Оглавление

[JDBC, ORM, Hibernate, JPA 5](#_Toc47979530)

[JDBC (Java Database Connectivity) 5](#_Toc47979531)

[ORM 5](#_Toc47979532)

[Hibernate 5](#_Toc47979533)

[JPA (Java Persistency API) 5](#_Toc47979534)

[JDBC и Hibernate 6](#_Toc47979535)

[Важные интерфейсы Hibernate: 8](#_Toc47979536)

[EntityManager 9](#_Toc47979537)

[Entity (требования) 10](#_Toc47979538)

[Встраиваемый класс (@Embeddable и @Embedded) 12](#_Toc47979539)

[@MappedSuperClass 13](#_Toc47979540)

[Типы связей 13](#_Toc47979541)

[Владелец связи 13](#_Toc47979542)

[mappedBy 13](#_Toc47979543)

[@JoinColumn и @JoinTable 15](#_Toc47979544)

[Каскадирование 17](#_Toc47979545)

[FetchType 19](#_Toc47979546)

[orphanRemoval 19](#_Toc47979547)

[@Basic и @Column 19](#_Toc47979548)

[@Basic 19](#_Toc47979549)

[@Column 20](#_Toc47979550)

[@Basic и @Column 20](#_Toc47979551)

[Маппинг 21](#_Toc47979552)

[enum 21](#_Toc47979553)

[дат 21](#_Toc47979554)

[коллекция примитивов 21](#_Toc47979555)

[Жизненный цикл сущности 21](#_Toc47979556)

[new 21](#_Toc47979557)

[managed 21](#_Toc47979558)

[detached 21](#_Toc47979559)

[removed 21](#_Toc47979560)

[Операции над сущностью в разных состояниях жизненного цикла 22](#_Toc47979561)

[persist() 22](#_Toc47979562)

[merge() 22](#_Toc47979563)

[detach() 22](#_Toc47979564)

[remove() 22](#_Toc47979565)

[refresh() 22](#_Toc47979566)

[Три стратегии построения иерархии 23](#_Toc47979567)

[n + 1 select (описание и решения) 23](#_Toc47979568)

[Entity Grpah 25](#_Toc47979569)

[Блокировки (оптимистические и пессимистические). 27](#_Toc47979570)

[Кеширование (уровни кэширования, @Cacheable, @Cache, ehcache). 28](#_Toc47979571)

[Как работает первый уровень кэша: когда он есть, когда нет, к чему он привязан (к какому объекту) 30](#_Toc47979572)

[К какому объекту привязан кэш второго уровня (к EntityManagerFactory) 34](#_Toc47979573)

[Как настроить кэш второго уровня. 36](#_Toc47979574)

[Какой кэш еще есть. Кэш запросов — как настроить. Желательно понимать как объекты хранятся в кэше второго уровня и в кэше запросов. 37](#_Toc47979575)

[Как контролировать объекты второго уровня кэша: как удалить, как посмотреть. 38](#_Toc47979576)

[HQL, JPQL и SQL 39](#_Toc47979577)

[Criteria API 39](#_Toc47979578)

[Hibernate proxy (lazy load) 40](#_Toc47979579)

[Транзакции в Hibernate 41](#_Toc47979580)

[@Access 43](#_Toc47979581)

[@Id (первичный ключ) 43](#_Toc47979582)

[Первичный ключ 43](#_Toc47979583)

[@GenerationType (стратегии генерации) 44](#_Toc47979584)

[@EmbeddedId 45](#_Toc47979585)

[@OrderBy 45](#_Toc47979586)

[@OrderColumn — как работает, где ставится 46](#_Toc47979587)

[Различия между @OrderBy и @OrderColumn — пример с БД 46](#_Toc47979588)

[@Transient 47](#_Toc47979589)

# JDBC, ORM, Hibernate, JPA

JDBC (Java Database Connectivity) — API (Java SE) между Java-программистами и разработчиков БД.

ORM — преобразование объектов в реляционные БД и наоборот. В Java делается с помощью рефлексия и JDBC.

Hibernate — реализация ORM.

JPA (Java Persistency API) — API (Java SE) между Java-программистами и разработчиками ORM решений.

На один шаг выше ORM. Его высокоуровневый API и спецификация, позволяющие реализовать различные инструменты ORM, что дает разработчику возможность гибко изменять реализацию с одной ORM на другую.

Например, если приложение использует API‑интерфейс JPA и реализация находится в спящем режиме. В будущем он может переключиться на IBatis, если требуется. Но, с другой стороны, если приложение напрямую блокирует реализацию с помощью Hibernate без платформы JPA, переключение будет непростой задачей.

