The Object Cache

by Armin Waibel, Thomas Mahler

Table of contents

1. Introduction

OJB supports several caching strategies and allow to pluggin own caching solutions by implementing the ObjectCache interface. All implementations shipped with OJB can be found in package **`org.apache.ojb.broker.cache`**. The naming convention of the implementation classes is `ObjectCacheXXXImpl`.

To classify the different implementations we differ *local/session cache* and *shared/global/application cache* implementations (we use the different terms synonymous). The ObjectCacheTwoLevelImpl use both characteristics.

- Local cache implementation mean that each instance use its own map to manage cached objects.
- Shared/global cache implementations share one (in most cases static) map to manage cached objects.

A distributed object cache implementation supports caching of objects across different JVM.

2. Why a cache and how it works?

OJB provides a pluggable object cache provided by the ObjectCache interface:

```
public interface ObjectCache
{
    /**
    * Write to cache.
    */
    public void cache(Identity oid, Object obj);

    /**
    * Lookup object from cache.
    */
    public Object lookup(Identity oid);

    /**
    * Removes an Object from the cache.
    */
    public void remove(Identity oid);

    /**
    * Clear the ObjectCache.
    */
    public void clear();
}
```

Each PersistenceBroker instance (PersistenceBroker is a standalone api and the basic layer for all top-level api's like ODMG) use it's own `ObjectCache` instance. The `ObjectCache` instances are created by the `ObjectCacheFactory` class on `PersistenceBroker` instantiation.

Each cache implementation holds objects previously loaded or stored by the `PersistenceBroker` - dependend on the implementation.
Using a Cache has several advantages:

- It increases performance as it reduces database lookups or/and object materialization. If an object is looked up by Identity the associated PersistenceBroker instance does not perform a `SELECT` against the database immediately but first looks up the cache if the requested object is already loaded. If the object is cached it is returned as the lookup result. If it is not cached a `SELECT` is performed.
  Other queries were performed against the database, but before an object from the ResultSet was materialized the object identity was looked up in cache. If not found the whole object was materialized.

- It allows to perform circular lookups (as by crossreferenced objects) that would result in non-terminating loops without such a cache (Note: Since OJB 1.0.2 this is handled internally by OJB and does not depend on the used cache implementation).

3. How to declare and change the used ObjectCache implementation

The `object-cache` element can be used to specify the ObjectCache implementation used by OJB. If no *object-cache* is declared in configuration files (see below), OJB use by default a noop-implementation of the `ObjectCache` interface.
There are two levels of declaration:

- jdbc-connection-descriptor level
- class-descriptor level

and the possibility to exclude all persistent objects of specified package names.

Use a **jdbc-connection-descriptor level** declaration to declare ObjectCache implementation on a per connection/user level. Additional configuration properties can be passed by using custom attributes entries:

```
<jdbc-connection-descriptor ...>
...
 <object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
     <attribute attribute-name="timeout" attribute-value="900"/>
     <attribute attribute-name="useAutoSync" attribute-value="true"/>
 </object-cache>
...
</jdbc-connection-descriptor>
```

Set an *object-cache* tag on **class-descriptor level** , to declare ObjectCache implementation on a per class level:

```
<class-descriptor
    class="org.apache.ojb.broker.util.sequence.HighLowSequence"
    table="OJB_HL_SEQ"
>
    <object-cache class="org.apache.ojb.broker.cache.ObjectCacheEmptyImpl">
    </object-cache>
...
</class-descriptor>
```

Additional configuration properties can be passed by using custom attributes entries.

Note:
If polymorphism was used it's only possible to declare the *object-cache* element in the class-descriptor of the top-level class/interface (root class), all *object-cache* declarations in the sub-classes will be ignored by OJB.

3.1. Priority of Cache Level

Since it is possible to mix the different levels of *object-cache* element declaration a ordering of priority is needed:

Note:
The order of priority of declared *object-cache* elements in metadata are:
per class > *excluded packages* > *per jdbc-connection-descriptor*

E.g. if you declare ObjectCache 'OC1' on connection level and set ObjectCache 'OC2' in class-descriptor of class A. Then OJB use 'OC2' as ObjectCache for class A instances and 'OC1' for all other classes.

