Consejos para un mejor rendimiento de Hibernate

Bases de Datos - 38210

Master Oficial en Desarrollo de Aplicaciones y Servicios Web

Miquel Esplà Gomis
Armando Suarez Cueto

Objetivo

- Describir algunas opciones y herramientas de hibernate que pueden ayudar a obtener un mejor rendimiento:
  - caché
  - fetching
  - batching

Estadísticas de rendimiento de hibernate

- Antes de empezar, aprenderemos a activar las estadísticas de rendimiento
- Estas estadísticas proporcionan tiempo y cantidad de determinadas operaciones de JDBC
- Las estadísticas se pueden activar con la siguiente opción en el fichero de configuración

```
<property name="hibernate.generate_statistics">true</property>
```

Estadísticas de rendimiento de hibernate

- Al cerrar la sesión obtendremos un mensaje como el que sigue:

```
INFO: Session Metrics {
    318965 nanoseconds spent acquiring 1 JDBC connections;
    260935 nanoseconds spent releasing 1 JDBC connections;
    9872101 nanoseconds spent preparing 254 JDBC statements;
    163852115 nanoseconds spent executing 254 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
   ...
```

Estadísticas de rendimiento de hibernate

```
    ...
    60425579 nanoseconds spent performing 470 L2C puts;
    874785 nanoseconds spent performing 16 L2C hits;
    1996686 nanoseconds spent performing 13 L2C misses;
    267609059 nanoseconds spent executing 4 flushes (flushing
a total of 269 entities and 68 collections);
    56055 nanoseconds spent executing 1 partial-flushes
(flushing a total of 0 entities and 0 collections)
}
```

Caché

Enlace al manual

Tres tipos de caché

- Caché de primer nivel: nativo, relativo a una Session
- Caché de segundo nivel: implementación externa, compartida entre todas las Session
- Caché de consultas: parte de la caché de segundo nivel

Caché de primer nivel

- **Activada** por defecto
- Asociada a una **sesión**: guarda todos los objetos utilizados en una sesión y se borra al cerrarla
- Capa intermedia entre la BDR y la aplicación: la aplicación únicamente escribe en la caché y Hibernate se encarga de realizar escrituras y lecturas de la BDR según convenga
- No tenemos demasiado control sobre ella, aunque podemos realizar algunas operaciones, tales como limpiarla (`session.clear()`), cargar una entidad de la BDR (`session.get()`) o borrarla (`session.evict()`)

Caché de segundo nivel

- Desactivada por defecto
- Diferentes implementaciones posibles (veremos EHCache)
- Asociada a la `SessionFactory`: compartida entre todas las `Session` de una misma BDR
- Las entidades que se guardarán en caché deben implementar la interfaz `Serializable`
- Ejemplo de activación y uso de esta caché

Orden de acceso a caché y BDR

- Cuando recuperamos una entidad persistida (`get()` o `load()`):
  a. caché de primer nivel; si no la encontramos,
  b. caché de segundo nivel; si no la encontramos,
  c. base de datos; al leerla se escribe en las dos cachés para futuros usos

- Cuando persistimos una entidad:
  a. se escribe en las dos cachés
  b. cuando se cierra o limpia la sesión (o cuando la caché de primer nivel se llena) se escribe en la base de datos

Usemos bien la caché de segundo nivel

- Las colecciones dentro de una entidad no se almacenan en caché si no se explicita

```
...
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@OneToMany
private Collection<Room> rooms;
...
```

Usemos bien la caché de segundo nivel

- Las consultas HQL son compatibles con la caché de segundo nivel, las SQL no:
  - una consulta HQL de inserción/actualización/borrado de una entidad concreta **invalidará las entidades afectadas** de la caché
  - una consulta SQL de actualización **invalidará toda la caché** (se puede especificar con (`addSynchronizedEntityClass(Foo.class)`)

Usemos bien la caché de segundo nivel

- Alerta con el *overhead* de las entidades que cambian muy a menudo: puede ser contraproducente almacenarlas en la caché
- Elijamos bien la estrategia de concurrencia sabiendo cuales son más baratas:
  - *READ_ONLY*
  - *NONSTRICT_READ_WRITE*
  - *READ_WRITE*
  - *TRANSACTIONAL*

```
...
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
...
```

Usemos bien la caché de segundo nivel

- Si algún dato de una entidad se genera por parte de la base de datos, ésta no se podrá guardar en la caché la primera vez que se defina:
  - primero deberá escribirse en la base de datos para obtener los campos que faltan
  - a partir de entonces, cuando vuelva a usarse sí que se podrá guardar en la caché
- Un caso típico son los ID generados con el generador *identity*

