**Hibernate Caching – First Level Cache**

Hibernate Cache can be very useful in gaining fast application performance if used correctly. The idea behind cache is to reduce the number of database queries, hence reducing the throughput time of the application.

Hibernate comes with different types of Cache:

1. **First Level Cache**: Hibernate first level cache is associated with the Session object. Hibernate first level cache is enabled by default and there is no way to disable it. However hibernate provides methods through which we can delete selected objects from the cache or clear the cache completely.

Any object cached in a session will not be visible to other sessions and when the session is closed, all the cached objects will also be lost.

1. **Second Level Cache**: Hibernate Second Level cache is disabled by default but we can enable it through configuration. Currently EHCache and Infinispan provides implementation for Hibernate Second level cache and we can use them. We will look into this in the next tutorial for hibernate caching.
2. **Query Cache**: Hibernate can also cache result set of a query. Hibernate Query Cache doesn’t cache the state of the actual entities in the cache; it caches only identifier values and results of value type. So it should always be used in conjunction with the second-level cache.

## ****Overview****

Caching is a feature that is being used in applications to improve the performance. Cache is positioned between the database and the application and usually the data of database queries gets cached (local copy) in disk or memory so that subsequent calls for the same data can be served by cache only.

Hibernate supports caching at a class level, collections level and even at a query level.

## ****Caching Scopes****

Cache can be in any one of the below three scopes

1. **Transactional Scope** – The scope of this type of cache is limited to the unit of work (transaction). Each transaction has its own  copy so it will not be shared and will not be accessed concurrently.
   1. If the same query is fired in same session again, it will be served by cache second time but in different session, it will not be served by cache.
2. **Process Scope -**  In this scope, cache is shared between transactions (or we can say between sessions) and between multiple threads. If the same query is fired in different session also, it will be served by cache. As data retrieved by one session will be visible to another session as well, we have to use this scope carefully.
   1. Note: Either a  persistent instances itself or  persistent state in a disassembled format can be stored in process cache. Each transaction  accessing the cache  needs to reassemble at instance from  cached data.
3. **Cluster Scope-**This scope is shared between process running on different machines (in cluster )as well.

**Hibernate Cache Architecture,** Hibernate has a two levels of  cache.

1. **First Level Cache –** The scope of this transaction is at a session level and  is  enabled for session by default. If the same query is fired in same session again, it will be served by cache second time, but in a different session, it will not be served by cache.  (transactional scope)
2. **Second Level Cache –** Second level cache can be enabled for a process or cluster scope. In this cache the state of a persistent object is stored (in a disassembled) form and not as a complete instance. Second level cache is optional and can be configured to cache at a class, collection level.

        There are several cache providers available and supported by Hibernate which can be plugged to use.

        It is very important to know that second  level cache is at a SessionFactory level, which means all sessions of same **session factory** will share the cache data.

**Concurrency Strategies**

The main responsibility of Concurrency Strategy is to store and retrieve data from the cache. There are four in-built concurrency strategies –

1. **Read Only** –As its name suggests, it is appropriate for the data which is never changed like reference data (like month in a year)
2. **Non Strict Read Write** – High possibility that we can read stale data and inconsistency between the database and a cache. Should be used when data hardly changes.
3. **Read-Write-**  this strategy maintains read-committed isolation level (read a committed data) and is not available in clustered environments.
4. **Transactional**-It prevents stale data, so should be used when stale data is not acceptable at all.

Custom concurrency strategy can be written by implementing org.hibernate.cache.CacheConcurrencyStrategy interface .