### JDBC и Hibernate

Hibernate:  
∙ один из самых востребованных ORM фреймворков для Java;  
∙ устраняет повторяющийся код (в отличии JDBC);  
∙ скрывает от разработчика множество кода, необходимого для управления ресурсами, и позволяет сосредоточиться на бизнес логике;  
∙ поддерживает XML и аннотации JPA (позволяет сделать реализацию кода независимой);  
∙ предоставляет собственный мощный язык запросов (HQL полностью объектно-ориентирован и понимает принципы наследования, полиморфизма и ассоциации (связи));  
∙ широко распространенный open source проект (тысячи открытых статей, примеров и документация по использованию);  
∙ легко интегрируется с другими Java EE фреймворками (Spring Framework);  
∙ поддерживает ленивую инициализацию (прокси объекты) и выполняет запросы к БД только по необходимости;  
∙ поддерживает разные уровни cache (может повысить производительность);  
∙ может использовать чистый SQL (поддерживает возможность оптимизации запросов и работы с любым сторонним вендором БД и его фичами);  
∙ неявно использует управление транзакциями (большинство запросов нельзя выполнить вне транзакций);  
∙ использует HibernateException (непроверяемые; нет необходимости проверять их в коде каждый раз).

О прокси:  
∙ Hibernate прокси используется для замены реальной сущности POJO (Plain Old Java Object);  
∙ класс Proxy генерируется во время выполнения и расширяет исходный класс сущности;  
∙ Hibernate использует объекты Proxy для объектов, чтобы разрешить отложенную загрузку;  
∙ при доступе к основным свойствам прокси просто делегирует вызов исходной сущности;  
∙ каждый List, Set, Map тип в классе сущностей замещен PersistentList, PersistentSet, PersistentMap. Эти классы отвечают за перехват вызова неинициализированной коллекции;  
∙ прокси не выдает никаких операторов SQL, он просто запускает InitializeCollectionEvent, который обрабатывается связанным прослушивателем, который знает, какой запрос инициализации выпустить (зависит от настроенного плана выборки);  
∙ прокси объект получаем через метод session.load(), если вызываем геттеры и сеттеры — выполняется select в базу на получение реального объекта.

# EntityManager

Главный API для работы с JPA.

Описывает API для основных операций:

операции над Entity:  
∙ persist (добавление Entity под управление JPA),  
∙ merge (обновление),  
∙ remove (удаления),  
∙ refresh (4) (обновление данных),  
∙ detach (удаление из управление JPA),  
∙ lock (2) (блокирование Entity от изменений в других thread);

получение данных:  
∙ find (4) (поиск и получение Entity),  
∙ createQuery (5),  
∙ createNamedQuery (2),  
∙ createNativeQuery (3),  
∙ contains,  
∙ createNamedStoredProcedureQuery,  
∙ createStoredProcedureQuery (3);

получение других сущностей JPA:  
∙ getTransaction,  
∙ getEntityManagerFactory,  
∙ getCriteriaBuilder,  
∙ getMetamodel,  
∙ getDelegate;

работа с EntityGraph:  
∙ createEntityGraph (3),  
∙ getEntityGraphs;

общие операции над EntityManager или всеми Entities:  
∙ close,  
∙ isOpen,  
∙ getProperties,  
∙ setProperty,  
∙ clear;

другие:  
∙ getReference,  
∙ flush,  
∙ setFlushMode,  
∙ getFlushMode,  
∙ getLockMode,  
∙ joinTransaction,  
∙ isJoinedToTransaction,  
∙ unwrap.

# Entity (требования)

Entity класс обязан  
∙ быть помечен аннотацией @Entity или описан в XML файле,  
∙ содержать конструктор без аргументов (public или protected; другие конструкторы содержать не запрещается),  
∙ быть классом верхнего уровня,  
∙ содержать первичный ключ;

Entity класс не может  
∙ быть перечислением (enum), интерфейсом или финальным классом,  
∙ содержать финальные поля или методы (участвующие в маппинге);

∙ Поля Entity класса должны быть доступны только его методам;

∙ Если объект Entity класса будет передаваться по значению как отдельный объект (через удаленный интерфейс), он так же должен реализовывать Serializable интерфейс.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | @Entity  public class classEntity {  @Id  @GeneratedValue(...)  private Long id;  ...  public classEntity() {  }  ...  // геттеры и сеттеры  } |

∙ Entity класс может наследоваться от любых классов (не Entity и Entity);  
∙ не Entity класс может наследоваться от Entity класса;  
∙ Entity может быть абстрактным классом (сохраняет свойства Entity, но нельзя непосредственно инициазировать).

# Встраиваемый класс (@Embeddable и @Embedded)

Встраиваемый класс:

∙ выносить общие атрибуты для нескольких Entity в отдельный класс;

∙ должен быть помечен аннотацией @Embeddable или описан в XML файле.

∙ должны удовлетворять правилам Entity класса (кроме первичного ключа и @Entity);

∙ используется только как часть Entity классов (одного или нескольких);

∙ Entity класс могут содержать как одиночные встраиваемые классы, так и коллекции таких классов;

∙ такие классы могут быть использованы как ключи или значения map;

∙ во время выполнения каждый встраиваемый класс принадлежит только одному объекту Entity класса и не может быть использован для передачи данных между объектами Entity классов (т. е. такой класс не является общей структурой данных для разных объектов);

# @MappedSuperClass

∙ класс, от которого наследуются Entity (может содержать аннотации JPA);

∙ не является Entity (не обязан выполнять требования Entity).

∙ не может использоваться в операциях EntityManager или Query.

∙ должен быть помечен аннотацией @MappedSuperclass или описан в xml файле.

# Типы связей

@OneToOne, @OneToMany, @ManyToOne и @ManyToMany.

