Вопросы и ответы на собеседование по Hibernate

(31-45)

31). Что вы знаете о Hibernate прокси и как это помогает в ленивой загрузке (lazy load)?

Hibernate **Proxy** - это мощный механизм, который позволяет оптимизировать загрузку данных и улучшить производительность приложения. Он позволяет вам работать с ассоциациями как с обычными объектами, но при этом загружать данные только по мере необходимости(**lazy loading**).
Ленивая загрузка позволяет отложить загрузку связанных данных до тех пор, пока они действительно не понадобятся, что улучшает производительность приложения.

Вот как Hibernate Proxy работает и как он помогает в ленивой загрузке:

1. Генерация Прокси-Объектов: При загрузке объекта с ассоциацией, помеченной как ленивая (например, @ManyToOne или @OneToMany с опцией FetchType.LAZY), Hibernate не загружает связанный объект сразу. Вместо этого он создает специальный прокси-объект, который является подклассом вашего класса и наследует его методы.
2. Инициализация Прокси: Когда вы пытаетесь получить доступ к данным в ассоциированном объекте (например, вызываете метод get() на ассоциированном поле), Hibernate проверяет, инициализирован ли уже прокси. Если нет, то он инициализирует его, выполнив запрос к базе данных для загрузки данных связанного объекта.
3. Преимущества Ленивой Загрузки:
    - Экономия ресурсов: Ленивая загрузка позволяет избегать избыточной загрузки данных, которые могут не понадобиться.
    - Улучшенная производительность: Меньше запросов к базе данных при загрузке объектов.
    - Гибкость: Вы можете выбирать, какие ассоциации загружать лениво, а какие - сразу.
4. Типы Прокси: Hibernate поддерживает два типа прокси-объектов:
    - **Runtime Proxy**: Этот тип прокси создается во время выполнения с использованием байткода. Он обеспечивает более высокую производительность, но требует наличия библиотеки байткода (например, CGLIB).
    - **Javassist Proxy**: Этот тип прокси создается с использованием библиотеки Javassist. Он менее производительный, но не требует наличия библиотеки байткода.

32). Как реализованы отношения в Hibernate?

Hibernate предоставляет различные способы для определения и реализации отношений между сущностями. Вы можете использовать JPA аннотации или конфигурацию XML файлов для описания этих отношений. Вот как это можно сделать:

1) One-to-One (Один-к-Одному):

- С JPA аннотациями: Используйте аннотации @OneToOne на поле или методе, чтобы определить отношение. Например:

```
@Entity
public class Person {
@OneToOne
private Address address;
    // ...
}
```

- С использованием XML: Определите соответствующий элемент <one-to-one> в вашем файле конфигурации XML.

2) One-to-Many (Один-к-Многим):

- С JPA аннотациями: Используйте аннотации @OneToMany на поле или методе в родительской сущности, чтобы указать отношение. Например:

```
@Entity
public class Department {
    @OneToMany(mappedBy = "department")
    private List<Employee> employees;
    // ...
}
```

- С использованием XML: Определите соответствующий элемент <one-to-many>.

32).
Продолжение

3) Many-to-Many (Многие-ко-Многим):

- С JPA аннотациями: Используйте аннотации @ManyToMany на поле или методе в обеих сущностях, чтобы определить отношение. Например:

```
@Entity
public class Student {
    @ManyToMany
    @JoinTable(name = "student_course",
            joinColumns = @JoinColumn(name = "student_id"),
            inverseJoinColumns = @JoinColumn(name = "course_id"))
    private List<Course> courses;
    // ...
}
```

- С использованием XML: Определите соответствующий элемент <many-to-many> и настройте таблицу промежуточного соединения.

33). Какие типы менеджмента транзакций поддерживаются в Hibernate?

Hibernate поддерживает два основных типа менеджмента транзакций:

1) Управление транзакциями с помощью **API** языка программирования (**Programmatic Transaction Management**): Этот метод позволяет программисту явно управлять транзакциями с использованием API, предоставляемого Hibernate или платформой, на которой работает ваше приложение. Программист может начать, зафиксировать или откатить транзакцию вручную с помощью соответствующих методов. Этот метод предоставляет максимальную гибкость, но может привести к более многословному коду.

