JPA Caching

by Carol McDonald MVB · **Aug. 24, 09 · Java Zone · Not set**

JPA has 2 levels of caching. The first level of caching is the persistence context.

The JPA Entity Manager maintains a set of Managed Entities in the Persistence Context.

The Entity Manager guarantees that within a single Persistence Context, for any particular database row, there will be only one object instance. However the same entity could be managed in another User's transaction, so you should use either optimistic or pessimistic locking as explained in JPA 2.0 Concurrency and locking

The code below shows that a find on a managed entity with the same id and class as another in the same persistence context , will return the same instance.

```
@Stateless public ShoppingCartBean implements ShoppingCart {

 @PersistenceContext EntityManager entityManager;

 public OrderLine createOrderLine(Product product,Order order) {
        OrderLine orderLine = new OrderLine(order, product);
        entityManager.persist(orderLine);   //Managed
        OrderLine orderLine2 =entityManager.find(OrderLine,
orderLine.getId());
```

```
orderLine.getId());
     (orderLine == orderLine2)   // TRUE
        return (orderLine);
    }

}
```

The diagram below shows the life cycle of an Entity in relation to the Persistent Context.

The code below illustrates the life cycle of an Entity. A reference to a container managed EntityManager is injected using the persistence context annotation. A new order entity is created and the entity has the state of new. Persist is called, making this a managed entity. because it is a stateless session bean it is by default using container managed transactions , when this transaction commits , the order is made persistent in the database. When the orderline entity is returned at the end of the transaction it is a detached entity.

The Persistence Context can be either Transaction Scoped-- the Persistence Context 'lives' for the length of the transaction, or Extended-- the Persistence Context spans multiple transactions. With a Transaction scoped Persistence Context, Entities are "Detached" at the end of a transaction.

As shown below, to persist the changes on a detached entity, you call the EntityManager's merge() operation, which returns an updated managed entity, the entity updates will be persisted to the database at the end of the transaction.

An Extended Persistence Context spans multiple transactions, and the set of Entities in the Persistence Context stay Managed. This can be useful in a work flow scenario where a "conversation" with a user spans multiple requests.

The code below shows an example of a Stateful Session EJB with an Extended Persistence Context in a use case scenario to add line Items to an Order. After the Order is persisted in the createOrder method, it remains managed until the EJB remove method is called. In the addLineItem method , the Order Entity can be updated because it is managed, and the updates will be persisted at the end of the transaction.

The example below contrasts updating the Order using a transaction scoped Persistence Context verses an extended Persistence context. With the transaction scoped persistence context, an Entity Manager find must be done to look up the Order, this returns a Managed Entity which can be updated. With the Extended Persistence Context the find is not

necessary. The performance advantage of not doing a database read to look up the Entity, must be weighed against the disadvantages of memory consumption for caching, and the risk of cached entities being updated by another transaction. Depending on the application and the risk of contention among concurrent transactions this may or may not give better performance / scalability.

JPA second level (L2) caching

JPA second level (L2) caching shares entity state across various persistence contexts.

JPA 1.0 did not specify support of a second level cache, however, most of the persistence providers provided support for second level cache(s). JPA 2.0 specifies support for basic cache operations with the new Cache API, which is accessible from the EntityManagerFactory, shown below:

If L2 caching is enabled, entities not found in persistence context, will be loaded from L2 cache, if found.

The advantages of L2 caching are:

- avoids database access for already loaded entities
- faster for reading frequently accessed unmodified entities

The disadvantages of L2 caching are:

- memory consumption for large amount of objects
- Stale data for updated objects
- Concurrency for write (optimistic lock exception, or pessimistic lock)
- Bad scalability for frequent or concurrently updated entities

You should configure L2 caching for entities that are:

- read often

- read often
- modified infrequently
- Not critical if stale

You should protect any data that can be concurrently modified with a locking strategy:

- Must handle optimistic lock failures on flush/commit
- configure expiration, refresh policy to minimize lock failures

The Query cache is useful for queries that are run frequently with the same parameters, for not modified tables.

The EclipseLink JPA persistence provider caching Architecture

The EclipseLink caching Architecture is shown below.

