is exceeded, elements are flushed to a Terracotta server instance but not evicted. The default of 0 gives elements an infinite lifetime.
targetMaxTotalCount	0	Target maximum number of values stored for a region in the cluster. If the target is exceeded, elements are evicted to bring the total back under the limit. The default of 0 gives elements an infinite lifetime.

Usage Example

The following is an example of a cache that implements the Terracotta distributed cache:

```
import org.terracotta.cache.*;
import static org.terracotta.cache.CacheConfigFactory.*;


DisributedCache<String,String> cache = CacheConfigFactory.newConfig()
.setMaxTTLSeconds(10)
.setMaxTTISeconds(5)
.newCache();


// start() method not needed; start is automatic.

cache.put("Rabbit", "Carrots");
cache.put("Dog", "Bone");
cache.put("Owl", "Mouse");
// wait 3 seconds
cache.get("Rabbit");


// wait 2 seconds - expire Dog and Owl due to TTI
assert ! cache.containsKey("Dog");
assert ! cache.containsKey("Owl");
assert cache.containsKey("Rabbit");


// wait 5 seconds - expire Rabbit due to TTL
assert ! cache.containsKey("Rabbit");
```

Terracotta Distributed Cache in a Reference Application

The Examinator reference application uses the Terracotta Distributed Cache to handle pending user registrations. This type of data has a "medium-term" lifetime which needs to be persisted long enough to give prospective registrants a chance to verify their registrations. If a registration isn't verified by the time TTL is reached, it can be evicted from the cache. Only if the registration is verified is it written to the database.

The combination of Terracotta and the Terracotta Distributed Cache gives Examinator the following advantages:

- The simple Terracotta Distributed Cache's API makes it easy to integrate with Examinator and to maintain and troubleshoot.
- Medium-term data is not written to the database unnecessarily, improving application performance.
- Terracotta persists the pending registrations so they can survive node failure.
- Terracotta clusters (shares) the pending registration data so that any node can handle validation.