Пример использования управления транзакциями с помощью API:

```
Session session = sessionFactory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    // Выполнение операций с базой данных
    tx.commit();
} catch (Exception e) {
    if (tx != null) {
        tx.rollback();
    }
    e.printStackTrace();
} finally {
    session.close();
}
```

33). Продолжение

2) Управление транзакциями с помощью декларативного подхода (**Declarative Transaction Management**): В этом методе управление транзакциями выносится за пределы кода приложения и настраивается с помощью конфигурации. Обычно этот метод используется в комбинации с контейнером Spring или другой платформой, которая предоставляет декларативную конфигурацию транзакций. Транзакции объявляются как аспекты, и их поведение настраивается в конфигурационных файлах. Этот метод обеспечивает сокращение кода и упрощение управления транзакциями.

Пример декларативного управления транзакциями в Spring с использованием аннотаций:

```
@Service
public class MyService {
    @Autowired
    private MyRepository repository;

    @Transactional
    public void doSomething() {
        // Выполнение операций с базой данных
    }
}
```

Оба метода имеют свои преимущества и подходят для разных сценариев. Выбор метода зависит от требований вашего приложения и вашей предпочтительной архитектуры.

34). Что такое каскадные связи (обновления) и какие каскадные типы есть в Hibernate?

Если у нас имеются зависимости между сущностями (entities), то нам необходимо определить как различные операции будут влиять на другую сущность. Это реализуется с помощью каскадных связей (или обновлений). Вот пример кода с использованием аннотации **@Cascade**:

```
import org.hibernate.annotations.Cascade;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {

@OneToOne(mappedBy = "employee")
@Cascade(value = org.hibernate.annotations.CascadeType.ALL)
private Address address;

}
```

Обратите внимание, что есть некоторые различия между enum CascadeType в Hibernate и в JPA. Поэтому обращайте внимание какой пакет вы импортируете при использовании аннотации и константы типа. Наиболее часто используемые CascadeType перечисления описаны ниже.

1. **None**: без Cascading. Формально это не тип, но если мы не указали каскадной связи, то никакая операция для родителя не будет иметь эффекта для ребенка.
2. **ALL**: Cascades save, delete, update, evict, lock, replicate, merge, persist. В общем — всё.
3. **SAVE_UPDATE**: Cascades save и update. Доступно только для hibernate.
4. **DELETE**: передает в Hibernate native **DELETE** действие. Только для hibernate.
5. **DETATCH**, **MERGE**, **PERSIST**, **REFRESH** и **REMOVE** – для простых операций.
6. **LOCK**: передает в Hibernate native **LOCK** действие.
7. **REPLICATE**: передает в Hibernate native **REPLICATE** действие.

35). Что вы знаете о классе HibernateTemplate?

Spring Framework предоставляет различные подходы для интеграции с Hibernate. Тем не менее, наиболее часто используется подход, использующий HibernateTemplate. Есть две основные причины:

- Класс скрывает детали управления сессиями и транзакциями.
- Предоставляет подход основанный на шаблонах

HibernateTemplate класс скрывает трудности управления сессиями и транзакциями при использовании Hibernate для доступа к данным. Нужно только инициализировать HibernateTemplate путем передачи экземпляра SessionFactory. Spring Framework берет на себя беспокойство за детали связанные с сессиями и транзакциями. Это помогает устранить инфраструктурный код, который может вносить суматоху при увеличении сложности.

HibernateTemplate, так же как и JdbcTemplate, предоставляет шаблонный подход для доступа к данным. Когда вы используете HibernateTemplate, вы будете работать с callbacks. Обратные вызовы — это единственный механизм в шаблонном подходе, который уведомляет шаблон запускать нужную задачу. Преимущество наличия обратного вызова в том, что там только одна точка входа в слой доступа к данным. И эта точка входа определяется шаблоном, в этом случае HibernateTemplate.