Каждая разделяется еще на два вида:

Unidirectional (ссылка устанавливается только с одной стороны).

Bidirectional (ссылка устанавливается с обеих сторон, один считается владельцем связи — важно для случаев каскадного удаления данных, т. е при удалении владельца удаляется объект).

## Владелец связи

### mappedBy

Используется с @JoinColumns. @JoinColumns (value, foreignKey) определяет владельца связи, атрибут mappedBy аннотаций @OneToOne, @OneToMany, @ManyToOne и @ManyToMany определяет колонку у владельца связи.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | @Entity  public class Email {  @Id  @GeneratedValue(  strategy = GenerationType.AUTO)  private Long id;  @ManyToOne(fetch = FetchType.LAZY)  @JoinColumn(name = "employee\_id")  private Employee employee;  // ...  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | @Entity  public class Employee {  @Id  @GeneratedValue(  strategy = GenerationType.AUTO)  private Long id;  @OneToMany(  fetch = FetchType.LAZY,  mappedBy = "employee")  private List<Email> emails;  // ...  } |

У таблицы с mappedBy удалится колонка с внешним ключом, т. к. она будет на другой стороне и так.

Этот атрибут говорит хибернейту, что ключ для связи лежит на другой стороне. Это значит, что несмотря на то, что у нас есть две таблицы — только одна из них содержит ограничение на внешний ключ. Этот атрибут позволяет по‑прежнему ссылаться из таблицы, которая не содержит ограничения на другую таблицу. Атрибут mappedBy тесно связан с аннотацией @JoinColumn. Если применить атрибут mappedBy на одной стороне связи — хибернейт не станет создавать смежную таблицу.

### @JoinColumn и @JoinTable

@JoinColumn

Указывает столбец для присоединения к связной сущности или коллекции элементов. Если сама аннотация @JoinColumn имеет значение по умолчанию, то предполагается наличие одного столбца соединения и применяются значения по умолчанию.

@JoinColumn — в связях вида One2Many/Many2One сторона, находящаяся в собственности обычно называется стороной “многих”. Обычно, это сторона, которая держит внешний ключ. Эта аннотация, фактически, описывает физический (как в БД) маппинг на стороне “многих”. В атрибут name этой аннотации дается название колонки, которая будет присоединена к таблице “многих” и которая будет заполняться значениями первичных ключей из таблицы-собственника. Т. о. — мы даем название для внешнего ключа. По факту — использование этой аннотации опционально, т. к. хибернейт, проанализировав сущность — поймет сам, что в таблице для этого класса нужно создать колонку, обозначающую внешний ключ, и даст ей соответствующее название “{name}\_id”.

@JoinTable

Определяет сопоставление ассоциации. Он применяется к владельцу ассоциации.

@JoinTable обычно используется при отображении связей «многие ко многим» и однонаправленных связей «один ко многим». Он также может использоваться для сопоставления двунаправленных ассоциаций «многие к одному» или «один ко многим», однонаправленных связей «многие к одному» и связей «один к одному» (как двунаправленных, так и однонаправленных).

Когда @JoinTable используется при отображении отношения с встраиваемым классом на стороне-владельце отношения, содержащая сущность, а не встраиваемый класс считается владельцем отношения.

Если @JoinTable аннотация отсутствует, применяются значения по умолчанию для элементов аннотации.

@JoinColumn и @JoinTable

@JoinTable хранит идентификатор обеих таблиц в отдельной таблице, а @JoinColumn хранит идентификатор другой таблицы в новом столбце.

@JoinTable: это тип по умолчанию. Используйте это, когда вам нужна более нормализованная БД, т. е. для уменьшения избыточности.

@JoinColumn: используйте это для лучшей производительности, т. к. ему не нужно присоединяться к дополнительной таблице.

## Каскадирование

Действие над целевой сущностью (владельцем) будет применено к связанной сущности.

|  |  |
| --- | --- |
| 1. | @OneToOne(cascade=CascadeType.ALL) |

CascadeType (JPA):  
∙ ALL — распространяет все операции, включая специфичные для Hibernate, от родительского объекта к дочернему объекту.  
∙ PERSIST — делает временный экземпляр постоянным. CascadeType.PERSIST передает операцию persist от родительского объекта к дочернему объекту. Когда мы сохраняем личность лица, адрес также будет сохранена.  
∙ MERGE — операция слияния копирует состояние данного объекта в постоянный объект с тем же идентификатором. CascadeType.MERGE передает операцию слияния от родителя к дочерней сущности.  
∙ REMOVE — удаляет строку, соответствующую объекту из БД, а также из постоянного контекста.  
∙ DETACH — удаляет объект из постоянного контекста. Когда мы используем CascadeType.DETACH, дочерняя сущность также будет удалена из постоянного контекста.

CascadeType (Hibernate):  
∙ LOCK — повторно присоединяет сущность и связанную дочернюю сущность с постоянным контекстом снова.  
∙ REFRESH — повторно считывают значение данного экземпляра из БД. В некоторых случаях мы можем изменить экземпляр после сохранения в БД, но позже нам нужно отменить эти изменения.  
∙ REPLICATE — используется, когда у нас более одного источника данных, и мы хотим, чтобы данные были синхронизированы. С CascadeType.REPLICATE операция синхронизации также распространяется на дочерние объекты всякий раз, когда выполняется над родительским объектом.  
∙ SAVE\_UPDATE — распространяет ту же операцию на связанный дочерний объект. Это полезно, когда мы используем специфичные для Hibernate операции, такие как save, update и saveOrUpdate.

