Frequently Asked Questions about PojoCache

Ben Wang <ben.wang@jboss.com>

Scott Marlow <smarlow@novell.com>

Release 2.0.0

June 2007

These are frequently asked questions regarding Pojocache.

General Information

Q:

What is PojoCache?

A:

PojoCache is a fine-grained field-level replicated and transactional POJO (plain old Java object) cache. By POJO, we mean that the cache: 1) automatically manages object mapping and relationship for a client under both local and replicated cache mode, 2) provides support for inheritance relationship between "aspectized" POJOs. By leveraging the dynamic AOP in JBossAop, it is able to map a complex object into the cache store, preserve and manage the object relationship behind the scene. During replication mode, it performs fine-granularity (i.e., on a per-field basis) update, and thus has the potential to boost cache performance and minimize network traffic.

From a user perspective, once your POJO is managed by the cache, all cache operations are transparent. Therefore, all the usual in-VM POJO method semantics are still preserved, providing ease of use. For example, if a POJO has been put in PojoCache (by calling `attach` , for example), then any POJO get/set method will be intercepted by PojoCache to provide the data from the cache.

Q:

What is the relationship between Cache and PojoCache?

A:

The core JBoss Cache library `Cache` is a traditional generic distributed cache system. PojoCache uses Cache as the underlying distributed state system to achieve POJO caching. It uses Cache as a delegate. As a result, all the replication aspects are configured with the Cache configuration XML. Additionally, PojoCache also has API to expose the Cache interface (via `getCache()` API).

Q:

What is the difference between Cache and PojoCache?

A:

Think of PojoCache as a Cache on steroids. :-) Seriously, both are cache stores-- one is a generic cache and the other other one POJO Cache. However, while Cache only provides pure object reference storage (e.g., `put(FQN fqn, Object key, Object value)`), PojoCache goes beyond that and performs fine-grained field level replication object mapping and relationship management for a user behind the scenes. As a result, if you have complex object systems that you would like to cache, you can have PojoCache manage it for you. You simply treat your object systems as they are residing in-memory, e.g., use your regular POJO methods without worrying about cache management. Furthermore, this is true in replication mode as well.

Q:
How does PojoCache work then?

A:
PojoCache uses the so-called AOP technology (aspect oriented programming) to do field level interception. Currently, it uses `JBoss Aop` library to do it.

Q:
What's changed between 1.x and 2.x release then?

A:
Starting in 2.0 release, we have a separate library for PojoCache, `pojocache.jar` that is extra to the core `jboss-cache.jar` . Since we uses Cache as a delegate, user will need to have a regular xml to configure the core Cache functionality (e.g., replication and locking aspect). In addition, there is also the `pojocache-aop.xml` that specifies the PojoCache interceptor stack (that can be left as default).

Additionally, here are the changed features:

- New APIs. It replaces `putObject, removeObject, and get` with `attach, detach, and find` .
- New POJO based events that a user can subscribe to.
- New configuration pojocache-aop.xml specifically for PojoCache, in addition to the regular cache-service.xml for the delegating Cache.
- New package namespace (`org.jboss.cache.pojo`) for PojoCache. The previous `org.jboss.cache.aop` space has been deprecated.

Q:
What are the steps to use the PojoCache feature?

A:
In order to use PojoCache, you will need to:

- prepare POJO. You can do either via xml declaration or JDK50 annotation. This is the step to declare your POJO such that it will be instrumented by `JBoss Aop` .
- instrumentation. You will need to instrument your POJO either at compile- or load-time. If you do it during compile-time, you use so-called an aop pre-compiler (aopc) to do bytecode manipulation. If you do it via load-time, however, you need either a special system class loader or, in JDK50, you can use the javaagent option. Either way, `JBoss Aop` will byte code manipulate your POJO class such that all field access can be intercepted.

So if you use JDK50, for example, with annotation and load-time instrumentation, then you won't need any pre-processing step to use PojoCache. For a full example, please refer to the distro examples directory. There are numerous PojoCache examples that uses different options.

Q:

What is the JDK version required to run PojoCache 2.x?

A:

PojoCache 2.x requires JDK5.0 since it uses the annotation extensively.

Q:

Can I run PojoCache as a standalone mode?

A:

Yes, same as the core Cache library, you can run PojoCache either as a standalone or inside an application server.

Q:

What is the JBoss AS recommended version to run PojoCache 2.x?

A:

PojoCache can be run either in AS4.0.5 (and up) and 5.0. But either way, it will require JDK5.0 though.

Q:

Can I pre-compile the aop classes such that I don't need to use the system classloader and jboss-aop configuration xml during runtime?

A:

Yes. The latest versions of JBossCache have a pre-compiler option called `aopc` . You can use this option to pre-compile your "aspectized" POJO. Once the classes have been byte code generated, they can be treated as regular class files, i.e., you will not need to include any `jboss-aop.xml` that specifies the advisable POJO and to specify the JBossAop system class loader.