В комментариях дополнили, что использование HibernateTemplate не является рекомендуемым. Вместо использования HibernateTemplate из пакета org.springframework.orm рекомендуется использовать декларативный подход (@Transactional). Таким образом фреймворк сам позаботится об операциях **open, commit, close, flush.**

36). Какие паттерны применяются в Hibernate?

- Domain Model Pattern – объектная модель предметной области, включающая в себя как поведение так и данные.
- Data Mapper – слой мапперов (Mappers), который передает данные между объектами и базой данных, сохраняя их независимыми друг от друга и себя.
- Proxy Pattern — применяется для ленивой загрузки.
- Factory pattern — используется в SessionFactory

37). Расскажите о Hibernate Validator Framework.

Проверка данных является неотъемлемой частью любого приложения. Hibernate Validator обеспечивает эталонную реализацию двух спецификаций **JSR-303** и **JSR-349** применяемых в Java. Для настройки валидации в Hibernate необходимо сделать следующие шаги.

- **Добавить hibernate validation зависимости в проект.**

```
<dependency>
   <groupId>javax.validation</groupId>
   <artifactId>validation-api</artifactId>
   <version>1.1.0.Final</version>
</dependency>
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-validator</artifactId>
   <version>5.1.1.Final</version>
</dependency>
```

- **Так же требуются зависимости из JSR 341, реализующие Unified Expression Language для обработки динамических выражений и сообщений о нарушении ограничений.**

37). Продолжение

```
<dependency>
   <groupId>javax.el</groupId>
   <artifactId>javax.el-api</artifactId>
   <version>2.2.4</version>
</dependency>
<dependency>
   <groupId>org.glassfish.web</groupId>
   <artifactId>javax.el</artifactId>
   <version>2.2.4</version>
</dependency>
```

- **Использовать необходимые аннотации в бинах.**

```
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.Email;

public class Employee {

   @Min(value=1, groups=EmpIdCheck.class)
   private int id;

   @NotNull(message="Name cannot be null")
   @Size(min=5, max=30)
   private String name;

   @Email
   private String email;

   @CreditCardNumber
   private String creditCardNumber;
....
```

38). Какие преимущества дает использование плагина Hibernate Tools Eclipse?

Плагин Hibernate Tools упрощает настройку маппинга, конфигурационного файла. Упрощает работы с файлами свойств или xml тегами. Помогает минимизировать ошибки написания кода.

39). Чем отличается Lazy от Eager в Hibernate?

- **Eager Loading** — стратегия загрузки, при которой подгрузка связанных сущностей происходит сразу. Для применения необходимо в аннотацию отношения (@OneToOne, @ManyToOne, @OneToMany, @ManyToMany) передать fetch = FetchType.EAGER. Используется по умолчанию для отношений @OneToOne и @ManyToOne.
- **Lazy Loading** — стратегия загрузки, при которой подгрузка связанных сущностей откладывается как можно дольше. Чтобы задать такое поведение, нужно в аннотацию отношения (@OneToOne, @ManyToOne, @OneToMany, @ManyToMany) передать fetch = FetchType.LAZY. Используется по умолчанию для отношений @OneToMany, @ManyToMany. До момента загрузки используется proxy-объект, вместо реального. Если обратиться к такому LAZY-полю после закрытия сессии Hibernate, то получим **LazyInitializationException**.

40). Что такое 'проблема N+1 запроса' при использовании Hibernate? Когда возникает? Как решить? Как обнаружить?

Проблема N+1 запроса (или проблема N+1 select) - это ситуация, которая возникает при использовании ORM-фреймворка, такого как Hibernate, когда при загрузке сущностей и связанных с ними объектов выполняется гораздо больше запросов к базе данных, чем ожидалось. Это может существенно ухудшить производительность приложения и привести к ненужной нагрузке на базу данных.
Пример ситуации, в которой возникает проблема N+1 запроса:

1. Пусть у вас есть сущность **Author**, представляющая авторов книг, и сущность **Book**, представляющая книги.
2. Существует отношение между **Author** и **Book**, где один автор может иметь много книг (one-to-many).
3. Вы хотите загрузить всех авторов и их книги.