# FetchType

|  |  |
| --- | --- |
| 1. | @OneToOne(fetch = FetchType.LAZY) |

В JPA описаны два типа fetch стратегии:  
∙ LAZY — данные поля будут загружены только во время первого доступа к этому полю;  
∙ EAGER — данные поля будут загружены немедленно.

Используются по умолчанию:  
∙ EAGER — @Basic и ToOne;  
∙ LAZY — @Collection и ToMany.

# orphanRemoval

Директива orphanRemoval объявляет, что связанные экземпляры сущностей должны быть удалены, когда они отсоединены от родителя, или эквивалентно, когда родитель удален

# @Basic и @Column

## @Basic

Простейший тип маппинга колонок.

Параметры: fetch (стратегия доступа) и optional (проверка на null).

|  |  |
| --- | --- |
| 1. | FetchType fetch() default FetchType.EAGER; |
| 1. | boolean optional() default true; |

По умолчанию — FetchType.EAGER.

Типы: примитивы и обертки, String, BigInteger, BigDecimal, Date, Calendar, Time, Timestamp, массивы (byte[], Byte[], char[], Character[]), enum, и любой другой тип, который реализует java.io.Serializable.

## @Column

Указывает детали столбца в таблице.

Атрибуты:  
∙ name — имя столбца;  
∙ length — длина столбца;  
∙ nullable — является ли элемент обнуляемым;  
∙ unique — является ли уникальным столбец.

Если мы не укажем эту аннотацию, имя поля будет считаться именем столбца в таблице.

## @Basic и @Column

|  |  |  |  |
| --- | --- | --- | --- |
|  | указывает | применяются | атрибуты проверки на null |
| @Basic | поле загружено лениво | к сущностям | optional |
| @Column | имя столбца БД | к столбцам | nullable |

# Маппинг

## enum

@Enumerated указывает, следует ли сохранять Enum по имени или по порядковому номеру (по умолчанию).

∙ @Enumerated(EnumType.STRING) — в базе хранятся имена enam.

∙ @Enumerated(EnumType.ORDINAL) — в базе хранятся порядковые номера enum (по умолчанию).

## дат

С Java 8 специальной аннотации не требуется (@Temporal).

## коллекция примитивов

@ElementCollection, @CollectionOfElements.

Жизненный цикл сущности

new — объект создан, не имеет первичный ключ;

managed — имеет первичный ключ, управляется JPA;

detached — не управляется JPA;

removed — управляется JPA, будет удален при commit.

Операции над сущностью в разных состояниях жизненного цикла

## persist()

∙ new, managed, removed — managed;

∙ detached — exception.

## merge()

∙ new, managed, detached — managed;

∙ removed → exception.

## detach()

∙ new, detached — игнор;

∙ managed, removed — detached.

## remove()

∙ new, removed — игнор;

∙ managed — removed;

∙ detached — exception.

## refresh()

∙ new, removed, detached — exception.

∙ managed → будут восстановлены все изменения из БД данного Entity, также произойдет refresh всех каскадно зависимых объектов;

# Три стратегии построения иерархии

В JPA описаны три стратегии наследования маппинга, т. е. как JPA будет работать с классами-наследниками Entity:

∙ Одна общая таблица (Animals с колонкой animalType: Cat/Dog). Минус: наличие полей с null (уникальные поля классов наследников).

∙ Объединяющая стратегия (одна таблица Animals с общими полями и таблицы Cat и Dog с уникальными). Минус: потери производительности от объединения таблиц.

∙ Одна таблица для каждого класса (без таблицы Animals). Минус: плохая поддержка полиморфизма и большое количество sql‑запросов (или использование UNION запроса).

# n + 1 select (описание и решения)

Есть коллекция объектов Car (строк БД), содержащих коллекцию объектов Wheel (отношение один-ко-многим). Нужно пройтись по всем машинам, и для каждой распечатать список колес:

|  |  |
| --- | --- |
| 1. | SELECT \* FROM Cars; |

И тогда для каждого Car:

|  |  |
| --- | --- |
| 1. | SELECT \* FROM Wheel WHERE CarId = ? |

Другими словами, у вас есть один выбор для автомобилей, а затем N дополнительных выборов, где N — общее количество автомобилей.