For an example of how to use `aopc` , please see 1) `tools` directory for PojoCacheTasks14.xml and PojoCacheTasks50.xml. Both contain Ant tasks that you can import to your regular project for `aopc` . In addition, please also check out the `examples` directory for concrete examples.

Q:

In PojoCache 2.x release, do I still need `annoc` ?

A:

The annoc precompiler is needed for JDK1.4 style annotation. For 2.x release, since we require the use of JDK5.0, there is no need to use annoc anymore.

Q:

How do I use aopc on multiple module directories?

A:

In aopc, you specify the src path for a specific directory. To pre-compile multiple ones, you will need to invoke aopc multiple times.

Q:

Does PojoCache provide POJO event subscription?

A:

Yes, since 2.0, you can use PojoCacheListener to subscribe to events such as POJO attach and detach and field updates. And if you need some customization, you can also use the Obervable pattern directly. TO see an example, please check out the test case: `org.jboss.cache.pojo.observer.LocalTest.java`

Q:

What's in the `jboss-aop.xml` configuration?

A:

`jboss-aop.xml` is needed for POJO instrumentation. In `jboss-aop.xml` , you can declare your POJO (e.g., `Person`) to be "prepared", a JBossAop term to denote that the object will be "aspectized" by the system. After this declaration, JBossAop will invoke any interceptor that associates with this POJO. PojoCache will dynamically add an `org.jboss.cache.pojo.interceptor.dynamic.CacheFieldInterceptor` to this POJO to perform object mapping and relationship management.

Note that to add your POJO, you should declare all the fields to be "prepared" as in the example.

Q:

What's the difference between `jboss-aop.xml` `pojocache-aop.xml` ?

A:

`pojocache-aop.xml` is essentially a `jboss-aop.xml` , except it is used specifically for PojoCache. The analogy is similar to JBoss' own MBean service file `jboss-service.xml` , for example. So in our documentation, we will use these two terms interchangeably.

Q:

Can I use annotation instead of the xml declaration?

A:

Yes, in release 2.0, you can use JDK5.0 annotation to instrument your POJO. Check the documentation for details.

Q:

What are the pro and con of xml vs. annotation?

A:

It really depends on your organization environment, I'd say, since this can be turned into a hot debate. Having said that, I feel strongly that POJO annotation is well suited for PojoCache. This is because once you specify the annotation, you'd probably change it rarely since there is no parameters to tune, for example.

Q:

What are the `@org.jboss.cache.pojo.annotation.Transient` and `@org.jboss.cache.pojo.annotation.Serializable` field level annotations?

A:

In 2.0, we also offer two additional field-level annotations. The first one, `@org.jboss.cache.pojo.Transient` , when applied has the same effect as declaring a field `transient` . PojoCache won't put this field under management.

The second one, `@org.jboss.cache.pojo.Serializable` when applied, will cause PojoCache to treat the field as a Serializable object even when it is `@org.jboss.cache.pojo.Replicable` .

Q:

What about compile-time vs. load-time instrumentation then?

A:

Again it depends. But my preference is to do compile-time instrumentation via aopc. I prefer this approach because it is easier to debug (at least at the development stage). In addition, once I generate the new class, there is no more steps needed.

Q:

Is it possible to store the same object multiple times but with different Fqn paths? Like /foo/byName and /foo/byId ?

A:

Yes, you can use PojoCache to do that. It supports the notion of object reference. PojoCache manages the unique object through association of the dynamic cache interceptor.

Q:

Do I need to declare all my objects "prepared" in `jboss-aop.xml` ?

A:

Not necessarily. If there is an object that you don't need the cache to manage for you, you can leave it out of the declaration. The cache will treat this object as a "primitive" type. However, the object will need to implement `Serializable` interface for replication.

Q:

Can the cache aop intercept update via reflection?

A:

No. The update via reflection will not be intercepted in JBossAop and therefore PojoCache will not be able to perform the necessary synchronization.

Q:

When I declare my POJO to be "aspectized", what happens to the fields with transient, static, and final modifiers?

A:

PojoCache currently will ignore the fields with these modifiers. That is, it won't put these fields into the cache (and thus no replication either).

Q:

What are those keys such as `JBoss:internal:class` and `PojoInstance` ?

A:

They are for internal use only. Users should ignore these keys and values in the node hashmap.

Q:

What about Collection classes? Do I need to declare them "prepared"?

A:

No. Since the Collection classes such as `ArrayList` are java util classes, aop by default won't instrument these classes. Instead, PojoCache will generate a dynamic class proxy for the Collection classes (upon the `attach` call is invoked). The proxy will delegate the operations to a cache interceptor that implements the actual Collection classes APIs. That is, the system classes won't be invoked when used in PojoCache.

Internally, the cache interceptor implements the APIs by direct interaction with respect to the underlying cache store. Note that this can have implications in performance for certain APIs. For example, both `ArrayList` and `LinkedList` will have the same implementation. Plan is currently underway to optimize these APIs.

Q:

How do I use `List` , `Set` , and `Map` dynamic proxy?

A:

PojoCache supports classes extending from `List` , `Set` , and `Map` without users to declare them "aspectized".