## ****Cache Providers****

Hibernate does provide built-in support for following four Cache Providers

1. **EHCache**- This is a widely used open source cache framework and can cache the data in memory or in disk.  EHCache also supports Query cache. Complete Details about EHCache can be grabbed from[http://ehcache.org](http://ehcache.org/). EHCahce does not support Transaction Concurrency Strategy.
2. **OSCache (Open Symphony)** – Similar to EHCahce , OSCache is also open source cache framework and supports cluster cache as well. Details can be found at [www.opensymphony.com/oscache/](http://www.opensymphony.com/oscache/)
3. **SwarmCache**- SwarmCache does not support Hibernate Query cache and uses IP multicasting to support distributed cache. More details about SwarmCache is available on [http://swarmcache.sourceforge.net](http://swarmcache.sourceforge.net/).
4. **JBoss Cache** –This is also a distributed Cache framework and supports Hibernate Query Cache. JBoss cache replicates the data across all nodes in distributed system. More details about JBoss Cache can be read at [www.jboss.org/jbosscache/](http://www.jboss.org/jbosscache/)

We need to configure the cache provider in hibernate.cfg.xml file using “hibernate.cache.provider\_class” property. Value of this property is the cache provider class name (refer Section 15.6)

**Cache Providers Support for Concurrency Strategies**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Provider** | **Provider Class** | **Read Only** | **Non Strict Read Write** | **Read Write** | **Transactional** |
| EHCache | org.hibernate.cache.EHCacheProvider | Yes | Yes | Yes |  |
| OSCache | org.hibernate.cache.OSCacheProvider | Yes | Yes | Yes |  |
| Swarm Cache | org.hibernate.cache.SwarmCacheProvider | Yes | Yes | Yes |  |
| JBoss Cache | org.hibernate.cache.JBossCacheProvider | Yes |  |  | Yes |

## ****Cache Region****

Cache Region can be thought of as a named area and Hibernate does cache each class, collection in a different cache region. Each Cache Region conceptually stores the persistent state (values in disassembled form) and the identifier of the containing collections or other classes.

The name of a cache region is the fully qualified name of the class and for the  containing class/collections  it collection name together with qualified class name of the parent class. For example “if Book class has a collection of chapters “ then “com.hibenrate.tutorial.Book” and “com.hibernate.tutorial.Book.chapters” will be cache region names.

If the application is working with multiple Session Factories than cache names may conflict so we can use prefix support of Hibernate.The Hibernate supports a property named hibernate.cache.region\_prefix which can be used to specify cache region name prefix of  a SessionFactory. For example, if property is set as “sf1” then Book cache region will become “sf1.com.hibenrate.tutorial.Book”

## ****Example – First Level Cache****

As we discussed earlier that first level cache is the default cache and is at a session level. Let’s write an example to validate our understanding.

**Student Entity**

@Entity

@Table(name = "STUDENT")

**public** **class** Student {

@Id

**private** **int** id;

@Column(name = "NAME")

**private** String name;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

**Test Program**

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**import** org.hibernate.cfg.Configuration;

**public** **class** FirstLevelCacheMain {

**public** **static** **void** main(String args[]) {

Configuration cfg = **new** Configuration().configure();

SessionFactory factory = cfg.~~buildSessionFactory~~();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Student student = **new** Student();

student.setName("Student A");

**int** id = (Integer) session.save(student);

tx.commit();

session.close();

session = factory.openSession();

Student student1 = (Student) session.get(Student.**class**, id);

Student student2 = (Student) session.get(Student.**class**, id);

session.close();

session = factory.openSession();

Student student3 = (Student) session.get(Student.**class**, id);

session.close();

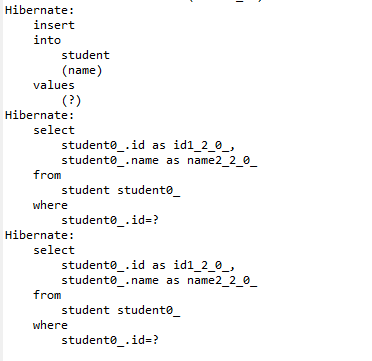
factory.close();

}

}

**Result**

On running Test Program we ca\n validate that only one select query is being fired for st1 and st2 because they are in the same session, but when again same student with same identifier is fetched in a different session, query is again fired because First level cache is session or transaction based.