3.2. Exclude classes from being cached

If it's undesirable to cache an persistent object (e.g. persistent objects with BLOB fields or large binary fields) declare an `object-cache` descriptor with the *noop-cache* implementation called ObjectCacheEmptyImpl.

```
<class-descriptor
    class="org.apache.ojb.broker.util.sequence.HighLowSequence"
    table="OJB_HL_SEQ"
>
    <object-cache class="org.apache.ojb.broker.cache.ObjectCacheEmptyImpl">
    </object-cache>
...
</class-descriptor>
```

Note:

If polymorphism was used and the class to exclude is part of an inheritance hierarchy **and** it's declared in in OJB metadata, it's not possible to exclude it. Only for the top-level class/interface (root class) it's allowed to specify the *object-cache* element in metadata. So it's only possible to exclude all sub-classes of the top-level class/interface (root class). More info see here.

3.3. Exclude packages from being cached

To exclude all persistent objects of a whole package from being cached use the custom attribute *cacheExcludes* on connection level within the object-cache declaration. To declare several packages use a comma seperated list.

```
<jdbc-connection-descriptor
    jcd-alias="myDefault"
    ...>

    <object-cache class="org.apache.ojb.broker.cache.ObjectCacheTwoLevelImpl">
        <attribute attribute-name="cacheExcludes"
            attribute-value="my.core, my.persistent.local"/>
        ... more attributes
    </object-cache>
</jdbc-connection-descriptor
```

To include a persistent class of a excluded package, simply declare an object-cache descriptor on class-descriptor level of the class to include, object cache declarations on class-descriptor level have a higher priority as the excluded packages - see more.

3.4. Turn off caching

If you don't declare a *object-cache* element in configuration files (see here), OJB doesn't cache persistent objects by default.
To explicitly turn off caching declare a *no-op* implementation of the ObjectCache interface as caching implementation. OJB was shipped with such a class called ObjectCacheEmptyImpl. To explicitly turn off caching for a used database look like this:

```
<jdbc-connection-descriptor ...>
    ...
    <object-cache class="org.apache.ojb.broker.cache.ObjectCacheEmptyImpl">
    </object-cache>
    ...
</jdbc-connection-descriptor>
```

To get more detailed info about the different level of cache declaration, please see here.

4. Shipped cache implementations:

4.1. ObjectCacheDefaultImpl

Per default OJB use a shared reference based ObjectCache implementation - ObjectCacheDefaultImpl. It's a really fast cache but there are a few drawbacks:

- There is no transaction isolation, when thread one modify an object, thread two will see the modification when lookup the same object or use a reference of the same object, so "dirty-reads" can happen.
- If you rollback/abort a transaction the modified/corrupted objects will **not** be removed from the cache by default(when using PB-api, top-level api may support automatic cache synchronization). You have to do this by your own using a service method to remove cached objects or enable the autoSync property.

```
broker.removeFromCache(obj);

// or (using Identity object)
ObjectCache cache = broker.serviceObjectCache();
cache.remove(oid);
```

- This implementation cache full object graphs (the object with all referenced objects) and does **not** synchronize the references. So if cached object *ProductGroup* has a 1:n reference to *Article*, e.g. article1, article2, article3 and another thread delete article2, the ProductGroup still has a reference to article2. To avoid such a behavior you can use the collection-descriptor 'refresh' attribute to force OJB to query the referenced objects when the main object is loaded from cache or use another `ObjectCache` implementation supporting synchronization of references (e.g. ObjectCacheTwoLevelImpl).

This implementation use by default `SoftReference` to wrap all cached objects. If the cached object was not longer referenced by your application but only by the cache, it can be reclaimed by the garbage collector.
As we don't know when the garbage collector reclaims the freed objects, it is possible to set a `timeout` property. So an cached object was only returned from cache if it was not garbage collected and was not timed out.

To enable this `ObjectCache` implementation declare

```
<object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
    <attribute attribute-name="cacheExcludes" attribute-value=""/>
    <attribute attribute-name="timeout" attribute-value="900"/>
    <attribute attribute-name="autoSync" attribute-value="true"/>
    <attribute attribute-name="cachingKeyType" attribute-value="0"/>
    <attribute attribute-name="useSoftReferences" attribute-value="true"/>
</object-cache>
```

Implementation configuration properties:

Property Key	Property Values
timeout	Lifetime of the cached objects in seconds. If expired, the cached object was discarded - default was 900 sec. When set to *-1* the lifetime of the cached object never expire.
autoSync	If set *true* all cached/looked up objects within a PB-transaction are traced. If the the PB-transaction was aborted all traced objects will be removed from cache. Default is *false*. **NOTE:** This does not prevent "dirty-reads" by concurrent threads (more info see above). It's not a smart solution for keeping cache in sync

	with DB but should do the job in most cases. E.g. if OJB read 1000 objects from the database within a transaction, one object was modified and the transaction will be aborted, then 1000 objects will be passed to the cache on lookup, 1000 objects will be traced and all 1000 objects will be removed from cache on abort. Read these objects without running tx or in a former tx and then modify one object in a tx and abort the tx, only one object was traced/removed. Keep in mind that this property counteract the *useSoftReferences* property as long as the PB-transaction is running, because all traced objects will have strong references.
cachingKeyType	Determines how the key was build for the cached objects: 0 - Identity object was used as key, this was the *default* setting. 1 - Idenity + jcdAlias name was used as key. Useful when the same object metadata model (DescriptorRepository instance) are used for different databases (JdbcConnectionDescriptor), because different databases should use separated caches (persistent object instances). 2 - Identity + model (DescriptorRepository) was used as key. Useful when different metadata model (DescriptorRepository instance) are used for the same database. Keep in mind that there was no synchronization between cached objects with same Identity but different metadata model. E.g. when the same database use different metadata versions of the same persistent object class. 3 - all together (Idenity + jcdAlias + model) If possible '0' is recommended, because it will be the best performing setting.
useSoftReferences	If set *true* this class use {@link java.lang.ref.SoftReference} to cache objects. Default value is *true*. If set *true* and the cached object was not longer referenced by your application but only by the cache, it can be reclaimed by the garbage collector. If set *false* it's strongly recommended to the *timeout* property to prevent memory problems of the JVM.

Recommendation:
If you take care of cache synchronization (or use autoSync property) and be aware of dirty reads, this implementation is useful for read-only or less update centric classes.

4.2. ObjectCacheTwoLevelImpl

ObjectCacheTwoLevelImpl is a two level ObjectCache implementation with a transactional session- and a shared application-cache part.

The first level is a transactional session cache that cache objects till PersistenceBroker#close() or if a PB-tx is running till `#abortTransaction()` or `#commitTransaction()` was called.
On commit all objects reside in the session cache will be pushed to the application cache.
If objects be new materialized from the database (e.g. when achieve a query), the full materialized

objects will be pushed immediately to the application cache (more precisely, if the application cache doesn't contain the "new materialized" objects).

The second level cache can be specified with the *applicationCache* property. Properties of the specified application cache are allowed too. Here is an example how to use the two level cache with `ObjectCacheDefaultImpl` as second level cache.

```
<object-cache class="org.apache.ojb.broker.cache.ObjectCacheTwoLevelImpl">
    <!-- meaning of attributes, please see docs section "Caching" -->
    <!-- common attributes -->
    <attribute attribute-name="cacheExcludes" attribute-value=""/>

    <!-- ObjectCacheTwoLevelImpl attributes -->
    <attribute attribute-name="applicationCache"
        attribute-value="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl"/>
    <attribute attribute-name="copyStrategy"
attribute-value="org.apache.ojb.broker.cache.ObjectCacheTwoLevelImpl$CopyStrategyImpl"/>
    <attribute attribute-name="forceProxies" attribute-value="true"/>

    <!-- ObjectCacheDefaultImpl attributes -->
    <attribute attribute-name="timeout" attribute-value="900"/>
    <attribute attribute-name="autoSync" attribute-value="true"/>
    <attribute attribute-name="cachingKeyType" attribute-value="0"/>
    <attribute attribute-name="useSoftReferences" attribute-value="true"/>
</object-cache>
```

The most important characteristic of the two-level cache is that all objects put to or read from the application cache are copies of the target object, so the cached objects never could be corrupted by the user when changing fields, because all operations done on copies of objects cached in the application cache (in contrast to ObjectCacheDefaultImpl).

The strategy to make copies of the persistent objects is pluggable and can be specified by the *copyStrategy* property which expects an implementation of the `ObjectCacheTwoLevelImpl.CopyStrategy` interface.

The *default* `ObjectCacheTwoLevelImpl.CopyStrategy` implementation make copies based on the field-descriptors of the cached object and set these values in a new instance of the cached object. If you lookup a cached object with *1:n or m:n* relation a query is needed to get the ID's of the referenced objects, because in application cache only *"flat"* objects without references/reference-info will be cached.