It is done via a dynamic proxy. Here is a code snippet to use an `ArrayList` proxy class.

```
ArrayList list = new ArrayList();
               list.add("first");

               cache.attach("list/test", list); // Put the list under the aop cache
               list.add("second"); // Won't work since AOP intercepts the dynamic proxy not the original

               ArrayList myList = (List)cache.find("list/test"); // we are getting a dynamic proxy instea
               myList.add("second"); // it works now
               myList.add("third");
               myList.remove("third");
```

Q:

What is the proper way of assigning two different keys with Collection class object?

A:

Let's say you want to assign a `List` object under two different names, you will need to use the class proxy to insert the second time to ensure both are managed by the cache. Here is the code snippet.

```
ArrayList list = new ArrayList();
               list.add("first");

               cache.attach("list", list); // Put the list under the aop cache

               ArrayList myList = (List)cache.find("list"); // we are getting a dynamic proxy instead
               myList.add("second"); // it works now

               cache.attach("list_alias", myList); // Note you will need to use the proxy here!!
               myList.remove("second");
```

Q:

OK, so I know I am supposed to use proxy when manipulating the Collection classes once they are managed by the cache. But what happens to Pojos that share the Collection objects, e.g., a `List` instance that is shared by 2 Pojos?

A:

Pojos that share Collection instance references will be handled by the cache automatically. That is, when you ask the Cache to manage it, the Cache will dynamically swap out the regular Collection references with the dynamic proxy ones. As a result, it is transparent to the users.

Q:

What happens when my "aspectized" POJO has field members that are of Collection class ?

A:

When a user puts a POJO into the cache through the call `attach` , it will recursively map the field members into the cache store as well. When the field member is of a Collection class (e.g., List, Set, or Map), PojoCache will first map the collection into cache. Then, it will swap out dynamically the field reference with an corresponding proxy reference.

This is necessary so that an internal update on the field member will be intercepted by the cache.

Q:

What are the limitation of Collection classes in PojoCache?

A:

Use of Collection class in PojoCache helps you to track fine-grained changes in your collection fields automatically. However, current implementation has the follow limitation that we plan to address soon.

Currently, we only support a limited implementation of Collection classes. That is, we support APIs in List, Set, and Map. However, since the APIs do not stipulate of constraints like NULL key or value, it makes mapping of user instance to our proxy tricky. For example, ArrayList would allow NULL value and some other implementation would not. The Set interface maps to java.util.HashSet implementation. The List interface maps to java.util.ArrayList implementation. The Map interface maps to java.util.HashMap implementation.

Another related issue is the expected performance. For example, the current implementation is ordered, so that makes insert/delete from the Collection slow. Performance between Set, Map and List collections also vary. Adding items to a Set is slower than a List or Map, since Set does not allow duplicate entries.

Q:

What are the pros and cons of PojoCache?

A:

As mentioned in the reference doc, PojoCache has the following advantages:

- Fine-grained replication and/or persistency. If you use a distributed PojoCache and once your POJO is put in the cache store, there is no need to use another API to trigger your changes. Furthermore, the replication are fine-grained field level. Note this also applies to persistency.
- Fine-grained replication can have potential performance gain if your POJO is big and the changes are fine-grained, e.g., only to some selected fields.
- POJO can posses object relationship, e.g., multiple referenced. Distributed PojoCache will handle this transparently for you.

And here are some cases that you may not want to use PojoCache:

- You use only cache. That is you don't need replication or persistency. Then since everything is operated on the in-memory POJO reference, there is no need for PojoCache.
- You have simple and small POJOs. Your POJO is small in size and also there is no object relationship, then PojoCache possess not clear advantage to plain cache.
- Your application is bounded by memory usage. Because PojoCache need almost twice as much of memory (the original POJO in-memory space and also the additional cache store for the primitive fields), you may not want to use PojoCache.
- Your POJO lifetime is short. That is, you need to create and destroy your POJO often. Then you need to do "attach" and "detach" often, it will be slow in performance.

Passiviation and eviction

Q:

Can I use eviction to evict POJO from the memory?

A:
No. In 2.0 release, we have deprecated the POJO-based eviction policy since it has always been problematic in earlier release. The main reason is that when we evict a POJO from the memory, the user has no ways of knowing it. So if the POJO is accessed after the eviction, there won't be any PojoCache interception (e.g., it will be just like ordinary POJO), but user may still expect that it will be managed by PojoCache.

Q:
So what do I do now?

A:
In order to keep your memory from overflowing, you can use the passivation feature that comes with the core Cache. Passivation uses the combination of eviction and cache loader such that when the items are old, it will be evicted from memory and store in a cache store (can be DB or file). Next time, when the item needs to be accessed again, we will retrieve it from the cache store.

In this sense, PojoCache level is not aware of the passivation aspect. It is configured through the underlying cache xml.

Troubleshooting

Q:
I am having problems getting PojoCache to work, where can I get information on troubleshooting?

A:
Troubleshooting section can be found in the following wiki link [1] .

[1] http://wiki.jboss.org/wiki/Wiki.jsp?page=PojoCacheTroubleshooting