Если вы используете стандартный способ загрузки данных с помощью Hibernate, то для каждого автора будет выполнен запрос к базе данных для загрузки его книг. То есть, сначала выполняется запрос для выбора всех авторов (это один запрос), а затем для каждого автора выполняется отдельный запрос для выбора его книг. Таким образом, если у вас есть N авторов, будет выполнено N+1 запросов к базе данных (1 запрос для выбора авторов и по одному запросу для каждого автора для выбора его книг).

40).
Продолжение

Для решения проблемы N+1 запроса в Hibernate можно использовать следующие подходы:

1. **Eager Loading** (Жадная загрузка): Вместо ленивой загрузки (по умолчанию) можно использовать жадную загрузку, чтобы загрузить все связанные объекты сразу. Для этого используется аннотация @OneToMany(fetch = FetchType.EAGER) (или аналогичный XML-конфигурационный параметр) над связью. Однако этот подход может загрузить слишком много данных, если они необходимы не всегда.
2. **Join Fetch**: Можно использовать ключевое слово JOIN FETCH в HQL или JPQL запросах для выборки связанных объектов одним запросом. Это позволяет сократить количество запросов, но также может привести к дублированию данных в результатах запроса.
3. **Batch Fetching**: Hibernate поддерживает пакетную загрузку (batch fetching), которая позволяет выполнить один запрос для загрузки коллекции объектов. Это особенно полезно, когда у вас есть связь многие-к-одному (one-to-many), и вы хотите загрузить все связанные объекты для нескольких родительских объектов одним запросом.
4. **Second-Level Cache**: Использование вторичного кэша Hibernate (например, Ehcache или Infinispan) может уменьшить количество запросов к базе данных, храня часто используемые данные в памяти.

Для обнаружения проблемы N+1 запроса можно воспользоваться мониторингом базы данных, логированием SQL-запросов или использовать специализированные инструменты для профилирования приложений. Часто такие инструменты могут выявить множество однотипных запросов к базе данных, что является признаком проблемы N+1 запроса.

41). Как описать составной ключ при использовании Hibernate?

Для описания составного ключа (**composite key**) при использовании Hibernate можно воспользоваться аннотациями или XML-конфигурацией. Сначала определите класс, который будет представлять составной ключ. Затем укажите его в сущности (entity) как поле с помощью аннотации **@EmbeddedId** (для аннотаций) или <composite-id> (для XML-конфигурации).
Вот пример того, как описать составной ключ с использованием аннотаций:

- Создайте класс, представляющий составной ключ. Этот класс должен реализовывать интерфейс **Serializable** и переопределить методы **equals**() и **hashCode**(). Например:

```
import java.io.Serializable;
import java.util.Objects;

public class MyCompositeKey implements Serializable {
    private Long keyPart1;
    private String keyPart2;

    // Геттеры и сеттеры для keyPart1 и keyPart2

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        MyCompositeKey that = (MyCompositeKey) o;
        return Objects.equals(keyPart1, that.keyPart1) &&
                Objects.equals(keyPart2, that.keyPart2);
    }

    @Override
    public int hashCode() {
        return Objects.hash(keyPart1, keyPart2);
    }
}
```

41).
Продолжение

- В сущности, которая будет использовать составной ключ, создайте поле с типом вашего составного ключа и пометьте его аннотацией @EmbeddedId:

```
@Entity
public class MyEntity {
    @EmbeddedId
    private MyCompositeKey id;

    // Другие поля сущности

    // Геттеры и сеттеры для id и других полей
}
```

Теперь Hibernate знает, что **MyCompositeKey** представляет составной ключ для сущности **MyEntity**. Вы можете использовать **MyCompositeKey** в качестве ключа для поиска сущностей или устанавливать его значения при сохранении новых сущностей. Помните, что составные ключи могут использоваться только сущностями, которые используются в режиме чтения и записи (можно модифицировать). Они не могут быть использованы в сущностях, которые только для чтения (read-only).

42). Как можно отобразить наследование на БД с помощью JPA (Hibernate)?