Note:

This two-level cache implementation does not guarantee that cache and persistent storage (e.g. database) are always consistent, because the session cache push the persistent objects to application cache *after* the PB-tx was commited.
Let us assume that thread 1 (using broker 1) update objects A1, A2, ... within a transaction and does commit the tx. Now before OJB could execute the after commit call on thread 1 to force session cache to push the objects to the application cache, thread 2 (using broker 2) lookup and update object A2 too (improbably but could happen, because thread 1 has already commited the objects A1, A2,... to the persistent storage) and push A2 to application cache. After this thread 1 was able to perform the after commit call and the 'outdated' version of A2 was pushed to the application cache overwriting the actual version of A2 in cache - cache and persistent storage are out of synchronization.
To avoid writing of outdated data to the persistence storage optimistic locking can be used. OL will not prevent the above scenario, but if it happens and e.g. broker 3 read the outdated object A1 from the cache and try to perform an update of A1, an optimistic locking exception will be thrown. So it is guaranteed that the persistent storage is always consistent.
A possibility to completely prevent synchronization problems of cache and persistent storage is the usage of pessimistic locking (if the used api supports it) with an adequate locking isolation level. If only one thread/broker could modify an object at the same time and the lock will be released after all work is done, the above scenario can't happen.

To avoid corrupted data, all objects cached by users (using the methods of the *ObjectCache* interface) will never be pushed to the application cache, they will be buffered in the session cache till it was cleared.

Implementation configuration properties:

Property Key	Property Values

applicationCache	Specifies the ObjectCache implementation used as application cache (second level cache). By default ObjectCacheDefaultImpl was used. It's recommended to use a shared cache implementation (all used PB instances should access the same pool of objects - e.g. by using a static Map in cache implementation).
copyStrategy	Specifies the implementation class of the `ObjectCacheTwoLevelImpl.CopyStrategy` interface, which was used to copy objects on read and write operations to application cache. If not set, a default implementation was used (`ObjectCacheTwoLevelImpl.CopyStrategyImpl` make field-descriptor based copies of the cached objects).
forceProxies	If *true* on materialization of cached objects, all referenced objects will be represented by proxy objects (independent from the proxy settings in reference- or collection-descriptor). **Note:** To use this feature all persistence capable objects have to be interface based **or** the `ProxyFactory` and `IndirectionHandler` implementation classes supporting dynamic proxy enhancement for all classes (see OJB.properties, find more information about proxy settings here).

4.3. ObjectCachePerBrokerImpl

ObjectCachePerBrokerImpl is a local/session cache implementation allows to have dedicated caches per PersistenceBroker instance.

Note: When the used broker instance was closed (returned to pool) the cache was cleared.

This cache implementation is not synchronized with the other `ObjectCache` instances, there will be no automatic refresh of objects modified/updated by other threads (*PersistenceBroker* instances).

So, objects modified by other threads will not influence the cached objects, because for each broker instance the objects will be cached separately and each thread should use it's own *PersistenceBroker* instance.

4.4. ObjectCacheEmptyImpl

This is an *no-op* ObjectCache implementation. Useful when caching was not desired.

Note:

This implementaion supports *circular references* as well (since OJB 1.0.2, materialization of object graphs with circular references will be handled internally by OJB).

4.5. ObjectCacheJCSImpl

A shared `ObjectCache` implementation using a JCS region for each classname. More info see turbine-JCS.

4.6. ObjectCacheOSCacheImpl

You're basically in good shape at this point. Now you've just got to set up OSCache to work with OJB. Here are the steps for that:

- Download OSCache from OSCache. Add the oscache-2.0.x.jar to your project so that it is in your classpath (for Servlet/J2EE users place in your WEB-INF/lib directory).
- Download JavaGroups from JavaGroups. Add the javagroups-all.jar to your classpath (for Servlet/J2EE users place in your WEB-INF/lib directory).
- Add oscache.properties from your OSCache distribution to your project so that it is in the classpath (for Servlet/J2EE users place in your WEB-INF/classes directory). Open the file and make the following changes:
  1. Add the following line to the CACHE LISTENERS section of your oscache.properties file:
     cache.event.listeners=com.opensymphony.oscache.plugins.clustersupport.JavaGroupsBroadcastingListe
  2. Add the following line at the end of the oscache.properties file (your network must support multicast):
     `cache.cluster.multicast.ip=231.12.21.132`
- Add the following class to your project (feel free to change package name, but make sure that you specify the full qualified class name in configuration files). You can find source of this class under `db-ojb/contrib/src/ObjectCacheOSCacheImpl` or copy this source:

```
public class ObjectCacheOSCacheImpl implements ObjectCacheInternal
{
    private Logger log = LoggerFactory.getLogger(ObjectCacheOSCacheImpl.class);
    private static GeneralCacheAdministrator admin = new
GeneralCacheAdministrator();
    private static final int REFRESH_PERIOD =
com.opensymphony.oscache.base.CacheEntry.INDEFINITE_EXPIRY;

    public ObjectCacheOSCacheImpl()
    {
    }

    public ObjectCacheOSCacheImpl(PersistenceBroker broker, Properties prop)
    {
    }

    public void cache(Identity oid, Object obj)
    {
        try
        {
            /*
            Actually, OSCache sends notifications (Events) only on flush
            events. The putInCache method do not flush the cache, so no event is
sent.
            The ObjectCacheOSCacheInternalImpl should force OSCache to flush the
entry
            in order to generate an event. This guarantee that other nodes
always
            in sync with the DB.
            Alternative a non-indefinite refresh-period could be used in
conjunction
            with optimistic-locking for persistent objects.
            */
            remove(oid);
            admin.putInCache(oid.toString(), obj);
        }
        catch(Exception e)
        {
            log.error("Error while try to cache object: " + oid, e);
        }
    }

    public void doInternalCache(Identity oid, Object obj, int type)
```

```
    {
        cache(oid, obj);
    }

    public boolean cacheIfNew(Identity oid, Object obj)
    {
        boolean result = false;
        Cache cache = admin.getCache();
        try
        {
            cache.getFromCache(oid.toString());
        }
        catch(NeedsRefreshException e)
        {
            try
            {
                cache.putInCache(oid.toString(), obj);
                result = true;
            }
            catch(Exception e1)
            {
                cache.cancelUpdate(oid.toString());
                log.error("Error while try to cache object: " + oid, e);
            }
        }
        return result;
    }

    public Object lookup(Identity oid)
    {
        Cache cache = admin.getCache();
        try
        {
            return cache.getFromCache(oid.toString(), REFRESH_PERIOD);
        }
        catch(NeedsRefreshException e)
        {
            // not found in cache
            if(log.isDebugEnabled()) log.debug("Not found in cache: " + oid);
            cache.cancelUpdate(oid.toString());
            return null;
        }
        catch(Exception e)
        {
            log.error("Unexpected error when lookup object from cache: " + oid,
e);
            cache.cancelUpdate(oid.toString());
            return null;
        }
    }

    public void remove(Identity oid)
    {
        try
        {
            if(log.isDebugEnabled()) log.debug("Remove from cache: " + oid);
            admin.flushEntry(oid.toString());
        }
        catch(Exception e)
        {
            throw new RuntimeCacheException("Unexpected error when remove object
from cache: " + oid, e);
        }
    }

    public void clear()
    {
        try
        {
            if(log.isDebugEnabled()) log.debug("Clear cache");
            admin.flushAll();
```

```
        }
        catch(Exception e)
        {
            throw new RuntimeCacheException("Unexpected error while clear
cache", e);
        }
    }
}
```

To allow usage of this implementation as application cache level in the two-level cache implement the internal object cache interface instead of the standard one.

Now *OSCache* can be used by OJB as standalone cache (by declaring the implementation class on connection- or class-level) or as application cache in the two-level cache.

4.7. More implementations ...

Additional *ObjectCache* implementations can be found in *org.apache.ojb.broker.cache* package.

5. Distributed ObjectCache?

If OJB was used in a clustered enviroment it is mandatory to distribute all shared cached objects across different JVM. OJB does not support distributed caching "out of the box", to do this a external caching library is needed, e.g. the OSCache implementation supports distributed caching. More information how to setup OJB in clustered enviroments see clustering howto.

6. Implement your own cache

The OJB cache implementations are quite simple but should do a good job for most scenarios. If you need a more sophisticated cache or need to pluggin a proprietary caching library you'll write your own implementation of the ObjectCache interface.
Integration of your implementation in OJB is easy since the object cache is a pluggable component. All you have to do, is to declare it on connection- or class-level. Here an example howto declare the new implementation on connection level:

```
<jdbc-connection-descriptor
    jcd-alias="myDefault"
    ...
>
    <object-cache class="my.ObjectCacheMyImpl">
        <attribute attribute-name="cacheExcludes" attribute-value=""/>
        ... additional attributes of the cache
     </object-cache>
</jdbc-connection-descriptor
```

If interested to get more detailed information about the "type" of the objects to cache (objects written to DB, new materialized objects,...) implement the ObjectCacheInternal interface (For an implementation example see source for ObjectCacheTwoLevelImpl).

Note:
Of course we interested in your solutions! If you have implemented something interesting, just contact us.

7. Future prospects

In OJB 1.1 the caching part will be rewritten to get rid of static classes, factories and member variables.