В качестве альтернативы можно получить все колеса и выполнить поиск в памяти:

|  |  |
| --- | --- |
| 1. | SELECT \* FROM Wheel |

Это уменьшает количество обращений к БД с N + 1 до 2. Большинство инструментов ORM предоставляют несколько способов предотвратить выбор N + 1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | unidirectional OneToMany | | unidirectional ManyToone | | bidirectional OneToMany/ManyToOne | |
| solution | jpql | native | jpql | native | jpql | native |
| join fetch | + | - | + | - | + | - |
| FetchMode.SUBSELECT | + | - | - | - | +/-\*\* | - |
| BatchSize | + | + | - | - | +/-\*\* | +/-\*\* |
| EntityGraph | + | - | + | - | + | - |
| SqlResultSetMapping | - | - | - | + | - | -/+\*\*\* |
| HibernateSpecificMapping | - | + | - | + | - | + |

\* если не используем аннотацию JoinColumn и оставляем связанную третью таблицу

\*\* работает только при выборке собственника с коллекцией зависимых сущностей

\*\*\* работает только при выборке дочерней сущности с ссылкой на  родительскую сущность

Выводы:

∙ Лучшим вариантом решения N+1 проблемы для простых запросов (1-3 уровня вложенности связанных объектов) будет join fetch и jpql запрос. Следует придерживаться тактики, когда мы выбираем из jpql и нативного запроса jpql

∙ Если у нас имеется нативный запрос, и мы не заботимся о слабой связанности кода — то хорошим вариантом будет использование Hibernate Specific Mapping. В противном случае стоит использовать @SqlResultSetMapping

∙ В случаях, когда нам нужно получить по‑настоящему много данных, и у нас jpql запрос — лучше всего использовать EntityGraph

∙ Если мы знаем примерное количество коллекций, которые будут использоваться в любом месте приложения — можно использовать @BatchSize

# Entity Grpah

FetchType.LAZY используется почти во всех случаях, чтобы получить хорошо работающее и масштабируемое приложение. Определение графа сущностей не зависит от запроса и определяет, какие атрибуты нужно извлечь из БД. Граф сущностей может использоваться в качестве выборки или графика загрузки. Если используется график выборки, только атрибуты, указанные в графе сущностей, будут обрабатываться как FetchType.EAGER. Все остальные атрибуты будут ленивыми. Если используется график загрузки, все атрибуты, которые не указаны в графе объектов, сохранят свой тип выборки по умолчанию.

Для этого существует EntityGraph API, используется он так: с помощью аннотации @NamedEntityGraph для Entity, создаются именованные EntityGraph объекты, которые содержат список атрибутов, у которых нужно поменять fetchType на EAGER, а потом данное имя указывается в hits запросов или метода find. В результате fetchType атрибутов Entity меняется, но только для этого запроса. Существует две стандартных property для указания EntityGraph в hit:

∙ javax.persistence.fetchgraph — все атрибуты перечисленные в EntityGraph меняют fetchType на EAGER, все остальные на LAZY.

∙ javax.persistence.loadgraph — все атрибуты перечисленные в EntityGraph меняют fetchType на EAGER, все остальные сохраняют свой fetchType (т. е. если у атрибута, не указанного в EntityGraph, fetchType был EAGER, то он и останется EAGER). С помощью NamedSubgraph можно также изменить fetchType вложенных объектов Entity.

Определение именованного графа сущностей выполняется аннотацией @NamedEntityGraph в сущности. Он определяет уникальное имя и список атрибутов (attributeNodes), которые должны быть загружены.

# Блокировки (оптимистические и пессимистические).

Блокировки — параллельная работа с одними и теми же данными в БД.

Оптимистический подход не предполагает параллельной работы (производится проверка и выбрасывание исключений).

Пессимистичный подход — предполагает. Другие транзакции останавливаются при обращении к заблокированным данным и ждут снятия блокировки (или кидают исключение).

LockModeType.OPTIMISTIC (блокировка на чтение),

LockModeType.OPTIMISTIC\_FORCE\_INCREMENT (блокировка на запись),

LockModeType.PESSIMISTIC\_READ (данные блокируются в момент чтения),

LockModeType.PESSIMISTIC\_WRITE (данные блокируются в момент записи).

Форс инкремент

# Кеширование (уровни кэширования, @Cacheable, @Cache, ehcache).

Способ оптимизации работы приложения (уменьшить количество прямых обращений к БД).

Кэш первого уровня  
∙ кэш сессии (Session) (обязательный);  
∙ через него проходят все запросы;  
∙ сессия хранит объект за счет своих ресурсов перед отправкой в БД;

В том случае, если мы выполняем несколько обновлений объекта, Hibernate старается отсрочить (насколько это возможно) обновление для того, чтобы сократить количество выполненных запросов. Если мы закроем сессию, то все объекты, находящиеся в кэше, теряются, а далее — либо сохраняются, либо обновляются. Кэш первого уровня это и есть PersistenceContext.

Кэш второго уровня является необязательным (опциональным) и изначально Hibernate будет искать необходимый объект в кэше первого уровня. В основном, кэширование второго уровня отвечает за кэширование объектов. Кэш второго уровня привязан к EntityManagerFactory.

В Hibernate предусмотрен кэш для запросов, и он интегрирован с кэшем второго уровня. Это требует двух дополнительных физических мест для хранения кэшированных запросов и временных меток для обновления таблицы БД. Этот вид кэширования эффективен только для часто используемых запросов с одинаковыми параметрами.

Одновременного доступа к объектам в кэше в hibernate существует четыре:  
∙ transactional – полноценное разделение транзакций. Каждая сессия и каждая транзакция видят объекты, как если бы только они с ним работали последовательно одна транзакция за другой. Минус — блокировки и потеря производительности.  
∙ read-write — полноценный доступ к одной конкретной записи и разделение ее состояния между транзакциями. Однако суммарное состояние нескольких объектов в разных транзакциях может отличаться.  
∙ nonstrict-read-write — аналогичен read-write, но изменения объектов могут запаздывать и транзакции могут видеть старые версии объектов. Рекомендуется использовать в случаях, когда одновременное обновление объектов маловероятно и не может привести к проблемам.  
∙ read-only — объекты кэшируются только для чтения и изменение удаляет их из кэша.

