Frequently Asked Questions about POJO Cache

Ben Wang

Scott Marlow

Jason Greene

Release 2.1.0.GA

March 2008

These are frequently asked questions regarding POJO Cache.

General Information

Q: What is POJO Cache?

A: POJO Cache 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 POJO Cache (by calling `attach` , for example), then any POJO get/set method will be intercepted by POJO Cache to provide the data from the cache.

Q: What is the relationship between Core Cache and POJO Cache?

A: Cores Cache is a traditional generic distributed cache system. POJO Cache uses Core Cache as the underlying distributed state system to achieve object caching. As a result, all the replication aspects are configured with the Cache configuration XML. Additionally, POJO Cache also has API to expose the Cache interface (via `getCache()` API).

Q: What is the difference between Core Cache and POJO Cache?

A: Think of POJO Cache 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)`), POJO Cache 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 POJO Cache 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.

Q: How does POJO Cache work then?

A: POJO Cache uses the JBoss AOP project to perform field level interception. This allows POJO Cache to monitor changes to your object model, and react accordingly.

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

A: Starting in 2.0 release, we have a separate library for POJO Cache, `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 POJO Cache 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 POJO Cache, in addition to the regular cache-service.xml for the delegating Cache.
- New package namespace (`org.jboss.cache.pojo`) for POJO Cache. The previous `org.jboss.cache.aop` space has been deprecated.

Q: How do you use POJO Cache?

A: In order to use POJO Cache, you will need to:

- Annotate your POJOt with @Replicable.
- Instrument your POJO. This can be done at load-time using special JVM arguments (prefered), or at compile time using the AOP precompiler tool (aopc). See the user guide for more specific details on instrumentation.

Q: What is the JDK version required to run POJO Cache 2.x?

A: POJO Cache 2.x requires Java 5 or newer.

Q: Can I run POJO Cache as a standalone mode?

A: Yes, same as the Core Cache library, you can run POJO Cache either as a standalone or inside an application server.

Q: What is the JBoss AS recommended version to run POJO Cache 2.x?

A: POJO Cache 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 objects used in POJO Cache, so that I don't have to provide an AOP descriptor?

A: Yes. The AOP library included with POJO Cache has a pre-compiler called `aopc` that can be used to instrument objects in advance. However, this is not the recommended approach, since your classes become tied to a specific AOP version. See the instrumentation chapter in the user guide for more information.

Q: In POJO Cache 2.x release, do I still need `annoc` ?

A: No. POJO Cache 2.x requires JDK 5, and recommends load-time instrumentation. Alternatively the offline aopc tool may be used.

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 POJO Cache provide a listener/event model for catching changes?

A: Yes. See the javadoc for PojoCache.addListener() and @PojoCacheListener.

Q: What's in the `pojocaches-aop.xml` configuration?

A: These descriptors are necessary for instrumentation. However, you typically do not need to touch them since they include a rule which matches the classes with an @Replicable annotation. Therefore, all you need to do, is just annotate your class with @Replicable. Advanced users may decide to customize them with special AOP prepare statements that match classes which do not have @Replicable.

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 POJO Cache. 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 annotations instead of editing the AOP XML descriptors?

A: Yes, in release 2.0, we recommend you use the @Replicable annotation, and don't bother with editing the AOP files.

Q: Is there a problem with using a custom AOP descriptor over the provided annotations?

A: The only real benefit to a custom AOP descriptor is if you can't easily add the annotation to the class source (it's not under your control).

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` . POJO Cache won't put this field under management.

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

Q: Why do you recommend load-time over compile-time instrumentation?

A: The major problem with compile-time instrumentation is that it adds a binary dependency on your class files to whatever version of JBoss AOP that was used to run aopc. Once this has been done, the class may not work with a future version of JBoss AOP (although the AOP team tries to ensure binary compatibility across minor revisions). Load-time doesn't have this problem since the class is instrumented only in memory, and only when it is loaded.

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 POJO Cache to do that. It supports the notion of multiple object references. POJO Cache manages the unique object through association of the dynamic cache interceptor.

Q: Do I have to instrument my objects?

A: You can also attach objects that implement `Serializable`. However, you lose field-level replication and object identity preservation. This is really only supported as a compatibility measure. It is definately worth useing instrumentation.

Q: Will POJO Cache intercept changes made from Java Reflection?

A: Yes and No. Since POJO Cache intercepts field changes, any method of an object that has been annotated with @Replicable will be handled properly when

called with reflection. However, modifying fields using reflection is not currently supported.

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

A: POJO Cache 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, POJO Cache 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 POJO Cache.

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` with POJO Cache?

A: POJO Cache supports all classes that implement `List` , `Set` , and `Map` without instrumentation. This is done using a dynamic proxy. Here is an example using `ArrayList`:

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

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

  ArrayList myList = (List)cache.find("/list/test"); // we
are getting a dynamic proxy instead
  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 two objects??

A: POJOss 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 you.

Q: What happens when my instrumented object contains collections?

A: When an object is passed to `attach` , it will recursively map the field members into the cache store as well. If the field member is of a Collection class (e.g., List, Set, or Map), POJO Cache will first map the collection into cache. Then, it will swap out 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 limitations of using Java Collections in POJO Cache?

A: List, Set, and Map are supported; however, these APIs do not stipulate of constraints like whether a null key or value is allowed. Therefore the

behavior of an attached collection may differ slightly from the originals Java implementation. The behavior implemented by POJO Cache follows java.util.HashSet for any Set type, java.util.ArrayList for any List type, and java.util.HashMap for any Map type.s

Passiviation and eviction

Q: Can I use eviction to evict a POJO from the memory?

A: In 2.0 release, we removed the POJO-based eviction policy since it has always been problematic in earlier release. You can, however, use the standard Core Cache eviction system to evict the data that backs the POJO.

Q: Is passivation supported?

A: Yes, in order to reduce memory consumption, you can use the passivation feature that comes with Core Cache. Passivation uses the combination of eviction and a 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.

Troubleshooting

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

A: Troubleshooting section can be found in the following *wiki link* [http://wiki.jboss.org/wiki/Wiki.jsp?page=PojoCacheTroubleshooting] .