**Second Level Caching Configuration**

There are certain steps need to follow to enable and configure the second level cache.

* 1. Select the Cache Provider and configure it in a hibernate.cfg.xml file. To do so

1. Use hibernate.cache.provider\_class property for Hibernate 3.
2. Use hibernate.cache.region.factory\_class  property cxif you are using hibernate 4
   1. Each cache provider has a configuration file where we need to configure the cache configuration.
   2. Enable Caching at a class or collection level in hbm xml file.
   3. Add required jar files in the build path.
   4. We will use ehcache provider to understand the caching.

### ****Configure EHCache Provider****

To configure EHCahce , add below lines in hibernate.cfg.xml.

**for Hibernate3**

 <property name=*"hibernate.cache.provider\_class"*>

org.hibernate.cache.EhCacheProvider

</property>

**for Hibernate 4**

  <property name=*"hibernate.cache.region.factory\_class"*>

org.hibernate.cache.ehcache.EhCacheRegionFactory

</property>

To use the singleton factory class,  use org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory class.

### ****EHCache Confguration****

We can configure the location of the ehcache configuration file in hibernate.cfg.xml file using net.sf.ehcache.configurationResourceName property. If we do not define then by default it will look for ehcache.xml file in the root directory. Below configuration in hibernate.cfg file will load myehcache.xml file available in the src root directory.

Note: If we don’t create ehcache.xml , it will not throw an error because it is available with default configuration in ehcache.jar file so that file will be picked.

<property name=*"net.sf.ehcache.configurationResourceName"*>

/myehcache.xml

</property>

EHCache Configuration looks like below

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<ehcache xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:noNamespaceSchemaLocation=*"ehcache.xsd"*>

<diskStore path=*"java.io.tmpdir/ehcache"* />

<defaultCache maxElementsInMemory=*"500"*

eternal=*"false"*

timeToIdleSeconds=*"220"*

timeToLiveSeconds=*"320"*

overflowToDisk=*"true"* />

<cache name=*"in.spring4buddies.application.caching.Student"*

maxElementsInMemory=*"500"*

eternal=*"false"*

timeToIdleSeconds=*"1800"*

timeToLiveSeconds=*"2400"*

overflowToDisk=*"false"* />

</ehcache>

**Details**

* **diskStore**:  EHCache does cache data in memory  and this property is used to specify the location where EHCache will store the data when it starts overflowing from memory.
* **defaultCache**: This configuration will be used there is no configuration defined for an object which needs to be cached. This is a  mandatory configuration.
* **cache name**=”com.tutorial.hibernate.Student” defines the cache region for Student class.
* **eternal**=false means cache will be evicted/expire based on time
* **timeToIdleSeconds**- specifies the expiry time in seconds the object is not accessed.
* **timeToLiveSeconds**- specify the maximum time in seconds the object is will be available in cache
* **maxElementsInMemory**- is the possible maximum elements of this class in cache.
* **overflowToDisk**= enable or disable writing to disk when memory overflows.

So the above Student Cache configurations says to expire the cache if  an element is not accessed for 3 mins OR number of elements exceeds 500 or element is there in a cache for 4 mins.

### ****Enable Caching at a class level****

To do so add  <cache usage="read-only" />   in class mapping (hbm) . Usage attribute defines the concurrency strategy

### ****Add Required jar files****

Copy the below three jar files from the optional/ehcache directory of hibernate download and add it in a build path.(refer Chapter 3 for more details).

1. hibernate-ehcache-4.3.7.Final.jar
2. slf4j-api-1.6.1.jar
3. ehcache-core-2.4.3.jar

## ****Example – Second Level Cache****

Let’s play with ehcache in our example.

