Вопросы и ответы на собеседование по Hibernate

(16-30)

16). Как настроить кэш второго уровня в Hibernate с помощью EHCache?

EHCache является лучшим выбором для организации кэширования второго уровня в Hibernate. Для настройки второго уровня кэширования в Hibernate требуется выполнить несколько шагов.

- **Добавить зависимость hibernate-ehcache в проект**.

```
<dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-ehcache</artifactId>
      <version>4.3.5.Final</version>
</dependency>
```

- **Добавить несколько записей в конфигурационный файл Hibernate**.

```
<property name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

<!-- For singleton factory -->
<!-- <property
name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory</property>
-->

<!-- enable second level cache and query cache -->
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property name="net.sf.ehcache.configurationResourceName">/myehcache.xml</property>
```

16). Продолжение

- **Создать файл конфигурации EHCache.**

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="true"
   monitoring="autodetect" dynamicConfig="true">

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

   <defaultCache maxEntriesLocalHeap="10000" eternal="false"
      timeToIdleSeconds="120" timeToLiveSeconds="120" diskSpoolBufferSizeMB="30"
      maxEntriesLocalDisk="10000000" diskExpiryThreadIntervalSeconds="120"
      memoryStoreEvictionPolicy="LRU" statistics="true">
      <persistence strategy="localTempSwap" />
   </defaultCache>

   <cache name="employee" maxEntriesLocalHeap="10000" eternal="false"
      timeToIdleSeconds="5" timeToLiveSeconds="10">
      <persistence strategy="localTempSwap" />
   </cache>

   <cache name="org.hibernate.cache.internal.StandardQueryCache"
      maxEntriesLocalHeap="5" eternal="false" timeToLiveSeconds="120">
      <persistence strategy="localTempSwap" />
   </cache>

   <cache name="org.hibernate.cache.spi.UpdateTimestampsCache"
      maxEntriesLocalHeap="5000" eternal="true">
      <persistence strategy="localTempSwap" />
   </cache>
</ehcache>
```

16). Продолжение

- **Использовать аннотацию @Cache и указание настройки стратегии кэширование над entity bean**.

```
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Table(name = "ADDRESS")
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY, region="employee")
public class Address {

}
```

17). Какие существуют различные состояния у entity bean?

1. **Transient**: состояние, при котором объект **никогда не был** связан с какой-либо сессией и не является персистентностью. Этот объект находится во временном состоянии. Объект в этом состоянии может стать персистентным при вызове метода save(), persist() или saveOrUpdate(). Объект персистентности может перейти в transient состоянии после вызова метода delete().
2. **Persistent**: когда объект **связан с** уникальной сессией он находится в состоянии persistent (персистентности). Любой экземпляр, возвращаемый методами get() или load() находится в состоянии persistent.
3. **Detached**: если объект **был** персистентным, но сейчас не связан с какой-либо сессией, то он находится в отвязанном (detached) состоянии. Такой объект можно сделать персистентным используя методы update(), saveOrUpdate(), lock() или replicate(). Состояния transient или detached так же могут перейти в состояние persistent как новый объект персистентности после вызова метода merge().

18). Как используется вызов метода Hibernate Session merge()?

Hibernate merge() может быть использован **для обновления** существующих значений, однако этот метод создает копию из переданного объекта сущности и возвращает его. Возвращаемый объект является частью контекста персистентности и отслеживает любые изменения, а переданный объект не отслеживается.

19). В чем разница между Hibernate save(), saveOrUpdate() и persist()?

Hibernate **save**() используется для сохранения сущности в базу данных. Проблема с использованием метода save() заключается в том, что он может быть вызван без транзакции. А следовательно если у нас имеется отображение нескольких объектов, то только первичный объект будет сохранен и мы получим несогласованные данные. Также save() немедленно возвращает сгенерированный идентификатор.

Hibernate **persist**() аналогичен save() с транзакцией. persist() не возвращает сгенерированный идентификатор сразу.

Hibernate **saveOrUpdate**() использует запрос для вставки или обновления, основываясь на предоставленных данных. Если данные уже присутствуют в базе данных, то **будет выполнен запрос обновления**. Метод saveOrUpdate() можно применять без транзакции, но это может привести к аналогичным проблемам, как и в случае с методом save().

20). Что произойдет, если будет отсутствовать конструктор без аргументов у Entity Bean?

Hibernate использует **рефлексию** для создания экземпляров Entity бинов при вызове методов **get**() или **load**(). Для этого используется метод **Class.newInstance()**, который **требует** наличия конструктора без параметров. Поэтому, в случае его отсутствия, вы получите ошибку **HibernateException**.

21). В чем разница между sorted collection и ordered collection? Какая из них лучше?

При использовании алгоритмов сортировки из Collection API для сортировки коллекции используется сортированный список (**sorted list**). Для маленьких коллекций это не приводит к излишнему расходу ресурсов, но на больших коллекциях это может привести к потере производительности и ошибкам OutOfMemory. Так же entity бины должны реализовывать интерфейс **Comparable** или **Comparator** для работы с сортированными коллекциями.
При использовании фреймворка Hibernate для загрузки данных из базы данных мы можем применить **Criteria API** и команду **order by** для получения отсортированного списка (**ordered list**). Ordered list является **лучшим** выбором к sorted list, т.к. он использует сортировку на уровне базы данных. Она быстрее и не может привести к утечке памяти. Пример запроса к БД для получения ordered list:

```
List<Employee> empList = session.createCriteria(Employee.class)
                        .addOrder(Order.desc("id")).list();
```

22). Какие типы коллекций в Hibernate вы знаете?

1. **Set**: Коллекция, представляющая собой множество уникальных элементов. В Hibernate, это часто используется для представления множественных связей "многие-ко-многим".
2. **List**: Упорядоченная коллекция элементов, которая может содержать дубликаты. Она часто используется для представления списков данных, например, список заказанных товаров в заказе.
3. **Map**: Коллекция, которая представляет собой отображение ключ-значение. Может быть использована для представления связей "один-ко-многим" или для хранения пар ключ-значение.
4. **Bag**: Это аналог List, но без гарантии порядка элементов. Она менее эффективна с точки зрения производительности, чем List или Set.
5. **Array**: Используется для хранения данных в виде массива.
6. **Primitive Collections**: Коллекции, специализированные для хранения примитивных типов данных (например, int, double).
7. **Sorted Collections**: Коллекции, которые автоматически сортируют элементы в заданном порядке (например, отсортированный Set или List).
8. **Identifier Bag**: Похожа на Bag, но также содержит идентификаторы объектов, что может быть полезно при выполнении дополнительных запросов.

23). Как реализованы Join'ы Hibernate?

Существует несколько способов реализовать связи в Hibernate.

- Использовать ассоциации, такие как **one-to-one**, **one-to-many**, **many-to-many**.
- Использовать в HQL запросе команду JOIN. Существует другая форма <<join fetch>>, позволяющая загружать данные немедленно (**не lazy**).
- Использовать чистый SQL запрос с командой join.

24). Почему мы не должны делать Entity class как final?

Hibernate использует **прокси классы** для ленивой загрузки данных (т.е. по необходимости, а не сразу). Это достигается с помощью расширения entity bean и, следовательно, если бы он был final, то это было бы невозможно. Ленивая загрузка данных во многих случаях повышает производительность, а следовательно важна.

25). Что вы знаете о HQL и какие его преимущества?

HQL (Hibernate Query Language) - это язык запросов, разработанный для выполнения запросов к данным в Hibernate, фреймворке объектно-реляционного отображения (ORM) для Java. HQL аналогичен **SQL** (Structured Query Language), но использует объекты и свойства Java-классов вместо таблиц и столбцов базы данных. Вот некоторые основные аспекты HQL и его преимущества:

1. **Объектно-Ориентированный Подход**: HQL позволяет разработчикам создавать запросы, используя объектную модель данных, а не таблицы и столбцы. Это делает код более читаемым и обеспечивает более натуральное взаимодействие с данными.
2. **Портабельность**: HQL позволяет писать запросы, которые могут быть перенесены между различными базами данных без изменения кода. Hibernate обрабатывает различия в SQL-диалектах различных СУБД.
3. **Использование Отображений Объектов**: HQL позволяет работать с объектами и их ассоциациями, а не требует прямого написания SQL-кода. Например, вы можете обращаться к свойствам объектов в HQL-запросах, а не к столбцам базы данных.
4. **Поддержка Явных и Неявных Join'ов**: HQL поддерживает как явные, так и неявные (автоматически генерируемые) объединения, что делает запросы более гибкими.
5. **Функции и Агрегатные Функции**: HQL предоставляет множество встроенных функций и агрегатных функций, аналогичных SQL, которые можно использовать в запросах.
6. **Поддержка Параметров**: Вы можете передавать параметры в HQL-запросы, что делает их параметризованными и улучшает безопасность и производительность.
7. **Легкая Интеграция с Hibernate**: HQL является частью Hibernate и интегрируется непосредственно с маппингом объектов Hibernate, что облегчает работу с данными и уменьшает необходимость писать сложный SQL-код.

26). Что такое Query Cache в Hibernate?

Hibernate реализует область кэша для запросов **resultset**, который тесно взаимодействует с кэшем второго уровня Hibernate. Для подключения этой дополнительной функции требуется несколько дополнительных шагов в коде. Query Cache полезны только для часто выполняющихся запросов с повторяющимися параметрами. Для начала необходимо добавить эту запись в файле конфигурации Hibernate:

```
<property name="hibernate.cache.use_query_cache">true</property>
```

Уже внутри кода приложения для запроса применяется метод **setCacheable**(**true**), как показано ниже:

```
Query query = session.createQuery("from Employee");
query.setCacheable(true);
query.setCacheRegion("ALL_EMP");
```

27). Можем ли мы выполнить нативный запрос SQL (sql native) в Hibernate?

С помощью использования SQLQuery можно выполнять чистый запрос SQL. В общем случае это не рекомендуется, т.к. вы потеряете все преимущества HQL (ассоциации, кэширование). Выполнить можно примерно так:

```
Transaction tx = session.beginTransaction();

SQLQuery query = session.createSQLQuery("select emp_id, emp_name, emp_salary from Employee");

List<Object[]> rows = query.list();
for(Object[] row : rows){
    Employee emp = new Employee();
    emp.setId(Long.parseLong(row[0].toString()));
    emp.setName(row[1].toString());
    emp.setSalary(Double.parseDouble(row[2].toString()));
    System.out.println(emp);
}
```

28). Назовите преимущества поддержки нативного sql в Hibernate.

Использование нативного SQL может быть необходимо при выполнении запросов к некоторым базам данных, которые могут не поддерживаться в Hibernate. Примером может служить некоторые специфичные запросы и «фишки» при работе с БД от Oracle.

29). Расскажите о преимуществах использования Hibernate Criteria API.

Hibernate **Criteria API** является более объектно-ориентированным для запросов, которые получают результат из базы данных. Для операций update, delete или других DDL манипуляций использовать Criteria API **нельзя**. Критерии используются **только** для выборки из базы данных в более объектно-ориентированном стиле.
Вот некоторые области применения Criteria API:

- Criteria API поддерживает проекцию, которую мы можем использовать для агрегатных функций вроде **sum(), min(), max()** и т.д.
- Criteria API может использовать **ProjectionList** для извлечения данных только из выбранных колонок.
- Criteria API может быть использована для **join** запросов с помощью соединения нескольких таблиц, используя методы **createAlias()**, **setFetchMode()** и **setProjection()**.
- Criteria API поддерживает выборку результатов согласно условиям (ограничениям). Для этого используется метод **add()** с помощью которого добавляются ограничения (**Restrictions**).
- Criteria API позволяет добавлять порядок (сортировку) к результату с помощью метода **addOrder()**.

30). Как логировать созданные Hibernate SQL запросы в лог-файлы?

Для логирования SQL запросов, созданных Hibernate, вы можете использовать инструменты и настройки логирования, предоставляемые Hibernate, в сочетании с популярными библиотеками логирования, такими как Log4j, SLF4J или Java Util Logging. Вот как это можно сделать:

- **Выбор Библиотеки Логирования**: Выберите библиотеку логирования, которую вы хотите использовать, и добавьте ее зависимость в ваш проект. Например, вы можете использовать Log4j2, добавив его зависимость в файл **pom.xml** вашего проекта

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.x.x</version> <!-- Замените на актуальную версию Log4j2 -->
</dependency>
```

- **Настройка Логирования Hibernate**: В файле конфигурации Hibernate (например, hibernate.cfg.xml или через Java классы конфигурации) добавьте следующие настройки:

```
<property name="hibernate.show_sql">true</property>
<property name="hibernate.format_sql">true</property>
```

hibernate.show_sql: Установите это свойство в true, чтобы Hibernate показывал созданные SQL запросы в консоли.
hibernate.format_sql: Установите это свойство в true, чтобы запросы были красиво форматированы.

30).
Продолжение

- **Настройка Логирования в Log4j (или Другой Логгер)**: Создайте файл конфигурации для вашего логгера (например, log4j2.xml для Log4j2) и настройте логирование Hibernate. В примере для Log4j2 это может выглядеть так:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="org.hibernate.SQL" level="DEBUG"/>
    <Logger name="org.hibernate.type.descriptor.sql.BasicBinder" level="TRACE"/>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

Этот файл конфигурации настраивает логирование Hibernate SQL запросов на уровне DEBUG и выше.

- **Запустите Приложение**: Теперь, при запуске вашего приложения, Hibernate будет логировать созданные SQL запросы в вашем выбранном лог-файле или консоли в зависимости от настроек логгера.