Hibernate Performance Tuning

by Ming Jiang · Jun. 10, 09 · Performance Zone

Hibernate is a powerful, high performance object/relational persistence and query service. Hibernate lets you develop persistent classes following object-oriented idiom - including association, inheritance, polymorphism, composition, and collections. Hibernate allows you to express queries in its own portable SQL extension (HQL), as well as in native SQL, or with an object-oriented Criteria and Example API.

Quintessential to using any ORM framework like hibernate is to know how to leverage the various performance tuning methods supported by the framework.

In this volume Wings Jiang discusses three performance tuning strategies for hibernate:

1. **SQL Optimization**
2. **Session Management**
3. **Data Caching**

SQL Optimization

When using Hibernate in your application, you already have been coding HQL (Hibernate Query Language) somewhere. For example, "**from User user where user.name = 'John'**". If issuing your SQL statement like this, Hibernate cannot use the SQL cache implemented by database because name of the user, in most scenarios, is extremely distinct. On the contrary, while using placeholder to achieve this, like "**from User user where user.name =?**" will be cached by the Database to fulfill the performance improvement. You can also set some Hibernate properties to improve performance, such as setting the number of records retrieved while fetching records via configuring property **hibernate.jdbc.fetch_size**, setting the batch size when committing the batch processing via configuring property **hibernate.jdbc.batch_size** and switching off the SQL output via setting property **hibernate.show_sql** to false in product environments. In addition, the performance tuning of your target Database is also significant, like SQL clauses tuning, reasonable indexes, delicate table structures, data partitions etc.

Session Management

Undoubtedly, Session is the pith of Hibernate. It manages the Database related attributes, such as JDBC connections, data entities' states. Managing the Session efficiently is the key to getting high performance in enterprise applications. One of the many commonly used and equally elegant approaches to session management in hibernate is to use ThreadLocal. Threadlocal will create a local copy of session for every thread. Thus

In hibernate is to use ThreadLocal. ThreadLocal will create a local copy of session for every thread. Thus synchronization problems are averted, when objects are put in the Threadlocal, . To understand how ThreadLocal variables are used in Java, refer to Sun Java Documentation at http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ThreadLocal.html

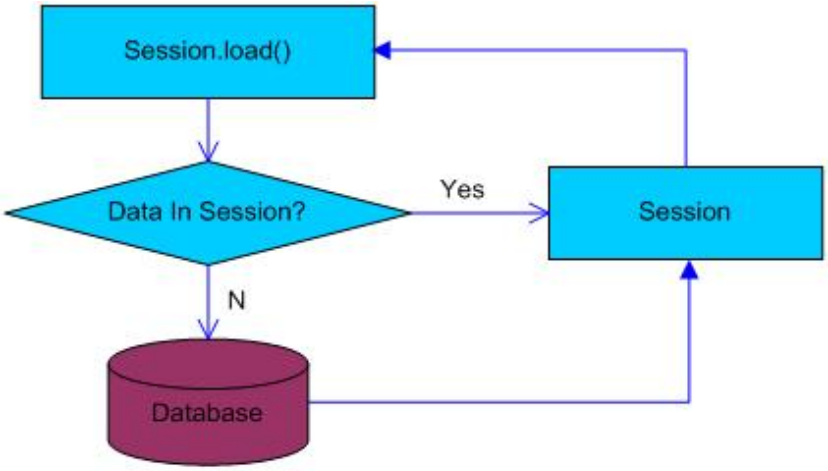
Data Caching

Before accomplishing any data caching, it is essential to set the property **hibernate.cache.user_query_cache = true**.
There are three kinds of commonly used Caching Strategies in Hibernate:
Using cache based on Session level (aka Transaction layer level cache). This is also called first-level cache.

Using cache based on SessionFactory level (Application layer level cache). This is also called second-level cache.

Using cluster cache which is employed in distributed application (in different JVMs). In fact, some techniques, like loading data by id, lazy initialization which betokens loading appropriate data in proper time rather than obtaining a titanic number of useless records, which are fairly useless in the subsequent operations are consummated via data caching.