Caché de consultas

- Desactivada por defecto
- Almacena tuplas consulta-parámetros y la clave de las entidades devueltas: es importante que las entidades estén *cacheadas*!
- No recomendable para consultas que:
    - cambian a menudo de parámetros
    - devuelven valores entidades que no están en caché o se actualizan a menudo

Caché de consultas

Se activa añadiendo al fichero de configuración:

```
<!-- activando la caché de consultas -->
<property name="hibernate.cache.use_query_cache">true</property>
```

Las queries se guardan en la región de caché de segundo nivel:

```
<cache name="org.hibernate.cache.internal.StandardQueryCache" ...
```

Caché de consultas

Especificar que una consulta debe ser guardada en caché:

```
org.hibernate.query.Query query = session.createQuery(
    "select p from Person p where p.name like :name" )
.setParameter( "name", "J%")
.setCacheable(true)
.setCacheRegion( "query.cache.person")
.list();
```

Carga de entidades relacionadas

Enlace al manual

Leyendo una entidad

- Cuando se instancia una entidad en Hibernate, se cargan sus datos automáticamente desde la base de datos (o la caché)
- Se puede especificar la forma de cargar los datos relacionados con *fetch* = ; por ejemplo:

```
@ManyToOne(fetch = FetchType.EAGER)
@Fetch(FetchMode.SELECT)
private Subject subject;
```

Tipos de *fetching*

- Hay dos tipos de fetching:
  - ***`FetchType.EAGER`***: carga los datos en el momento de instanciar el objeto
  - ***`FetchType.LAZY`***: carga los datos cuando se intente acceder a ellos a través del método getter
- Por defecto: @OneToOne y @ManyToOne son Eager, mientras que @OneToMany y @ManyToMany son Lazy

Estrategias de *fetching*

- Hay tres estrategias de fetching:
  - ***`FetchMode.SELECT`***: por defecto; realiza una consulta para la entidad y tantas como sean necesarias para los elementos de la colección (**problema N+1**)
  - ***`FetchMode.JOIN`***: realiza una un *outer join* para cargar los datos, tanto de la entidad como los de la relación. **No recomendada**, ya que produce duplicados y sólo funciona con las consultas criteria.
  - ***`FetchMode.SUBSELECT`***: realiza una consulta para obtener la entidad y una sub-consulta para el resto de elementos de la colección. Sólo disponible para relaciones XtoMany

Qué es el problema N+1?

- Consultas que recuperan entidades con relaciones y se requieren una consulta para recuperar estas entidades, más *N* consultas para recuperar cada una de las asociaciones
- Afecta a consultas que recuperan entidades con relaciones. Puede darse en relaciones:
  - **XtoMany** cuando se utiliza una estrategia de *fetching* ***SELECT***
  - **XtoOne** cuando el tipo de *fetching* es ***EAGER***
  - **Cualquiera** de ellas cuando el tipo de *fetching* es ***LAZY*** y se acceden a los elementos de la relación en nuestro código después de haber hecho la carga
- **Ejemplo**: en vuestra práctica, imaginad que recuperamos todas las conexiones y queremos imprimir cada uno de los IDs del usuario que se ha conectado.

Solución

- En el caso de las relaciones XtoMany se puede optar por una estrategia de *fetching* de tipo *SUBSELECT* que realice solo dos consultas (una para las entidades y una para la relación)

```
@Fetch(FetchMode.SUBSELECT)
```

- Utilizar `JOIN FETCH` cuando hagamos las consultas para forzar a cargar las asociaciones

```
session.createQuery("SELECT p FROM Persona p LEFT JOIN FETCH p.direccion")
```

Batching

Enlace al manual

Para qué sirve el *batching*?

- Múltiples inserciones y borrados: operaciones caras si se ejecutan de una en una
- Las opciones de *batching* permiten realizar estas operaciones en grupo de forma automática, mejorando el rendimiento

```
<property name = "hibernate.jdbc.batch_size">50</property>
<property name = "hibernate.order_inserts">true</property>
<property name = "hibernate.order_updates">true</property>
```

Para qué sirve el *batching*?

- `hibernate.jdbc.batch_size`: número de operaciones que se agruparán en un sólo *batch*
- `hibernate.order_inserts` y `hibernate.order_updates`: opción que permite a hibernate decidir el orden óptimo de las operaciones de inserción y actualización
- las operaciones de borrado se pueden enviar a la base de datos en *batches*, aunque no pueden ser re-ordenadas y debe ser el programador quien decida su orden