Есть 4 способа отобразить наследование на БД с помощью JPA (Hibernate):

- **MappedSuperclass** — поля родителя содержатся в каждой таблице для каждого дочернего класса. Базовый класс отдельной таблицы не имеет. На базовый класс навешиваем **@MappedSuperClass**, а вот на дочерние @Entity. Если в таблице потомка поле родителя называется не так, как указано в родительском классе, то его нужно смаппить с помощью аннотации **@AttributeOverride** в классе этого потомка. Родитель не может участвовать в ассоциации. При полиморфных запросах у нас будут отдельные запросы для каждой таблицы.
- **Single table** — вся иерархия классов в одной таблице. Чтобы различать классы, необходимо добавить колонку-дискриминатор. В данной стратегии на родительский @Entity-класс навешивается @Inheritance(strategy = InheritanceType.SINGLE_TABLE) и @DiscriminatorColumn(name = "YOUR_DISCRIMINATOR_COLUMN_NAME") (по умолчанию имя колонки DTYPE и тип VARCHAR). В каждом подклассе указываем @DiscriminatorValue("ThisChildName") со значением, которое будет храниться в колонке-дискриминаторе для данного класса. Если нет возможности добавить колонку, то можно использовать аннотацию **@DiscriminatorFormula**, в которой указать выражение CASE...WHEN — это не по JPA, фишка Hibernate. Денормализация. Простые запросы к одной таблице. Возможное нарушение целостности — столбцы подклассов могут содержать NULL.

42).
Продолжение

- **Joined table** — отдельные таблицы для всех классов иерархии, включая родителя. В каждой таблице только свои поля, а в дочерних добавляется внешний (он же первичный) ключ для связи с родительской таблицей. В @Entity-класс родителя добавляем @Inheritance(strategy = InheritanceType.JOINED). Для полиморфных запросов используются JOIN, а также выражение CASE...WHEN, вычисляющее значение поля _clazz, которое заполняется литералами (0 (родитель), 1, 2 и т.д.) и помогает Hibernate определить какого класса будет экземпляр.
- **Table per class** — также как и в MappedSuperclass, имеем отдельные таблицы для каждого подкласса. Базовый класс отдельной таблицы не имеет. По **спецификации** JPA 2.2 (раздел 2.12) данная стратегия является опциональной, но в Hibernate реализована, поэтому продолжим. В данном случае на базовый класс мы навешиваем @Entity и @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS). Поле первичного ключа (@Id) обязательно для родительского класса. Также аннотация @AttributeOverride в этой стратегии не работает — называйте родительские поля в таблицах сразу единообразно. Полиморфный запрос будет использовать UNION для объединения таблиц. Чтобы различить при создании экземпляров подклассы, Hibernate добавляет поле _clazz в запросы, содержащие литералы (1, 2 и т.д.). А одинаковый набор столбцов для объединения добирается как NULL AS some_field. Родитель может участвовать в ассоциации с другими сущностями.

43). Что такое диалект?

Диалект — это набор файлов кода или иногда один файл, который определяет процесс подключения базы данных к классу Java. Диалект в Hibernate играет роль понимания связи, происходящей с базовой базой данных. Всякий раз, когда изменяется базовая база данных, все, что вам нужно изменить в конфигурации Hibernate, — это диалект и учетные данные базы данных. Это верно, пока код использует HQL-запросы.

44). Как Hibernate создает соединение с базой данных?

Hibernate читает настроенный диалект, чтобы решить, какой драйвер использовать. Hibernate поставляется в комплекте с jar-файлами драйверов баз данных. Эти банки аналогичны тем, которые используются для подключения к базе данных с использованием JDBC. На основе диалекта Hibernate динамически регистрирует соответствующий класс драйверов и использует URL-адрес и учетные данные для подключения к базе данных с использованием JDBC в бэкэнде.

45). Какая аннотация используется для объявления класса как сущность ?

Аннотация @Entity используется для объявления класса как объекта. Простой пример показан ниже.

```
@Entity
@Table(name="users")
public class User{
String username;
String password;
}
```