Support for second level cache in EclipseLink is turned on by default, entities read are L2 cached. You can disable the L2 cache. EclipseLink caches entities in L2, Hibernate caches entity id and state in L2. You can configure caching by Entity type or Persistence Unit with the following configuration parameters:

- Cache isolation, type, size, expiration, coordination, invalidation,refreshing
- Coordination (cluster-messaging)
- Messaging: JMS, RMI, RMI-IIOP, ...
- Mode: SYNC, SYNC+NEW, INVALIDATE, NONE

The example below shows configuring the L2 cache for an entity using the @Cache annotation

The Hibernate JPA persistence provider caching Architecture

The Hibernate JPA persistence provider caching architecture is different than EclipseLink: it is not configured by default, it does not cache enities just id and state, and you can plug

in different L2 caches. The diagram below shows the different L2 cache types that you can plug into Hibernate.

The configuration of the cache depends on the type of caching plugged in. The example below shows configuring the hibernate L2 cache for an entity using the @Cache annotation

For More Information:

Introducing EclipseLink

EclipseLink JPA User Guide

Hibernate Second Level Cache
Speed Up Your Hibernate Applications with Second-Level Caching
Hibernate caching
Java Persistence API 2.0: What's New ?
Beginning Java™ EE 6 Platform with GlassFish™ 3
Pro EJB 3: Java Persistence API (JPA 1.0)

Like This Article? Read More From DZone

Proposed Jakarta EE Design Principles

Advanced Research Initiatives: Focus on the Future

How to Configure NGINX High Availability Cluster Using Pacemaker on Ubuntu 16.04

Free DZone Refcard
Getting Started With Kotlin

Topics:

Opinions expressed by DZone contributors are their own.

Java Partner Resources

Learning Kotlin: Object Expressions and SAM Conversions

In this post from "Learning Kotlin," we take a look at object expressions and SAM conversions using the Comparator class in Java.

by Robert Maclean MVB · Jul 02, 18 · Java Zone · Tutorial

For this post on learning Kotlin, we get to look at something new to an old C# person — object expressions! Object expressions are very similar to anonymous classes in Java where you can declare a class inline rather than entirely separately in its own file.

In the first Koan, we need to implement the `Comparator<Int>` inline:

```
1  fun task10(): List<Int> {
2      val arrayList = arrayListOf(1, 5, 2)
3      Collections.sort(arrayList, object : Comparator<Int> {
4          override fun compare(o1:Int, o2:Int):Int {
5              return o2 - o1;
6          }
7      })
8      return arrayList
9  }
```

You can see on line three, we define the new object with the `object` keyword and, then, use `: Comparator<Int>` to state that it implements that interface. The `Comparator` has a single function that needs to be implemented, as you can see in line four.

The second Koan takes this further and states that, if there is a single method in an abstract class or interface, we can use the Single Abstract Method (SAM). This is done to avoid needing the class since we just need to implement the single function. To achieve this with the Koan, we use an anonymous function that handles the compare function of the `Comparator`:

```
1  fun task11(): List<Int> {
2      val arrayList = arrayListOf(1, 5, 2)
3      Collections.sort(arrayList, { x, y -> y - x})
4      return arrayList
5  }
```

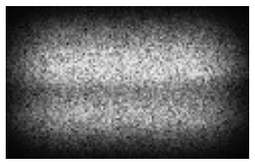
Lastly, if we look back to the previous post, we can use the extension method to simplify it further:

```
1  fun task12(): List<Int> {
2      return arrayListOf(1, 5, 2).sortedDescending()
3  }
```

These Koans have given me more thoughts about the language than probably any previous Koans:

1. Why do classes that implement an interface use override? In C#, when you implement an interface, you do not need to state that you are not overriding the functions (see this example). In C#, the only state that you're overriding is when you inherit from a function and you override a function. The reason is that an interface in Kotlin is closer to an abstract class than in C#, to the point it can have functionality. Yep, interfaces can have functions and logic!
2. So, why does Kotlin have interfaces and abstract classes? The key difference is that an abstract class can have a state, while the logic in an interface needs to be stateless!
3. Why bother having SAM? As I was working on the Koan, I was delighted by the SAM syntax. But, then, I wondered why I needed this at all? Why is `Collections.sort` taking a class as the second parameter? Why not just pass in a function, since that is all that is actually needed? Both C# and Kotlin support passing functions so that this is possible. However, something I never knew about Java is that it doesn't support passing functions! You have to use Callable, a class, to pass functions.

Like This Article? Read More From DZone



Kotlin: Static Methods

Testing Kotlin With Spock (Part 1): Object

Creating Stubs Using Java 8 Lambdas

Free DZone Refcard

Getting Started With Vaadin 10

Topics: JAVA, TUTORIAL, KOTLIN, COMPARATOR, OBJECT EXPRESSIONS, SAM

Published at DZone with permission of Robert Maclean , DZone MVB. See the original article here.

Opinions expressed by DZone contributors are their own.