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:

Уже внутри кода приложения для запроса применяется метод setCacheable(true).

# Как работает первый уровень кэша: когда он есть, когда нет, к чему он привязан (к какому объекту)

Кэширование — средство, предоставляемое средами ORM, которое помогает пользователям быстро запустить веб‑приложение, а сама структура помогает сократить количество запросов к БД за одну транзакцию. Hibernate достигает второй цели, внедряя кэш первого уровня.

Кэш первого уровня в hibernate включен по умолчанию, и вам не нужно ничего делать, чтобы эта функция работала. На самом деле, вы не можете отключить его даже принудительно.

Кэш первого уровня легко понять, если мы понимаем тот факт, что он связан с объектом Session. Как мы знаем, объект сеанса создается по требованию из фабрики сеансов и теряется при закрытии сеанса. Аналогично, кэш первого уровня, связанный с объектом сеанса, доступен только до тех пор, пока объект сеанса не станет активным. Он доступен только для объекта сеанса и не доступен для любого другого объекта сеанса в любой другой части приложения.

Важные факты:

∙ кэш первого уровня связан с объектом Session, а другие объекты сеанса в приложении его не видят.

∙ область действия объектов кэша имеет сессию. Как только сессия закрыта, кэшированные объекты исчезают навсегда.

∙ Кэш первого уровня включен по умолчанию, и вы не можете его отключить.

∙ Когда мы запрашиваем объект в первый раз, он извлекается из БД и сохраняется в кэше первого уровня, связанном с сессией хибернейта.

∙ Если мы снова запросим тот же объект с тем же объектом сеанса, он будет загружен из кэша, и никакой SQL‑запрос не будет выполнен.

∙ Загруженный объект можно удалить из сеанса с помощью метода evict(). Следующая загрузка этого объекта снова вызовет БД, если она была удалена с помощью метода evict().

∙ Весь кэш сеанса можно удалить с помощью метода clear(). Это удалит все сущности, хранящиеся в кэше.

Несколько фактов про кэш первого уровня:

∙ Кэш первого уровня не является потокобезопасным.

∙ Кэш первого уровня привязан к сессии и уничтожается следом за уничтожением сессии.

Из этого следует один важный вывод: кэш первого уровня не является средством оптимизации большого количества повторяющихся запросов на выборку со стороны клиента, т. к. каждый запрос будет обрабатываться в отдельной транзакции, на которую будет выделен новый объект entityManager, который связан напрямую с новой сессией. Соответственно, на 20 одинаковых запросов пользователя будет создано 20 entityManager и 20 сессий. Будет выделено 20 транзакций, даже если запросы обрабатываются и поступают одновременно.

Кэш первого уровня нужен:

∙ Для сохранения целостности данных.

∙ Оптимизации запросов на изменение/удаление.

∙ Оптимизация запросов на выборку в рамках одной транзакции.

В пределах жизненного цикла одной сессии и в рамках одной транзакции мы можем изменить внутреннее состояние сущности неограниченное количество раз, каждое изменение будет вноситься в кэш первого уровня. Но в базу запрос отправится только тогда, когда будет сделан комит транзакции. В базу отправятся те данные, которые содержит сущность на момент последнего изменения. До тех пор, пока транзакция не будет закончена — все изменения будут храниться в кэше. Даже если мы вызовем 20 раз метод setField() у любой сущности — в базу в итоге отправится только один запрос.

Если же мы вынуждены читать в рамках одной транзакции несколько раз одни и те же данные, то, единожды загрузив данные запросом из базы, мы будем в дальнейшем работать с данными внутри кэша, не повторяя дополнительных запросов. Например, если достать List<User> и затем достать конкретного юзера с id = 2, то запрос в базу не будет произведен, т. к. список всех пользователей уже лежит в кэше. Так же если мы уже после того, как достали пользователя с id = 2 изменили 10 раз его имя, а затем снова выберем список всех пользователей — мы и в этом случае не получим дополнительных запросов. В описанном выше случае будет произведено только два запроса: на выборку списка всех пользователей в самом начала и один запрос на изменение состояния пользователя уже в конце транзакции.

# К какому объекту привязан кэш второго уровня (к EntityManagerFactory)

Кэш второго уровня создается в области фабрики EntityManagerFactory и доступен для использования во всех EntityManager, которые создаются с использованием этой конкретной фабрики.

Это также означает, что после закрытия фабрики весь кэш, связанный с ним, умирает, а менеджер кэша также закрывается.

Кроме того, это также означает, что если у вас есть два экземпляра фабрики, в вашем приложении будет два менеджера кэша, и при доступе к кэшу, хранящемуся в физическом хранилище, вы можете получить непредсказуемые результаты, такие как пропадание кеша.

∙ Всякий раз, когда сессия пытается загрузить объект, самое первое место, где он ищет кэшированную копию объекта в кэше первого уровня.