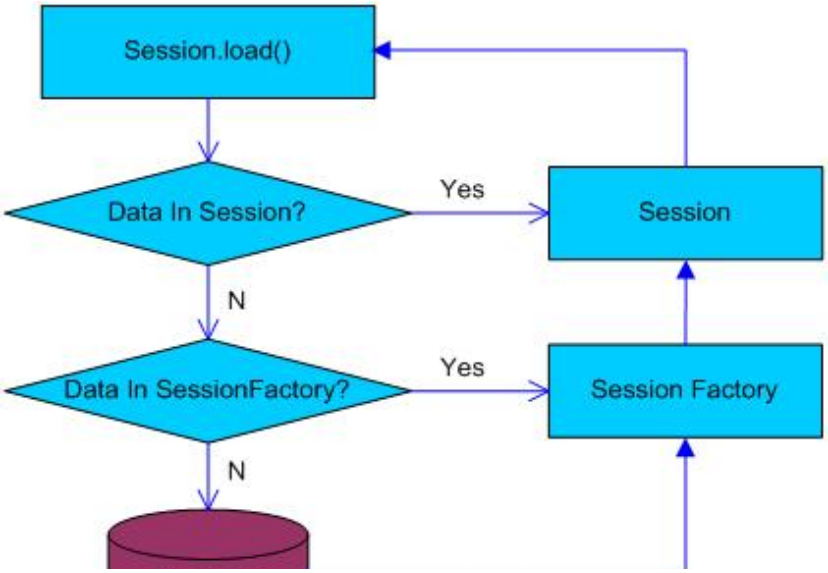
First Level Cache (aka Transaction layer level cache)



Fetching an object from database always has a cost associated with it. This can be offset by storing the entities in hibernate session. Next time the entities are required, they are fetched from the session, rather than fetching from the database.

To clear an object from the session use: **session.evict(object)**. To clear all the objects from the session use **session.clear()**.

Second Level Cache (aka Application layer level cache)



In this approach, if an object is not found in session, it is searched for in the session factory before querying the database for the object.
If an object is indeed fetched from database, the selected data should be put in session cache. This would improve the performance when the object is required next time. To remove an entity from session factory use the various overloaded implementations of **evict()** method of SessionFactory.

In fact, Hibernate lets you tailor your own

In fact, Hibernate lets you tailor your own caching implementation by specifying the name of a class that implements **org.hibernate.cache.CacheProvider** using the property **hibernate.cache.provider_class**. But it is recommended to employ a few built-in integrations with open source cache providers (listed below).

CACHE	TYPE	CLUSTER SAFE	QUERY CACHE SUPPORTED
HASHTABLE	MEMORY	NO	YES
EHCACHE	MEMORY, DISK	NO	YES
OSCACHE	MEMORY, DISK	NO	YES
SWARMCACHE	CLUSTERED	YES (CLUSTERED INVALIDATION)	NO
JBOSS TREECACHE	CLUSTERED	YES (REPLICATION)	YES
TERRACOTA	CLUSTERED	YES	YES

In order to use second level caching, developers have to append some configurations in **hibernate.cfg.xml** (for example, using EHCache here).

```
<hibernate-configuration>    <session-factory>        <property name="hibernate.cache.provi
```

In addition, developers also need to create a cache specific configuration file (Example: **ehcache.xml** for EHCache).

```
<ehcache>    <diskStore path="java.io.tmpdir"/>      (1)    <defaultCache        maxElement
```

1. **diskStore** : Sets the path to the directory where cache .data files are created. The following properties are translated:
   a.user.home - User's home directory
   b.user.dir - User's current working directory
   c.java.io.tmpdir (Default temp file path)
2. **maxElementsInMemory** : Sets the maximum number of objects that will be created in memory.
3. **eternal** : Sets whether elements are eternal. If eternal, timeouts are ignored and the element is never expired.

4. **timeToIdleSeconds** : Sets the time to idle for an element before it expires. Is only used if the element is not eternal. Idle time is now - last accessed time.
5. **timeToLiveSeconds** : Sets the time to live for an element before it expires. Is only used if the element is not eternal. TTL is now - creation time
6. **overflowToDisk** : Sets whether elements can overflow to disk when the in-memory cache has reached the maxInMemory limit.