**Student Entity**

@Entity

@Table(name = "STUDENT")

@Cache(usage = CacheConcurrencyStrategy.***READ\_ONLY***)

**public** **class** Student {

@Id

**private** **int** id;

@Column(name = "NAME")

**private** String name;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

**Hibernate.cfg.xml**

<property name=*"hibernate.cache.region.factory\_class"*>

org.hibernate.cache.ehcache.EhCacheRegionFactory

</property>

<property name=*"net.sf.ehcache.configurationResourceName"*>/Myehcache.xml</property>

**Myehcahce.xml**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<ehcache xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:noNamespaceSchemaLocation=*"ehcache.xsd"*>

<diskStore path=*"java.io.tmpdir/ehcache"* />

<defaultCache maxElementsInMemory=*"500"*

eternal=*"false"*

timeToIdleSeconds=*"220"*

timeToLiveSeconds=*"320"*

overflowToDisk=*"true"* />

<cache name=*"in.spring4buddies.application.caching.Student"*

maxElementsInMemory=*"500"*

eternal=*"false"*

timeToIdleSeconds=*"1800"*

timeToLiveSeconds=*"2400"*

overflowToDisk=*"false"* />

</ehcache>

**Test Program**

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**import** org.hibernate.cfg.Configuration;

**public** **class** FirstLevelCacheMain {

**public** **static** **void** main(String args[]) {

Configuration cfg = **new** Configuration().configure();

SessionFactory factory = cfg.~~buildSessionFactory~~();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Student student = **new** Student();

student.setName("Student A");

**int** id = (Integer) session.save(student);

tx.commit();

session.close();

session = factory.openSession();

Student student1 = (Student) session.get(Student.**class**, id);

Student student2 = (Student) session.get(Student.**class**, id);

session.close();

session = factory.openSession();

Student student3 = (Student) session.get(Student.**class**, id);

session.close();

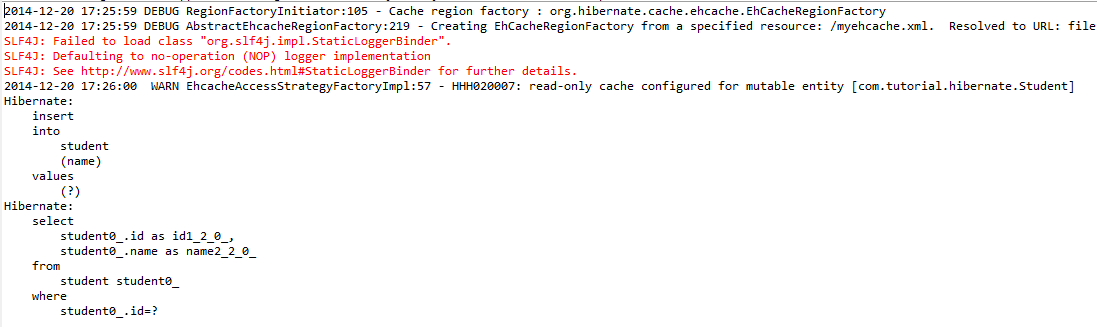
factory.close();

}

}

**Run Test Program**

Only one select statement is fired even though we fetched it three times (in different session also)

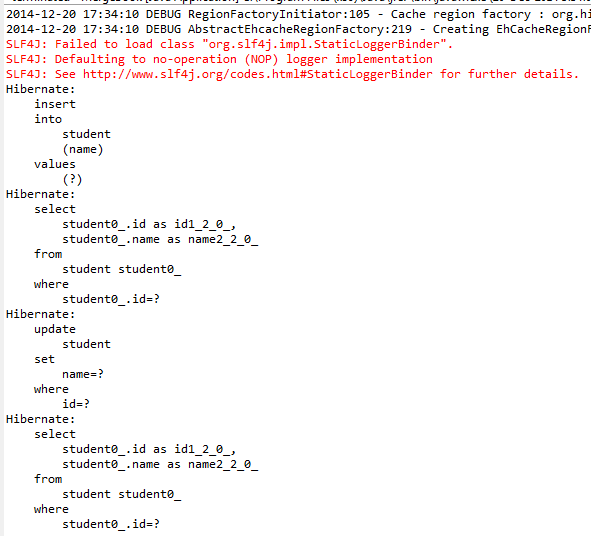


We configured **read-only** concurrency strategy so  if you will try to update the student like below, you will get an error. To solve this you can configure read-write  strategy

    st1.setName("XYZ");

    session.flush();