∙ Если кэшированная копия объекта присутствует в кэше первого уровня, она возвращается как результат метода загрузки.

∙ Если в кэше первого уровня нет кэшированной сущности, то для кэшированной сущности ищется кэш второго уровня.

∙ Если кэш второго уровня имеет кэшированный объект, он возвращается как результат метода load(). Но перед возвратом объекта он также сохраняется в кэше первого уровня, так что при следующем вызове метода загрузки объект будет возвращен из самого кэша первого уровня, и больше не потребуется обращаться в кэш второго уровня.

∙ Если объект не найден в кэше первого уровня и кэше второго уровня, то выполняется запрос к БД, и объект сохраняется на обоих уровнях кэша перед возвратом в качестве ответа метода load().

∙ Кэш второго уровня проверяет себя для измененных объектов.

∙ Если какой‑либо пользователь или процесс вносят изменения непосредственно в БД, то само по себе кэширование второго уровня не может обновляться до тех пор, пока не истечет время «timeToLiveSeconds» для этой области кэша. В этом случае хорошей идеей будет сделать недействительным весь кеш и позволить hibernate снова построить кэш.

# Как настроить кэш второго уровня.

Со следующими двумя свойствами мы сообщаем Hibernate, что кэширование L2 включено, и даем ему имя класса фабрики региона:

|  |  |
| --- | --- |
| 1.  2.  3.  4. | hibernate.cache.use\_second\_level\_cache=true  hibernate.cache.region.factory\_class=  org.hibernate.cache.ehcache  .EhCacheRegionFactory |

Чтобы сделать объект пригодным для кэширования второго уровня, мы помечаем его аннотацией @Cache или @Cacheable, специфичной для Hibernate, и указываем стратегию параллельного использования кэша:

∙ ALL — все Entity могут кэшироваться в кэше второго уровня;

∙ NONE — кеширование отключено для всех Entity;

∙ ENABLE\_SELECTIVE — кэширование работает только для тех Entity, у которых установлена аннотация Cacheable(true), для всех остальных кэширование отключено;

∙ DISABLE\_SELECTIVE — кэширование работает для всех Entity, за исключением тех у которых установлена аннотация Cacheable(false);

∙ UNSPECIFIED — кеширование не определенно, каждый провайдер JPA использует свою значение по умолчанию для кэширования.

# Какой кэш еще есть. Кэш запросов — как настроить. Желательно понимать как объекты хранятся в кэше второго уровня и в кэше запросов.

Hibernate также поддерживает QueryCache, который может хранить результаты запроса. Вам необходимо активировать его в файле persistence.xml, установив для параметра

|  |  |
| --- | --- |
| 1. | hibernate.cache.use\_query\_cache=true |

и определив

|  |  |
| --- | --- |
| 1. | hibernate.cache.region.factory\_class |

Кроме того, вам также необходимо активировать кэширование для конкретного запроса, для которого вы хотите кэшировать результаты, вызывая setCacheable(true).

# Как контролировать объекты второго уровня кэша: как удалить, как посмотреть.

Сохранение или обновление элемента: save(), update(), saveOrUpdate().

Получение предмета: load(), get(), list(), iterate(), scroll().

Состояние объекта синхронизируется с БД при вызове метода flush(). Чтобы избежать этой синхронизации, вы можете удалить объект и все коллекции из кэша первого уровня с помощью evict() метода. Чтобы удалить все элементы из кэша сеанса, используйте метод Session.clear():

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | ScrollableResult cats =  sess.createQuery("from Cat as cat")  .scroll();  while ( cats.next() ) {  Cat cat = (Cat) cats.get(0);  doSomethingWithACat(cat);  sess.evict(cat);  } |

Определение того, принадлежит ли элемент кешу сеанса. Сеанс предоставляет contains() метод для определения того, принадлежит ли экземпляр кешу сеанса.

# HQL, JPQL и SQL

HQL и JPQL работают с сущностями, а SQL работает с таблицей.

JPQL основан на HQL (Hibernate Query Language), более раннем не стандартизированном языке запросов, включенном в библиотеку объектно-реляционного отображения Hibernate.

Hibernate и HQL были созданы до появления спецификации JPA. JPQL является подмножеством языка запросов HQ.

# Criteria API

HQL и JPQL.

Hibernate Criteria API является более объектно-ориентированным для запросов, которые получают результат из БД. Для операций update, delete или других DDL манипуляций использовать Criteria API нельзя. Критерии используются только для выборки из БД в более объектно-ориентированном стиле. Используется для динамических запросов.

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().

# Hibernate proxy (lazy load)

Hibernate использует прокси объект для поддержки отложенной загрузки. Обычно при загрузке данных из таблицы Hibernate не загружает все отображенные (замаппинные) объекты. Как только вы ссылаетесь на дочерний объект или ищите объект с помощью геттера, если связанная сущность не находиться в кэше сессии, то прокси код перейдет к БД для загрузки связанной сущности. Для этого используется javassist, чтобы эффективно и динамически создавать реализации подклассов ваших entity объектов.

# Транзакции в Hibernate

Hibernate построен поверх JDBC API и реализует модель транзакций JDBC. Если быть точным, Hibernate способен работать или с JDBC транзакциями или с JTA транзакциями — Java Transaction API.