Finally the cache concurrency strategy has to be specified in mapping files. For example, the following code fragment shows how to configure your cache strategy.

```
<class name="org.jmatrix.user.User" table="USER"><cache usage=”read-write” />…<set name="ad
```

Cache Concurrency Strategies

There are four kinds of built-in cache concurrency strategies provided by Hibernate. Chosing a right concurrency strategy for your hibernate implementation is the key to cache performance optimization. Besides to ensure data consistency and transaction integrity it is indispensable to master these strategies.

read-only
If your application needs to read but never modify instances of a persistent class, a read-only cache may be used. This is the simplest and best performing strategy. It's even perfectly safe for use in a cluster.

nonstrict-read-write
If the application only occasionally needs to update data (For example, if it is extremely unlikely that two transactions would try to update the same item simultaneously) and strict transaction isolation is not required, a nonstrict-read-write cache might be appropriate.

read-write
If the application needs to update data, a read-write cache might be appropriate. This cache strategy should never be used if serializable transaction isolation level is required.

transactional
If the application seldom needs to update data and at the same time, application also needs to avoid “dirty read” and “repeatable read”, this kind of concurrency strategy can be employed. The transactional cache strategy provides support for fully transactional cache providers such as JBoss TreeCache.

The following table lists cache concurrency strategy supported by various cache providers.

CACHE	READ-ONLY	NONSTRICT-READ-WRITE	READ-WRITE	TRANSACTIONAL
HASHTABLE	YES	YES	YES	N/A

EHCACHE	**YES**	**YES**	**YES**	**N/A**
OSCACHE	**YES**	**YES**	**YES**	**N/A**
SWARMCACHE	**YES**	**YES**	**N/A**	**N/A**
JBOSS TREECACHE	**YES**	**N/A**	**N/A**	**YES**

Cluster Cache (in different JVMs)

Hibernate also supports cluster caching in disparate JVMs. At present, both SwarmCache and JBoss TreeCache support cluster caching across multiple JVMs. In some situations, especially at the level of enterprise, certain application has to support the concurrency accessing of thousands of users, at that time, cluster cache can help you because the cluster can provide failover and load balancing which improve the performance of application.

Points to Note

When employing one of the four cache strategies above, pay close attention to the following situation:

1. Data cached almost immutable
   If data you want to cache is almost constant, you can use data caching which can improve the performance of the application. On the contrary, if the caching data are quiet volatile, Hibernate have to maintain and update the caching over time which extremely leads to performance hit.
2. Data sizes in reasonable range

   If the size of data you is caching is massive, Hibernate will occupy the most memories of system, which causes the long waiting time of the whole application.
3. Low frequency of data updating
   If data you are caching needs to be modified frequently, Hibernate have to take an array of time to update and modify the data in caching, which impacts the performance of the application as well.
4. High frequency of data querying
   If data you are caching is steady, which means that most of the operations are querying, searching, no updating and modifying, making the most use of caching will be affording huge performance improvement.
5. None crucial data
   Because of existing some incongruities when keeping the data in caching, so if the data you are caching is fairly crucial, do not use caching. By contrast, if the data in caching is insignificant, just use it without any vacillation.

Summary

Actually, after employing SQL Optimization, Session Management, Data Caching, we will obtain great battalions of performance gains, which make applications achieve acceptable waiting time for the final customers.

External Links for Further Study

http://www.hibernate.org/hib_docs/reference/en/html/performance.html
http://blogs.jboss.com/blog/acoliver/2006/01/23/Hibernate_EJB3_Tuning.txt

About Author

I am Wings Jiang from BCM China. I have mainly focused on J2EE technologies in recent years and worked in several projects involving Struts/Tapestry, Spring, Hibernate, WebLogic, Websphere, Oracle, DB2 etc. I have experience in design and code of several Java applications. Hibernate performance is one of the areas I pay close heed to in my current working.

Like This Article? Read More From DZone

How Does Hibernate Query Cache Work?

Hibernate Caching With Hazelcast: JPA Caching Basics

Hibernate Facts: Favoring bidirectional Set(s) vs List(s)

Free DZone Refcard
Scalability & High Availability

Topics: JAVA , HIBERNATE , PERFORMANCE , CACHING , SQL