By doing so , you will see an extra update statement (we are updating student) and extra select because data has been changed in database



## ****Caching Associations****

If a parent entity is cached, it will not cache the child objects. We need to turn on the caching of child entities.

Fetching associations in different session instead of having a parent object cached will fire select query to fetch the association.

If Student entity has a collections of books borrowed then fetching books like below will fire select statement on books for second session . This will be fired once per session because second time it will be available through the first level cache.

**public** **class** CacheAssociationMain {

**public** **static** **void** main(String args[]) {

Configuration cfg = **new** Configuration().configure();

SessionFactory factory = cfg.~~buildSessionFactory~~();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Book book1 = **new** Book();

book1.setName("Book1");

Book book2 = **new** Book();

book2.setName("Book2");

Set<Book> books = **new** HashSet<Book>();

books.add(book1);

books.add(book2);

session.save(book1);

session.save(book2);

Student student = **new** Student();

student.setName("Student A");

student.setBooksBorrowed(books);

**int** id = (Integer) session.save(student);

tx.commit();

session.close();

session = factory.openSession();

Student st1 = (Student) session.get(Student.**class**, id);

Student st2 = (Student) session.get(Student.**class**, id);

System.***out***.println(st1.getBooksBorrowed());

System.***out***.println(st2.getBooksBorrowed());

session.close();

session = factory.openSession();

Student st3 = (Student) session.get(Student.**class**, id);

System.***out***.println(st3.getBooksBorrowed());

Student st4 = (Student) session.get(Student.**class**, id);

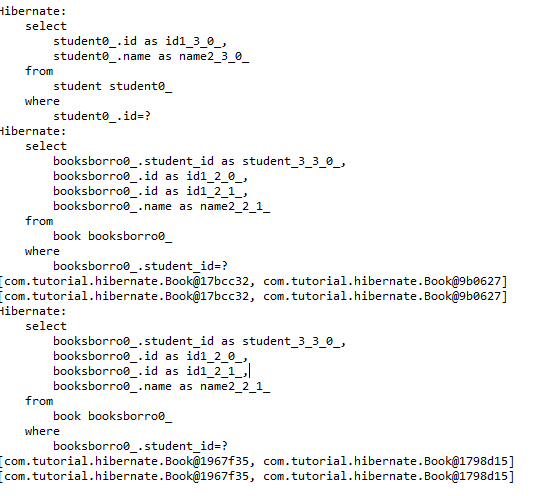
System.***out***.println(st4.getBooksBorrowed());

session.close();

factory.close();

}

}



If the collection is an entity, then object is cached with identifier (will have specific cache region with class name together with property) and if it is a value type then values are cached.

We need to enable caching at a Book level and also at a collection level

If we set caching in Book class as well then only one select for the book will be fired.

Adding <cache usage=”read-only”/ > at a Book level and in <set> will fire one select query on Book.

Need to add cache configuration in myehcache.xml as well.

## ****Query Level Cache****

By default the data of  HQL queries are not cached. If application fires HQL statement in the same or different session, multiple times, then data of the query can be cached.

To do so we need to enable, we need to first set **hibernate.cache.use\_query\_cache="true**" in hibernate.cfg.xml file . Setting up this property will create additional two required cache regions. One to store the data and one to hold the last updated timestamps.

Once done, we need to explicitly call setCacheable(true) on Query object before execution

By default Query results are cached with the region name “**org.hibernate.cache.QueryCache”**and we can change it by calling setCacheRegion(“Region Name”) on Query Object.