Транзакцию можно начать вызовом beginTransaction() объекта Session, либо запросить у Session связанный с ней объект Transaction и позвать у последнего метод begin(). С объектом Session всегда связан ровно один объект Transaction, доступ к которому может быть получен вызовом getTransaction().

Методов для подтверждения или отката транзакции у объекта Session нет, необходимо всегда обращаться к объекту Transaction.

В отличие от JDBC в Hibernate не поддерживаются Savepoints и транзакция может только быть подтверждена или откачена, без промежуточных вариантов.

Операции над транзакциями

У объекта Transaction есть еще несколько методов, кроме commit() и rollback(), которые позволяют тонко управлять поведением транзакции. Метод isActive() позволяет проверить, есть ли в рамках объекта Transaction управляемая им транзакция. Очевидно, что такая транзакция существует в промежутке времени между вызовами begin() и commit()/rollback().

Метод setRollbackOnly() помечает транзакцию как откаченную в будущем. В отличие от rollback() этот метод не закрывает транзакцию и все последующие запросы к базе будут продолжать выполняться в рамках той же самой транзакции, но завершить эту транзакцию можно будет только откатом и вызовом rollback(). Вызов commit() на такой транзакции выбросит исключение. Проверить состояние транзакции можно вызовом getRollbackOnly().

## @Access

Определяет тип доступа. Обращение к атрибутам Entity как к полям класса (AccessType.FIELD) или как к свойствам класса (AccessType.PROPERTY). Т. е. PROPERTY над методом дает возможность вернуть измененный элемент.

## @Id (первичный ключ)

Определяет primary key в entity bean.

Указываем первичный ключ.

Типы переменных: примитивы (оболочки), String, Date, BigDecimal, BigInteger.

### Первичный ключ

Допустимые типы:  
∙ примитивы (обертки);  
∙ строки;  
∙ BigDecimal и BigInteger;  
∙ java.util.Date и java.sql.Date.

Для автогенерируемого — числовые типы.

Другие типы поддерживают не все БД (непереносимым).

@EmbeddedId указывает на поле составного первичного ключа, а @Embeddable объявляет класс составным ключом.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | @Embeddable  public class BillingAddress implements Serializable {...}  @Entity  @Table(name = "PURCHASE\_ORDERS")  @IdClass(BillingAddress.class)  public class PurchaseOrder {...} |

Обратите внимание, что есть некоторые ключевые требования, которым должен соответствовать класс составного ключа:  
∙ мы должны пометить его с помощью @Embeddable;  
∙ он должен реализовать java.io.Serializable;  
∙ мы должны обеспечить реализацию hashCode() и equals() методы;  
∙ ни одно из полей не может быть сущностью.

## @GenerationType (стратегии генерации)

|  |  |
| --- | --- |
| 1.  2. | @Id  @GeneratedValue(strategy = GenerationType.AUTO) |

AUTO. Стратегия зависит от БД. Для большинства — SEQUENCE.

IDENTITY. Автоматически увеличивающийся столбец БД при каждой операции вставки.

Простой, но не самый производительный.

Hibernate требует значения первичного ключа для каждого управляемого объекта и поэтому должен немедленно выполнить оператор вставки. Это предотвращает использование различных методов оптимизации, таких как пакетная обработка JDBC. (Идентити делает инсерт до персиста).

Генерируется на стороне БД

SEQUENCE. Использует последовательность БД для генерации уникальных значений.

(Для получения следующего значения из последовательности БД требуются дополнительные операторы select.)

Но это не влияет на производительность для большинства приложений. (Секвенс делает селект, чтобы сгенерить id).

Генерируется на стороне Java

TABLE. используется редко. Он моделирует последовательность, сохраняя и обновляя ее текущее значение в таблице БД, что требует использования пессимистических блокировок, которые помещают все транзакции в последовательный порядок. Это замедляет работу вашего приложения

## @EmbeddedId

Используется для определения составного ключа в бине.

# @OrderBy

Определяет порядок элементов коллекции, оцениваемой ассоциацией или коллекцией элементов, в тот момент, когда ассоциация или коллекция извлекаются.

@OrderBy могут быть применены к элементу коллекции. Когда @OrderBy применяется к коллекции элементов базового типа, порядок будет по значению базовых объектов, а имя свойства или поля не используется. При указании порядка для коллекции элементов встраиваемого типа необходимо использовать точечную нотацию для указания атрибута или атрибутов, которые определяют порядок.

# @OrderColumn — как работает, где ставится

Указывает столбец, который используется для поддержания постоянного порядка списка. @OrderColumn указывается в отношении OneToMany или ManyToMany или в коллекции элементов. @OrderColumn указывается на стороне отношения, ссылающейся на коллекцию, которая должна быть упорядочена.

# Различия между @OrderBy и @OrderColumn — пример с БД

@OrderBy в запросе отсортирует, а в кэше вернет неотсортированный порядок. @OrderedColumn сортирует данные с учетом данных в колонке, и в кеше и в запросе.

Указанный порядок @OrderBy применяется только во время выполнения при получении результата запроса.

@OrderColumn приводит к постоянному упорядочению соответствующих данных.

# @Transient

Не будут мапяться.